POLITECNICO DI TORINO
Dipartimento di Automatica e Informatica

Dottorato di Ricerca in
Ingegneria Informatica e dei Sistemi
X ciclo

TESI

Understanding, Evaluating, Formalizing, and Exploiting Code Mobility

Candidato:
Gian Pietro Picco

Relatori:
Prof. Carlo Ghezzi
Prof. Gruia-Catalin Roman

Tutore:
Prof. Silvano Gai

Coordinatore del Dottorato:
Prof. Pietro Laface

1998

A mio padre
To my father

Prefazione

Le reti di calcolatori stanno cambiando rapidamente. È evidente come la loro *dimensione* stia aumentando e tale fenomeno non è confinato soltanto a Internet, il cui tremendo tasso di crescita è ben noto. Reti inter- e intra-organizzazione stanno avendo una diffusione nonché una crescita continua, alimentata dalla disponibilità di hardware a basso prezzo e motivata dalla necessità di canali informativi uniformi, aperti ed efficaci all'interno dell'organizzazione e fra organizzazioni diverse. Il sintomo di questa crescita, così come percepito dagli utenti, è l'aumento di traffico che oggigiorno motiva gli sforzi significativi spesi per migliorare le *prestazioni* dell'infrastruttura di comunicazione. I collegamenti di rete sono migliorati costantemente e gli sviluppi tecnologici stanno portando a un'aumentata potenza computazionale sia sui nodi intermedi che su quelli ai bordi della rete.

Tuttavia, l'aumento delle dimensioni e il miglioramento delle prestazioni delle reti di calcolatori è contemporaneamente la causa e l'effetto di un cambiamento più fondamentale: le reti stanno aumentando la loro diffusione e diventando onnipresenti. Con il termine *diffusione*, si intende sottolineare come la disponibilità di una rete non sia più una costosa aggiunta, bensì una presenza scontata in una piattaforma di elaborazione. Con il termine *onnipresente*, ci si riferisce alla disponibilità di collegamenti in rete indipendentemente dalla locazione fisica. Gli sviluppi nel campo delle reti *wireless* liberano i gli host dal vincolo di rimanere posizionati in una locazione prefissata e rendono possibile un nuovo scenario in cui gli host, collegati alla rete mediante collegamenti wireless, seguono i loro utenti mentre essi cambiano la loro posizione fisica. Tutto ciò accade mentre la disponibilità di sistemi e applicazioni distribuite accessibili a chiunque, come il World Wide Web, ha acceso l'interesse degli utenti finali e dei mercati motivando lo sviluppo di intere classi di applicazioni innovative. Questo fatto, in combinazione con la diffusione e l'onnipresenza delle reti, sta cambiando il modo con cui le reti in generale, e Internet in particolare, sono impiegate. Il loro *ruolo* si sta spostando progressivamente da quello di semplici strumenti di comunicazione verso una piattaforma distribuita su larga scala che supporta servizi innovativi e precedentemente non previsti. Termini come "commercio elettronico" o "telefonia su Internet" sono sintomatici di questo spostamento.

Questa evoluzione non è senza ostacoli. La dimensione crescente delle reti di calcolatori solleva un problema di *scalabilità*. La maggior parte dei risultati che sono significativi per reti di piccole dimensione sono spesso inapplicabili a reti di dimensione mondiale come Internet. Ad esempio, se può essere concepibile applicare un'algoritmo di snapshot a una LAN, le sue prestazioni sono inaccettabili in uno scenario Internet. I collegamenti wireless, il cui impiego nelle reti di calcolatori è spesso denominato *mobile computing* pongono problemi persino più impegnativi. I nodi di rete possono muoversi ed essere collegati in maniera discontinua, pertanto la topologia di rete non è più definita staticamente. Di conseguenza, ciò pone in discussione alcuni dei fondamenti della ricerca sui sistemi distribuiti e si rende necessario aggiornare i tradizionali risultati teorici e tecnologici. Inoltre, i nuovi servizi e modi di utilizzo delle reti spesso richiedono un grado più elevato di adattabilità nei confronti dell'utente finale, mentre la natura dinamica sia dell'infrastruttura di comunicazione che dei requisiti di mercato richiedono un aumento nella flessibilità ed estensibilità.

Vi sono stati molti tentativi di fornire risposte efficaci, attaccando diversi aspetti del problema a diversi livelli di astrazione. La maggior parte degli approcci, tuttavia, cercano di adattare al nuovo scenario i modelli e le tecnologie comunemente adottate, solitamente dando per scontata la tradizionale architettura *client-server*. Ad esempio, CORBA integra la chiamata a procedura remota (RPC) con il paradigma orientato agli oggetti nel tentativo di combinare i benefici del secondo in termini di modularità e riuso con l'ampiamente utilizzato meccanismo di comunicazione fornito dal primo. Tuttavia, ciò non sembra fornire il livello di flessibilità, adattabilità e riconfigurabilità necessario a soddisfare i nuovi requisiti.

Un approccio diverso e innovativo proviene da una promettente area di ricerca che sta affrontando il problema menzionato impiegando codice mobile. La *mobilità di codice* può essere definita informalmente come la possibilità di cambiare dinamicamente i legami tra porzioni di codice e la locazione dove esse vengono eseguite. Questo potente concetto ha generato sviluppi assai interessanti. Tuttavia, nonostante il diffuso interesse in linguaggi e applicazioni per la mobilità di codice, quest'area di ricerca è tuttora abbastanza immatura e questo problema si sta rendendo manifesto sotto vari aspetti.

Prima di tutto, manca ancora un bagaglio terminologico e metodologico comune, e non esiste neanche un termine comunemente accettato per definire l'oggetto di questa ricerca. Chiaramente, ciò ostacola la comunicazione e il confronto fra i vari risultati e, in una parola, la comprensione dei concetti fondamentali in quest'area di ricerca. Ad esempio, la mancanza di un bagaglio concettuale complica l'attività di selezione della tecnologia a codice mobile maggiormente adatta per un dato compito, nonché la valutazione dei benefici che essa può portare nel processo di sviluppo. Un interessante effetto collaterale di questa situazione è che le fondamenta teoriche per la mobilità di codice sono particolarmente povere e i primi modelli formali sono apparsi solo assai recentemente.

L'interesse dimostrato dai mercati e dai media, dovuto al fatto che la mobilità di codice è legata a filo doppio con Internet, ha aggiunto ulteriore rumore introducendo aspettative ingiustificate e informazioni non corrette. Ciò introduce un secondo problema, cioè la mancanza di prove chiare e possibilmente quantitative che confermino le promesse associate alla mobilità di codice. Ancora peggio, se non si considera l'uso di *applet* Java nei navigatori Web, di fatto un'applicazione abbastanza limitata della mobilità di codice, possiamo dire con sicurezza che oggigiorno esistono veramente poche applicazioni distribuite che impiegano il codice mobile in un dominio reale. Questa mancanza di evidenza sta ostacolando un vasto consenso all'approccio e ponendo domande fondamentali e impegnative a tutta l'area di ricerca.

Il contributo di questa tesi affronta i problemi sopramenzionati e, essendo il risultato dell'impatto con un'area di ricerca nuova e immatura, segue un percorso naturale di ricerca che inizia con la formulazione di fondamenta concettuali per il dominio e termina cercando applicazioni per i risultati teorici, così come illustrato dalla struttura stessa di questo documento.

La prima parte della tesi affronta la *comprensione* della mobilità di codice. Viene sviluppata un bagaglio concettuale strutturato secondo tre dimensioni: tecnologie, architetture e applicazioni. Esso è concepito per essere utile ai ricercatori nel fornire un mezzo per classificare e confrontare gli approcci esistenti. Inoltre, è concepito per guidare gli sviluppatori nel valutare le opportunità di impiego della mobilità di codice nelle loro applicazioni distribuite e guidare il corrispondente processo di sviluppo. Il soggetto di questo sforzo è originale nella ricerca sul codice mobile e fornisce guida e struttura al resto del lavoro descritto in questa tesi.

La parte II affronta la *valutazione* della mobilità di codice. Viene scelta la gestione di rete come dominio applicativo in quanto essa è spesso menzionata come banco di prova ideale per la mobilità di codice, e vengono valutati qualitativamente e quantitativamente i benefici derivanti dall'introduzione del codice mobile a livello di progetto e implementazione. In particolare, viene costruito un modello quantitativo di un'applicazione per la gestione di rete e si confrontano varie alternative di progetto con il codice mobile rispetto alla quantità di traffico generata. Nonostante l'applicazione del codice mobile in questo dominio non sia nuova, la valutazione quantitativa presentata qui è originale nell'area di ricerca sul codice mobile e, da un punto di vista più generale relativo all'ingegneria del software, costituisce inoltre un'esperienza degna di nota nel fornire criteri per valutare vantaggi e svantaggi fra diverse architetture.

La parte III affronta la *formalizzazione* della mobilità di codice. Viene introdotto un problema di simulazione distribuita usato per illustrate l'applicazione alla migrazione di componenti software di una notazione e logica originariamente concepite per descrivere il movimento di host fisici in uno scenario di mobile computing. Viene dimostrato come la notazione proposta possa modellare in maniera efficace le astrazioni e meccanismi descritti nella parte I e viene verificata formalmente la correttezza del modello usando la logica associata. Il formalismo impiegato favorisce un'analisi di correttezza assiomatica e basata sulla nozione di stato, in contrasto con altri modelli proposti recentemente che sono algebrici e derivati dal π-calcolo.

La parte IV affronta l'*impiego* della mobilità di codice. Essa rende conto circa attività più recenti e ancora attive che sfruttano quanto sviluppato nelle altre parti della tesi. Il capitolo 14 presenta una libreria per la mobilità di codice concepita sulle fondamenta di quanto sviluppato nella parte I e sull'analisi presentata nella parte II: essa mira a essere flessibile, supportando tutte le strategie di rilocazione del codice mobile, e leggera, minimizzando il carico sulla rete e sulla piattaforma elaborativa. Il capitolo 15 presenta un esempio originale di applicazione della mobilità di codice a un'applicazione convenzionale: la videoconferenza. L'approccio presuppone una *rete attiva* dove i dispositivi di rete posseggono la capacità di eseguire codice mobile e sfrutta questo fatto per fornire adattabilità e flessibilità al server di videoconferenza. Infine, il capitolo 16 descrive un sistema ispirato al modello Linda che fornisce coordinamento tra componenti mobili logici e fisici, cioè host mobili e agenti mobili, cercando di scoprire le simmetrie tra questi due tipi di mobilità. La semantica formale del sistema è definita con il medesimo formalismo descritto nella parte III.

Ringraziamenti

Molte persone hanno contribuito, direttamente o indirettamente, a questa tesi. Prima di tutto voglio ringraziare i miei relatori, Carlo Ghezzi e Catalin Roman. Lavorare con loro è intellettualmente stimolante e, in una parola, divertente. Queste persone, insieme ad Alfonso Fuggetta che ha supervisionato lavori precedenti a questa tesi, mi hanno insegnato la maggior parte di ciò che conosco circa il fare ricerca. Sono profondamente in debito con tutti loro. Voglio ringraziare anche il mio tutore, Silvano Gai, il quale mi ha fornito costantemente supporto e consiglio durante il dottorato e mi ha lasciato libero di stabilire collaborazioni di ricerca con persone di altre università.

Oltre alle persone sopra menzionate, ho avuto inoltre la fortuna di lavorare con colleghi dottorandi dalle eccezionali qualità professionali e umane. Molti di loro sono stati co-autori di articoli che descrivono lavori contenuti in questa tesi e, ancora più importante, molti di loro sono diventati miei carissimi amici durante questi anni. Voglio quindi ringraziare Mario Baldi, Antonio Carzaniga, Gianpaolo Cugola, Jean-Philippe Martin-Flatin, Pete McCann, Amy Murphy, Fulvio Risso e Giovanni Vigna. Molti altri al Politecnico di Torino, Politecnico di Milano e Washington University, St. Louis, hanno contribuito in diversi modi, dandomi consigli o ponendo domande impegnative che hanno innescato discussioni stimolanti nonché interminabili. Grazie a tutti loro.

Questo per quanto riguarda l'ambiente di ricerca. Al di fuori di esso, voglio menzionare almeno due persone che hanno giocato, sebbene indirettamente, un ruolo chiave nella mia tesi—e non solo in essa. Voglio esprimere la mia gratitudine a mia madre, che mi ha sempre incentivato nel perseguire i miei obiettivi, qualunque essi fossero, ma mi ha dato anche saggi consigli nel ricordarmi ciò che è realmente importante nella vita. Dulcis in fundo, *voglio ringraziare Cristina, la quale mi è sempre stata accanto in questi anni, condividendo con me i momenti di gioia e quelli difficili e incoraggiandomi sempre con entusiasmo, anche nella mia decisione di intraprendere un periodo di ricerca all'estero.*
È bello avere qualcuno su cui contare.

Grazie.

Acknowledgments

Many persons contributed, directly or indirectly, to this thesis. First of all I want to thank my advisors, Carlo Ghezzi and Catalin Roman. Working with them is intellectually stimulating and, well, in one word just fun. These persons, together with Alfonso Fuggetta who supervised earlier work prior to this thesis, taught me most of what I know about doing research. I am profoundly indebted with all of them. I want to thank also my tutor Silvano Gai. He provided me with constant support and advice during the Ph.D. and set me free to establish research collaborations with people at different universities.

Besides the aforementioned people, I had also the fortune to work with doctoral colleagues of exceptional professional and human qualities. Many of them co-authored some of the papers describing work contained in this thesis and, even more important, many of them have become close friends of mine during these years. So, I want to thank Mario Baldi, Antonio Carzaniga, Gianpaolo Cugola, Jean-Philippe Martin-Flatin, Pete McCann, Amy Murphy, Fulvio Risso, and Giovanni Vigna. Many others at Politecnico di Torino, Politecnico di Milano, and Washington University, St. Louis, have contributed in many ways, providing me with advice or bringing up challenging questions that triggered endless as well as stimulating discussions. Thanks to all of them.

So much for the research ambit. Outside of it, I wish to mention at least two persons who played, although indirectly, a key role in this thesis—and not only in it. I want to express my gratitude to my mother, who has always incentivated me in pursuing my goals, whichever they were, but has given me also wise advice in constantly reminding me what really matters in life. *Dulcis in fundo*, I want to thank Cristina. She has always been close to me in these years, sharing with me both happy and difficult moments, and she always encouraged me with enthusiasm, also in my decision of spending a research period abroad. It is good to have someone you can count on.

Thank you.

Contents

Introduction

Computer networks are changing rapidly. It is evident that their *size* is increasing, and this phenomenon is not confined just to the Internet, whose tremendous growth rate is well-known. Intra- and inter-organization networks experience an increasing diffusion and growth as well, fostered by the availability of cheap hardware and motivated by the need for uniform, open, and effective information channels inside and across the organizations. The symptom of this growth, as perceived by the users, is the increase in traffic which is nowadays motivating significant efforts to enhance the *performance* of the communication infrastructure. Network links are improved constantly and technological developments lead to increased computational power on both intermediate and end network nodes.

However, the increase in size and performance of computer networks is both the cause and the effect of a more fundamental change: networks are becoming pervasive and ubiquitous. By *pervasive*, we mean that network connectivity is no longer an expensive add-on, rather it is often assumed implicitly when considering a computing platform. By *ubiquitous*, we refer to the availability of network connectivity independently of the physical location. Developments in wireless technology free network hosts from the constraint of being placed at a fixed physical location and enable a new scenario where the hosts, still connected to the net by wireless links, follow their users while their physical location is changing. This happens while the availability of distributed applications and systems accessible to the general public (e.g., the World Wide Web) has triggered the interest of end users and markets, motivating the development of entirely new classes of applications. This, combined with pervasiveness and ubiquity, is changing the way the networks in general, and Internet in particular, are exploited. Their *role* is shifting progressively from that of a plain communication media to a large scale distributed platform that supports innovative and previously unforeseen services. Terms like "electronic commerce", or "Internet telephony" are symptomatic of this shift.

This evolution is not without obstacles. The growing size of networks raises a problem of *scalability*. Most of the results that are significant for small networks are often inapplicable when scaled to a world-wide network like the Internet. For instance, while it might be conceivable to apply a global snapshot algorithm to a LAN, its performance is unacceptable in an Internet setting. Wireless connectivity, whose exploitation in computer networks is often referred to as *mobile computing*, poses even tougher problems [35, 51]. Network nodes may move and be connected discontinuously, hence the topology of the network is no longer defined statically. As a consequence, some of the basic tenets of research on distributed systems are undermined, and adaptation of well-known theoretical and technological results is needed. On the other hand, the new services and modes of use of computer networks often demand for a higher degree of *customizability* by the end user, whereas the dynamic nature of both the underlying communication infrastructure and the market requirements demand for increased *flexibility* and *extensibility*.

There have been many attempts to provide effective answers, attacking different facets of the problem at different layers of abstraction. Most of the approaches, however, try to adapt well-established models and technologies to the new scenario and usually take for granted the traditional client-server architecture. For example, CORBA [79] integrates remote procedure calls (RPCs) with the object-oriented paradigm in an attempt to combine the benefits of the latter in terms of modularity and reuse with the well-established communication mechanism of the former. However, this does not seem to provide the intended degree of flexibility, customizability, and reconfigurability needed to cope with the new requirements.

A different and innovative approach comes from a promising research area which is attacking the problems mentioned so far by exploiting mobile code. *Code mobility* can be defined informally as the capability to change dynamically the bindings between code fragments and the location where they are executed [21]. This powerful concept has originated a very interesting range of developments. However, despite the widespread interest in mobile code technology and applications, this research area is still quite immature, and this problem is becoming manifest under many respects.

First of all, a sound terminological and methodological common framework is still missing, and there is not even a commonly agreed term to qualify the subject of this research[1]. Clearly, this hampers communication and comparison of results and, in one word, the understanding of the fundamental concepts of this research area. For instance, the lack of a conceptual framework complicates the activity of selecting the mobile code technology that is best suited for a given task, as well as to estimate the benefits it can bring in the development process. An interesting side effect of this situation is that the theoretical background for code mobility is particularly poor, and only until very recently formal models for mobility have begun to appear.

The interest demonstrated by markets and media, due to the fact that mobile code research is tightly bound to the Internet, has added an extra level of noise by introducing hypes and sometimes unjustified expectations. This brings in a second problem, i.e. the lack of sound and possibly quantitative evidence confirming that the promises associated with the mobile code approach really hold. Even worse, if we do not consider the use of Java applets in Web browsers—indeed, a limited application of code mobility—we can safely say that there are presently very few distributed applications that exploit mobile code in a real domain. This lack of evidence is hampering a widespread acceptance of the approach and posing fundamental challenging question to the whole research area.

[1] Hereafter, we use interchangeably the terms *code mobility* and *mobile code*, although other authors prefer different terms such as *mobile computations*, *mobile object systems*, or *program mobility*.

The contribution of this thesis addresses the aforementioned problem areas and, being the result of the impact with a new and immature research area, follows a natural path of discovery that starts from the formulation of a conceptual framework for the field and terminates by seeking for application of the theoretical results, as illustrated by the very structure of this document.

The first part of the thesis is concerned with *understanding* code mobility. It develops a conceptual framework structured along three dimensions: technologies, architectures, and applications. The framework is meant to be useful to researchers in providing a means to classify and compare existing approaches. Also, it is conceived to guide the developers in assessing the opportunities for the exploitation of code mobility in their distributed applications and guide the corresponding choices in the development process. The subject of this endeavor is original in research on mobile code and has provided guidance and structure to the remainder of the work described in this thesis.

Part II is concerned with *evaluating* code mobility. We choose network management as our reference applicative domain as it is often mentioned as an ideal testbed for code mobility, and we evaluate qualitatively and quantitatively the benefits that are expected from the introduction of mobile code in the design and implementation. In particular, we build a quantitative model of a network management application and compare different mobile code designs with respect to the amount of network traffic generated. Although the application of mobile code is not new in this domain, the quantitative evaluation presented here is novel in the research area of code mobility and, from a more general software engineering perspective, constitutes also a valuable experience in providing quantitative criteria to evaluate the tradeoffs of different architectures.

Part III is concerned with *formalizing* code mobility. We introduce a distributed simulation problem and use it to illustrate the application to the migration of software components of a notation and logic originally conceived to describe the movement of physical hosts in a mobile computing scenario. We demonstrate how the notation proposed can model effectively the abstractions and mechanisms discussed in Part I and then verify formally the correctness of the model using the logic associated. The formalism employed fosters axiomatic state-based reasoning, in contrast with other models proposed recently that are algebraic approaches derived from π-calculus.

Part IV is concerned with *exploiting* code mobility. It reports about recent and still on-going activities that leverage off of what developed in the other parts of the thesis. Chapter 14 presents a mobile code toolkit conceived on the grounds of the framework developed in Part I and of the analysis presented in Part II: it is flexible, by supporting all the relocation strategies for mobile code, and lightweight, by minimizing the overhead on the network and the computing platform. Chapter 15 presents an original example of application of a mobile code design to a conventional distributed application: videoconferencing. The approach assumes an *active network* where network devices can execute mobile code and exploits this fact to provide customizability and adaptability of the conference server. Finally, Chapter 16 describes a system inspired by the Linda model that provides coordination of physical and logical mobile components, i.e. mobile hosts and mobile code, trying to uncover the simmetries between these two notions of mobility. The formal semantics of the system is specified with the same formalism described in Part III.

Part I

Understanding Code Mobility

Chapter 1

Motivations, Background, and Approach

Code mobility is not a new concept. In the recent past, several mechanisms and facilities have been designed and implemented in order to move code among the nodes of a network. Examples are remote batch job submission [16] and the use of PostScript [3] to control printers. A more structured approach has been followed in the research work on distributed operating systems. In this context, the main problem is to support the migration of active processes and objects (together with their state and the associated code) at the operating system level. A survey of systems providing process or object migration can be found in [78].

Process migration concerns the transfer of an operating system process from the machine where it is running to a different one. Migration mechanisms manage the bindings between the process and its execution environment (e.g., open file descriptors and environment variables) to allow the process to resume its execution seamlessly in the remote environment. Process migration facilities have been introduced at the operating system level to achieve load balancing across network nodes. Therefore, most of these facilities provide transparent process migration, in which the programmer has neither control nor visibility of the migration process. Other systems provide some form of control over the migration process. For example, in the Locus system [111] process migration can be triggered either by an external signal or by the invocation of the `migrate` system call.

Object migration makes it possible to move objects among address spaces, implementing a finer grained mobility with respect to systems providing migration at the process level. For example, the Emerald system [57] provides object migration at any level of granularity ranging from small, atomic data to complex objects. Emerald does not provide complete transparency since the programmer can determine objects locations and may request explicitly the migration of an object to a particular node. An example of system providing transparent migration is the COOL [63] object-oriented subsystem. It is built on top of the Chorus operating system [95] and is able to move objects among address spaces without user intervention or knowledge.

Process and object migration address the issues that arise when code and state are moved among the hosts of a loosely coupled, small scale distributed system. These results have been taken as a starting point by a new breed of systems providing different forms of code mobility. These systems, often referred to as *Mobile Code Systems* (MCSs), exhibit several innovations with respect to the existing approaches:

Code mobility is exploited on an Internet-scale Distributed systems providing process or object migration have been designed with small-scale computer networks in mind, thus assuming high bandwidth, small predictable latency, trust, and often homogeneity. Mobile code systems address instead a large scale setting where networks are composed of heterogeneous hosts, managed by different authorities with different levels of trust, and connected by links with different bandwidths ranging from wireless slow connections to fast optical links.

Programming is location aware Location is a pervasive abstraction that has a strong influence on both the design and the implementation of distributed applications. Mobile code systems do not paper over the location of application components, rather, applications are location-aware and may take actions based on such knowledge.

Mobility is under programmer's control The programmer is provided with mechanisms and abstractions that enable the shipping and fetching of code fragments and entire components to remote nodes. The underlying run-time support provides basic functionalities (e.g., data marshaling, code check-in, and security), but it does not have any control over migration policies.

Mobility is not performed just for load balancing Process and object migration aim at supporting load balancing and performance optimization. Mobile code systems address, service customization, dynamic extension of application functionality, autonomy, fault tolerance, support for disconnected operations.

Many MCSs have been proposed. This lively and sometimes chaotic scenario has generated some confusion about concepts, abstractions, terms, and semantics of code mobility.

First of all, the fundamental distinction among implementation technologies, specific applications, and paradigms used to design these applications is not always made clear. In an early and yet valuable assessment of code mobility [46] the authors analyze and compare issues that belong to different abstraction levels. Analogously, in a recent work about autonomous objects [14], *mechanisms* like REV [100] and RPC [15] are compared to the Echo distributed *algorithms* [26], to *applications* like "intelligent e-mail" and Web browsers, and to *paradigms* for structuring distributed applications, like mobile agents. These are different concepts and notions that cannot be compared directly. It is almost like comparing the `emacs` editor with the `fork` UNIX system call and the client-server design paradigm.

There is also confusion on terms. For instance, several systems [54, 83] claim to be able to move the *state* of a component together with its code, based on the fact that they allow the programmer to pack some portion of the data space of an executing component before the component's code is sent to a remote destination. This is quite different from the situation where the run-time image of the component is transferred as a whole, including its execution state, i.e. program counter, call stack, and so on. In the former case, it is the programmer's task to rebuild the execution state of component after its migration, using the data transferred with the code. Conversely, in the latter case this task is carried out by the run-time support of the MCS.

Another terminological confusion stems out from the excessive overload of the term "mobile agent". This term is used with different and somewhat overlapping meanings in both the distributed systems and artificial intelligence research communities. In the distributed system community the term "mobile agent" is used to denote a software component that is able to move between different execution environments. This definition has actually different interpretations. For example, while in Telescript [122] an agent is represented by a thread that can migrate among different nodes carrying its execution state, in TACOMA [54] agents are just code fragments associated with initialization data that can be shipped to a remote host, without the ability to migrate again while in execution. On the other hand, in the artificial intelligence community the term "agent" denotes a software[1] component that is able to achieve a goal by performing actions and reacting to events in a dynamic environment [65]. This component is directed by knowledge of the relationships among events, actions, and goals. Moreover, knowledge can be exchanged with other agents, or increased by some inferential activity [41]. Although mobility is not the most characterizing aspect of these entities [125] there is a tendency to blend this notion of intelligent agent with the one coming from distributed systems and thus assume implicitly that a mobile agent is also intelligent and vice versa. This is actually creating confusion about between the layer providing mobility and the one exploiting mobility.

Finally, there is no definition or agreement about the distinguishing characteristics of languages supporting code mobility. In [59], Knabe lists the essential characteristics of a mobile code language. They include support for manipulating, transmitting, receiving, and executing "code-containing objects". However, there is no discussion about how to manage the state of mobile components. Other contributions [84, 46] consider only the support for mobility of both code and state, without mentioning weaker forms of code mobility involving code migration alone—as we discuss later on in this part of the thesis.

Certainly, confusion and disagreement are typical of a new and still immature research field. Nevertheless, research developments are fostered not only by novel ideas, mechanisms, and systems, but also by a rationalization and conceptualization effort that re-elaborates on the raw ideas seeking for a common and stable ground on which to base further endeavors. Research on code mobility is no exception. The technical concerns raised by performance and security of MCSs are not the only factors hampering full acceptance and exploitation of mobile code. A conceptual framework is needed to foster understanding of the multi-faceted mobile code scenario and enable researchers and practitioners to

[1]For the purpose of what discussed in this thesis we ignore the implications of broader notions of agent which are not restricted to the software domain.

assess and compare different solutions with respect to a common set of reference concepts and abstractions—and go beyond it. To be effective, this conceptual framework should also provide valuable information to application developers, actually guiding the evaluation of opportunities for exploitation of code mobility during the different phases of application development.

These considerations provide the rationale for the classification presented in this part of the thesis. The classification introduces abstractions, models, and terms to characterize the different approaches to code mobility proposed so far, highlighting commonalities, differences, and applicability. The classification is organized along three dimensions that are of paramount importance during the actual development process: *technologies*, *design paradigms*, and *application domains.* Mobile code technologies are the languages and systems that provide *mechanisms* enabling and supporting code mobility. Some of these technologies have been already mentioned and are discussed in greater detail in the following section. Mobile code technologies are used by the application developer in the implementation stage. Design paradigms are the *architectural styles* that the application designer uses as building blocks in defining the application architecture. An architectural style identifies a specific configuration for the components and their mutual interactions. Client-server and peer-to-peer are well-known examples of design paradigms. Application domains are *classes of applications* that share the same general goal, e.g., distributed information retrieval or electronic commerce. They play a role in defining the application requirements. The expected benefits of code mobility in a number of application domains is the motivating force behind this research field.

Our classification will break down in a vertical distinction among these three layers, as well as in an horizontal distinction among the peculiarities of the various approaches found in literature, proving useful in accomodating the current situation of the field. Chapter 2 presents a general model and a classification of the mechanisms provided by mobile code technologies. The classification is then used to survey and characterize several MCSs. Chapter 3 presents mobile code design paradigms and discusses their relationships with mobile code technologies. Chapter 4 discusses the advantages of the mobile code approach and presents some application domains that are supposed to benefit from the use of some form of code mobility.

The work described in this part of the thesis has been published in [30, 29, 21, 37].

Chapter 2

Technologies

Mobile code technologies are constituted mainly by programming languages and the corresponding run-time support. These technologies apparently provide very diverse abstractions and primitives. In this chapter we first introduce some reference abstractions and then seek out and classify the different mechanisms that allow an application to move code and state among the nodes of a network, as well as other mechanisms and issues in language design that are affected by mobility. Then, we validate our classification by characterizing the features provided by several of the MCSs proposed in literature.

2.1 A Virtual Machine for Code Mobility

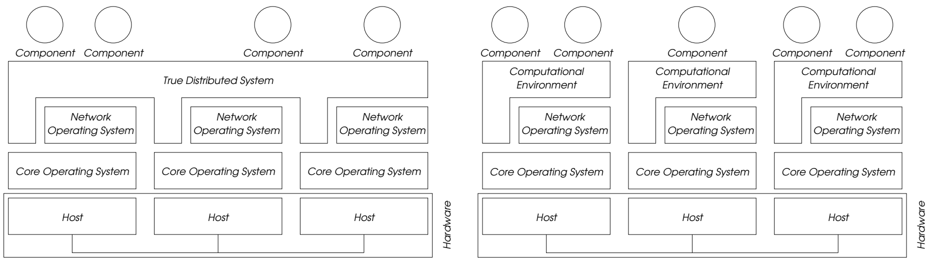


Figure 2.1: Traditional systems vs. MCSs. Traditional systems, on the left hand side, may provide a TDS layer that hides the distribution from the programmer. Technologies supporting code mobility, on the right hand side, explicitly represent the location concept, thus the programmer needs to specify *where*—i.e. in which CE—a computation has to take place.

Traditional distributed systems can be accommodated in the virtual machine shown on the left-hand side of Figure 2.1. The lowest layer, just upon the hardware, is constituted by the *Core Operating System* (COS). The COS can be regarded as the layer providing the basic operating system functionalities, such as file system, memory management, and process support. No support for communication or distribution is provided by this layer. Non-transparent communication services are provided by the *Network Operating System* (NOS)

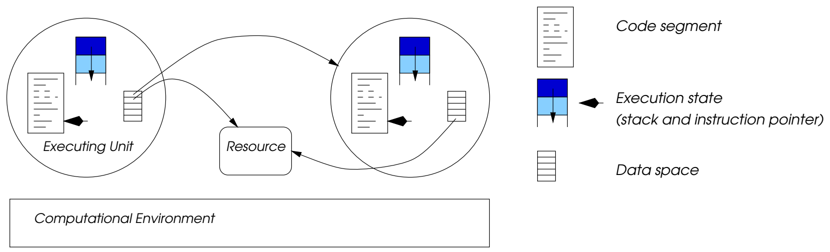


Figure 2.2: The internal structure of an executing unit.

layer. Applications using NOS services address explicitly the host targeted by communication. For instance, socket services can be regarded as belonging to the NOS layer, since a socket must be opened by specifying explicitly a destination network node. The NOS, at least conceptually, uses the services provided by the COS, e.g., memory management. Network transparency is provided by the *True Distributed System* (TDS) layer. A TDS implements a platform where components, located at different sites of a network, are perceived as local. Users of TDS services do not need to be aware of the underlying structure of the network. When a service is invoked, there is no clue about the node of the network that will actually provide the service, and even about the presence of a network at all. As an example, CORBA [79] services can be regarded as TDS services since a CORBA programmer is usually unaware of the network topology and always interacts with a single well-known object broker. At least in principle, the TDS is built upon the services provided by the underlying NOS.

Technologies supporting code mobility take a different perspective. The structure of the underlying computer network is not hidden from the programmer, rather it is made manifest. In the right-hand side of Figure 2.1 the TDS is replaced by *Computational Environments* (CEs) layered upon the NOS of each network host. In contrast with the TDS, the CE retains the “identity” of the host where it is located. The purpose of the CE is to provide applications with the capability to dynamically relocate their components on different hosts. Hence, it leverages off of the communication channels managed by the NOS and of the low-level resource access provided by the COS to handle the relocation of code, and possibly of state, of the hosted software components.

To give uniformity to the treatment of our taxonomy, we distinguish the components hosted by the CE in *executing units* (EUs) and *resources*. Executing units represent sequential flows of computation. Typical examples of EUs are single-threaded processes or individual threads of a multi-threaded process. Resources represent entities which can be shared among multiple EUs, such as a file in a file system, an object shared by threads in a multi-threaded object-oriented language, or an operating system variable. Figure 2.2 illustrates our modeling of EUs as the composition of a *code segment*, which provides the static description for the behavior of a computation, and a *state* composed of a *data space* and an *execution state*. The data space is the set of references to components that can be accessed by the EU. As it will be explained later, these components are not necessarily co-located with the EU on the same CE. The execution state contains private data that cannot be shared, as well as control information related to the EU state, such as the call stack and the instruction pointer. For example, a Tcl interpreter P_X executing a Tcl script X can be regarded as an EU where the code segment is X; the data space is composed of variables containing the handles for files and references to system environment variables used by P_X; the execution state is composed of the program counter and the call stack maintained by the interpreter, along with the other variables of X.

2.2 Mobility Mechanisms

In conventional systems, each EU is bound to a single CE for its entire lifetime. Moreover, the binding between the EU and its code segment is generally static. Even in environments that support dynamic linking, the code linked belongs to the local CE. This is not true for MCSs. In MCSs, the code segment, the execution state, and the data space of an EU can be relocated to a different CE. In principle, each of these EU constituents might move independently. However, we will limit our discussion to the alternatives adopted by existing systems. Figure 2.3 presents a classification of mobility mechanisms.

The portion of an EU that needs to be moved is determined by composing orthogonal mechanisms supporting mobility of code and execution state with mechanisms for data space management. For this reason, we will analyze them separately.

2.2.1 Code and Execution State Mobility

Existing MCSs offer two forms of mobility, characterized by the EU constituents that can be migrated. *Strong mobility* is the ability of an MCS (called *strong MCS*) to allow migration of both the code and the execution state of an EU to a different CE. *Weak mobility* is the ability of an MCS (called *weak MCS*) to allow code transfer across different CEs; code may be accompanied by some initialization data, but no migration of execution state is involved.

Strong mobility is supported by two mechanisms: *migration* and *remote cloning*. The migration mechanism suspends an EU, transmits it to the destination CE, and then resumes it. Migration can be either proactive or reactive. In *proactive* migration, the time and destination for migration are determined autonomously by the migrating EU. In *reactive* migration, movement is triggered by a different EU that has some kind of relationship with the EU to be migrated, e.g., an EU acting as a manager of roaming EUs. The remote cloning mechanism creates a copy of an EU at a remote CE. It differs from the migration mechanism because the original EU is not detached from its current CE. As in migration, remote cloning can be either proactive or reactive.

Mechanisms supporting weak mobility provide the capability to transfer code across CEs and either link it dynamically to a running EU or use it as the code segment for a new EU. Such mechanisms can be classified according to the direction of code transfer, the nature of the code being moved, the synchronization involved, and the time when code is actually executed at the destination site.

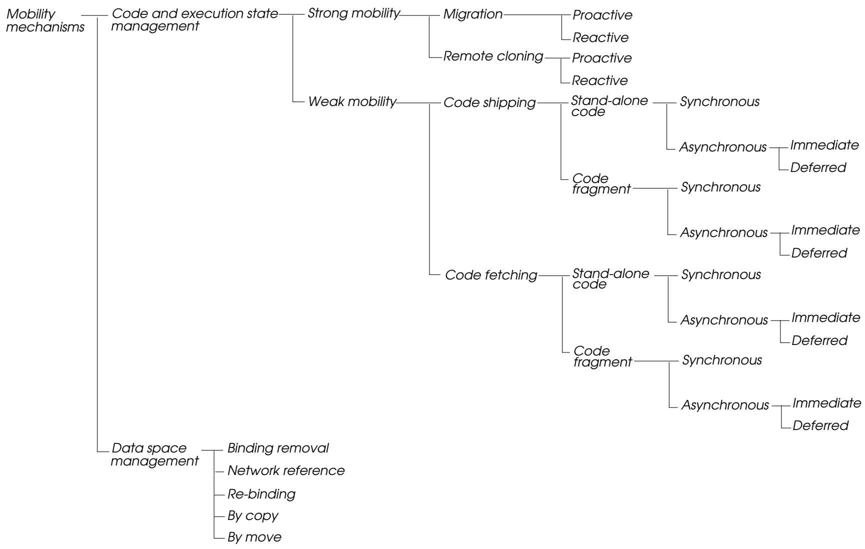


Figure 2.3: A classification of mobility mechanisms.

As for direction of code transfer, an EU can either *fetch* the code to be dynamically linked and/or executed, or *ship* such code to another CE. The code can be migrated either as *stand-alone code* or as a *code fragment*. Stand-alone code is self-contained and will be used to instantiate a new EU on the destination site. Conversely, a code fragment must be linked in the context of already running code and eventually executed. Mechanisms supporting weak mobility can be either *synchronous* or *asynchronous*, depending on whether the EU requesting the transfer suspends or not until the code is executed. In asynchronous mechanisms actual execution of the code transferred may take place either in an *immediate* or *deferred* fashion. In the first case, the code is executed as soon as it is received, while in a deferred scheme execution is performed only when a given condition is satisfied—e.g., upon first invocation of a portion of the code fragment or as a consequence of an application event.

2.2.2 Data Space Management

Upon migration of an EU to a new CE, its data space, i.e. the set of bindings to components accessible by the EU, must be rearranged. This may involve voiding bindings to resources, reestablishing new bindings, or even migrating some resources to the destination CE along with the EU. The choice depends on the nature of the resources involved, the type of binding to such resources, as well as on the requirements posed by the application.

We model resources as a triple $Resource = \langle I,V,T \rangle$, where I is a unique identifier, V is the value of the resource, and T is its type, which determines the structure of the information contained in the resource as well as its interface. The type of the resource determines also whether the resource is *transferrable* or *not transferrable*, i.e. whether, in principle, it can be migrated over the network or not. For example, a resource of type "stock data" is likely to be transferrable, while a resource of type "printer" probably is not. Transferrable resource instances can be marked as *free* or *fixed*. The former can be migrated to another CE, while the latter are associated permanently with a CE. This characteristic is determined according to application requirements. For instance, even if it might be conceivable to transfer a huge file or an entire database over the network, this might be undesirable for performance reasons. Similarly, it might be desirable to prevent transfer of classified resources, even independently of performance considerations.

Resources can be bound to an EU through three forms of binding, which constrain the data space management mechanisms that can be exploited upon migration. The strongest form of binding is *by identifier*. In this case, the EU requires that, at any moment, it must be bound to a given uniquely identified resource. Binding by identifier is exploited when an EU requires to be bound to a resource that cannot be substituted equivalently by some other resource.

A binding established *by value* declares that, at any moment, the resource must be compliant with a given type and its value cannot change as a consequence of migration. This kind of binding is usually exploited when an EU is interested in the *contents* of a resource and wants to be able to access them locally. In this case, the identity of the resource is not relevant, rather the resource migrated must have the same type and value of the one present on the source CE.

The weakest form of binding is *by type*. In this case, the EU requires that, at any moment, the bound resource is compliant with a given type, no matter what its actual value or identity are. This kind of binding is exploited typically to bind resources that are available on every CE, like system variables, libraries, or network devices. For example, if a roaming EU needs to access the local display of a machine to interact with the user through a graphical interface, it may exploit a binding with a resource of type "display". The actual value and identifier of the resource are not relevant, and the resource actually bound is determined by the current CE. Note that it is possible to have different types of binding to the *same* resource. In the example above, suppose that the roaming EU, in addition to interact with the local user through the display, needs to report progress back to the user that "owns" the EU. This is accomplished by creating, at startup, a binding by identifier to the display of the owner and a binding by type to the same resource. As we will explain shortly, after the first migration the bindings will be reconfigured so that the binding by identifier will retain its association with the owner's display, while the binding by type will be associated with the display on the destination CE.

The discussion above evidences that two classes of problems must be addressed by data space management mechanisms upon migration of an EU: resource relocation and binding reconfiguration. The way mechanisms tackle these problems is constrained both by the nature of the resources involved and the forms of binding to such resources. These relationships are analyzed hereafter and summarized in Table 2.1.

Let us consider a migrating executing unit U whose data space contains a binding B to a resource R. A first general mechanism, which is independent of the type of binding or resource is *binding removal*. In this case, when U migrates, B is simply discarded. If access to bound resources must be preserved, different

	Free Transferrable	*Fixed Transferrable*	*Fixed Not Transferrable*
By Identifier	By move (Network reference)	Network reference	Network reference
By Value	By copy (By move, Network reference)	By copy (Network reference)	(Network reference)
By Type	Re-binding (Network reference, By copy, By move)	Re-binding (Network reference, By copy)	Re-binding (Network reference)

Table 2.1: Bindings, resources and data space management mechanisms.

mechanisms must be exploited.

If U is bound to R *by identifier*, two data space management mechanisms are suitable to preserve resource identity. The first is relocation *by move*. In this case, R is transferred, along with U to the destination CE and the binding is not modified (Figure 2.4a). Clearly, this mechanisms can be exploited only if R is a free transferrable resource. Otherwise, a *network reference* mechanism must be used. In this case, R is not transferred and once U has reached its target CE, B is modified to reference R in the source CE. Every subsequent attempt of U to access R through B will involve some communication with the source CE (Figure 2.4b). The creation of inter-CE bindings is often not desirable because it exposes U to network problems—e.g., partitioning, or delays—and makes it difficult to manage state consistency since the data space is actually *distributed* over the network. On the other hand, moving away a resource from its CE may cause problems to other EUs that own bindings to the moved resource. This latter situation may be managed in different ways. A first approach is to apply binding removal, i.e. to void bindings to the resource moved (see top of Figure 2.4a). Subsequent attempts to access the resource through such bindings will rise an exception. A second approach is to retain the bindings to the resource at its new location by means of network references (see bottom of Figure 2.4a).

If B is *by value* and R is transferrable, the most convenient mechanism is data space management *by copy* because the identity of the resource is not relevant. In this case, a copy R' of R is created, the binding to R is modified in order to refer to R', and then R' is transferred to the destination CE along with U (see Figure 2.4c). Management *by move* satisfies the requirements posed by bindings by value but, in some cases, may be less convenient. In fact, in this case R would be removed from the source CE and other EUs owning bindings to R would have to cope with this event. If R cannot be transferred, the use of the network reference mechanisms is the only possible solution, with the drawbacks described previously.

If U is bound to R *by type*, the most convenient mechanism is *re-binding*. In this case B is voided and re-established after migration of U to another resource R' on the target CE having the same type of R (Figure 2.4d). Re-binding exploits the fact that the only requirement posed by the binding is the type of the resource, and avoids resource transfers or the creation of inter-CE bindings. Clearly, this mechanism requires that, at the destination site, a resource of the same type of R exists. Otherwise, the other mechanisms can be used depending

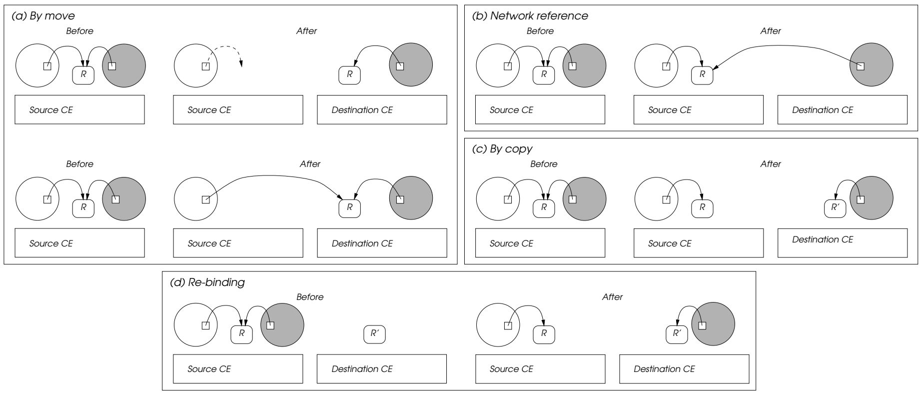


Figure 2.4: Data space management mechanisms. For each mechanism, the configuration of bindings before and after migration of the grayed EU is shown.

on the type and characteristics of the resource involved.

The existing MCSs exploit different strategies as far as data space management is concerned. The nature of the resource and the type of binding is often determined by the language definition or implementation, rather than by the application programmer, thus constraining the mechanisms exploited. For instance, files are usually considered a fixed unique resource, and migration is usually managed by voiding the corresponding bindings, although files in principle could be migrated along with an EU. Replicated resources are often provided as built-in to provide access to system features in a uniform way across all CEs. The next section will provide more insights about mobility mechanisms in existing MCSs.

2.3 Security Mechanisms

Security is a big concern in systems providing support to code mobility and is often considered the main limitation to the widespread use of mobile code technologies. MCSs provide a distributed computing infrastructure on which applications belonging to different (usually untrusted) users can execute concurrently. On the other side, the CEs that compose the infrastructure may be managed by different authorities (e.g., a university or a company) with different and possibly conflicting objectives and may communicate across untrusted communication infrastructures, e.g., the Internet. In this scenario several attacks are possible. Unauthorized people may eavesdrop network traffic to access or modify application data while it is in transit over the network. Applications may try to masquerade as privileged applications or claim to belong to particular unaware users when trying to move code to remote CEs. A CE, on the side, could try to impersonate another legitimate CE in order to receive application data and gain access to application private information. In addition, applications may try to build several kinds of attacks against the CEs supporting their execution. First of all, an application may try to gain privileged or unrestricted access to private information belonging to the CE, e.g., its secret key, or to the underlying operating system, e.g., the password file of a UNIX machine. In addition, an application may try to misuse services offered by the CE to attack the CE itself or to use the CE as the basis for probing and exploiting the vulnerabilities of other systems. For example, unrestricted access to the socket facility of the operating system underlying the CE could be used to attack hosts behind a firewall. As another example, unregulated access to the file system may allow an application to install virii or trojan horses. Unrestricted access to CE resources may also allow an application to build denial-of-service attacks, in which an application tries to jeopardize the functionalities of the CEs by exhausting its memory, storage system or CPU cycles. Applications may also try to disclosure information, misuse services or denial normal execution of other applications. For example, if applications share the same address space, an application could examine the data or code of other applications in order to steal strategic information or tamper with their execution. An application may even try to induce another application to execute arbitrary code that carries out malicious operations, or to avoid its correct or complete execution by flooding the application with messages or simply by requesting its termination. MCSs introduce a completely new security issue: since application or their EUs may travel across different CEs with different level of trust, CEs could try to tamper with EUs code or state in order to disclosure application private information, gain competitive advantage with respect to other CEs, or modify application code so that they will attack other CEs or applications. Consider for example an application composed of an EU that roams the network in search of the best offert for a compact disc. A CE executing the EU may try to modify the EU data structure containing the prices offered by other companies and then change the code of the agent so that information collected by visiting other CEs will be ignored during the determination of the best offert.

The scenario described so far suggests that security issues could be divided into two classes: *inter-CE security*, i.e. the issues related to the problem of protecting entities inside a CE, and *intra-CE security* that includes the issues raised by interactions among remote entities. The classification of security mechanisms is shown in Figure 2.5.

2.3.1 Inter-CE Security

Inter-CE security issues encompasses authentication, integrity, and privacy of the communication between two CEs, an EU and a remote CE, and two remote EUs.

When interacting, CEs need to authenticate each other in order to be protected from spoofing [27]. Authentication mechanisms can be based simply on identifiers (network addresses or names) that provide little or no security or may use some stronger mechanism based on cryptography [97], namely, secret-key authentication systems (a la Kerberos [102]), or authentication systems based on public-key cryptography and digital signatures, like the ones conforming to the X.509 standard [52]. In addition, authentication need to be accompanied by mechanisms that provide for integrity and privacy of the exchanged data, in particular of code or EUs that are moved from CE to CE. Integrity mechanisms ensure that transmitted data is not modified, either by malicious intent or by errors in the transmission process, using cryptographic checksums (e.g.,

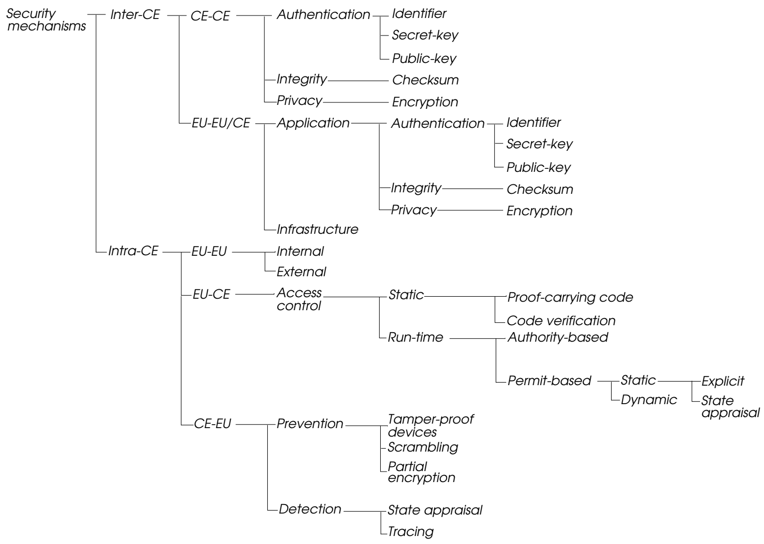


Figure 2.5: A classification of security mechanisms.

MD5 [90]). Privacy mechanisms ensure that third parties not involved in inter-CE communication cannot access the information transmitted over the network using encryption (e.g., DES [76]).

Similar to the previous case, interactions between an EU and a remote CE or between two remote EUs requires mechanisms to authenticate the partners of communication and to protect data during transmission over an untrusted channel. In the case of EUs, these mechanisms can be implemented at the application level including security information in the data being exchanged (e.g., agent id and IP address or stronger checksums and digital signatures) or can be mediated by the underlying CEs that use CE-to-CE security mechanisms in order to protect EU-to-CE or EU-to-EU communication.

2.3.2 Intra-CE Security

Intra-CE security addresses the problem of protecting the entities that compose a single computational environment. Intra-CE security encompasses security among different EUs, protection of the hosting CE and the associated resources from EUs, and protection of EUs from the CE. Most of these issues are addressed by *authorization* that determines the acceptable actions of an entity on the basis of the its identity, as determined by the authentication process.

EUs hosted by a CE must be protected by unauthorized or malicious attempts to tamper with each other. Protection is achieved by means of access control mechanisms. These mechanisms can be *internal* or *external*. Internal mechanisms allow an EU to examine requests for services and explicitly granting access on the basis of the identity of the requesting EU and the service parameters. For example, an internal communication security mechanism may act as a wrapper, allowing an EU to examine the identity of the partner that requested the setup of a communication channel before actually accepting the connection.

External security mechanisms limit the actions of an EU with respect to another EU using some information maintained in the CE that specifies the associated security policy for EU interactions. For example an EU may configure the CE access control mechanisms so that only EUs of a certain group are allowed to stop its execution. Requests of termination coming from EUs outside this group would be intercepted and blocked by the CE before they reach the intended EU.

CE resources and functionalities are protected from attacks or faulty actions of EUs by means of access control mechanisms, that determine *which* resources can be accessed and *how much* of them can be used, protecting the CE from denial-of-service attacks that, exhausting CE resources, prevent CE from proper functioning. Access control can be *static* or *run-time*. Static access control mechanisms try to determine, before EU execution, if, given the EU code and state, execution will respect the access control policy enforced by the system. If the verification succeeds, no other check will be performed during EU execution. For example the *proof-carrying code* [77] mechanism associates code fragments with a formal proof of their correctness that is verified before code execution. A different approach is followed by *code verifiers* that use theorem proving techniques to determine if code preserves some security invariants (e.g., address space containment). In run-time access control mechanisms each EU has an associated set of access rights to local resources (e.g., permission to read/write a file or to use a certain amount of CPU time) that is determined by the security policy of the CE. Every attempt of an EU to access CE resources is intercepted and examined with respect to the access rights of the EU. As a consequence, the required operation can be performed or denied. This access control mechanism is often called *safe execution* or *sandboxing*. Run-time access control mechanisms can be distinguished on the basis of the information used to determine the set of access rights of an EU. *Authority-based* mechanisms determine access rights on the basis of the authority under which the EU is running. *Permit-based* mechanisms determine access rights on the basis of a *permit* that is associated to the EU. Permits can be statically associated to the EU for its whole lifetime or can be determined dynamically. For example the *state appraisal* mechanism [34], defines an EU permit on the basis of a state appraisal function that is associated with the migrating EU. The function is evaluated before EU execution passing as parameter the EU current state. The function returns the set of access rights that the EU needs in its current state. Other dynamic permit-based mechanisms allow an EU to re-contract its access rights during execution using some kind of electronic currency.

One of the most difficult problem in intra-CE security is represented by protection of EUs from the hosting CE. Attacks may include denial-of-service, service overcharging, private information disclosure, and code/data modification. To execute EUs, CEs must access EUs code and execution state. It is therefore very difficult to protect EUs from malicious CEs. Protection mechanisms can be aimed at *prevention* or *detection*. Prevention mechanisms try to make it impossible to access or modify an EU data and code in a useful way. For example, *tamper-proof devices* [123] avoid unauthorized modification of EU code and state by executing EUs in a physically sealed environment that is not accessible even by the owner of the system without disrupting the system itself. A different approach is followed by the code scrambling mechanism, in which the EU code is rearranged before EU moves to a CE in order to make almost impossible reverse-engineering [50] while maintaining the EU's original behavior. A simpler solution consists in *partial encryption* of the EU content. Using this mechanism, an EU could protect data that has to be used in a particular CE, by encrypting it with the CE's public-key. This way data will be accessible only when the EU reaches the intended execution environment. Detection mechanisms aim at detecting, after EU execution, if some illegal modification of its code and state occurred in the EU lifetime. A first mechanism is based on the state appraisal functions described earlier. State appraisal is used after execution to detect illegal modifications in the EU state that invalidate some invariants of the EU. A second mechanisms is *cryptographic tracing* [115] that, using execution traces and digital signatures, allows the owner of an EU to detect, after EU termination, every possible illegal modification of EU's code and state.

As a matter of fact, usually MCSs consider CEs as trusted entities and the problem of protecting EUs form CEs is not tackled.

2.4 Translation Mechanisms

The choice of executing a program either through direct interpretation or through compilation is not distinctive of MCSs. Nonetheless, the nature of MCSs introduces new requirements as well as new degrees of freedom, which affect the criteria determining the choice. Code migrated among MCSs should be:

Portable The target platform is a computer network made of heterogeneous machines. The obvious goal is to write the mobile code once, and then be able to unleash it for execution on any machine, without being aware of the hardware and software requirements specific to it.

Secure Code being executed within a CE must be checked, in order to protect the system from malicious or accidental damage, as discussed in Section 2.3.

As for security, interpretation enables load-time or run-time checks on the source code in order to verify that only legal instructions are executed. Interpretation is exploited frequently in MCSs because it supports portability as well, given that an interpreter is provided for each platform involved. A compiled approach, instead, would force the run-time support of the sender machine to be aware of the platform of the receiver machine in order to select the appropriate native executable code for transmission—or, if the destination platform is unknown, to send the native code for each platform. A common strategy among MCSs is the adoption of a hybrid approach, consisting of compiling source programs into an *intermediate language* which is used for transmission and interpretation on the target machine. Using this approach, higher-level source code is compiled into a lower-level intermediate, portable code, designed to improve efficiency and safety of transmission and execution.

Using this technique, some MCSs push even further the portability issue, by achieving platform independence as well as language independence. In this setting, CEs may support the execution of EUs whose code is written using different languages. Each CE manages several *virtual machines* each specialized for the execution of programs written in a particular language. A MCSs

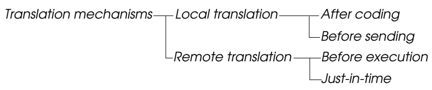


Figure 2.6: A classification of translation mechanisms.

featuring more than one virtual machine, called a *multi-language MCS*, is a generalization of conventional MCSs where the language is fixed, e.g., Java or Telescript. Hence, we will focus our discussion on multi-language MCSs.

The basic problem of multi-language MCSs is the following: given a program written in a language L, how can it be executed on a platform that supports a set of languages $\{L_1, L_2, \ldots, L_n\}$? The solution is given by two mechanisms: *local translation* and *remote translation*.

Local translation is performed at the source CE and produces a program written in language $L' = T(L)$ that will be shipped to the remote CE. If the set of languages available at the remote site is not known in advance, the program written in L can be translated into different languages, in order to enhance the probability to find a suitable virtual machine at the remote CE. Local translation mechanisms can be further distinguished with respect to the *time* when translation is performed. A first solution, which is currently the most commonly employed, is to perform the translation once for all *after program coding*. Clearly, this simplifies the design of the run-time support system. An alternative is to defer the translation to the moment when the code needs to be actually sent, i.e. *before transmission*. This way, the run-time support may exploit information about the set of languages that are supported by the destination CE, and perform only the appropriate translation.

Remote translation is performed at the destination CE and produces a program written in a language $T(L') = L'' \in \{L_1, L_2, \ldots, L_n\}$, whose execution is supported by the CE. Again, these mechanisms can be distinguished with respect to translation time. Translation can be performed on the whole program just *before execution*, or it can be performed *just-in-time* as soon as the execution flow reaches untranslated portions of the original program.

A real MCS does not necessarily need to exploit all the mechanisms described so far, which are summarized in Figure 2.6. For example, the common scheme where a single high-level language is compiled into a single bytecode interpreted language needs just local translation after coding, which directly generates the only language supported on the destination CE, the bytecode. Section 2.6 will provide more insights about how MCSs support translation. Anyway, how these mechanisms are composed and applied may vary, depending on a variety of considerations and trade-off related to code size, efficiency, and portability.

2.5 Communication Mechanisms

Code migration can be regarded as a form of communication per se, which takes place among processes distributed at different locations—the CEs. All in all, the first approaches to code mobility [100, 33] where conceived as an extension of the remote procedure call communication mechanism. Furthermore, a mobile EU can be used to disseminate information among a set of CEs, thus realizing

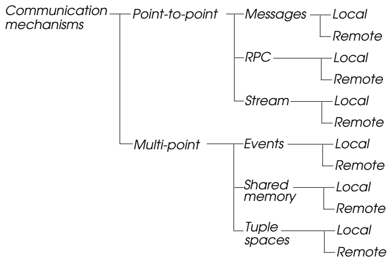


Figure 2.7: A classification of communication mechanisms.

a form of communication. However, within the context of MCSs the issue is usually to enable communication among migrating EUs than among the CEs supporting their execution.

To this end, the alternative could in principle encompass many of the mechanisms found in literature. Nevertheless, EUs roaming in a distributed system pose a problem which is related intrinsically with mobility: some of the communication mechanisms need to be extended to work properly in a mobile setting. Message passing, for instance, assumes that the recipient of a message is identified with an address, which is statically associated with a location. Clearly, if the recipient is allowed to detach from its current physical location and move to a different one, new protocols must be devised to avoid message loss. Analogolously, an EU migrating to a new CE may need to communicate with other EUs present there: mechanisms are needed to establish such communication. However, this problem is not specific to EUs moving across CEs. Similar problems arise in the context of mobile computing, where packet routing must be changed dynamically according to the movement of the physical host, which in addition needs to be aware of the facilities available at a given access point.

Development of communication mechanisms in MCSs is still in its early phase, the reason for this being that until now research has been focused mainly on the mechanisms needed to provide EU mobility. After the initial successes in this effort, researchers are now investigating more complex forms of interactions among EUs.

Mechanisms supported by existing MCSs can be distinguished along two orthogonal dimensions: the number of EUs involved in communication, and their mutual location. Hence, the first dimension partitions the space of mechanisms in *point-to-point* and *multi-point* communication mechanisms, while the second distinguishes among *local* and *remote* communication, depending on whether the EUs are co-located in the same CE or not. At a first glance, remote communication seems a contradiction with the goal of code mobility, i.e. to minimize communication over the network. However, as it will be discussed later, mobile code is an option rather than *the* solution, and remote communication mechanisms may be helpful even when used in conjunction with mobile EUs, e.g., to monitor EU activity or to interact with systems which do not offer support for code mobility. The classification for communication mechanisms is summarized in Figure 2.7.

2.5.1 Point-to-point Mechanisms

Point-to-point mechanisms enable communication between two EUs. The more primitive mechanism is *message passing*, although no existing MCSs actually uses it for communication between EUs moving arbitrarily, due to the problems arising in presence of mobility discussed earlier. Usually, some form of *remote procedure call* (RPC) is provided instead. This may involve invocation of bare procedures or method invocation on remote objects, a la CORBA. No existing MCS allows an EU to migrate during the execution of an RPC. RPC is often exploited for both local and remote communication, in order to provide a uniform mechanism. *Streams* are also usually provided, which allow an EU to open a channel to another EU, and send it a continuous stream of information. Like RPC, they are often used for local communication as well, much like sockets in a Unix system—actually, streams are usually implemented as sockets. Notably, some MCSs manage to retain open sockets as part of the state of an EU during the migration process, thus providing primitives that abstract from the physical location of the EUs involved in communication.

2.5.2 Multi-point Mechanisms

Multi-point mechanisms enable communication among more than two EUs at a time. *Shared memory* is by far the mechanism most frequently used to achieve multi-point communication. Several EUs are given a reference to the same variable or object. Changes in the value associated to the reference are perceived by all the EU owning a reference to it. In existing MCSs, this mechanism is always used for local communication, although it could be extended to implement remote communication, for example with a distributed shared variable abstraction, as described in [6].

Event-based mechanisms, in turn, define an event bus constituting the logical channel through which events are dispatched, together with primitives that allow EUs to generate events and to subscribe for receiving events they are interested in. Upon generation of an event, the communication systems sends a copy of the event description to all the EUs subscribed to it. Hence, events provides a form of *implicit* communication. Variants of this scheme exist in MCSs. Many languages exploit this mechanism for local communication only—events are dispatched only to EUs within the CE where the event is generated. In fact, event dispatching suffers of the same problem highlighted for message passing in presence of mobile EUs. Furthermore, building scalable and efficient mechanisms for dispatching events over large-scale networks is still an open research area [93]. Nevertheless, the first proposals of architectures overcoming the problem are beginning to appear in [11, 124]. Some of these proposals combine the notion of event with the notion of *group*. The relationship determining whether an EU belongs or not to a given group is not necessarily related with communication: for instance, it can be used to specify that a set of EUs are under the same authority. In the context of communication, groups provide a naming scheme which can be used to support multicast or anycast schemes in message or event-based communication. A typical example is the following. Assume a group of EUs have been unleashed over the Internet to retrieve information. If an EU finds an information item, it might be desirable to inform the other EUs of this fact and communicate them that the next item has to be searched instead. The group naming scheme and an distributed event mechanism probably provide a good set of abstraction to accomplish this.

Another form of implicit communication is provided by *tuple spaces*. With this mechanism, pioneered by the Linda model [40], EUs communicate by either inserting the tuples containing the information to be communicated into a shared tuple space, or by searching it for a tuple using some form of pattern matching. This mechanism provides a powerful abstraction of a communication channel, however, its application to a large-scale setting is limited by the implementation problems in mantaining a notion of shared tuple space across the network. For this reason, remote communication through tuple space is not implemented by any MCSs, yet. Chapter 16 will present a system under development that exploits a novel mechanism encompassing tuple spaces to coordinate mobile agents and mobile hosts.

2.6 A Survey of Mobile Code Technologies

Currently available technologies differ in the support they provide for the mechanisms mentioned so far. In this section we map a number of existing MCSs onto the classification scheme we developed in the previous sections. The set of technologies considered, although representative of the various typologies of MCSs, is not exhaustive. The reader may verify the soundness of the taxonomy by applying it to other, more recent MCSs not considered here, like those described in [124, 38, 58].

2.6.1 Agent Tcl

Developed at Dartmouth College, NH, USA, Agent Tcl [44] provides a Tcl interpreter extended with support for strong mobility. In Agent Tcl, an EU (called *agent*) is implemented by a Unix process running the language interpreter. Since EUs run in separate address spaces, they can share only resources provided by the underlying operating system, like files. These resources are considered as not transferrable. The CE abstraction is implemented by the operating system plus the language run-time support. In Agent Tcl, EUs can `jump` to another CE, `fork` a new EU at a remote CE, or `submit` some code to a remote CE. In the first case, a proactive migration mechanism enables movement of a whole Tcl interpreter along with its code and execution state. In the second case, a proactive remote cloning mechanism is implemented. In both cases, bindings in the data space of a migrating EU are removed. In the third case, a code shipping mechanism for stand-alone code is exploited to perform remote execution of a Tcl script in a newly created EU at the destination CE. This mechanism is asynchronous and immediate. A copy of variables belonging to the execution state of the EU invoking the `submit` may be migrated together with the Tcl script by passing them explicitly as parameters.

Being Tcl the only language currently supported, no translation mechanisms are needed. Agents are transferred from site to site in source code form. However, there are plans for a new, multilanguage version of the system supporting Java and Scheme in addition to Tcl, and providing additional security features.

Presently, the CE can be configured in order to accept mobile EUs only from selected hosts only. Morevoer, at the moment Agent Tcl provides no other inter-CE security mechanisms. An extension that allows EUs and CEs to authenticate each other and to support privacy and integrity using PGP [130] is under development. Inside the CE there is no mechanism to protect EUs from each other and to protect EUs from CEs. Protection of the CE from EUs is delegated to the underlying operating system mechanisms, and therefore EUs may access any resource that is accessible to the user that is executing the run-time support.

Two communication mechanisms are provided: *messages* and *meetings*. Messages allow EUs to exchange messages composed of strings. The message recipient can be located in the same CE or in a remote CE. Meetings model, at the language level, socket-like stream connections between agents. An EU can request a meeting to another agent either in the same CE or in a remote CE. The agent that receives the connection request may accept or reject the connection.

2.6.2 Ara

Developed at University of Kaiserslautern, Germany, Ara [84] is a multi-language MCS that supports strong mobility. Ara EUs, called *agents*, are managed by a language-independent system core plus interpreters for the languages supported—at the time of writing C, C++, and Tcl. The core and the interpreters constitute the CE, whose services are made accessible to agents through the *place* abstraction. Mobility is supported through proactive migration, and data space management is simplified by the fact that agents cannot share anything but system resources—whose bindings are always removed upon migration.

Ara agents may be written in different languages. Tcl agents are not translated and, as in the case of Agent Tcl, they are transferred in source code form. C and C++ agents are translated, at the starting site, in a common lower-level language called MACE, after coding.

Ara does not provide any cryptographically strong means to authenticate and protect agents moving from site to site. The security mechanism to protect a CEs from EUs is the *allowance*. An allowance specifies a set of access rights to the system resources, like files, network connection, and CPU time. An agent migrating to a site carries an allowance specifying the desired access to the target place. In turn, the place before starting the agent, determines an actual allowance for the agent based on the requested allowance and the place security policy. Since EUs can interact by message passing only, EU-EU security is performed by the message recipient EU on the basis of some internally defined policy.

Communication in Ara is only local and is achieved by means of simple messaging. Server agents can register a service point with a well-known name in a place and client agents can send messages to the service point to have them delivered to the server. A mechanism supporting messaging among EUs running on different CEs is under development.

2.6.3 Facile

Developed at ECRC in Münich, Germany, Facile [112] is a functional language that extends the Standard ML language with primitives for distribution, concurrency and communication. The language has been extended further in [59] to support weak mobility, advanced translation techniques, and strongly typed resource linking. Executing units are implemented as threads that run in Facile CEs, called *nodes*. The *channel* abstraction is used for communication between threads. Channels can be used to communicate any legal value of the Facile language. In particular, functions may be transmitted through channels since they are first-class language elements. Communication follows the rendez-vous model: both the sender and the receiver are blocked until communication takes place. For this reason, mobility mechanisms can be regarded as supporting both code shipping and code fetching—depending on whether an EU is a sender or a receiver. In addition, the programmer can specify whether the function transmitted has to be considered as stand-alone code or as a code fragment. When the function has been transferred, the communication channel is closed, and the receiver EU is free to evaluate the function received or defer its evaluation. Therefore, the mechanism is asynchronous and supports both immediate and deferred execution. As for data space management, this takes place always by copy, except for special variables called *ubiquitous values*. They represent resources replicated in each Facile node and are always accessed with bindings by type, exploiting a re-binding mechanism.

Facile supports all the translation mechanisms described previously. The source code can be transmitted as it is, compiled locally into various intermediate representation with different abstraction levels, or directly into native code. In addition, to shorten compilation time on receipt in interactive applications, functions are compiled remotely using a just in time mechanism only when the application tries to invoke them [59].

At the time of writing, Facile lacks security mechanisms other than lexical scoping.

2.6.4 Java

Developed by Sun Microsystems, Java [104] has triggered most of the attention and expectations on code mobility. The original goal of the language designers was to provide a portable, clean, easy-to-learn, and general-purpose object-oriented language, which has been subsequently re-targeted by the growth of Internet. The Java compiler translates Java source programs into an intermediate, platform-independent language called *Java Byte Code*. The byte code is interpreted by the *Java Virtual Machine* (JVM)—the CE implementation. Java provides a programmable mechanism, the *class loader*, to retrieve and link dynamically classes in a running JVM. The class loader is invoked by the JVM run-time when the code currently in execution contains an unresolved class name. The class loader actually retrieves the corresponding class, possibly from a remote host, and then loads the class in the JVM. At this point, the corresponding code is executed. In addition, class downloading and linking may be triggered explicitly by the application, independently of the need to execute the class code. Therefore Java supports weak mobility using mechanisms for fetching of code fragments. Such mechanisms are asynchronous and support both immediate and deferred execution. In both cases, the code loaded is always executed from scratch and has neither execution state nor bindings to resources at the remote host—no data space management is needed.

One of the key success factors of Java is its integration with the World Wide Web technology. Web browsers have been extended to include a JVM, and particular Java classes called *applets* can be downloaded together with HTML pages to allow for active presentation of information and interactive access to a server. We consider this as a particular *application* of mobile code technology. On the other hand, the combination of a Web browser and a JVM is so frequent that it can be regarded as a technology per se, supporting the development of Web applications. The presence of a JVM is hidden and its mechanisms are used to provide a higher-level view where browsers constitute the CEs and applets are EUs executing concurrently within them. In this context, the downloading of applets can be regarded as a mechanism provided by the browser to support fetching of stand-alone code even if, in principle, it is based on the code fragment code fetching mechanism provided by the underlying JVM.

Java uses a stack-based intermediate language, the *Java Byte Code*, which is interpreted by the *Java Virtual Machine*. The bytecode can also be compiled into native code either on the fly, using one of the several just in time compilers available, some of which enable local offline generation of native code as well.

Java Byte Code received from the network is not directly interpreted by the Java Virtual Machine. It is first scanned by a run-time module, the *Byte Code Verifier*, to check for the absence of potentially dangerous constructs and then it is controlled by a programmable *Security Manager*. Hence, security mechanisms are partly hard-wired into the language run-time support and partly programmable.

Since networked applications is the intended domain for Java, a rich set of communication-oriented APIs is provided together with the language. Support for messaging, and stream communication mechanism is provided. Moreover, the Remote Method Invocation (RMI) [108] allows a Java application to invoke methods exported by other applications and provides a distributed objects platform that is similar to CORBA under many respects. All the mechanisms are both local and remote. Also, mechanisms for communication based on events and tuple spaces are under development.

2.6.5 Java Aglets

The Java Aglets API (J-AAPI) [61], developed by IBM Tokyo Research Laboratory in Japan, extends Java with support for weak mobility. Aglets [62], the EUs, are threads in a Java interpreter which constitutes the CE. The API provides the notion of *context* as an abstraction of the CE. The context of an aglet provides a set of basic services, e.g., retrieval of the list of aglets currently contained in that context or creation of new aglets within the context. Java Aglets provides two migration primitives: `dispatch` is the primitive that performs code shipping of stand-alone code (the code segment of the aglet) to the context specified as parameter. The mechanism is asynchronous and immediate. The symmetrical primitive `retract` performs code fetching of stand-alone code, and is used to force an aglet to come back to the context where `retract` is executed, with a synchronous, immediate mechanism. In both cases, the aglet is re-executed from scratch after migration, although it retains the value of its object attributes which are used to provide an initial state for its computation. The attribute values may contain references to resources, which are always managed by copy.

Aglets can communicate explicitly through message passing; moreover, an aglet can invoke operations on other aglets. In both cases, this encompass invocation of operations on a *proxy object* which "protects" the aglet. For example, upon invocation of a `dispatch` operation on an aglet proxy, a predefined sequence of method invocation (whose code is under the control of the programmer) is automatically started, which allows the aglet to determine whether the request must be accepted, rejected, or delayed. The documentation mentions an additional level of security provided by the run-time support, but at the time of writing information is lacking on this issue. A multi-point mechanism for communication is also provided, that implements a local event-based communication system that allows an aglet to send a message to all the aglets in the same context that subscribed to the event type.

Finally, a limited form of persistency is provided, which enables to `deactivate` an aglet by removing it from memory and storing it on the persistency store. Nevertheless, analogously to migration, this operation causes the loss of the execution state of every thread contained in the aglet.

2.6.6 M0

Implemented at the University of Geneva, Switzerland, M0 [113] is a stack-based interpreted language that implements the concept of *messengers*. Messengers—representing EUs—are sequences of instructions that are transmitted among *platforms*—representing CEs—and executed unconditionally upon receipt. Messengers [114], in turn, can *submit* the code of other messengers to remote platforms. Resources are always considered transferrable and fixed, and the submitting messenger may copy them in the message containing the submitted code, in order to make them available at the destination CE. Therefore, M0 is a weak MCS providing shipping of stand-alone code whose execution is asynchronous and immediate, and data space management by copy.

Local communication is achieved through an associative array called *dictionary* that is used to allow memory sharing among different EUs, similar to a tuple space. Also, channels connecting platforms provide remote communication between different hosts.

In M0, integrity of messengers transmitted over channels is supported via checksums included in the messenger message. While no mechanisms are provided for authentication, M0 provides the programmer with privacy and access control mechanisms. References to shared objects may have *attributes* that specify Unix-like access rights for a messenger with respect to the referenced object. In addition, messengers may use asymmetric encryption to create protected entries in shared memory, accessible by means of a public key, but modifiable only by means of a private key.

2.6.7 Mole

Developed at University of Stuttgart, Germany, Mole [103, 11] is a Java API that supports weak mobility. Mole agents are Java objects which run as threads of the JVM, which is abstracted into a *place*, the Mole CE. A place provides access to the underlying operating system through *service agents* which, differently from *user agents*, are always stationary. Shipping of stand-alone code is provided with an asynchronous, immediate mechanism. The code and data to be sent are determined automatically upon migration using the notion of island [49]. An island is the transitive closure over all the objects referenced by the main agent object. Islands, which are generated automatically starting from the main agent object, cannot have object references to the outside; inter-agent references are symbolic and become void upon migration. Hence, data space management by move is exploited.

The Mole project puts a strong emphasis on inter-agent communication which, according to the description in [11] is centered around three main concepts: agent groups, sessions, and events. Agent groups can be created by defining a *badge*, i.e.., an application-dependent identifier which an agent can "pin" on and off. Predicates on badge values can be specified within the application code, thus enabling the dynamic creation of groups of related agents and (non-deterministic) selection of their elements for communication. Actual communication, supported either through message passing or RPC, requires the setup of a session. During this phase, each agent can specify its will to participate in the session with either blocking or non-blocking operations; moreover, it describes the agent which it intends to communicate with by means of a pair `(placeId, PeerQualifier)`, where the second component can be either a globally assigned agent identifier or a badge predicate. Finally, anonymous communication is encompassed by event management, which is still under design.

As far as security is concerned, Mole does not provide anything but the underlying Java mechanisms, although work is on-going on this issue [50].

2.6.8 Obliq

Developed at DEC, Obliq [19] is an untyped, object-based, lexically scoped, interpreted language. Obliq allows for remote execution of procedures by means of *execution engines* which implement the CE concept. A thread, the Obliq EU, can request the execution of a procedure on a remote execution engine. The code for such procedure is sent to the destination engine and executed there by a newly created EU. The sending EU suspends until the execution of the procedure terminates. Thus, Obliq supports weak mobility using a mechanism for synchronous shipping of stand-alone code. Obliq objects are transferrable fixed resources, i.e. they are bound for their whole lifetime to the CE where they are created even if in principle they could be moved across CEs. When an EU requests the execution of a procedure on a remote CE, the references to the local objects used by the procedure are automatically translated into network references. Thus, the data state of the program might actually become distributed across several machines.

Obliq threads may communicate, both locally and remotely, by method invocation or using shared memory.

In Obliq there are no explicit security mechanisms. Lexical scope rules support an implicit form of safe execution. In fact, procedures evaluated by an Obliq engine, by default, may only access objects in their original scope. Therefore, access to objects and files residing at the engine CE must be granted explicitly by the remote engine passing a reference to a local component when invoking the procedure.

2.6.9 Safe-Tcl

Initially developed by the authors of the Internet MIME standard, Safe-Tcl [17] is an extension of Tcl [81] conceived to support active e-mail. In active e-mail, messages may include code to be executed when the recipient receives or reads the message. Hence, in Safe-Tcl there are no mobility or communication mechanisms at the language level—they must be achieved using some external support, like e-mail. Rather, mechanisms are provided to protect the recipient's CE, which is realized following a *twin interpreter* scheme. The twin interpreter consists of a *trusted* interpreter, which is a full-fledged Tcl interpreter, and an *untrusted* interpreter, whose capabilities have been restricted severely, so that one can execute code of uncertain origin without being damaged. The owner of the interpreter may decide to export procedures which are guaranteed to be safe from the trusted interpreter to the untrusted one. Presently, most of the fundamental features of Safe-Tcl have been included in the latest release of Tcl/Tk, and a plug-in for the Netscape browser has been developed, allowing Safe-Tcl scripts to be included in HTML pages [82], much like Java applets.

In Safe-Tcl, security is based on a *twin interpreter* scheme which consists of a *trusted* interpreter, which is a full-fledged Tcl interpreter, and an *untrusted interpreter*, whose capabilities have been severely restricted, so that one can execute code of uncertain origin without being damaged. The owner of the interpreter may decide to export procedures which are ensured to be safe from the trusted interpreter to the untrusted one. For example, the trusted interpreter could provide procedures to access just a limited portion of the file system. When an e-mail message containing Safe-Tcl code is received, such code is passed to the untrusted interpreter for execution. The untrusted interpreter may require the loading of a Safe-Tcl library. Thanks to Tcl mechanisms, the loading mechanism can be programmed by the owner of the interpreter. The dynamically loaded code is evaluated by the trusted interpreter and therefore great care must be taken in order to avoid loading of code of unknown or uncertain source.

2.6.10 Sumatra

Sumatra [2], developed at University of Maryland, is a Java extension designed expressly to support the implementation of *resource-aware* mobile programs, i.e. programs which are able to adapt to resource changes by exploiting mobility. Sumatra provides support for strong mobility of Java threads, which are Sumatra EUs. They are executed within *execution engines*, i.e. dynamically created interpreters which extend the abstract machine provided by the JVM with methods which embody proactive migration mechanisms, proactive remote cloning, and shipping of stand-alone code with synchronous, immediate execution. Threads or stand-alone code can be migrated separately from the objects they need. The *object-group* abstraction is provided to represent dynamically created object aggregates which determine the unit of mobility as well as the unit of persistency. Objects belonging to a group must be explicitly checked in and out, and thread objects cannot be checked in an object-group. The rationale for the absence of an automatic mechanism is to give the programmer the ability to modify dynamically the granularity of the unit of mobility. Data space management in an object-group is always by move; bindings to migrated objects owned by EUs in the source CE are transformed into network references.

Remote access is achieved through the Java RMI API [108]. An additional communication mechanism is represented by *asynchronous events*, which are provided mainly to support *resource monitoring*, the main focus in Sumatra. Although, differently from other approaches, events are dispatched only locally, in order to inform a thread of resource changes or other site-related information, and are not conceived to be used as an inter-EU communication mechanism. No security features are provided.

2.6.11 TACOMA

In TACOMA [54] (Tromsø And COrnell Mobile Agents), the Tcl language is extended to include primitives that support weak mobility. Executing units, called *agents*, are implemented as Unix processes running the Tcl interpreter. The functionality of the CE is implemented by the Unix operating system plus a dedicated run-time supporting agent check-in and check-out. Code shipping of stand-alone code is supported by mechanisms providing both synchronous and asynchronous immediate execution. Initialization data for the new EU are encapsulated in a data structure called *briefcase*, while resources in the CE are contained in stationary data structures called *cabinets*. Upon migration, data space management by copy can be exploited to provide the new EU with a resource present within the source CE cabinet.

TACOMA provides two basic security mechanisms. First, the CE owner can specify the set of TACOMA hosts from which agents are accepted. Second the execution of the whole system can be confined in a subtree of the host file system. Apart from these mechanisms no inter-CE or intra CE mechanisms are currently implemented.

TACOMA agents can communicate using briefcases, but only at startup time. In fact, the `meet` primitive—a rather least counter-intuitive name—invokes a *new* program specified by a primitive parameter, passing on the standard input the contents of a briefcase specified as another parameter. Communication between active agents can only be performed using *cabinets*, i.e., shared data containers mantained in the file system and can involve only agents residing on the same CE.

TACOMA is a multi-language system supporting, in version 1.2, Tcl, Python, Scheme, Perl, and C. Code of agents written in interpreted language is shipped in source code form and no translation occurs. C source code is shipped to the remote site where is compiled and then executed. Therefore in this case a remote translation before execution mechanism is used.

2.6.12 Telescript

Developed by General Magic, Telescript [122] is an object-oriented language conceived for the development of large distributed applications. Security has been one of the driving factors in the language design, together with a focus on strong mobility. Telescript employs an intermediate, portable language called *Low Telescript*, which is the representation actually transmitted among *engines*, the Telescript CEs. Engines are in charge of executing *agents* and *places*, i.e. the Telescript EUs. Agents can move by using the `go` operation, which implements a proactive migration mechanism. A `send` operation is also available which implements proactive remote cloning. Places are stationary EUs that can contain other EUs. Data space management is ruled by the *ownership* concept which associates each resource with an owner EU. Upon migration, this information is used to determine automatically the set of objects that must be carried along with the EU. Data space management always exploits management by move for the migrating EU. Bindings to migrated resources owned by other EUs in the source site are always removed.

Low Telescript is the name of the stack-based language which constitutes the intermediate language for Telescript (more properly called High Telescript). In opposition to Java, the only purpose of this intermediate form is a more efficient transfer and execution. No security controls are performed, since security in Telescript is ensured at the language level. No compilation of Byte Code is allowed; Low Telescript is strictly interpreted.

Of all reviewed MCLs, Telescript provides the most powerful mechanisms to support security [109]. In Telescript, each thread object has attributes that can be used to determine its security-related characteristics. For example, attributes are provided to hold the *authority* of the thread, i.e. the real-world person or organization which it belongs to, and can be accounted for. A particular set of these attributes, called *permits*, grant the agent the right of performing a given set of operations (e.g., the `go` operation), or to use engine resources (e.g., the CPU time) in a specified amount. Permit violation is controlled by the programmer through the exception handling mechanism provided by Telescript. The values of the security attributes are partly specified by the programmer and partly set by the Telescript engine where the trip originates, or by the engine where the trip ends. The mechanism by which the engine sets these values is fully programmable by the engine owner through the methods of the engine place, that is, the particular thread that represents the engine. Agent integrity is provided by packing and encrypting the agents before their trip.

2.7 Other Relevant Language Issues

Mobility has a strong impact on programming language features. This section analyzes both the characteristics of conventional languages that must be extended to take into account mobility, and the completely new issues that must be considered. For each issue, a general discussion of the problem is presented, followed by an analysis of how such problems have been tackled in the MCLs surveyed in the previous section.

Type System

The decision about which type system to adopt is one of the most important design choices that have to be taken when defining a new programming language. The language can be defined to be *typeless*, i.e. all data belong to an untyped universe and can be interpreted as values of different types when manipulated by different operators. Typeless languages impose no restrictions on how data are manipulated. The resulting power and flexibility are counterbalanced by the impossibility of protecting data against nonsensical or erroneous manipulations.

At the other extreme, a language may adopt a *strong type system*. In such case, the language definition ensures that the absence of type errors can be guaranteed for a program statically. Languages of this class support the development of verifiable programs. To increase flexibility, however, languages may weaken the type system in such a way that full type checking may only be achieved dynamically, by verifying at run-time that typed data objects are manipulated only through legal operations. This design choice weakens program verifiability, since type errors can only be uncovered if particular input data are used during execution.

In languages supporting mobile code, strong typing can be practically impossible to achieve for two main reasons. First, code can be downloaded from a remote site and linked dynamically to a running program. Second, resources may be bound to a program as it is executing. In both cases, type correctness of a program cannot be verified statically. Mobile code languages adopt different strategies to weaken their type system in order to allow dynamic linking.

A first strategy, exploited by several languages, is to complement compile-time type checking with several type checks performed at run-time. This is the approach taken by Telescript, Facile, as well as Java and its derivatives, where type checking is performed at run-time to ensure that the dynamically downloaded and linked code obeys the language type rules. Similarly, in Telescript and Facile the run-time system performs type checking to evaluate the type correctness of remote resources accessed after migration. Another strategy, adopted by Obliq and M0, is to perform all type checking at run-time only. Finally the last strategy is to define a typeless language. This is the approach taken by Tcl derivatives, like Safe-Tcl, Agent Tcl, and TACOMA, Tcl variables contain just strings. Tcl instructions elaborate these strings as appropriate. For example, arithmetic operators convert strings to the numeric values they represent, operate on such values and then translate the numeric results back to strings.

Scoping and Name Resolution

The scope of an identifier is the range of instructions over which the identifier is known. Name resolution rules determine which computational entity is bound to each identifier in any point of a given program.

All of the MCSs examined, with the exception of M0, use static scoping rules. This means that the scope of a variable name is determined by the lexical structure of the program. As will be explained later, in M0 the scope of a name can be modified at run time by the programmer. Name resolution is critical in MCSs. In fact, during the execution of an application the names that appear in the code may be bound to entities that may be located on remote CEs. Name resolution can either be performed automatically by the run-time support or hooks in the language run-time support can be provided to allow programmers to define their own name resolution rules.

Most of the existing MCSs do not offer particular features to perform automatic name resolution for remote resources. Name resolution for local resources is often performed statically and remote resources have to be accessed explicitly. This means that the programmer is aware of the location of the various remote resources. For example, a Java program can download and link code from the network but the name resolution of dynamically downloaded classes has to be explicitly programmed. A Java programmer must write her own class loader that resolves the names of the classes that have to be downloaded from the network.

In Obliq, new objects created by an EU are stored in the local data space, that is, in the address space of the interpreer where the EU is actually running. The identifier for this object is bound to the storage area of the object. When the EU moves, references to objects in the origin data space are automatically transformed into *network references*. Further accesses to such objects will be transparently translated into callbacks to the interpreter that holds the variable value. Hence, the binding between the object identifier and the object storage area is established at run-time and cannot be changed. Moreover, name resolution rules hide the actual physical location of remote resources to the programmer.

Telescript has a sophisticated name resolution scheme too. As mentioned in Section 2.6, the migration unit in Telescript is the thread. Each thread, among its attributes, has a list (called *package*) that contains the code for some of the classes it uses. Moreover, the language formally defines the *ownership* relation between the threads and the objects they own. When an object `Obj` requests an operation that forces the engine to resolve the name of a class (e.g., a `new` operation), the class name is searched into the package of the thread that owns `Obj`. If the class name is not found there, it is searched in the package of the place containing the thread. If the class name is still missing, it is searched backwards in the enclosing places, until the root of the place hierarchy is reached. Hence, in principle, the name space of a Telescript engine can be enriched by sending "installer" agents that are able to put packages in the appropriate place at run-time, in a fully programmable way.

In M0, the name space of a messenger is defined by its *dictionary stack*. Each dictionary that composes the dictionary stack is an associative array that contains names and their definitions. Whenever an interpreter has to resolve a name in the messenger code, it looks up the dictionaries in the dictionary stack starting from the topmost, hence the first matching definition is used. The dictionary stack can be manipulated by the programmer. In addition to this run-time mechanism, M0 provides, at the language level, a mechanism to force the binding of procedure names in a particular name space. In fact, using the `bind` command, a messenger is able to substitute all operators and procedures names present in a procedure with their corresponding values, according to the messenger dictionary stack. This means that in M0 the scope of a variable can be programmed explicitly.

Chapter 3

Architectures

Mobile code technologies are only one of the ingredients needed to build a software system. Software development is a complex process where a variety of factors must be taken into account: technology, organization, and methodology. In particular, a very critical issue is the relationship between technology and methodology. This relationship is often ignored or misinterpreted. Quite often, researchers and practitioners tend to believe that a technology inherently induces a methodology. Thus “it is sufficient to build good development tools and efficient languages”. This is particularly evident in a critical phase of software development: software design. The goal of design is the creation of a software architecture, which can be defined as the decomposition of a software system in terms of software components and interactions among them [99]. Software architectures with similar characteristics can be represented by *architectural styles* [1] or *design paradigms*, which define architectural abstractions and reference structures that may be instantiated into actual software architectures. A design paradigm is not necessarily induced by the technology used to develop the software system—it is a conceptually separate entity. This distinction is not merely philosophical: the evolution of programming languages has clearly emphasized the issue. A *modular* system can be built even using an assembly language, and at the same time, the adoption of sophisticated languages such as Modula-2 does not guarantee *per se* that the developed system will be really modular. Certainly, specific features of a language can be particularly well-suited to guarantee some program property, but a “good” program is not just the direct consequence of selecting a “good” language.

Traditional approaches to software design are not sufficient when designing large scale distributed applications that exploit code mobility and dynamic reconfiguration of software components. In these cases, the concepts of location, distribution of components among locations, and migration of components to different locations need to be taken explicitly into account during the design stage. As stated in [118], interaction among components residing on the same host is remarkably different from the case where components reside on different hosts of a computer network in terms of latency, access to memory, partial failure, and concurrency. Trying to paper over differences between local and remote interactions can lead to unexpected performance and reliability problems after the implementation phase.

It is therefore important to identify reasonable design paradigms for distributed systems exploiting code mobility[1], and to discuss their relationships with the technology that can be used to implement them. It is also important to notice that each of the languages mentioned in the previous chapter embodies mechanisms that can be used to implement one or more design paradigms. On the other hand, the paradigms themselves are independent of a particular technology, and could even be implemented without using mobile technology at all, as described in the case study presented in [42].

3.1 Basic Design Abstractions

Before introducing design paradigms we present some basic concepts that abstract out the entities that constitute a software system, such as files, variable values, executable code, or processes. In particular, we introduce three architectural concepts: *components*, *interactions*, and *sites*.

Components are the constituents of a software architecture. They can be further divided into *code components*, that encapsulate the know-how to perform a particular computation, *resource components*, that represent data or devices used during the computation, and *computational components*, that are active executors capable to carry out a computation, as specified by a corresponding know-how. *Interactions* are events that involve two or more components, e.g., a message exchanged among two computational components. *Sites* host components and support the execution of computational components. A site represents the intuitive notion of location. Interactions among components residing at the same site are considered less expensive than interactions taking place among components located in different sites. In addition, a computation can be actually carried out only when the know-how describing the computation, the resources used during the computation, and the computational component responsible for execution are located *at the same site*.

Design paradigms are described in terms of interaction patterns that define the relocation of and coordination among components needed to perform a service. We will consider a scenario where a computational component A, located at site S_A needs the results of a service. We assume the existence of another site S_B, which will be involved in the accomplishment of the service.

We identify three main design paradigms exploiting code mobility: *remote evaluation*, *code on demand*, and *mobile agent*. These paradigms are characterized by the location of components before and after the execution of the service, by the computational component which is responsible for execution of code, and by the location where the computation of the service actually takes place (see Table 3.1).

[1]The reader interested in the original formulation of the paradigms described here is directed to [21].

Paradigm	Before		After	
	S_A	S_B	S_A	S_B
Client-Server	A	know-how resource B	A	know-how resource **B**
Remote Evaluation	know-how A	resource B	A	*know-how* resource **B**
Code on Demand	resource A	know-how B	resource *know-how* **A**	B
Mobile Agent	know-how A	resource	—	*know-how* resource ***A***

Table 3.1: Mobile code paradigms. This table shows the location of the components before and after the service execution. For each paradigm, the computational component in bold face is the one that executes the code. Components in italics are those that have been moved.

The presentation of the paradigms is based on a metaphor where two friends—Louise and Christine[2]—interact and cooperate to make a chocolate cake. In order to make the cake (the results of a service), a recipe is needed (the know-how about the service), as well as the ingredients (movable resources), an oven to bake the cake (a resource that can hardly be moved), and a person to mix the ingredients following the recipe (a computational component responsible for the execution of the code). To prepare the cake (to execute the service) all these elements must be co-located in the same home (site). In the following, Louise will play the role of component A, i.e. she is the initiator of the interaction and the one interested in its final effects.

3.2 Client-Server (CS)

> *Louise would like to have a chocolate cake, but she doesn't know the recipe, and she does not have at home either the required ingredients or an oven. Fortunately, she knows that her friend Christine knows how to make a chocolate cake, and that she has a well supplied kitchen at her place. Since Christine is usually quite happy to prepare cakes on request, Louise phones her asking: "Can you make me a chocolate cake, please?". Christine makes the chocolate cake and delivers it back to Louise.*

The client-server paradigm is well-known and widely used. In this paradigm, a computational component B (the server) offering a set of services is placed at site S_B. Resources and know-how needed for service execution are hosted by site S_B as well. The client component A, located at S_A, requests the execution

[2]The "cake makers" Cristina and Luisa, in real life are my fiancee and Vigna's wife, respectively. While giving a talk about this work, I was informed that our examples seem to convey an image of the woman which is not "politically correct". It is not a matter of political correctness. Simply, when publishing the original paper we thought that nobody—at least among our colleagues—would have believed in examples in which we were in the role of cooks. Furthermore, we found that using the names of our significant others (who actually bake delicious cakes) if compared with sentences like "X wants to prepare a cake", was far better than dealing with aseptic symbolic names.

of a service with an interaction with the server component B. As a response, B performs the service requested by executing the corresponding know-how and accessing the involved resources co-located with B. In general, the service produces some sort of result that will be delivered back to the client with an additional interaction.

3.3 Remote Evaluation (REV)

> *Louise wants to prepare a chocolate cake. She knows the recipe but she has at home neither the required ingredients nor an oven. Her friend Christine has both at her place, yet she doesn't know how to make a chocolate cake. Louise knows that Christine is happy to try new recipes, therefore she phones Christine asking: "Can you make me a chocolate cake? Here is the recipe: take three eggs...". Christine prepares the chocolate cake following Louise's recipe and delivers it back to her.*

In the REV paradigm[3], a component A has the know-how necessary to perform the service but it lacks the resources required, which happen to be located at a remote site S_B. Consequently, A sends the service know-how to a computational component B located at the remote site. B, in turn, executes the code using the resources available there. An additional interaction delivers the results back to A.

3.4 Code on Demand (COD)

> *Louise wants to prepare a chocolate cake. She has at home both the required ingredients and an oven, but she lacks the proper recipe. However, Louise knows that her friend Christine has the right recipe and she has already lent it to many friends. So, Louise phones Christine asking "Can you tell me your chocolate cake recipe?". Christine tells her the recipe and Louise prepares the chocolate cake at home.*

In the COD paradigm, component A is already able to access the resources it needs, which are co-located with it at S_A. However, no information about how to manipulate such resources is available at S_A. Thus, A interacts with a component B at S_B by requesting the service know-how, which is located at S_B as well. A second interaction takes place when B delivers the know-how to A, which can subsequently execute it.

3.5 Mobile agent (MA)

> *Louise wants to prepare a chocolate cake. She has the right recipe and ingredients, but she does not have an oven at home. However, she knows that her friend Christine has an oven at her place, and that she is very happy to lend it. So, Louise prepares the chocolate batter and then goes to Christine's home, where she bakes the cake.*

[3] Hereafter, by "remote evaluation" we will refer to the design paradigm presented in this section. Although it has been inspired by work on the REV system [100], they have to be kept definitely distinct. Our REV is a design paradigm, while the REV system is a *technology* that may be used to actually implement an application designed using the REV paradigm.

In the MA paradigm, the service know-how is owned by A, which is initially hosted by S_A, but some of the required resources are located on S_B. Hence, A *migrates* to S_B carrying the know-how and possibly some intermediate results with itself. After it has moved to S_B, A completes the service using the resources available there. The *mobile agent* paradigm is different from other mobile code paradigms in that the associated interactions involve the mobility of an *existing* computational component. In other words, while in REV and COD the focus is on the transfer of code between components, in the mobile agent paradigm a whole computational component, together with its state, the code it needs, and some resources required to perform the task, is moved to a remote site.

3.6 Discussion and Comparison

The mobile code design paradigms introduced in the previous sections define a number of abstractions for representing the bindings among components, locations, and code, and their dynamic reconfiguration. Our initial experience in applying the paradigms [7, 42] suggests that these abstractions are effective in the design of distributed applications. Furthermore, they are fairly independent of the particular language or system in which they are eventually implemented.

Mobile code paradigms model explicitly the concept of location. The *site* abstraction is introduced at the architectural level in order to take into account the location of the different components. Following this approach, the types of interaction between two components is determined by *both* components' code and location. Introducing the concept of location makes it possible to model the cost of the interaction between components at the design level. In particular, an interaction between components that share the same location is considered to have a negligible cost when compared to an interaction involving communication through the network.

Most well-known paradigms are static with respect to code and location. Once created, components cannot change either their location or their code during their lifetime. Therefore, the types of interaction and its quality (local or remote) cannot change. Mobile code paradigms overcome these limits by providing component mobility. By changing their location, components may change dynamically the quality of interaction, reducing interaction costs. To this end, the REV and MA paradigms allow the execution of code on a remote site, encompassing local interactions with components located there. In addition, the COD paradigm enables computational components to retrieve code from other remote components, providing a flexible way to extend dynamically their behavior and the types of interaction they support.

Flexibility and dynamicity are useful, but it is not clear when these paradigms should be used, and how one can choose the right paradigm in designing a distributed application. We think that no paradigm is the best in absolute terms. In particular the mobile code paradigms we described do not necessarily prove to be better suited for a particular application with respect to the traditional ones. The choice of the paradigms to exploit must be performed on a case-by-case basis, according to the specific type of application and to the particular functionality being designed within the application. For each case, some parameters that describe the application behavior have to be chosen, together with some criteria to evaluate the parameters values. For example, one may want to minimize the number of interactions, the CPU costs or the generated network traffic. In addition, a model of the underlying distributed system should be adopted to support reasoning about the criteria. For each paradigm considered, an analysis should be carried out in order to determine which paradigm optimizes the chosen criteria. This phase cannot take into account all the characteristics and constraints, which probably will be fully understood only after the detailed design, but it should provide hints about the most reasonable paradigm to follow in the design. Part II provides guidelines on how such analysis can be carried out.

Once an application has been designed, developers are faced with the choice of a suitable technology for its implementation. Even if technologies are somewhat orthogonal with respect to paradigms, some technologies are better suited to implement application designed according to particular paradigms. For example, one can implement an application designed following the REV paradigm with a technology that allows EUs to exchange just messages. In this case, the programmer has the burden to translate the code to be shipped to the remote site into the data format used in message payloads. Moreover, the receiving EU has to explicitly extract the code and invoke an intepreter in order to execute it. A mobile code technology providing mechanisms for code shipping would be more convenient and would manage marshaling, shipping, and remote interpretation tasks at the system level.

A common case is represented by the use of a weak MCS that allows for code shipping for implementing applications designed following the MA paradigm [12]. In this case, the architectural concept of a moving component must be implemented using a technology that allows just for mobile code modules. Therefore the programmer has to build explicitly some appropriate data structures that allows for saving and restoring the execution state of the component in case of migration. Upon migration, the EU has to pack such data structures and send them along with the code to the remote location; then the original EU terminates. When the new EU is started on the remote CE to execute the code, it must use explicitly the encoded representation of the component's state in order to reconstruct, at the program level, the component's execution state. If a strongly mobile technology is used, the component can be directly mapped into a migrating EU and mobility is reduced to a single instruction. Therefore the programmer is set free from handling the management of the component's state and can concentrate on the problem to solve. A case study that analyzes these relationships in detail can be found in [42].

Chapter 4

Applications

At the time of writing, applications exploiting code mobility can still be considered as relegated to a niche, at least if compared to traditional client-server based applications. This is a consequence of the immaturity of technology—mostly as far as performance and security are concerned—and of the lack of methodologies for application development. Nevertheless, the interest in mobile code is not motivated by the technology per se, rather by the benefits that it is supposed to provide by enabling new ways of building distributed applications and even of creating brand new applications. The advantages expected from the introduction of mobile code into distributed applications are particularly appealing in some specific application domains. This fact has sometimes led to identifying entire application classes with terms like "mobile agent systems" or "Internet agents" that refer more to how the applications are structured rather than to the functionality they implement. Therefore, in order to understand mobile code it is important to distinguish clearly between what is an application (e.g., a system to control a remote telescope) and what is the paradigm used to design it (e.g., the REV paradigm to identify control modules that are sent to the remote telescope) or the technology used to implement it (e.g., Java Aglets).

Hence, the purpose of this chapter is to provide the reader both with a grasp on the key benefits which mobile code is expected to bring, and with a non-exhaustive review of application domains which are being identified by researchers in the field as suitable for the exploitation of mobile code. This completes our conceptual framework and provides the reader with a path from the problem to the implementation, spanning application, design, and technology issues.

4.1 Key Benefits of Mobile Code

A major asset provided by code mobility is that it enables *service customization.* In conventional distributed systems built following a CS paradigm, servers provide an *a-priori* fixed set of services accessible through a *statically* defined interface. It is often the case that this set of services, or their interfaces, are not suitable for unforeseen client needs. A common solution to this problem is to upgrade the server with new functionality, thus increasing both its complexity and its size without increasing its flexibility. The ability to request the remote execution of code, by converse, helps in increasing server flexibility without affecting permanently the size or complexity of the server. In this case, in fact, the server actually provides very simple and low-level services that seldom need to be changed. These services are then composed by the client to obtain a customized high-level functionality that meets the specific client's needs.

Mobile code is proving useful in supporting the last phases of the software development process, namely, *deployment* and *maintenance*. Software engineering addressed the problem of minimizing the work needed to extend an application and to keep trace of the changes in a rational way, by emphasizing design for change and the provision of better development tools. In a distributed setting, however, the action of installing or rebuilding the application at each site still has to be performed locally and with human intervention. Some products, notably some Web browsers, already use some limited form of program downloading to perform automatic upgrade over the Internet. Mobile code helps in providing more sophisticated automation for the installation process. For instance, a scheme could be devised where installation actions (that, by their nature, can usually be automated) are coded in a mobile program roaming across a set of hosts. There, the program could analyze the features of the local platform and operate the correct configuration and installation steps. Pushing even further these concepts, let us suppose that a new functionality is needed by an application, say, a new dialog box must be shown when a particular button is pushed on the user interface. In a distributed application designed with conventional techniques, the new functionality needs to be introduced by reinstalling or patching the application at each site. This process could be lenghty and, even worse, if the functionality is not fundamental for application operativity there is no guarantee that it will be actually used. In this respect, the ability to request on demand the dynamic linking of the code fragment implementing the new functionality provides several benefits.

First, all changes would be centralized in the code server repository, where the last version is always present and consistent. Moreover, changes would not be performed proactively by an operator on each site, rather they could be performed reactively by the application itself, that would request automatically the new version of the code to the central repository. Hence, changes could be propagated in a lazy way, concentrating the upgrade effort only where it is really needed.

Mobile code concepts and technology embody also a notion of *autonomy* of application components. Autonomy is a useful property for applications that use a heterogeneous communication infrastructure where the nodes of a network may be connected by a variety of physical links with different performances. These differences must be taken into account since from the design level. For instance, recent developments in mobile computing evidenced that low-bandwidth and low-reliable communication channels require new design methodologies for applications in a mobile setting [35, 51]. In networks where some regions are connected through wireless links while others are connected through conventional links the design becomes complex, in that it must cope with frequent disconnections and aim at avoiding as much as possible the generation of traffic over the low-bandwidth links. The CS paradigm has a unique alternative to achieve this objective: to raise the granularity level of the services offered by the server. This way, a single interaction between client and server is sufficient to specify a high number of lower level operations, which are performed locally on the server without involving communication over the physical link. Nevertheless, this solution may be impossible to achieve in certain cases given the specific application requirements and, in any case, it leads to increased complexity and size as well as reduced flexibility of the server. Code mobility overcomes these limits because it allows for specifying complex computations that are able to move across a network. This way, the services that need to be executed by a server residing in a portion of the network reachable only through an unreliable and slow link could be described in a program. This should pass once through the wireless link and be injected into the reliable network. There, it could execute autonomously and independently. In particular, it would not need any connection with the node that sent it, except for the transmission of the final results of its computation.

Autonomy of application components brings improved *fault tolerance* as a side-effect. In conventional client-server systems, the state of the computation is distributed between the client and the server. A client program is made of statements that are executed in the local environment, interleaved with statements that invoke remote services on the server. The server contains (copies of) data that belong to the environment of the client program, and will eventually return a result that has to be inserted into the same environment. This structure leads to well-known problems in presence of partial failures, because it is very difficult to determine where and how to intervene in order to reconstruct a consistent state. The action of *migrating* code, and possibly sending back the results, is not immune from this problem. In order to determine whether the code has been received and avoid duplicates or lost mobile code, an appropriate protocol must be in place. However, the action of *executing* code that embodies a set of interactions that should otherwise take place across the network is actually immune from partial failure. An autonomous component encapsulates all the state involving a distributed computation, and can be easily traced, checkpointed, and possibly recovered *locally*, without any need for knowledge of the global state.

Another advantage that comes from the introduction of code mobility in a distributed application is *data management flexibility* and *protocol encapsulation*. In conventional systems, when data are exchanged among components belonging to a distributed application, each component owns the code describing the protocol necessary to interpret the data correctly. However, it is often the case for the "know-how" related to the data to change frequently or to be determined case by case according to some external condition—thus making impractical to hard-wire the corresponding code into the application components. Code mobility enables more efficient and flexible solutions. For example, if protocols are only seldom modified and are loosely coupled with data, an application may download the code that implements a particular protocol only when the data involved in the computation need a protocol unknown to the application. Instead, if protocols are tightly coupled with the data they accompany, components could exchange messages composed by both the data and the code needed to access and manage such data.

4.2 Application Domains for Mobile Code

The following review of application domains for mobile code serves two purposes. First, we want to describe some of the domains which are expected to exploit in the near future the benefits described previously, in order to provide the reader with an idea of the applicability of the concepts presented so far. Second, we want to point out that some concepts which are often associated *tout court* with code mobility are not mobile code approaches per se, rather they are examples of the exploitation of mobile code in a given application domain.

4.2.1 Distributed Information Retrieval

Distributed information retrieval applications gather information matching some specified criteria from a set of information sources dispersed in the network. The information sources to be visited can be defined statically or determined dynamically during the retrieval process. This is a wide application domain, encompassing very diverse applications. For instance, the information to be retrieved in principle could range from the list of all the publications of a given author to the software configuration of hosts in a network collected for administrative purposes. Code mobility could improve efficiency by migrating the code that performs the search process close to the (possibly huge) information base to be analyzed [60]. This type of application has been often considered "the killer application" motivating a design based on the MA paradigm. However, analysis to determine the network traffic in some typical cases evidenced that, according to the parameters of the application, the CS paradigm sometimes can be the best choice [21].

4.2.2 Active Documents

In active documents applications, traditionally passive data like e-mail or Web pages, are enhanced with the capability of executing programs which are somewhat related with the document contents, enabling enhanced presentation and interaction. Code mobility concepts are fundamental for these applications since they enable the embedding of code and state into documents, and support the execution of the dynamic contents during document fruition. A paradigmatic example is represented by an application that uses graphic forms to compose and submit queries to a remote database. The interaction with the user is modeled by using the COD paradigm, i.e. the user requests the active document component to the server and then performs some computation using the document as an interface. This type of application can be easily implemented by using a technology that enables fetching of remote code fragments. A typical choice is a combination of WWW technology and Java applets.

4.2.3 Advanced Telecommunication Services

Support, management, and accounting of advanced telecommunication services like videoconference, video on demand, or telemeeting, demand for a specialized "middleware" providing mechanisms for dynamic reconfiguration and user customization—benefits provided by code mobility. For example, the application components managing the setup, signalling, and presentation services for a videoconference could be dispatched to the users by a service broker. Examples of approaches exploiting code mobility can be found in [64] and [66]. A particular class of advanced telecommunications services are those supporting mobile users. In this case, as discussed earlier, autonomous components can provide support for disconnected operations, as discussed in [45].

4.2.4 Remote Device Control and Configuration

Remote device control applications are aimed at configuring a network of devices and monitoring their status. This domain encompasses several other application domains, e.g., industrial process control and network management. In the classical approach, monitoring is achieved by periodical polling of the resource state and configuration is performed using a predefined set of services. This approach, based on the CS paradigm, can lead to a number of problems [127]. Code mobility could be used to design and implement monitoring components that are co-located with the devices being monitored and report events that represent the evolution of the device state. In addition, the shipment of management components to remote sites could improve both performance and flexibility [7, 43]. A case study focused on the application of our taxonomy to the network management application domain is presented in Part II.

4.2.5 Workflow Management and Cooperation

Workflow management applications support the cooperation of persons and tools involved in a development process. The workflow defines which activities must be carried out in order to accomplish a given task as well as how, where, and when these activities involve each party. A way to model this is to represent activities as autonomous entities that, during their evolution, are circulated among the entities involved in the workflow. Code mobility could be used to provide support for mobility of activities that encapsulate their definition and state. For example, a mobile component could encapsulate a text document that undergoes several revisions. The component maintains information about the document state, the legal operations on its contents, and the next scheduled step in the revision process. An application of these concepts can be found in [18].

4.2.6 Active Networks

The idea of active networks has been proposed recently [110, 128] as a means to introduce flexibility into networks and provide more powerful mechanisms to "program" the network according to applications' needs. Although some interpret the idea of active networks without any relation with code mobility [13], most of the approaches rely on it and can be classified along a spectrum delimited by two extremes represented by the programmable switch and the capsule approaches [110]. The *programmable switch* approach is basically an instantiation of the COD paradigm, and aims at providing dynamic extensibility of network devices through dynamic linking of code. On the other hand, the *capsule* approach proposes to attach to every packet flowing in the network some code describing a computation that must be performed on packet data, at each node. Clearly, active networks aim at leveraging off of the advantages provided by code mobility in terms of deployment and maintenance, customization of services, and protocol encapsulation. As an example, in this scenario a multiprotocol router could dynamically download on demand the code needed to handle a packet corresponding to an unknown protocol, or even receive the protocol together with the packet. The work described in [120] is an example of an active network architecture exploiting the COD paradigm. In Chapter 15 we present the design of a videoconference system that exploits code mobility in active networks.

4.2.7 Electronic Commerce

Electronic commerce applications enable users to perform business transactions through the network. The application environment is composed of several independent and possibly competing business entities. A transaction may involve negotiation with remote entities and may require access to information that is continuously evolving, e.g., stock exchange quotations. In this context, there is the need to customize the behavior of the parties involved in order to match a particular negotiation protocol and it is desirable to move application components close to the information relevant to the transaction. This problems make mobile code appealing for this kind of applications. Actually, Telescript [121] was conceived espressly to support electronic commerce. For this reason, the term "mobile agent" is often related with electronic commerce. Another application of code mobility to electronic commerce can be found in [72].

Final Remarks on Understanding Code Mobility

Mobile code appears to be promising as a solution for the design and implementation of large scale distributed applications, in that it overcomes many of the drawbacks of the traditional client-server approach. Most research efforts in this field have been focused on the development of mobile code technologies, rather than on the formulation of a sound conceptual framework for code mobility. Hence, research developments in this area have not been supported by a comprehensive characterization and comparison of the technologies proposed and by a corresponding development of methodologies.

In part of the thesis we proposed a conceptual framework structured along three classes of concepts: *applications*, *design paradigms*, and *technologies*. Applications are the solutions to specific problems. Paradigms guide the design of applications. Technologies support their development. We surveyed each of these concepts and pointed out features, advantages, and disadvantages of existing approaches and proposals.

The intent of the framework presented in this part of the thesis is to foster the progress towards a common understanding of the issues and contributions in the area of mobile code. Also, we expect it to be useful to practitioners, who might find it useful as a guideline during the development process to identify the opportunities for exploitation of mobile code at the appropriate abstraction level. Certainly, the work presented in this part of the thesis needs to be incrementally enriched and revised, taking into account experiences, results, and innovations that will be achieved progressively. We need to improve our understanding of the properties and weaknesses of the existing design paradigms. We also need to consolidate a conceptual framework for mobile code languages, that makes it possible to compare them as we do for traditional programming languages. Another issue is the development of models enabling formal reasoning and verification, that we address in Part III. Finally, we need to explore further the relatively unknown world of applications and problems that can benefit from the adoption of technology and methodology based on mobile code. Nonetheless, we believe that the concepts presented in this part of the thesis can be instrumental in the creation of a mature and comprehensive background for the evolution and further diffusion of mobile code applications and techniques.

Part II

Evaluating Code Mobility

Chapter 5

Motivations, Background, and Approach

In this part of the thesis we focus on evaluating the benefits that code mobility can provide in a real world application domain: network management. In doing this, we pursue the goal of illustrating how the taxonomy presented earlier can be useful to guide the software engineer through the design and implementation phases of the application development process. The suggested development process, guided by our taxonomy, is as follows. Given an application whose requirements have been already specified, the first step is to determine if the mobile code approach is suited to meet the application needs—that is, whether code mobility should be used at all. This early evaluation is performed on the basis of the discussion at the beginning of Chapter 4. The second step involves identifying the suitable paradigms for the design of the application at hand, through some form of qualitative or quantitative analysis. Finally, after the suitable paradigms have been chosen, the technology for implementation has to be selected by examining the tradeoffs highlighted at the end of Chapter 3, e.g., trading ease of programming for lightweight implementation. Clearly, the core of this process is the selection of the right design paradigms. As we discussed earlier, the exploitation of a mobile code design is not an advantage per se, rather it must be evaluated as an alternative to traditional design in the context of the particular application being designed, *for each application functionality* whose design could involve code mobility. This can be done informally and qualitatively but, as evidenced in Chapter 3, a quantitative evaluation provides more insight and confidence about the existing tradeoffs.

The development of an analytical model to guide such a quantitative evaluation constitues a novel contribution that can be regarded under different perspectives. First of all, it provides formal arguments for the pro and cons of mobile code in a relevant application domain. At the time of writing, almost no contribution exists on this subject. Arguments in favor of code mobility are usually supported only by qualitative and superficial considerations, often biased towards a particular technology. The methodological contribution we present here can hopefully be extended and adapted to application domains other than network management. From a strict network management perspective, it provides managers with a formal tool to determine the best way to perform management operations. One could envision a scenario where the network manager can select from a library of management tasks the implementation that best suits management needs, as identified by a refinement of the model presented here. Finally, from a software engineering point of view it constitutes an experience in evaluating design alternatives quantitatively.

We chose this application domain because the synergy between research on network management and on code mobility is promising. Network management is looking for ways to overcome the limitations of mainstream client-server architectures, and code mobility now provides technologies and design paradigms for this goal. On the other hand, network management provides a good test-bed for the mobile code approach. Nevertheless, efforts in this direction are only in their early stages [43, 110, 66].

This part of the thesis is structured as follows. The remainder of this chapter provides some background information on the goals and challenges of network management and describes *management by delegation*, an early application of code mobility to network management. Chapter 6 analyzes the benefits that the mobile code design paradigms discussed in Chapter 3 are expected to bring in network management. The analysis is qualitative and informal; a quantitative analysis is the subject of Chapter 7, where an analytical model of management functionality is developed. The results developed in these two chapters, which enable the selection of the most convenient design paradigm, are finally extended in Chapter 8 to take into account the choice of the appropriate technologies for its implementation.

The work presented in this part of the thesis has been published in [7, 9].

5.1 The Challenges of Network Management

The world of network management research can be split grossly in two worlds: management of IP networks, where the *Simple Network Management Protocol* [22] proposed by IETF is the dominant protocol, and management of ISO networks, dominant in the telecommunication field and based on the *Common Management Information Protocol* [80].

The two approaches are very similar in the architecture they adopt though they differ slightly in the way they operate [56]. Both assume a centralized management architecture based on a client-server paradigm. A *network management station* (NMS), operated by the network manager, interacts with *agents*[1] running on network nodes like bridges, routers, and workstations. Agents are computational entities whose purpose is to provide a standardized interface for accessing information related to the network device on which they reside. Each agent stores the relevant parameters of the corresponnding device in a hierarchical base of information called *management information base*[2]. The NMS

[1] Despite the name, management agents are conventional programs which cannot move and in general do not exhibit a great deal of intelligence.

[2] MIB is actually the term used in SNMP only. CMIP uses the term *management information tree* (MIT) database, instead. Hereafter, we ignore the difference for the sake of simplicity.

plays the role of the client that provides a user interface to the network manager and requests information from the agents. Communication exploits a a *management protocol* that specifies the packet format for a set of basic operations. Messages containing data and operation codes are transmitted using transport layer services.

5.1.1 The Drawbacks of Centralization

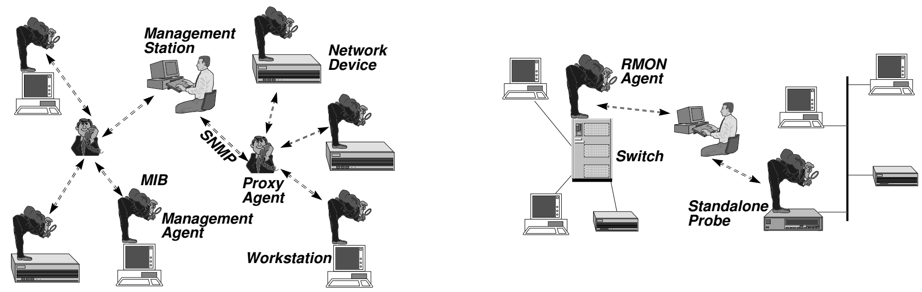
In the setting described so far, all the computation related to management, e.g., statistics, is demanded to the NMS. The operations available to the NMS for accessing a MIB are very low level. In SNMP, for instance, the NMS can only `get` and `set` atomic values in the MIB. This fine grained client-server interaction is often called *micro-management*, and leads to the generation of intense traffic and computational overload on the NMS.

Thus, both the IETF and the ISO approach are characterized by a high degree of centralization, since their architecture puts almost all the computational burden on the NMS. Centralization has proven to limit seriously the scalability of network management [127]. As the dimension of the network grows, the NMS needs to communicate with a larger number of devices, as well as store and process an ever increasing amount of data. This leads to the need for high cost hardware dedicated to the NMS, to poor performance, or even to the impossibility to cope with the dimension of the network. In some cases, a separated network is employed to carry management data, but in most cases *in band management* is employed, i.e. the very network that is being managed is used to carry management information. Consequently, the area of the network around the NMS experiences heavy traffic due to the combination of messages sent around by the management station and those containing data from the devices. The worst shortcomings of the centralized in band approach show up during periods of heavy congestion, when management intervention is particularly important. In fact, during these periods: *i)* the NMS increases its interactions with the devices and possibly downloads configuration changes, thus increasing congestion, *ii)* access to devices in the congested area becomes difficult and slow (sometimes even impossible), and *iii)* congestion, as an abnormal status, is likely to trigger notifications to the NMS which worsen congestion.

Similar problems affected routing table computation some years ago when it was centralized. A routing control center gathered information on network topology, calculated the routing table for each router in the network, and downloaded it into the proper device. The heavy traffic load in the area around the routing control center and the difficulty of managing network areas far from it led to the development of distributed routing, which is still being applied successfully. Analogously, in order to cope with problems that arise from centralization, the network management functionality must be decentralized, i.e. the complex diagnosing and information gathering activities must be moved from the NMS into the network.

5.1.2 Decentralization Features

The disadvantages of centralization have been recognized by the network management community, and both IETF and ISO have already introduced some



(a) SNMPv2 proxy agents. **(b)** RMON probes.

Figure 5.1: Decentralization features in the IETF approach.

primitive form of decentralization in the latest definitions of their management architectures.

Event notification is a first step in decentralization and aims at relieving the NMS from part of its processing burden. SNMP agents are designed to notify upon the occurrence of a few significant events by issuing *traps*, i.e. messages sent without an explicit request from the NMS. Since the IETF approach aims at keeping agents as simple as possible, they can notify only upon events that can be identified with a lightweight computation. Typical examples are changes in the status of a link (from down to up, or vice versa) or a device that has been reset. In turn, the ISO approach encompasses more complex agents, able to perform more processing on the data collected, thus increasing the number of significant situations (*events* or *alarms* in the OSI jargon) possibly dependent on several parameters that can be recognized and notified. Anyway, in both the approaches the agent is responsible only for the notification of an event and has no means of recovering from it. In other words, all the "management knowledge" resides on the NMS.

According to the ISO approach, the NMS can invoke actions on the objects stored in the MIT database, which is object-oriented. The execution of these actions, which can be fairly complex, reduces both the processing load on the NMS and the management traffic. The IETF approach does not provide an explicit remote action invocation mechanism. Nevertheless, this can be implemented by forcing the agent to execute actions as a side effect of a change, operated by the NMS, in predefined data items in the MIB. In other words, changes in the information base trigger actions performed remotely by the agent. SNMPv2 [23] takes a step further towards decentralization by defining hierarchical decentralization through the concept of *proxy agents*. A proxy agent is responsible for the management of a pool of devices (towards which it acts as a client) on behalf of the NMS (towards which it acts as a server). However, the use of SNMPv2 is presently limited by availability of implementations and by the many concerns raised about its definition, mostly as far as security is concerned.

IETF has proposed another approach, *Remote MONitoring* (RMON) [117], that introduces a higher degree of decentralization. RMON assumes the availability of network monitoring devices called *monitors* or *probes*. By monitoring packet traffic and analyzing the headers, probes provide information about links, connections among stations, traffic patterns, and status of network nodes. Hence, RMON can be regarded as a *traffic-oriented* approach because the status of the network is determined by direct inspection of the packets flowing in it rather than inspection of the status of each device, like in the *device-oriented* approaches reviewed before. As shown in Figure 5.1(b), probe implementations can be stand alone devices dedicated to link monitoring or can be embedded into network devices. SNMP is used for communication between the NMS and the agent running on each probe. Some of the processing that in traditional approaches is performed by the powerful hardware of the NMS, in RMON is performed by the agents on the probes *locally* and *independently* of the NMS. A probe can detect failures, misbehaviors, and identify complex relevant events even when not in contact with the NMS, which is likely to happen when the network is overloaded or in critical conditions—i.e. when network management is vital. The concept of *alarm* increases decentralization: the agent on the probe is configured to take periodic statistical samples of relevant parameters and, whenever a predefined threshold is crossed by any of them, to notify the NMS. In addition, the agent on the probe can perform semantic compression of data by pre-processing the information collected before sending it to the NMS, thus reducing the traffic.

As discussed in Chapter 4, network management applications may overcome some of these limits by taking advantage of the benefits of the mobile code approach, such as opportunities for semantic compression, dynamicity in service deployment and customization, autonomy, and fault tolerance.

5.2 Management by Delegation

The approaches to network management that involve code mobility are usually grouped under the label *management by delegation*. Management by delegation has been conceived originally before the appearance of Internet-based MCSs, as a means to cope with the micro-management phenomenon of centralized management. The original proposal [129], further extended in [43], identifies an architecture for the dynamic uploading of management scripts on network devices using a combination of the REV and CS paradigms. The management architecture still includes a management protocol and agents, yet an *elastic process run-time support* is assumed on each device. Instead of exchanging bare messages, the management station can specify a task by packing into a program a set of commands to agents and send it to the devices involved, thus *delegating* to them the actual execution of the task. This execution is completely asynchronous, enabling the management station to perform other tasks in the meantime and introducing a higher degree of parallelism in the management architecture. Moreover, since the code fragments (which the authors claim can be written in any language) are not statically bound to devices, they can be changed and re-sent by the management station at any time. This enables more flexibility, because the management station can customize and enhance *dynamically* the services provided by the agents on the devices.

In management by delegation code migration is always triggered by the NMS, and there is no support for autonomous mobility of delegated scripts. The original idea has been interpreted in many ways by researchers leading to prototypes which are often based on relatively limited "mobile code" technology, like interpreted Perl scripts. Management by delegation is presently under standardization by IETF and ISO working groups [98].

Management by delegation can be considered a precursor of the ideas discussed in the remainder of this part of the thesis. Nevertheless, the use of the paradigms described in Chapter 3 covers different and broader aspects of mobility and can include management by delegation as a special case. Furthermore, they can be implemented by leveraging off of a number of languages providing abstraction conceived expressly for code mobility, as described in Chapter 2.

The reader interested in an extensive survey and comparison of the approaches to network management found in literature is directed to [68].

Chapter 6

A Qualitative Analysis

Mobile code provides the technology needed to enhance network management with the degree of flexibility needed to cope with the problems mentioned in the previous chapter. Mobile code can be linked dynamically on network devices either proactively by the NMS or reactively by the network device; also, the capability to embed in the mobile code the strategy for migration, provides an extra level of autonomy to management operations. A NMS can then extend dynamically a management agent with mobile code as well as delegate to such code the capability to migrate autonomously among devices to perform complex management tasks without involving the NMS.

In this chapter we analyze qualitatively and informally the advantages of an approaches that move "management functions to the data rather than move data to these functions"[43] and exploit the mobile code design paradigms described in Chapter 3.

6.1 Code on Demand: Flexibility

In the IETF approach the MIB structure is implemented by the code of the management agent and cannot be changed at run-time. This lack of flexibility represents a limit in particular for event notification, because the definition of significant events is hard-wired in the MIB structure and customized events cannot be defined. It is desirable to extend agents so that they can check arbitrarily complex event and possibly react locally without requiring the intervention of the NMS. The extension of a management agent can be achieved with traditional technology based on the client-server paradigm: the ISO approach allows the MIT to be changed dynamically, and also several CORBA-based approaches to network management enable this feature. However, these solutions present some relevant drawbacks.

First of all, one could argue that the reason why the primitives offered by management agents are so poor is that agents must be lightweight, since they can be embedded in network devices equipped with limited computational resources. In this setting, adding management primitives to agents is just not desirable. To be fair, this "myth of the dumb agent" [119], indeed true some years ago, is being undermined by the technological developments of network devices which are equipped with increasing amount of memory and computational power.

However, even if adequate computational resources are available another relevant issue is the frequency of invocation of primitives. A function may be used only every now and then, e.g., because its execution is needed only during periods of heavy congestion or because its code may change slightly according to operating conditions. In these cases, hard-wiring the function into the agent just makes it bigger, wasting device resources without appreciable gain. In general, the dynamic nature of management operations demands for dynamic customizability of agent primitives.

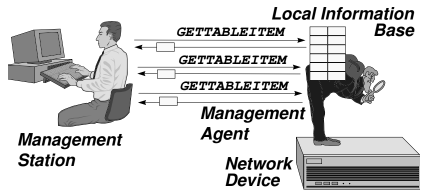
Code on Demand can increase flexibility while keeping agents simple and small, thus combining the advantages of IETF and ISO approaches. Whenever the structure of the local information base needs to be changed, e.g., to modify the criteria to detect an event, the corresponding portion of the agent code can be invalidated and replaced automatically with a new one retrieved from a code server. This way, the management primitives embedded in it become available on the device only when requested by management operations, thus consuming device resources only when this is really needed. Also, the agent can be "patched" without any need to rebuild and reinstall it, and this operation does not require direct operation by the network administrator, since it is made once on the code server and propagated *on demand* on every node. Consequently, this approach can be exploited to perform *i) perfective* maintenance as in this case, *ii) adaptive* maintenance, i.e. to upgrade the agent code as a consequence of a change in the underlying software platform, like a change in the operating system version. As an aside, it can be noted that a COD paradigm is likely to be of great help also in network domains other than network management. For example, a new protocol module could be added to a router by downloading it on demand from a central repository.

6.2 Remote Evaluation: Bandwidth Savings and Processing Distribution

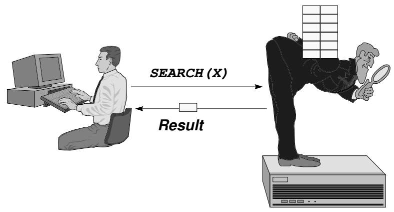
In mainstream network management approaches, data are moved from agents to the NMS and processed there. Afterwards, some data are possibly sent back to agents. Again, this aims at keeping agents small, without overloading them with the code implementing the operations the NMS is in charge of. As an example, let us consider the case in which the NMS needs to search for a single value in a table, which is commonly used to store information inside devices. With mainstream approaches this can be done in one of the following ways:

1. The whole table is transferred into the NMS, where its rows are searched for the value (Figure 6.1(a)). Hence, the larger the table the more bandwidth is wasted and the more computational overload is generated on the NMS.

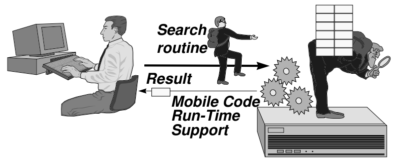
2. The searching routine can be implemented into the agent code, as in Figure 6.1(b). As we already pointed out, this increases the size of the agent, and may not make sense for a function that is used only seldom.



(a) Client-server.



(b) Client-server.



(c) Mobile code.

Figure 6.1: Three different ways to perform table lookup.

The REV paradigm could be used to pack together the set of SNMP operations describing the search and send them as a single function on the device holding the table, exploit local interaction there. After execution, only the target value should be sent back—thus *semantic compression of data* is performed. Intuitively, this solution is likely to save bandwidth at least for big tables and small routines and distributes the computational load among the agents, rather than concentrating it on the NMS. Semantic compression is a benefit that in principle can be provided by all the mobile code paradigms: the function installed on a device by the COD paradigm can perform semantic compression and the MA paradigm, as described in the next section, pushes this idea even further.

The solution described above provides also a desirable side-effect: it raises the level of abstraction of the operations available to the network manager. One could envision a scenario where the manager builds her own management procedures upon lower level primitives, stores them on the NMS, and invokes their remote evaluation on the appropriate device whenever needed. This brings a nice side-effect of modularity into the management architecture, and provides better performance by reducing the number of messages exchanged between the agent and the NMS, thus limiting the load in the area around it.

We agree with this observation as we do not propose to exploit REV for the decentralization of every management task. REV and the other paradigms, as already discussed in Chapter 3, are not a silver bullet and their use has to be assessed case by case according to a set of parameters specific to the functionality that has to be implemented. This observation is the rationale for the analysis in Chapter 7.

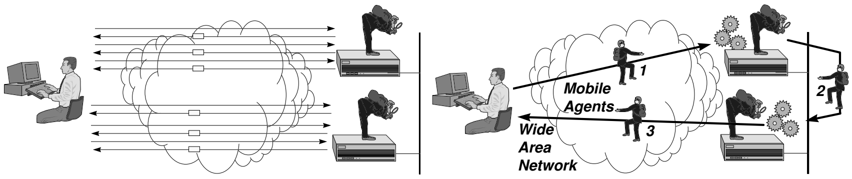
6.3 Mobile Agent: Autonomous and Mobile Management Functionality

The capability to retain the state across several hops implicit in an MA design adds a new dimension to the benefits achievable through an REV design: autonomy. In the REV paradigm each remote evaluation on a device must be initiated explicitly by the NMS, and this has been highlighted as a shortcoming of management by delegation. In the MA paradigm, the NMS can exploit the capability of a mobile component to retain its state and demand to it the retrieval of information from a specified pool of devices. Thus, it can delegate to it the decision about when and where to migrate, according to its current state. Besides providing increased flexibility and distributing processing load like in the other paradigms, this scheme is likely to reduce the traffic traffic around the NMS, because each query to the network devices being visited by the mobile component is performed locally by the mobile agent without involving a message on the net—only the injection of the mobile component and the transmission of the final results need to go through the NMS.

By visiting a number of devices, the mobile component can also extend the concept of semantic compression enabled by REV and COD. These paradigms can provide only a form of semantic compression that is limited to a single device, and local to it. The MA paradigm, in contrast, enables *global semantic compression* of data across all the network devices visited. Whether this is actually improving traffic load is still unclear at this point, and depends on the information that needs to be collected that may cause the state of the mobile component is to grow from hop to hop. This issue will be analyzed quantitatively in the next chapter.

Some other advantages can determine the choice of this paradigm, independently of the issue of traffic. For instance, let us consider a scenario where the pool of devices to be managed resides in a LAN, and assume that the NMS is connected to the managed devices by a long-haul link, likely to be unreliable and slow. In traditional approaches this operation would require many data exchanges on slow links between the NMS and the devices (Figure 6.2(a)) with higher latency and risk of partial failure. With MA, the mobile component, once injected into the LAN, can collect information about all the managed devices without any need to be connected with the NMS (Figure 6.2(b)). Even if the state of the mobile component increases during this operation, bandwidth is assumed to be cheaper within the LAN than on the long-haul link.

The mainstream approaches to network management presented in Section 5.1



(a) Client-server. **(b)** Mobile Agent.

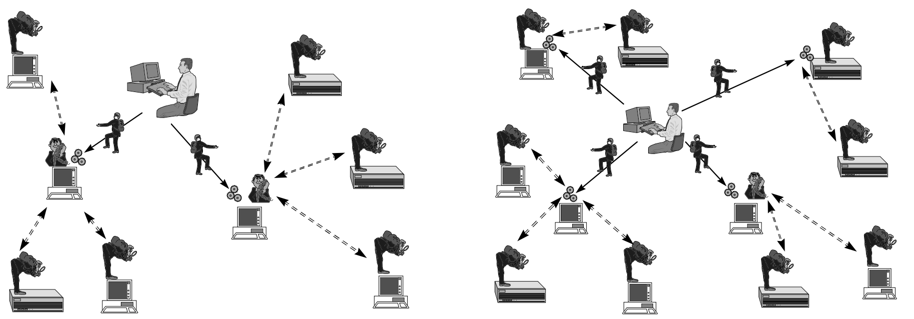
Figure 6.2: Interaction between the NMS and the devices.

exploit transport layer protocols to carry management messages. This allows the NMS to communicate transparently with agents all over the network but, on the other hand, it makes management completely impossible when network level routing is disrupted. Mobile agents can help in overcoming this problem. If the NMS does not have network level connectivity with a node to be managed through an agent, it can provide its agent with a route calculated from historical routing information and send it to the first hop on the route. Whenever the agent resumes execution on an intermediate hop, it tries to reach one of the next hops towards the target node using its internal route, until it reaches the target and performs its management task.

Finally, mobile agents can also be effective in detecting and bypassing routing loops. If the NMS has clues to the presence of a loop, it can send a mobile agent to the first router towards the destination. This agent uses the information in its local routing table to move closer to the destination, while keeping track of the routers it is traversing. The presence of a routing loop is detected whenever the agent finds itself on a router already visited. At this point the mobile agent can try to: *i)* bypass the routing loop, trying to reach the destination through an interface chosen randomly, or *ii)* resolve the routing loop, e.g., disabling the interface on the loop and monitoring the reaction of the routing protocol. In principle, the NMS could detect the routing loop on its own by reading the routing tables with the management protocol. Nevertheless, due to the presence of routing problems, it is likely that the NMS does not have connectivity to some of the routers, and thus the partial or out-of-date information is not sufficient to identify the loop.

6.4 Feasibility Considerations and Interim Solutions

The lack of network devices supporting MCSs, which require a dedicated and often heavy run-time support, is today a major obstacle to the exploitation of code mobility in network management. Nevertheless, the opportunities offered



(a) Mobile code and proxy agents.

(b) Mixed interim solutions.

Figure 6.3: Interim solutions.

by code mobility within the networking domain are becoming recognized by an increasing number of researchers and organizations. A great deal of research effort is being invested in studying how to devise architectures that embed code mobility at the network device level [110]. In parallel, some vendors already announced on-chip support for Java in the next release of their devices, according to a trend where manufacturers are placing more and more computational power devoted to network management into their devices, thus enabling mobile code support to take advantage of existing hardware. For instance, the switches of some manufacturers are already equipped with a processor devoted to packet analysis and other functionalities required by RMON. Moreover, the problem of having a common format to ensure interoperability among the devices of different vendors, possibly using different languages, will exist. This problem is already evident in the mobile code arena, where the adoption of translation strategies based on an intermediate language has boosted portability but, on another hand, requires agreement on such a language in order to achieve full interoperability among different mobile code environments.

However, some interim solutions can be devised which, although less effective, are easier to implement and do not require support for code mobility on every node. These interim solutions could be a test-bed for code mobility in network management, in order to assess its effectiveness and possibly raise interest among the equipment manufacturers. Furthermore, they could be useful to manage legacy systems without built-in support for mobile code. In the remainder of this section we sketch briefly three architectures which can be used separately or in conjunction for this goals. A scenario where all the interim solutions are applied to the same network is shown in Figure 6.3(b).

1. Code mobility can be exploited to interact with SNMPv2 proxy agents running on workstations equipped with mobile code run-time support. In the interaction with proxy agents the NMS gets the advantages previously described for the interaction with management agents. For example, management functions can be evaluated on the proxy agents' workstations (Figure 6.3(a)) with a REV or MA paradigm, or the functionality of the

proxy agent can be updated using a COD paradigm.

2. Mobile code could be sent to workstations with proper support which are installed in the managed area. As shown on the left in Figure 6.3(b), this code should interact with management agents on the nodes in the area around such workstations by using a conventional management protocol, e.g., SNMP or CMIP. This solution is not as beneficial as mobile code executing directly on network devices. Nevertheless, it enables local exchange of management protocol messages (e.g., useful on LANs with large capacity), having mobile code traveling from the NMS to the managed areas. This solution is being implemented in a prototype study at our university.

3. When network devices will be equipped with mobile code run-time support (upper right part of Figure 6.3(b)), this will be exploited in place of the workstations above. Mobile code will interact not only with the devices hosting the run-time support, but also with legacy ones installed in their neighborhoods, using conventional management protocols.

Chapter 7

A Quantitative Analysis

The previous chapter evidenced informally and qualitatively several advantages that mobile code design paradigms provide over the CS paradigm. Nevertheless, we pointed out in Chapter 3 that mobile code design paradigms are not good per se, rather their application must be carefully analyzed on a case-by-case basis, taking into account traditional paradigms as well. In this chapter, we exemplify this concept by comparing formally and quantitatively different solutions to the problem of polling device data. We focus on the realization of a single management task, and characterize it by defining its relevant parameters. Our goal is to determine which are the performance of the paradigms with respect to the traffic they generate around the NMS as well as globally on the managed network. Hence, we do not consider other factors, like CPU utilization, and focus uniquely on the decentralization of network traffic.

The model is conceived to guide the manager during the selection of the right design choice for a management functionality. However, the model presented in this section is a general one, and has to be adapted to the management scenario where the manager operates. This scenario must take into account the actual management protocols in place, as well as the technology actually used to implement a given paradigm. An example of such a refinement is provided in Chapter 8.

In Section 7.1 we assume a uniform network, i.e. a network without differences among the bandwidth, latency or, in general, cost of the links. Although somehow constraining, this assumption is still meaningful when managing a single LAN. The assumption is then removed in Section 7.2, where a model encompassing non-uniform network is discussed.

7.1 Model of Management Traffic

We consider a scenario where a NMS retrieves management data from a pool of devices, e.g., the load on every network interface of each device.

In what follows, a measure of the dimension of the managed network is given in terms of the number N of managed devices. The complexity of the management task is taken into account by the number Q of queries performed on the MIB. The transmission overhead introduced by protocol encapsulation, and possibly traffic control or connection setup, is taken into account down to the network layer, as defined by the OSI model. In fact, the focus is on the traffic generated on the network as a whole; the amount of data exchanged between network layer entities is independent of the lower layer technology, and thus invariant throughout the whole network. If a chunk of data of size X is to be transmitted at the application level, we represent the actual amount of data exchanged at the network layer as $X' = \alpha(X) + \beta(X)X$. In this expression, $\beta(X)$ may account, for example, for the overhead introduced by message encapsulation, while $\alpha(X)$ may account for the control information exchanged during the setup phase in a connection-oriented protocol. In the remainder of the section, however, we use the equivalent expression $X' = \eta(X)X$ where $\eta(X) = \frac{\alpha(X)}{X} + \beta(X)$, $\eta(X) > 1$. This analytical transformation does not change the meaning of the model; it just provides a more compact representation. $\eta(X)$ is called *overhead function* since accounts for the control information (*protocol overhead*) added to X by the network and the above layers. In the following, we write simply ηX in place of $\eta(X)X$, in order to simplify formulae. Table 7.1 illustrates most of the symbols used in the remainder of this chapter.

symbol	unit	description
T_i	byte	traffic generated in case i
N	node	number of managed network devices
Q	instruction	number of SNMP instructions needed to perform a query
I_q	byte	size of the q^{th} instruction
R_{qn}	byte	size of q^{th} reply coming from the n^{th} node
C_i	byte	code size in case i
η_i	—	overhead function applied to requests in case i
$\tilde{\eta}_i$	—	overhead function applied to replies in case i

Table 7.1: Symbols used in the quantitative model.

7.1.1 Model of the Overall Traffic

We derive here a model for the traffic generated by a management task which involves retrieving a set of data from managed devices. In devising this model, we assume that the same management operations are executed on each device. This is not necessarily the case for every management task, but allows for a simpler notation without compromising the generality and significance of what can be inferred from the model.

Client-server

With the traditional client-server paradigm, the NMS requests an operation to the management agent by sending a request message to it. We assume that such information has size I_q. In order to perform the management task, Q request messages must be sent to each of the N managed devices. Device n answers the q^{th} request with a reply whose size is R_{qn}. The overall traffic is then:

$$T_{CS} = \sum_{n=1}^{N} \sum_{q=1}^{Q} (\eta_{CS} I_q + \tilde{\eta}_{CS} R_{qn}) \tag{7.1}$$

where we differentiate between the overhead function used to send message requests (η_{CS}) and the one used for replies ($\tilde{\eta}_{CS}$). Multicast communication is sometimes used to send the same management request to multiple devices. In our model we do not consider this case as it is not very common. Moreover, we assume that the queries contained in each request message are dependent on the previous reply, i.e. it is not possible to aggregate request messages.

Remote Evaluation

If the REV paradigm is exploited, the Q requests are embedded in a code fragment of size C_{REV} sent on managed device n. Remote evaluation of the code produces the Q results R_{qn} which are sent back collectively to the NMS. This pairwise interaction has to take place for each of the N managed nodes. The overall traffic generated is given by

$$T_{REV} = \sum_{n=1}^{N} (\eta_{REV} C_{REV} + \tilde{\eta}_{REV} \sum_{q=1}^{Q} R_{qn}) \tag{7.2}$$

Mobile Agent

The NMS unleashes a mobile component that visits each of the N nodes and collects information locally. When modeling such a component, we model the code and the portion of the state needed for its execution (C_{MA}) as separate from the portion of the state relevant to the application. The latter grows as long as this agent travels from node to node. In fact, if we denote with $S_{MA,n}$ the size of the state of the agent during the trip towards node n, then

$$S_{MA,n} = \begin{cases} 0 & \text{if } n = 1 \\ \sum_{m=1}^{n-1} \sum_{q=1}^{Q} R_{qm} & \text{if } n > 1 \end{cases}$$

The first expression accounts for the fact that the mobile agent has not yet collected any information when traveling from the NMS to the first node. The second one accounts for the fact that, when traveling to node n, the agent already carries all the replies collected on the previous $n-1$ nodes. After information on the last node has been collected, the mobile agent sends back to the NMS all the results collected. An alternative design would return the component back to the NMS. Although sometimes viable, we chose the first solution for uniformity with the REV case. The overall traffic generated is given by

$$T_{MA} = \sum_{n=1}^{N} \eta_{MA}(C_{MA} + S_{MA,n}) + \tilde{\eta}_{MA} \sum_{n=1}^{N} \sum_{q=1}^{Q} R_{qn} \tag{7.3}$$

Code On Demand

With the COD paradigm, management agents can be augmented dynamically with code implementing primitives at a higher level of abstraction, thus providing agents with the capability to perform the Q operations locally, rather than across the network. The protocol we assume in this model is the following. As in the CS paradigm, the NMS requests an operation by sending a message that contains the operation signature, I_{COD}. If the operation has already been installed on the managed node, a reply is sent which contains the result of the Q queries, like in a REV implementation. On the other hand, if the code for the operation has not been installed yet, the agent replies with a message (that we assume of size I_{fetch}) requesting the dynamic download. The code, of size C_{COD}, is transferred and linked on the agent device, where it becomes available for future invocations, and the corresponding operation is performed. Consequently, the expression of traffic at equilibrium is:

$$T_{COD,stable} = \sum_{n=1}^{N}(\eta_{COD} I_{COD} + \tilde{\eta}_{COD} \sum_{q=1}^{Q} R_{qn}) \tag{7.4}$$

By converse, during the "setup" phase, there is an overhead represented by the message sent by the agent which requests the download, plus the actual code transfer: $T_{COD} = T_{COD,stable} + T_{COD,setup}$, where[1]

$$T_{COD,setup} = \sum_{n=1}^{N}(\eta_{COD} I_{fetch} + \eta_{COD} C_{COD}) \tag{7.5}$$

Evaluation

We elaborate on the expressions given earlier in order to find out the tradeoffs that must be considered in order to minimize the overall traffic. To this end, we compare the equations determined for the various paradigms.

For instance, the use of a REV paradigm is an improvement over traditional centralized management only if $T_{CS} \geq T_{REV}$. After elaboration of (7.1) and (7.2):

$$\begin{array}{l} N \sum_{q=1}^{Q} \eta_{CS} I_q + \sum_{n=1}^{N} \sum_{q=1}^{Q} \tilde{\eta}_{CS} R_{qn} \quad \geq \\ N \eta_{REV} C_{REV} + \sum_{n=1}^{N} \tilde{\eta}_{REV} \sum_{q=1}^{Q} R_{qn}. \end{array}$$

In order to simplify notation and have a better insight into the meaning of the formulae, we introduce some assumptions. Instead of differentiating the contribution of each request I_q and reply R_{qn} in (7.1), we consider the average values $\bar{I}$and $\bar{R}$. Consequently, the formula can be rewritten as:

$$NQ\eta_{CS}\bar{I} + NQ\tilde{\eta}_{CS}\bar{R} \geq N(\eta_{REV} C_{REV} + \tilde{\eta}_{REV} Q\bar{R})$$

It is likely[2] that $Q\tilde{\eta}_{CS}\bar{R} \geq \tilde{\eta}_{REV} Q\bar{R}$ since usually a fixed overhead is associated to each packet and thus, the longer the message being segmented, the smaller the relative overhead. Hence, if more results $\bar{R}$ can be transmitted together in a single message, the overhead is likely to be smaller, although depending on the protocols used to implement communications in CS and REV. We call the difference in the overhead introduced to send the results of the queries $\Delta O_{CS,REV} \geq 0$; REV is more convenient than CS if

$$\eta_{REV} C_{REV} \leq \Delta O_{CS,REV} + Q\eta_{CS}\bar{I}, \tag{7.6}$$

that is, if the size of the message containing the code to be evaluated remotely is smaller than the overall size of the message requests needed in a CS paradigm

[1] We do not model the intermediate situation where some of the nodes already contain the code, since we do not consider it relevant to the goals of this paper and can be easily derived from the result presented.

[2] It must be reminded that we use ηX as a shortcut for $\eta(X)X$.

plus the difference in overhead introduced when transmitting the result. Clearly, REV is convenient when the number of instructions Q needed to perform a query is high and C_{REV} effectively compacts the representation of the local interactions I_q within the code, e.g., using loop control structures.

We can apply the same reasoning and assumptions in order to determine which is the most convenient paradigm between REV and MA. It can be seen that a MA implementation always generates more traffic than a REV one. In fact, substituting and elaborating (7.2) and (7.3) in $T_{MA} \geq T_{REV}$ within the assumptions above, we obtain:

$$\begin{aligned} N\eta_{MA}C_{MA} + \tilde{\eta}_{MA}QN\bar{R} + \textstyle\sum_{n=1}^{N} \eta_{MA}Q(n-1)\bar{R} &\geq \\ N(\eta_{REV}C_{REV} + \tilde{\eta}_{REV}Q\bar{R}). & \end{aligned}$$

A first simplification is to assume

$$N\tilde{\eta}_{REV}Q\bar{R} \simeq \tilde{\eta}_{MA}QN\bar{R}. \tag{7.7}$$

This assumption holds when Q is sufficiently large and $\tilde{\eta}_{REV} \simeq \tilde{\eta}_{MA}$: in this case, more results are packed together for both, rather than being sent individually like with CS, and the difference in overhead is likely to be negligible. A more gross simplification is to consider $C_{REV} \simeq C_{MA}$. Usually is $C_{REV} < C_{MA}$, since C_{MA} contains the execution state as well as the code determining the next migration. If we apply this oversimplification the equation above becomes

$$\sum_{n=1}^{N} \eta_{MA}Q(n-1)\bar{R} \geq 0,$$

that is, REV is always more convenient than MA, because the latter must carry the state which is growing at every hop. As we discuss in Section 7.1.3, the possibility of performing semantic compression introduces a different evaluation criteria.

The application of the COD paradigm depends on the frequency of invocation which has not been considered yet. So far, we have given the expression of the traffic generated for a single execution of a management task. However, in general it may be interesting to consider how this varies over U different invocations. For the other paradigms, this additional parameter does not affect the expression of the traffic. The traffic generated during U executions of an implementation with a paradigm $p \in \{\text{CS, REV, MA}\}$, $T_p(U) = UT_p$. The expression of $T_{COD}(U)$ when COD is used is

$$T_{COD}(U) = T_{COD,setup} + UT_{COD,stable}$$

Calculation of $T_{REV}(U) \geq T_{COD}(U)$ under the likely assumptions that $\bar{I} \simeq I_{COD} \simeq I_{fetch}$ and $\tilde{\eta}_{REV}Q\bar{R} \simeq \tilde{\eta}_{COD}Q\bar{R}$ yields

$$\eta_{REV}C_{REV} \geq \frac{U+1}{U}\eta_{COD}\bar{I} + \frac{1}{U}\eta_{COD}C_{COD}$$

Clearly, if U is large, i.e. the primitive is invoked many times before being upgraded or discarded, the disequation above can be approximated by

$$\eta_{REV}C_{REV} \geq \eta_{COD}\bar{I}$$

which is always satisfied, the threshold being a REV code composed by a single instruction. If, by converse, U is small, all the terms must be considered.

However, we can assume that, although shipped with a different paradigm, the code describing the management function at hand is the same, i.e. $C_{REV} = C_{COD} = C$, and that we are comparing implementations of different paradigms with the same technology, i.e. $\eta_{REV} = \eta_{COD} = \eta$. Under these assumptions, we obtain

$$\eta C \geq \frac{U+1}{U-1}\eta \bar{I}$$

which confirms the intuition that if a function is used at least two times in a row, caching its code saves bandwidth. Finally, comparing $T_{CS} \geq T_{COD}$ under the assumption $\bar{I} \simeq I_{COD} \simeq I_{fetch}$, we obtain that

$$U \geq \frac{\eta_{COD}\bar{I} + \eta_{COD}C_{COD}}{Q\eta_{CS}\bar{I} - \eta_{COD}\bar{I} + \Delta O_{CS,COD}} \tag{7.8}$$

where $\Delta O_{CS,COD}$ is the difference in the overhead introduced by the two implementations to send back the results.

7.1.2 Model of the Traffic around the NMS

As we pointed out in Section 5.1, the NMS is likely to represent a bottleneck of the management system in a centralized approach, due to the micro-management phenomenon. Thus, although the measure of the overall traffic generated is surely relevant, it is important to compare the performance of the different paradigms also in terms of the traffic generated from and to the NMS.

The measure of the overall traffic generated by management operations and the measure of the management traffic flowing through the NMS coincide for all the paradigms considered, except for the MA paradigm. In fact, in the other paradigms, the NMS is always interacting directly with all the devices being managed. Instead, when the MA paradigm is used, there is traffic through the NMS only when the mobile agent is injected into the network and when the agent reports back the collected results, that is

$$T_{MA,Mgm} = \eta_{MA}C_{MA} + \tilde{\eta}_{MA}\sum_{n=1}^{N}\sum_{q=1}^{Q} R_{qn}. \tag{7.9}$$

The remainder of the traffic is generated without involving the NMS. It is useful to compare $T_{REV} \geq T_{MA,Mgm}$. If we consider $\bar{R}$ instead of the single contributions, and apply assumption (7.7), then

$$\frac{\eta_{MA}C_{MA}}{\eta_{REV}C_{REV}} \leq N$$

holds, i.e. MA is convenient with respect to REV if the ratio between the traffic generated when unleashing the mobile agent and the one to move the code to be evaluated remotely is smaller than the number of nodes being managed. Notably, for a single node REV is going to be more convenient, as usually $C_{MA} > C_{REV}$. Here, the tradeoff is between the size of the code of the mobile agent and its autonomy as far as mobility is concerned.

A comparison with COD at the equilibrium point, within the assumptions enumerated before, yields $T_{COD,stable} \geq T_{MA,Mgm}$, that is,

$$\frac{\eta_{MA}C_{MA}}{\eta_{COD}\bar{I}} \leq N.$$

Hence, the MA paradigm is more convenient than COD only if the size of the code and execution state of the mobile agent are sufficiently small. If we assume that the size of a single instruction in the code of the mobile agent is $\bar{I}$, its code must contain a number of instructions smaller than the number of nodes being managed.

As for CS, always within the assumptions above, $T_{CS} \geq T_{MA,Mgm}$ if

$$\Delta O_{CS,MA} \geq \eta_{MA} C_{MA} - QN\eta_{CS}\bar{I},$$

that is, if the difference of the protocol overhead generated to send the results in the two cases is greater than the difference between the size of the code of the mobile agent and the overall size of request message. Note that, if $QN \gg 1$, the right hand side is likely to be negative, thus always verifying the inequality.

7.1.3 Semantic Compression

size of compression. the does not allow semantic compression to be performed whenever paradigm, on the other hand, may enable semantic proper functionality on the management

$$T'_{COD,stable} = \sum_{n=1}^{N} (\eta_{COD} I_{COD} + \tilde{\eta}_{COD} R_n)$$

where we assumed that semantic compression of q data values R_{qn} yields a single value R_n. Clearly, $T_{COD,setup}$ is unchanged. The REV and MA paradigms can achieve semantic compression by executing remotely a routine that performs the compression, like the search routine described in Chapter 6. This generates a traffic

$$\begin{aligned} T'_{REV} = &\quad \textstyle\sum_{n=1}^{N} (\eta_{REV} C_{REV} + \tilde{\eta}_{REV} R_n) \\ T'_{MA} = &\quad \textstyle\sum_{n=1}^{N} [\eta_{MA}(C_{MA} + S'_{MA,n}) + \tilde{\eta}_{MA} R_n] \end{aligned}$$

and $T'_{REV} < T'_{MA}$, since in the expression of the traffic generated by MA there is still a term $S'_{MA,n}$ which grows linearly with the number of nodes visited and, in addition, usually $C_{REV} < C_{MA}$.

If we consider only the management traffic involving the NMS, like in the previous section, we obtain that

$$T'_{MA,Mgr} = \eta_{MA} C_{MA} + \tilde{\eta}_{MA} \sum_{n=1}^{N} R_n \tag{7.10}$$

To evaluate the improvement introduced by semantic compression, we compute the difference $\Delta T'_p$, $p \in \{\text{REV, MA, COD}\}$, between the traffic generated with the traditional CS paradigm and the traffic generated in the situations above. In doing this, we apply the usual approximation of considering average sizes $\bar{I}$ and $\bar{R}$ for instructions and results, respectively. For the sake of simplicity, we consider a unique overhead function η across all the paradigms, without differentiating between requests and replies. For mobile code paradigms, this is not a big approximation if we restrict the evaluation to implementations that use the same application level protocol for transferring both the code and the results. The real approximation is introduced in considering as equivalent the overhead introduced by a particular MCS (e.g, Java Aglets) and the overhead introduced by a traditional CS-based management protocol (e.g., SNMP). Under this assumptions, however, we obtain:

$$\begin{aligned}
\Delta T'_{COD,stable} &= N(Q-1)(\eta\bar{I}+\eta\bar{R}) \\
\Delta T'_{REV} &= NQ\eta\bar{I}+N(Q-1)\eta\bar{R}-\eta C_{REV} \\
\Delta T'_{MA,Mgm} &= NQ\eta\bar{I}+N(Q-1)\eta\bar{R}+\Delta O_{CS,MA}-\eta C_{MA}
\end{aligned}$$

After some trivial manipulations of the formulae above, we can express the above as functions of $\Delta T'_{COD,stable}$:

$$\begin{aligned}
\Delta T'_{REV} &= \Delta T'_{COD,stable}+N(\eta\bar{I}-\eta C_{REV}) \\
\Delta T'_{MA,Mgm} &= \Delta T'_{COD,stable}+N\eta\bar{I}+\Delta O_{CS,MA}-\eta C_{MA}
\end{aligned}$$

These formulae show that REV is never better than COD when the management task is repeated a number of times large enough to neglect the COD setup traffic. This is not true only if $\eta C_{REV} \leq \eta\bar{I}$, which is clearly unlikely, as it means that the message containing the code being evaluated remotely is smaller than a simple request message. The traffic around the NMS is reduced with respect to COD if

$$\eta C_{MA} \leq N\eta\bar{I}+\Delta O_{CS,MA},$$

which highlights that, if the code of the mobile agent is kept sufficiently compact, the gain in traffic grows linearly with the number of nodes.

The MA paradigm enables also a *global* form of semantic compression across all the devices, while the other paradigms enable only a *local* semantic compression on each device. For example, a mobile agent may travel across a set of nodes looking for the most loaded network interface. In doing this, only one data value needs to be carried from hop to hop—the state of the agent does not increase. In other words, if we assume that such data value has a fixed size $\bar{R}$, then $S_{MA,n} = \bar{R}$, and

$$T''_{MA} = N(\eta_{MA}C_{MA}+\tilde{\eta}_{MA}\bar{R}).$$

Again, MA is not more advantageous than the COD paradigm, at least as far as overall traffic is concerned. The calculation of $\Delta T''_{MA} \geq \Delta T'_{COD}$, assuming that the overhead function is the same, shows that this condition is met only if $\eta C_{MA} \leq \eta\bar{I}$ which, as we discussed earlier for local compression with REV, is practically never met. The real advantage of global compression, however, shows up when considering traffic involving the NMS. In this case, the traffic is simply

$$T''_{MA,Mgm} = \eta_{MA}C_{MA}+\tilde{\eta}_{MA}\bar{R} \tag{7.11}$$

and calculation of $\Delta T''_{MA} \geq \Delta T'_{COD}$ shows that MA is better than COD if

$$\eta C_{MA} \leq N(\eta\bar{I}+\eta\bar{R})-\eta\bar{R}$$

The first addendum actually represents a whole CS interaction: basically, the message containing the code of the mobile agent must be smaller than N pairs of request and reply messages.

7.2 Model of Management Costs

The model devised so far assumed a uniform network, i.e. a network where all the links have the same communication cost. This assumption is acceptable in many cases; it applies for example to the management of a single LAN. However, corporate intranets and extranets spanning across long-haul links with different characteristics, require an explicit modeling of communication costs. A natural way to relate network traffic with the communication cost associated to the links is to assign to each link a weighting cost coefficient $0 \leq \lambda_l \leq 1$. The value of the cost coefficient has to be determined by the manager according to the notion of cost associated to the link, and may be actually a combination of several factors. For instance, a high cost may be due to high latency or low-bandwidth on the link, or to the fact that a link connected to the NMS should be kept as unloaded as possible, or to security considerations.

The formulae derived so far are still valid in the case of management of a single LAN (Fig. 7.1.a). We approximate the LAN as a shared media or as a mesh of identical cost paths between any pair of nodes. The same coefficient λ is associated to each link, and the overall cost of performing a given management task is $K = \lambda T$, where T can be each of the expressions derived in the previous section.

A common case that involves differentiation of costs is management of a remote LAN, shown in Fig. 7.1.b. In this case, the managed devices are placed in a high-speed network whose links are characterized by the same coefficient λ. The LAN has an entry point, e.g., a router, to which the NMS is connected through a link[3] with different, and most likely higher, coefficient λ_0. Thus, the cost of reaching any device from the NMS is weighted by a coefficient $(\lambda_0 + \lambda)$. In this situation, the cost for any management task performed using a paradigm $p \in \{\text{CS}, \text{REV}, \text{COD}\}$ is

$$K_p = (\lambda_0 + \lambda)T_p.$$

On the other hand, for the MA paradigm the expression of the cost is actually a generalization of the traffic $T_{Ma,Mgm}$ computed in the previous section:

$$K_{MA} = \lambda_0 T_{MA,Mgm} + \lambda T_{MA}$$

In fact, traffic across the link connecting the NMS to, say, the router of the LAN is concerned only with the first trip of the agent and the final delivery of aggregated results, and thus has the same expression derived earlier. Traffic within the managed LAN is generated since the mobile agent is injected from the ingress router into the managed LAN. If we imagine to place a virtual NMS on the ingress router, we can easily see that the traffic generated within the LAN has the same expression derived for the general case of the MA paradigm. The traffic $T_{MA,mgm}$ computed earlier is a special case of the formula above, where $\lambda_0 = 1$ and $\lambda = 0$.

A more general case is management of an internetwork, i.e. a collection of LANs interconnected through internetworking devices—usually routers—which enables communication among stations on different LANs. In this setting, the

[3] Here, the term link must be considered in a broad sense. The NMS can be actually connected to the managed LAN through a sequence of physical links and nodes. In this case λ_0 is the weight associated to the whole path from the NMS to the managed LAN.

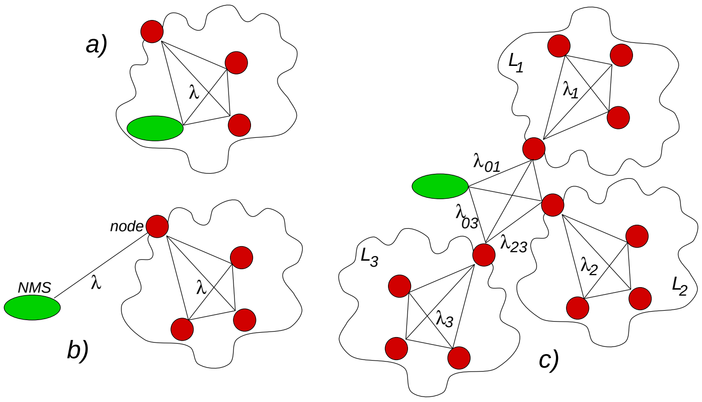


Figure 7.1: Different management configurations: a) local management of a single LAN, b) remote management of a LAN, and c) management of an internetwork.

NMS is connected through different links to each of the remote LANs (Fig. 7.1.c). In our model, the link connecting the NMS to the l^{th} of the L LANs is assigned a cost coefficient λ_{0l}; the link connecting the LANs l and m is assigned a coefficient λ_{lm}; each link within LAN l is assigned a coefficient λ_l. In this respect, remote management of a LAN can be seen as a special case of internetwork management where only one LAN must be managed. For the sake of simplicity, we assume that the cost coefficient is constant with respect to the direction of communication over the link, i.e. $\lambda_{lm} = \lambda_{ml}$. Under these assumptions, the overall cost for the CS paradigm is

$$K_{CS} = \sum_{l=1}^{L}\sum_{n=1}^{N}\sum_{q=1}^{Q}[(\lambda_{0l} + \lambda_l)(\eta_{CS} I_q + \tilde{\eta}_{CS} R_{lnq})]$$

where we assumed, in order to avoid cluttering the expression, that each LAN contains the same number N of managed nodes. The expressions for the cost of the REV, COD, and MA paradigms are derived similarly.

The cost of the MA paradigm, however, requires some additional elaboration. We define

$$S_{MA,ln} = S_{MA,l} + \hat{S}_{MA,ln}$$

as the state of the mobile agent while it is traveling towards node n in LAN l. The first term accounts for the state accumulated by the agent just before entering LAN l, while the second term accounts only for the state accumulated within the current LAN. Thus,

$$S_{MA,l} = \begin{cases} 0 & \text{if } l = 1 \\ \sum_{j=1}^{l-1}\sum_{n=1}^{N}\sum_{q=1}^{Q} R_{inq} & \text{otherwise} \end{cases}$$
$$\hat{S}_{MA,ln} = \sum_{i=1}^{n-1}\sum_{q=1}^{Q} R_{liq}$$

The overall expression of the cost for the MA paradigm is then

$$\begin{aligned} K_{MA} = \ & (\lambda_{01} + \lambda_1)\eta_{MA} C_{MA} + (\lambda_{0L} + \lambda_L)\tilde{\eta}_{MA} \textstyle\sum_{l,n,q=1}^{L,N,Q} R_{lnq} + \\ & \textstyle\sum_{l=2}^{L} \lambda_{l-1,l}\eta_{MA}(C_{MA} + S_{MA,l}) + \sum_{l=1}^{L}\sum_{n=2}^{N} \lambda_l \eta_{MA}(C_{MA} + S_{MA,ln}) \end{aligned}$$

Chapter 8

Selecting the Right Technology

In this chapter we focus on the implementation of a specific management task: collecting information about the load level of network interfaces. The goal is to show how to refine the model to cope with the details of the technologies used to implement design paradigms and how to use the model to actually determine the best design choice.

8.1 Refining the Model Towards Technology

We actually implemented the aforementioned management functionality by using SNMP as an implementation of the CS paradigm and Java Aglets [61] to implement the mobile code paradigms, because of their diffusion and level of support. For each implementation, the traffic generated has been measured and the overhead of each protocol layer down to the IP one has been isolated[1]. Measurements provided the information needed to calculate the value of the parameters of the traffic formulae presented in earlier sections.

SNMP requests and replies are encapsulated into UDP messages (Fig. 8.1). In our case study, requests and replies are short and are always carried within a single UDP message[2]—no fragmentation takes place. Hence, as described in Section 7.1, we can express the overhead function $\eta_{CS}(X) = \tilde{\eta}_{CS}(X) = \frac{\alpha_{CS}(X)}{X} + \beta_{CS}(X)$ by assuming $\alpha_{CS} = 60$ and $\beta_{CS} = 1$, independently of the values of I_q and R_{q_n}.

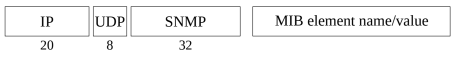


Figure 8.1: Encapsulation of SNMP requests and replies.

[1] The measurements have been performed over an Ethernet LAN which introduces an additional 26 byte overhead on each packet. Moreover, the overall traffic generated on the LAN is slightly larger than the aforementioned because Ethernet packets have a minimum size of 72 bytes.

[2] Some SNMP implementations may segment messages also in this case; we do not consider this here.

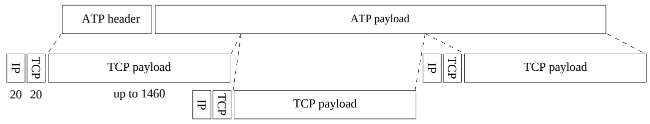


Figure 8.2: Segmentation of the ATP `dispatch`.

In Java Aglets, invocation of the `dispatch` method on an aglet triggers the migration of the classes describing its code and of the value of its attributes. Java Aglets supports only weak mobility, thus no execution state is retained across migration. In order to implement the MA paradigm, we added explicitly extra information to the aglet's state to obtain control of the execution flow after migration. Java Aglets communication facilities are built upon the Agent Transfer Protocol (ATP) which exploits the services offered by TCP. Upon migration, the code and state of an aglet are prepended by an ATP header of variable length. Reliable transfer is ensured by an acknowledgment ATP message sent by the receiver. To transmit the result to the NMS we used the message passing facility of Java Aglets, which is built upon ATP as well.

Sending an ATP message requires a TCP connection to be setup and subsequently torn down, i.e. 5 TCP messages[3]. As shown in Fig. 8.2, an ATP message is possibly segmented into TCP messages, whose maximum payload is 1460 bytes in our implementation. Each TCP message is prepended by a 20 byte TCP header and encapsulated into an IP packet that introduces a 20 byte header. A TCP message must be acknowledged by the receiver either explicitly or implicitly. We assume explicit acknowledgment and no traffic in the reverse direction of the TCP connection transporting the code, i.e. acknowledgment piggybacking is not used.

The traffic generated by an ATP message is given by

$$\alpha_{TCP} + \beta_{TCP}(H_{ATP} + C) + \beta_{TCP}\alpha_{ATP}$$

where α_{TCP} is the traffic generated for setting up and tearing down the TCP connection (200 bytes), α_{ATP} is the ATP acknowledgment message size (120 bytes), H_{ATP} is the ATP header size, C is the size of the ATP message payload—code and state in the case of aglet migration. β_{TCP} is the overhead introduced by the transmission and acknowledgment of TCP messages:

$$\beta_{TCP}(X) = \frac{2H_{TCP/IP}}{X}\left\lceil \frac{X}{p_{TCP}} \right\rceil + 1$$

where $H_{TCP/IP}$ is the size of the TCP and IP headers (40 bytes) present in data and acknowledgment TCP messages, and p_{TCP} is the size of the maximum TCP payload (1460 bytes). In our case study, since the same MCS is exploited for the implementation of all the three mobile code paradigms, and replies are sent using the same ATP facilities used for code migration, the overhead function $\eta_p = \tilde{\eta}_p = \eta$ is independent of the paradigm:

$$\begin{aligned} \alpha(X) = \quad & \alpha_{TCP} + \left(\tfrac{2H_{TCP/IP}}{H_{ATP}+X}\left\lceil \tfrac{H_{ATP}+X}{p_{TCP}} \right\rceil + 1\right) H_{ATP} \\ & + \left(\tfrac{2H_{TCP/IP}}{\alpha_{ATP}} + 1\right)\alpha_{ATP} \\ \beta(X) = \quad & \tfrac{2H_{TCP/IP}}{H_{ATP}+X}\left\lceil \tfrac{H_{ATP}+X}{p_{TCP}} \right\rceil + 1 \end{aligned} \qquad (8.1)$$

[3]The equations derived in this section assume that no message is lost in the network.

At this point, the model has been refined to take into account the details of the technology selected for the implementation. We can now evaluate the different architectures. However, this needs some information about the topology of the network and the characteristics of the task to be performed. We assume that the managed network is composed of $N = 50$ nodes, and we temporarily assume a uniform network. In our CS implementation, the load level of an interface can be obtained through an SNMP query which retrieves 5 MIB variables; assuming that each device has 30 interfaces, $Q = 30$. From our data, each SNMP request and reply is 48 and 66 byte long, respectively. Moreover, we assume that each query is sent in a separate request message. Substituting these values and $\eta_{CS} = 2.01$ in (7.1), the traffic generated by the SNMP implementation of the management task is $T_{CS} = 335.6$ Kbytes.

The same management task can be implemented with the REV paradigm. The Java bytecode performing the Q queries on each node is $C_{REV} = 5.6$ Kbytes and is sent on each node prepended by a header $H_{ATP} = 120$ bytes, and augmented by the overhead determined by (8.1), which yields $\eta(C_{REV}) = 1.32$ and, under similar conditions, $\eta(Q\bar{R}) = 2.91$. According to (7.2) we obtain that $T_{REV} = 437.1$ Kbytes. Thus, as far as global generated traffic is concerned, it is more convenient to implement the management task according to the CS paradigm. This can be inferred using (7.6) obtaining that REV is convenient over CS if $Q > 87$. However, this is not a general result and depends on the particular technology chosen. In particular, using ATP rather than directly TCP connections to send replies introduces a huge, unnecessary overhead. Furthermore, Java Aglets are probably overshooting for the implementation of a REV paradigm. For this task, scripting languages are probably much more efficient as witnessed by earlier work on management by delegation, because they provide a more compact code representation. Ongoing experimental work will provide better insight on these technological issues.

If the management task is repeated $U = 20$ times, the traffic generated by a COD implementation is $T_{COD} = 2.6$ Mbytes, as given by (7.4) and (7.5) where $C_{COD} = 5.1$ Kbytes and $\eta(\bar{I}) = 58.3$. In this case the COD design is more convenient than the CS one, which generates 6.6 Mbytes. Using (7.8) it is possible to foresee that the COD approach is convenient if the management task is supposed to be performed at least $U = 4$ times.

As we discussed earlier, an MA design reduces the traffic involving the NMS. In fact, according to (7.9) $T_{MA,Mgm} = 34.3$ Kbytes, with a code size for the aglet of 6.6 Kbytes. This can be further improved leveraging on semantic compression. We consider now a different management task: collecting load on the most loaded interface on each device. This task enables local semantic compression and, according to (7.10), $T'_{MA,Mgm} = 10$ Kbytes, being $\eta(N\bar{R}) = 2.9$. Moreover, if global compression is possible, for instance when the most loaded interface in the whole network has to be searched for, the traffic is reduced to $T''_{MA,Mgm} = 9.6$ Kbytes, as derived from (7.11) with $\eta(\bar{R}) = 65.9$. Again, in this case the difference in the traffic generated with local and global compression is small because the fixed overhead component in the transmission of the results to the NMS is large, due to the particular technology chosen.

As an example of application of our model of costs, let us consider that the most loaded interface has to be found in an internetwork of $L = 15$ LANs, each containing $N = 5$ nodes. A reasonable value for the cost associated to each

LAN is $\lambda_{LAN} = 0.01$, with a cost associated to WAN links of $\lambda_{WAN} = 1$. Under these assumptions, the cost of performing the management task using SNMP is given by $K_{SNMP} = 508.4$ Kbytes, while the cost with an implementation based on the MA paradigm is $K_{MA} = 130.3$ Kbytes. The movement of the agent among the managed devices generates a considerable amount of traffic, as we described earlier; nevertheless, these movements are mostly inside high bandwidth LANs and only a small percentage of them involve wide area links and the NMS. This confirms that the MA paradigm can be cost effective for the management of internetworks.

8.2 Discussion

In principle, design paradigms and the technology used for their implementation are orthogonal, as discussed at the end of Chapter 3. Nevertheless, we have already pointed out that this is true only partially, and that an inappropriate technology may put an unnecessary burden on the programmer—at least as far as code mobility is concerned. In particular, we have shown how a strong MCS is the natural choice for implementing an MA design. Mobility is reduced to a single instruction, and the migrating EU can be mapped directly to a roaming component in the higher-level design. Conversely, a weak MCS constrains the programmer to manage explicitly the execution state, which degradates programmer productivity, program readability, and ease of debugging. In the context of network management, however, there is an additional drawback. The formulae derived so far show how the size of the code transferred is a key parameter in the expressions of network traffic. Implementing an MA design with a weak MCS is likely to end up in creating bigger code (because of the explicit management of execution state), thus reducing the benefits potentially achievable.

Nevertheless, the final choice might be influenced by other considerations as well. The analysis described in the previous chapters aims at identifying the best paradigm to design a single functionality within an application. Of course, an application is composed of several functionalities, each with its own peculiarities that may lead to completely different designs. For instance, suppose we want to implement a network management application that provides, among the others, a first functionality to determine the most loaded interface on a given path, a second functionality that determines all the parameters for a given interface, and a third one that allows the manager to set a given value in a device's MIB. The analysis carried out earlier tells us that in the first case we may want to take advantage of the opportunity to perform global semantic compression, and exploit the MA paradigm; in the second, we may want to follow the REV paradigm, in order to save bandwidth—MA is overshooting. In the third one, CS will suffice.

The choice of the technology used to implement the application must take into account not only which is the best technology to implement a given functionality designed following a certain design paradigm, but also how the technology fits the global application development. For example, let us suppose that we are faced with the choice between a strong MCS which does not provide good support for stand-alone code shipping and a weak MCL which provides it. In the context depicted above, the first functionality is likely to be used less frequently than the others: in this case, we may want to sacrifice the traffic optimization achievable with the strong MCS and use the weak one, to obtain better support in the key functionalities and keep the uniformity of the development tools.

Final Remarks on Evaluating Code Mobility

In this part of the thesis we discovered that in the selected application domain and with the particular goal of optimizing network traffic, the design tradeoffs depend on the characteristics of the network being managed (costs, number of nodes, protocols) and of the management task (possibility of local/global semantic compression, expected frequency of invocation, complexity of the task, dimension of replies). The characteristics of the technology actually used for the implementation also affect these tradeoffs according to the management protocols (overhead) and the MCS (expressiveness of the language, formats used for transfer, overhead) used. Hence, determining when to use a mobile code design paradigm in place of a traditional client-server architecture requires *i)* a model of the management functionality to be implemented, together with information about the managed network; *ii)* a precise quantitative characterization of the management protocols and the MCS to be used for the implementation. This work is a partial answer to the first point. In fact, it provides a general framework that is conceived for being tailored and customized by the manager according to the peculiarities of her needs, as exemplified in Chapter 8.

Quantitative information about the performance and the overhead introduced by MCSs is still missing in literature. However, this measurement and modeling activity is done once and for all, and can guide a highly effective design of management tasks, as shown throughout the chapter.

A question can be raised about whether similar considerations can be drawn in different application domains. Only experimentation can provide this answer. However, the use of a model seems to be a general and effective technique for evaluating the benefits of mobile code, especially if evaluation of aspects concerned with network traffic is concerned.

Our long term work on the theme of this chapter envisions a scenario where the manager is provided with a software platform which guides the "management life cycle" as describe above. Management operations could be specified by "packing" sets of management instructions into reusable library components—possibly reusing off-the-shelf components. The same complex functions could be embedded in different components implementing different design paradigms. This platform would provide tools to derive customized models of the managed networks from general models as we exemplified, and to visualize and simulate the performance of the management operations according to different design and implementation choices. Ongoing work is headed towards the realization of such a platform, the refinement of our model, and a quantitative characterization of existing MCS.

Part III

Formalizing Code Mobility

Chapter 9

Motivations, Background, and Approach

By and large, the developments described so far in the research field concerned with code mobility fall outside the traditional concerns of distributed computing since much of the existing work on models, algorithms, proof systems, methodologies, and impossibility has been carried out assuming networks with a fixed topology and static binding between the application code and the hosts where it is being executed. In contrast, MCSs enable more dynamic solutions to distributed computing problems, such as design paradigms that encompass new forms of interaction among the components of a distributed application. Relating these new paradigms to previous research on distributed computing is the main theme of this part of the thesis. The focus is on the modeling of programs that exhibit code mobility and on the construction of assertional proofs for them. The approach we pursue is that of expressing various forms of code mobility in a model that was originally designed for the specification and verification of mobile computing systems, i.e. systems which involve software components bound to hosts that move in space and interact opportunistically when communication becomes possible. We believe this to be an important first step towards constructing formal models specific to code mobility as it highlights the differences and similarities between the two kinds of mobility. Furthermore, these kinds of studies may eventually lead to the development of a unified model of mobility.

The model we use in our study is called *Mobile UNITY* [92, 70, 69], an extension of work by Chandy and Misra on UNITY [25]. Mobile UNITY provides a programming notation that captures the notion of mobility and transient interactions among mobile nodes and includes an assertional-style proof logic. The model adheres to the minimalist philosophy of the original UNITY, supports text-based reasoning about programs, and focuses only on essential abstractions needed to cope with the presence of mobility. As we use this model to examine mobile code paradigms, the fundamental goal is to determine whether Mobile UNITY by itself is adequate to this modeling task.

In Mobile UNITY, the unit of mobility is a program. Migration is captured by augmenting the program state with a location attribute whose change in value is used to represent motion. In this manner, UNITY is augmented with an explicit representation of space and its properties—we contend that modeling the space explicitly is desirable when one hopes to take into account the physical reality of moving hosts and its implications on the behavior of the software they carry. While Mobile UNITY is usually concerned with movement in physical space, there is nothing in the model that precludes a more abstract view of space. Nodes of a network or even distinct memory partitions on a single node can be viewed as locations in a domain whose properties are distinct from those of the physical space, e.g., one in which all points are equidistant—a reasonable assumption for a model which ignores variations in communication delays. In the remainder of this part of thesis we show how one can treat mobile code fragments as Mobile UNITY programs, how one can specify code movement from one location to another by manipulating the location variable, and how one can capture dynamic binding of variables among the mobile and stationary code by means of transient interactions, a concept specific to Mobile UNITY, following the lines of [87]. Having accomplished this, we also show how the Mobile UNITY proof logic enables us to verify the correctness of programs using mobile code.

Until very recently the only formal model of concurrency to refer to mobility explicitly was π-calculus [74], a process algebra proposed by Milner and his colleagues. In π-calculus there is no formal concept of space. Mobility is equated to the ability to "express processes which have changing structure". Under this definition, any model able to pass processes as values, e.g. the Actor model, or link names as values (π-calculus) qualifies. This fact has been recognized, and a growing number of researchers [36, 32, 4] is concentrating on building variants and extensions of the π-calculus or in general process algebras providing a better notion of location. In particular, the work of Cardelli [20] seems directly inspired by the domain of mobile code, as its concept of nested localities whose access is regulated by capabilities associated to mobile processes closely recalls the Telescript model. The approach pursued by researchers in process algebra is clearly distinct from the modeling strategy adopted for Mobile UNITY and may offer yet another perspective on code mobility. An evaluation of π-calculus and its derivatives, however, is outside the scope of this thesis.

The remainder of this part of the thesis is structured as follows. Chapter 10 describes a simplified version of a distributed simulation problem and introduces standard UNITY via a centralized solution. Chapter 11 presents a first distributed solution to the problem using the client-server paradigm. In doing so, it also provides the reader with a gentle introduction to Mobile UNITY and its proof logic. Mobile code solutions to the same problem illustrate our strategies for modeling mobile code in Mobile UNITY. These solutions and their associated proof outlines appear in Chapter 12. Section 12.4 at the end of that chapter explores some of the issues raised by this investigation and discusses the kinds of features that may be needed in a richer model of mobility.

The work presented in this part of the thesis has been published in [87, 88].

Chapter 10

A Distributed Simulation Problem

In this chapter we present an example that will be used for illustration purposes throughout the remainder of this part of the thesis. We use it first as a vehicle to introduce the reader to the UNITY notation. The example is inspired by the work of Chandy and Misra who provided a formal characterization and solution for a distributed simulation problem [24]. The basic idea is to simulate the behavior of a physical system such as an electronic circuit on a network of computing nodes which communicate asynchronously and in the absence of global shared memory. Physical entities are allocated to nodes across the network and simulated according to their expected behavior. The nodes must communicate among themselves in order to simulate the interactions normally occuring among the physical components (e.g., passing a signal from one gate to the next) and also in order to preserve the correct temporal relationships among the actions associated with the various simulated entities. It is the latter aspect of the problem which is central to its solution. For this reason, in our simplified version we focus strictly on the temporal aspects of the problem and ignore any other interactions among the components. In other words, we assume that each simulated entity executes at most one action at a time in a deterministic manner and does not interact with any other entities being simulated at other nodes. However, because the simulation may be monitored by some external agent while in progress, the ordering of actions in time must be consistent with those occuring in the simulated system. These simplifying assumptions would be realistic, for instance, when particle movement is simulated in the absence of collisions.

The behavior of each node is very simple. A *local timer* holds the time value at which the next local action is to be executed. The action can be executed only when all nodes participating in the simulation reach that particular time, i.e. all actions scheduled for earlier times have been executed. The notion of *global virtual time (GVT)*, whose value is defined as the minimum among the values of all local timers, formalizes the intuitive idea that the simulation reached a particular point in time. In a centralized solution to the problem, such as the UNITY solution appearing later in this chapter, the value of the

GVT can be stored in a variable and can be updated by examining the value of each local timer. In a distributed solution, each node has to discover the GVT value by communicating with other participating nodes. Once GVT catches up with the local timer, the corresponding action is executed and the local timer is incremented to reflect the time when the subsequent action is scheduled to take place.

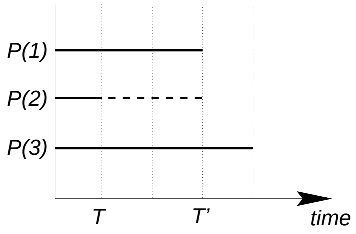


Figure 10.1: A snapshot of the distributed simulation.

In Figure 10.1 we show three processes P indexed by i which participate in a distributed simulation. Solid lines depict values of the corresponding local timers. The current GVT is shown below the horizontal axis and marked by the symbol T. The only process allowed to execute an action is $P(2)$. After the execution of its action, $P(2)$ is allowed to update the value of its local timer to a new value referring to some time in the future. As a result, in this example, a new GVT value T' is established and $P(1)$ and $P(2)$ are now allowed to execute their respective actions, independently of each other. In the remainder of this chapter, we define the problem formally, show a centralized solution for it, and state some properties that will be used to prove the correctness of the distributed solutions shown in Chapters 11 and 12.

10.1 Problem Formalization

A system is modeled by a set of N processes, indexed from 0 to $(N-1)$. Each process $P(i)$ has an associated local timer, $t(i)$. The GVT T is defined as the minimum of all the local timers, i.e. $T = \langle \min i : 0 \leq i \leq N-1 :: t(i) \rangle$[1]. The scheduling criterion used by a process to update its timer is embodied in the definitions for functions f and g, both indexed by a specific process identifier:

$$f_i(t(i),T,z) = \begin{cases} t(i) & \text{if } t(i) > T, \\ g_i(t(i),z) & \text{if } t(i) = T, \\ \perp & \text{if } t(i) < T. \end{cases} \tag{10.1}$$

$$g_i(t(i),z) > t(i) \tag{10.2}$$

[1]The three-part notation $\langle \mathbf{op}\, quantified\_variables : range :: expression \rangle$ is borrowed from UNITY and will be used throughout the paper. It is defined as follows: The variables from *quantified_variables* take on all possible values permitted by *range*. If *range* is missing, the first colon is omitted and the domain of the variables is restricted by context. Each such instantiation of the variables is substituted in *expression* producing a multiset of values in which **op** is applied. In the case above, it is equivalent to computing the minimum across the N values.

where $t(i)$ denotes the value for a local timer, T denotes the GVT value, and z identifies the simulation mode. These definitions capture the following requirements:

1. The local timer cannot change if it is ahead of the GVT.

2. The local timer, if permitted, can only increase, i.e. actions are always scheduled in the future (see the constraint on g).

3. The value of a local timer can never be behind the GVT. For such cases the function f is undefined.

10.2 A Centralized UNITY Solution

In this section, we discuss a solution for the distributed simulation problem using the UNITY notation described in [25] and use the associated logic to prove several key properties of the problem. The UNITY program appears in Figure 10.2. All variable declarations appear in the **declare** section. The array t contains the values of the local timers for each process i. T stores the current value for the GVT and z represents the simulation mode. The **initially** section contains a set of predicates, separated by the symbol $[]$, which define the allowed set of initial states. Uninitialized variables assume arbitrary values belonging to their declared type. All local timers $t(i)$ are initialized to zero and T is initialized consistently. The simulation mode z is initialized to some default initial value. The **assign** section is the core of the program. It consists of a set of assignment statements that specify the program behavior. Program execution starts in the state described in the **initially** section and evolves as a non-deterministic, fair interleaving of statements—in an infinite execution of the program each statement is executed infinitely often. The first statement computes the current GVT as the minimum among the values of the local timers, and stores it in T. The second statement is a set of asynchronous assignments each updating the local timer for a corresponding process $P(i)$. Due to the definition of f, the update is performed only when the timer value is equal to the value of T, and has no effect on T. The statement is defined using the three-part notation with $[]$ as a quantifier, hence it is equivalent to N assignments separated by $[]$ and executed non-deterministically and independently. If we wanted to specify synchronous execution instead, we could have used $||$ in place of $[]$. Note also that, because of fair execution, each of the assignments separated by $[]$ is non-deterministically interleaved with the one computing the GVT. In some fair execution, it could happen for each $t(i)$ to be updated before the new GVT is computed. In this case, T actually represents a lower bound for the GVT value, and the T parameter used as an argument for function f is actually an approximation of the GVT. In this solution, we abstract them into a function $SimulationWork(t)$, which is dropped in the other solutions. Finally, the value of the simulation mode is updated dynamically according to the definition of a function $d(T)$ whose details are left out.

Program properties may be stated formally and may be verified by using the UNITY proof logic. A specialization of temporal logic, the UNITY proof logic allows one to verify the correctness of a program directly from its text in

```
Program DistributedSimulation
    declare
        t : array of integer [] T,z : integer
    initially
        ⟨[] i :: t(i) = 0⟩ [] T = 0
    assign
        SimulationWork(t)
      [] T := ⟨min i :: t(i)⟩
      [] ⟨[] i :: t(i) := f_i(t(i),T,z)⟩
      [] z := d(T)
end
```

Figure 10.2: A simple UNITY solution for the distributed simulation problem.

the tradition of sequential programming. Progress properties are captured by a predicate relation called **leads-to** (written $\mapsto$). For instance,

$$t(i) = \langle \min j :: t(j) \rangle = \alpha \ \mapsto \ t(i) > \alpha$$

states that from a state in which the local timer of a process is the minimum of all the local timers, the program eventually enters a state where that local timer is advanced. (By convention, all free variables are assumed to be universally quantified.)

A **leads-to** property is most often derived by relying on the transitivity of this relation or on a primitive form of **leads-to** called **ensures**. In the UNITY program, for instance, one can show that

$$t(i) = \langle \min j :: t(j) \rangle = \alpha \ \textbf{ensures} \ T = \alpha \wedge t(i) = \alpha$$

This can be verified directly from the program text by observing that, for as long as the right hand side (RHS) of the **ensures** relation is false:

1. All program statements either leave the left hand side (LHS) of the **ensures** unchanged because they do not update T or they actually update T with the correct value. (Technically this expresses a safety property called **unless** in UNITY.)

2. There is a statement which, if selected, updates T to the correct value in a single atomic step. Fairness guarantees that this statement is eventually selected for execution.

To summarize, the three UNITY inference rules we used so far are:

$$\frac{p \ \textbf{ensures} \ q}{p \ \mapsto \ q}, \tag{$\mathcal{T}_1$}$$

$$\frac{p \ \textbf{unless} \ q, \ \langle \exists \, s :: \{p\} \ s \ \{q\} \rangle}{p \ \textbf{ensures} \ q}, \tag{$\mathcal{T}_2$}$$

and

$$\frac{\langle \forall \, s :: \{p \ \wedge \neg q\} \ s \ \{p \vee q\} \rangle}{p \ \textbf{unless} \ q}, \tag{$\mathcal{T}_3$}$$

where $\{p\} \ s \ \{q\}$ is the Hoare triple which can be proven using the standard assignment axiom and pure logic. The **unless** and **leads-to** relations can be combined in an **until** relation,

$$p \ \textbf{until} \ q \equiv (p \ \textbf{unless} \ q) \wedge (p \ \mapsto \ q), \tag{$\mathcal{T}_4$}$$

meaning that p holds as long as q does not, and that q holds eventually[2]. The correctness criteria for the distributed simulation problem are conveniently expressed as two **until** properties:

PROPERTY 1 $t(i) = \langle \min j :: t(j) \rangle = \alpha$ **until** $t(i) > \alpha$.
A local timer increases only at the point when its value equals the GVT.

PROPERTY 2 $t(i) = \alpha$ **until** $t(i) = \langle \min j :: t(j) \rangle = \alpha$.
The GVT eventually catches up with with the local timer, i.e. the simulation makes continuous progress.

The UNITY logic includes many other useful predicate relations and inference rules. We prefer to introduce them only when they are really needed. Some of them are needed immediately in verifying Property 2. First, is the fact that implication is a special case of a **leads-to**:

$$\frac{p \Rightarrow q}{p \mapsto q} \qquad (\mathcal{T}_5)$$

Second, is the definition of an invariant property. A predicate p is called an invariant (written **inv.**) if it holds in the initial state and forever throughout the execution of the program:

$$\frac{\mathit{INIT} \Rightarrow p,\ p \ \textbf{unless}\ \mathit{false}}{\textbf{inv.}\ p} \qquad (\mathcal{T}_6)$$

where *INIT* is a predicate which characterizes the set of acceptable initial states. As an example, we can show that

$$\textbf{inv.}\ t(i) \geq T$$

for every i. This is true initially, since both $t(i)$ and T are initialized to zero. Moreover, the execution of any statement in a state in which this property holds preserves the property—local timers can only advance past T and any updates to T guarantee that it acquires a value that is the minimum across all local timers. In addition, we apply the following *induction principle* for **leads-to**. If W is a set well-founded under the relation $\prec$, and M is a *metric* function mapping program states to W, then

$$\frac{\langle \forall m : m \in W :: p \wedge M = m \mapsto (p \wedge M \prec m) \vee q \rangle}{p \mapsto q} \qquad (\mathcal{T}_7)$$

holds. The hypotesis states that from any state where p holds, program execution eventually reaches a state where either q holds or the metric M is decreased and p holds. From the well-foundedness of W, M cannot decrease indefinitely, which leads to the conclusion that eventually a state is reached where q holds. Finally, we will use the *disjunction rule* of **leads-to**

$$\text{for any set } W,\ \frac{\langle \forall m : m \in W :: p(m) \mapsto q \rangle}{\langle \exists m : m \in W :: p(m) \rangle \mapsto q} \qquad (\mathcal{T}_8)$$

[2]The **ensures** differs from **until**, in that it requires a specific statement to establish the RHS as soon as it is scheduled for execution.

which has the simpler consequence

$$\frac{p \mapsto q,\ p' \mapsto q}{p \vee p' \mapsto q}$$

and allows to prove **leads-to** properties by case analysis.

Armed with this knowledge, we can prove the properties stated above. Proof of each **until** relation involves proving its **unless** and **leads-to** parts.

PROOF OF PROPERTY 1. We prove separately:

1. $t(i) = \langle \min j :: t(j) \rangle = \alpha$ **unless** $t(i) > \alpha$ can be proven by observing that, due to the definition of f and g, the set of asynchronous statements updating $t(i)$ always increase the timer value, while the other statements always leave $t(i)$ and the minimum unchanged. Hence, it is proven by application of $(\mathcal{T}_3)$.
2. $t(i) = \langle \min j :: t(j) \rangle = \alpha \ \mapsto\ t(i) > \alpha$ can be split by $(\mathcal{T}_1)$ and transitivity of **leads-to** into

$$\begin{array}{r} t(i) = \langle \min j :: t(j) \rangle = \alpha \ \textbf{ensures}\ T = \alpha \wedge t(i) = \alpha, \text{ and} \\ T = \alpha \wedge t(i) = \alpha \ \textbf{ensures}\ t(i) > \alpha. \end{array}$$

   The first obligation has been proven earlier. The second one is proven by arguments similar to the ones used to prove the **unless** above and observing that there exists one statement that increases $t(i)$, and fairness guarantees that eventually it is selected.

PROOF OF PROPERTY 2. Again, we prove separately:

1. $t(i) = \alpha$ **unless** $t(i) = \langle \min j :: t(j) \rangle = \alpha$ is proven observing that, after the execution of each statement in the program, $t(i)$ can either be the minimum or not.
2. $t(i) = \alpha \ \mapsto\ t(i) = \langle \min j :: t(j) \rangle = \alpha$ is proven by considering two cases, according to the value of $t(i)$:
   (a) $t(i) = \alpha \wedge t(i) = \langle \min j :: t(j) \rangle \ \mapsto\ t(i) = \langle \min j :: t(j) \rangle = \alpha$, which holds by virtue of the fact that

$$t(i) = \alpha \wedge t(i) = \langle \min j :: t(j) \rangle \Rightarrow t(i) = \langle \min j :: t(j) \rangle = \alpha,$$

   followed by application of $(\mathcal{T}_5)$.
   (b) $t(i) = \alpha \wedge t(i) \neq \langle \min j :: t(j) \rangle \mapsto t(i) = \langle \min j :: t(j) \rangle = \alpha$. We use the induction principle of **leads-to** stated in $(\mathcal{T}_7)$, where W is the set of natural numbers $\mathbb{N}$, $\prec$ is the $<$ relation, and M is the number of local timers that are behind $t(i)$, i.e. $M = \langle \sharp j :: t(j) < \alpha \rangle$ with $M \geq 0$ by construction[3]. Thus, we prove that

$$\begin{array}{rl} & t(i) = \alpha \wedge t(i) \neq \langle \min j :: t(j) \rangle \wedge M = m \\ \mapsto & (t(i) = \alpha \wedge t(i) \neq \langle \min j :: t(j) \rangle \wedge M < m) \vee \\ & t(i) = \langle \min j :: t(j) \rangle = \alpha. \end{array}$$

   We observe that, since $t(i)$ is not the minimum, some other timer must be the minimum and, because of Property 1, it eventually

[3]The operator $\sharp$ used in the three part notation is used to count the number of elements in the domain of j for which the provided expression holds.

increases, decreasing the value of M. Furthermore, $t(i)$ cannot increase unless it is the minimum, as proven earlier. Eventually no timer is behind $t(i)$, i.e. $t(i)$ holds the minimum value of all timers.

Application of the disjunction rule ($\mathcal{T}_8$) completes the proof.

Although the solution presented in Figure 10.2 is formally correct, it is not acceptable from a design point of view because every variable appears to be *shared* and, in particular, the GVT is shared among all processes—which has been explicitly forbidden by the statement of the problem. In addition, it is not apparent that the local timers are associated with processes, and the program text does not even capture explicitly the notion of independent processes. In the following chapter we show a client-server solution that does not employ shared variables and makes explicit the location and encapsulation embodied in each process. In doing this, we will introduce Mobile UNITY, which will be used in Chapter 12 to describe distributed solutions which exploit code mobility.

Chapter 11

A Client-Server Solution in Mobile UNITY

In a later chapter of this part of the thesis we give three alternative solutions to the distributed simulation problem, each modeled after a single mobile code design paradigm. Each will be expressed in Mobile UNITY [69], a modification of the standard UNITY notation presented in Section 10.2. Mobile UNITY offers novel abstractions designed to deal specifically with distributed systems that contain mobile components, as well as an associated proof logic for reasoning about the behavior of such systems. In this chapter we make use of Mobile UNITY to present our first non-centralized solution to the distributed simulation problem—one that embodies the client-server paradigm.

Although the solution we present does not contain mobile components, it will serve to introduce the reader to the new constructs available in Mobile UNITY, such as the system structuring conventions and simple interaction clauses. After the presentation of the example we discuss other features of Mobile UNITY not exhibited by the text of the client-server solution, such as transiently shared variables.

11.1 System Definition

As in our earlier centralized solution, we ignore the internal simulation steps except for their effect on the advancement of the local timers. Each process must calculate an estimate T of the GVT to determine whether the next step in its local simulation is allowed. For correctness, this estimate should never exceed the real GVT, otherwise a process might take a step even when its timer value is in the future with respect to some other component. The distributed solutions presented in this and the following sections must compute a lower bound on the GVT without using statements like $T := \langle \mathsf{min}\, i :: t(i) \rangle$ which imply centralized access to the local timers of each component. The client-server solution of Figure 11.1, graphically represented in Figure 11.2, provides for such distribution by breaking up the system into a single server and a set of clients. The server contains an array τ which attempts to maintain the global state of all the local timer values from each of the clients. This array is updated

```
System DSClientServer
    Program P(i) at λ
        declare
            t,z : integer [] T : integer ∪ {⊥} [] RQ : request ∪ {⊥}
        initially
            t = 0 [] T = ⊥ [] λ = Location(i) [] RQ = ⊥
        assign
            [] t,T := f_i(t,T,z),⊥                    if def(T)
            [] RQ := ⟨SERVER,CS,MINSERV,t⟩             if ¬def(RQ) ∧ ¬def(T)
            [] T,RQ := RQ ↑ 4,⊥                        if RQ ↑ 1 = i
    end
    Program Server at λ
        declare
            T : integer ∪ {⊥} [] τ : array of integer [] q : array of (request ∪ {⊥})
        initially
            T = ⊥ [] ⟨|| j :: τ(j) = 0⟩ [] ⟨|| j :: q(j) = ⊥⟩ [] λ = Location(SERVER)
        assign
            ⟨[] j :: τ(j),q(j) ↑ 2 := q(j) ↑ 4,WAIT       if q(j) ↑ 1 = SERVER ∧ q(j) ↑ 2 ≠ WAIT⟩
            [] T := ⟨min k :: τ(k)⟩                         if ⟨∃ j :: q(j) ↑ 2 = WAIT⟩ ∧ ¬def(T)
            [] ⟨|| j :: q(j),T := ⟨j,⊥,⊥,T⟩,⊥               if def(T) ∧ q(j) ↑ 2 = WAIT⟩
    end
    Components
            ⟨[] i :: P(i)⟩ [] Server
    Interactions
            Server.q(i) := P(i).RQ                         when ¬def(Server.q(i)) ∧
                                                                serviceRequest(CS,MINSERV,i)
        [] P(i).RQ,Server.q(i) := Server.q(i),⊥            when serviceReady(i)
end
```

Figure 11.1: Client-Server solution for the distributed simulation problem.

via asynchronous message passing, and therefore may sometimes contain old values of the local timers. A new estimate for the GVT is calculated at the server and returned to waiting clients again via asynchronous message passing.

Figure 11.1 is illustrative of the new structuring conventions provided by Mobile UNITY. The first line provides the system name, *DSClientServer*. Next are a set of program definitions, the first of which, *P(i)*, serves as the code executed by each client and is parameterized by a single index representing the client number. The second, *Server*, has no such parameter. The program definitions are treated as types that are instantiated in the **Components** section. The program instances listed there are considered to be the running components of a distributed computation. In this example, the **Components** section instantiates one client for every value of i in the appropriate range, and a singleton *Server* instance. Each component has a distinct name: the clients, because they are indexed, and the server, because it is instantiated only once.

Note that each program definition begins with a line like **Program** *name* **at** λ. This denotes that each program exists at a specific physical or logical location denoted by the distinguished program variable λ. Each program contains a predicate in its **initially** section that gives the initial position of the component. We assume the existence of a function Location() that returns the initial position of each component based on an address which is either a client number or the constant SERVER. Each program may also contain code in its **assign** section that reads or modifies the position of the component. An assignment to λ models actual migration of the component through some physical or logical address space. Throughout this chapter, we leave the type of λ unspecified. For many problems, a simple discrete domain that reflects the connectivity among components will suffice, for example, a single bit that denotes whether a mobile

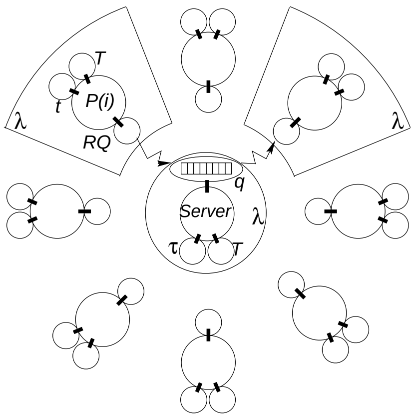


Figure 11.2: In the Client-Server solution, communication between the server and the clients takes place via asynchronous message passing.

host is within broadcast range of a fixed router. For other problems, the location type may be more complex and may exhibit much more structure. The type may be determined by the characteristics of a particular problem domain or may be implied by the way λ is used, e.g., how and whether it is incremented or checked for equality, but it is not necessary to completely specify this type in order to carry out useful reasoning about a system of mobile components. We make much more use of λ in later examples that, in contrast to *DSClientServer*, actually do contain mobile components, and where the connectivity among components depends on their dynamically changing locations.

In standard UNITY, there is no notion of changing connectivity among the programs making up a composed system, and two variables with the same name in different programs are considered shared throughout execution of the system. All variables of a Mobile UNITY component are considered local to that component. When dealing with a collection of instantiated components, a specific instance variable is referenced by prepending the name of the variable with the name of the program in which it appears. For example, the variables t and z of the program $P(1)$ have the fully qualified names $P(1).t$, and $P(1).z$. These fully qualified names should be used when carrying out formal reasoning about the behavior of the system.

Because variables are local, no communication can take place among components without the presence of interaction clauses spanning the scope of components. These appear in the **Interactions** section, and serve to provide implicit communication and coordination among the components. The two interaction clauses given in Figure 11.1 allow for communication between each client and the server. We assume that the index i is instantiated over the appropriate range. Note that these clauses look like ordinary UNITY assignment statements, except that they use the keyword **when** in place of the keyword **if**. Also, because they are not internal to some component, they may reference variables of any component, using the naming conventions given above. For example, the first interaction clause provides asynchronous message transfer from the client message buffer $P(i).RQ$ to the server message buffer $Server.q(i)$, when the server buffer is empty and there is a valid request message waiting in the client buffer:

$$Server.q(i) := P(i).RQ \qquad \textbf{when } \neg\mathsf{def}(Server.q(i)) \wedge \mathsf{serviceRequest}(\text{CS},\text{MINSERV},i)$$

The second interaction transfers a reply back to the client and empties the server buffer, when the reply is ready:

$$P(i).RQ,Server.q(i) := Server.q(i),\perp \qquad \textbf{when } \mathsf{serviceReady}(i)$$

Semantically, these two statements are treated like ordinary assignment statements, and we assume that execution consists of a fair interleaving of all assignment statements of each component as well as these "extra statements" in the **Interactions** section. As an aside, in the second interaction the symbol $\perp$ is used to denote that the buffer is empty, and consequently the value of $P(i).RQ$ undefined.

The meaning of the macros used in the guards of the interactions is shown in Figure 11.3. These macros are used throughout the remaining examples and have the only goal to improve readability. A request is a tuple consisting

clientAddress $= \langle \text{set } n : 0 \leq n \leq N-1 :: n \rangle$
address $= \{\text{SERVER}\} \cup$ clientAddress
opName $= \{\text{CS},\text{REV},\text{COD}\}$
opStatus $= \{\text{WAIT}\}$
request $= \langle$address, opName $\cup$ opStatus $\cup$ $\{\perp\}$, serviceName $\cup$ $\{\perp\}$, integer$\rangle$
serviceRequest(x : opName, y : serviceName, i : clientAddress) $\equiv$
$P(i).RQ \uparrow 1 = \text{SERVER} \wedge P(i).RQ \uparrow 2 = x \wedge P(i).RQ \uparrow 3 = y$
serviceReady(i : clientAddress) $\equiv Server.q(i) \uparrow 1 = i$

Figure 11.3: Macro definitions used in the example systems. Allowed values for each data element are represented as sets.

of four elements. We denote the tuple by enclosing it in angle brackets and separating its four elements by commas. We use the projection operator $\uparrow$ to access individual fields of a tuple, e.g., $P(i).RQ \uparrow 1$ represents the address field of the request variable $P(i).RQ$. This field is used to denote the destination of the message, and must be either a client index or the constant SERVER. The second field is used to denote the paradigm used to deliver the service, and must be an opName, which is one of CS for client-server, REV for remote evaluation, or COD for code on demand. This field may also represents the status of the request with an opStatus, if the server is in the process of constructing a reply. The third field denotes the specific service requested, which in the case of the client-server solution is specified by the client to be MINSERV, which indicates that the client wants an estimate of the minimum local time held by any client. This field is always MINSERV in the client-server example, but we provide it because a server, in general, may provide multiple services. This field takes on a different value for the code on demand example presented later. The fourth and final field of a request must be an integer data item, which the client uses to transmit its current value of t to the server. With this in mind, the reader can see that the predicate serviceRequest checks to see that the message buffer of client i contains a message destined for the SERVER, with the opName and serviceName given. The predicate serviceReady checks the $Server.q(i)$ buffer for a message destined for client i.

Now we examine the inner workings of each component, as given by the **assign** section of each program definition. The assignment statements in the client program *P(i)* look very much like the ones in the centralized solution *DistributedSimulation* presented in Section 10.2, except that now the variables t occurs once in each component instead of appearing as global arrays of values: we now use $P(i).t$ in place of $t(i)$. Also, T appears once in each program instead of being globally declared. Note that the simulation mode $P(i).z$, in contrast to the centralized solution, is initialized statically for each component and does not change during simulation execution. This is done for the sake of simplicity and clarity. In the code on demand solution presented later, we compute this value dynamically. The statement to update the local simulation time $P(i).t$ is the same as before, except that the estimated GVT variable $P(i).T$ is simultaneously set to $\perp$ when an update to the local time is made. Throughout the example, the notation $\mathsf{def}(v)$ denotes the predicate $v \neq \perp$, i.e. the variable v is defined. This is used in the guard of the update to the local timer, and signals the fact that the client needs a new value of T before it can proceed with another simulation step. The new estimate of GVT is computed in the server at the request of the client, and the request is made by the third statement in the program *P(i)*, which writes a request record to the message buffer $P(i).RQ$.

Once the assignment to $P(i).RQ$ has taken place, the first interaction clause is enabled. Its guard makes use of the macro serviceRequest, which detects the presence of a valid request. After the interaction is enabled it is eventually executed, which asynchronously transmits the request to the server.

The *Server* program consists of three groups of assignments. The first processes input requests by updating the array $Server.\tau$ with the local time sent by the client. The second computes a new estimate of the GVT based on the current values in $Server.\tau$, if some client request is waiting in a buffer. The third takes the estimate and constructs a reply message to all waiting clients, clearing the estimate. Once the reply has been written to $Server.q(i)$, the second interaction clause is enabled. Its guard makes use of the macro serviceReady, which detects the presence of a valid reply. After the interaction is enabled it is eventually executed, which asynchronously transfers the reply back to the client. The fourth assignment statement of the program *P(i)* processes the reply by updating the local GVT estimate $P(i).T$ and clearing the request buffer.

11.2 Verification

The solution presented in Figure 11.1 introduces a degree of distribution with respect to the one in Figure 10.2. Its correctness remains to be shown. The CS solution must maintain Property 1 and 2 stated for the centralized one. The proof can make use of the standard UNITY proof logic as long as we use the fully qualified names at every step; we simply have a larger number of statements that must be considered. Later, we will introduce some abstractions which do have an impact on the proof logic, e.g., transient shared variables. For now, we outline the proof of the following two properties, which are simply a rewriting of Property 1 and 2 stated in Section 10.2 with the appropriate naming conventions observed:

PROPERTY 1 $P(i).t = \langle \min j :: P(j).t \rangle = \alpha$ **until** $P(i).t > \alpha$
A client P(i) increases its timer only when its value equals the GVT,

PROPERTY 2 $P(i).t = \alpha$ **until** $P(i).t = \langle \min j :: P(j).t \rangle = \alpha$
The GVT will catch up eventually with the timer in P(i).

The properties and the structure of their proof outline is the same across all the solutions presented in this part of the thesis. Nevertheless, the proof makes use of some lemmas and invariants which are proven differently in each solution. In the remainder of this chapter, we present the general proof structure, together with the proofs specific to the client-server solution. In the following chapter, we will provide only the proofs specific to each solution.

11.2.1 General Proof Structure

The properties stated earlier can be proven for all the solutions presented in this part of the thesis by using the following proof structure.

PROOF OF PROPERTY 1. We prove separately the two parts of the **until** relation:

1. $P(i).t = \langle \min j :: P(j).t \rangle = \alpha$ **unless** $P(i).t > \alpha$ can be proven from the program text by observing that, for any of the solutions, every statement in the system leaves unchanged the LHS except possibly for the first statement of *P(i)*. This statement, according to the definition of f and strict monotonicity of g, may either leave unchanged the LHS or establish the RHS.
2. $P(i).t = \langle \min j :: P(j).t \rangle = \alpha \mapsto P(i).t > \alpha$ can be split by transitivity of **leads-to** into two separate proof obligations. First, we prove that whenever a client's timer corresponds to the GVT, the GVT estimate eventually catches up with the timer within the client, i.e.

$$P(i).t = \langle \min j :: P(i).t \rangle = \alpha \mapsto P(i).t = P(i).T = \alpha$$

This is proven by applying transitivity to the following two lemmas:

$$\begin{aligned} &P(k).t = \langle \min l :: P(l).t \rangle = \beta \\ \mapsto\ &P(k).t = \langle \min l :: Server.\tau(l) \rangle = \beta \end{aligned} \qquad (\mathcal{L}_1)$$

which asserts that if a given GVT value is reached, the timers whose values match the GVT are eventually mirrored on the *Server*, and lemma

$$\begin{aligned} &P(k).t = \langle \min l :: Server.\tau(l) \rangle = \beta \\ \mapsto\ &P(k).t = P(k).T = \beta \end{aligned} \qquad (\mathcal{L}_2)$$

which asserts that the minimum of all the values mirrored on the *Server* eventually corresponds to the GVT estimate communicated back to the client.

Then, we prove that, in the situation when the GVT estimate is equal to the timer within a client, the client eventually makes progress by increasing its timer, which is formally captured by the following lemma:

$$P(k).t = P(k).T = \beta \textbf{ ensures } P(k).t > \beta \tag{$\mathcal{L}_3$}$$

Lemma ($\mathcal{L}_1$) is proven the same way across all the solutions, so it is proven in the remainder of this section. On the other hand, lemmas ($\mathcal{L}_2$) and ($\mathcal{L}_3$) will be proven for each solution in the later sections.

PROOF OF PROPERTY 2. We prove separately the two parts of the **until** relation:

1. $P(i).t = \alpha$ **unless** $P(i).t = \langle \min j :: P(j).t \rangle = \alpha$. We observe that

$$P(i).t = \alpha \textbf{ unless } P(i).t = P(i).T = \alpha$$

holds because, in all the solutions, the only statement affecting $P(i).t$ is the first statement in *P(i)*. After execution of such a statement it might be the case that the RHS holds, according to the invariant

$$\textbf{inv. } \mathsf{def}(P(k).T) \Rightarrow P(k).T \le P(k).t, \tag{$\mathcal{I}_1$}$$

which asserts that whenever the GVT estimate is defined within a client, its value is always at most equal to the corresponding local timer value. Furthermore, the invariant

$$\textbf{inv. } P(k).t = P(k).T = \beta \Rightarrow \langle \min l :: P(l).t \rangle = \beta, \tag{$\mathcal{I}_2$}$$

states that whenever the values of the local timer and the GVT estimate are equal in a client, their value equal the GVT value as well. Hence, according to the *consequence weakening theorem* [25]

$$\frac{p \textbf{ unless } q, q \Rightarrow r}{p \textbf{ unless } r} \tag{$\mathcal{T}_9$}$$

the conclusion holds. Invariant ($\mathcal{I}_1$) is proven in the remainder of this section. Invariant ($\mathcal{I}_2$), as well as the other introduced in this section, are proven differently in each solution. The corresponding proof outlines can be found in the Appendix.

2. $P(i).t = \alpha \;\mapsto\; P(i).t = \langle \min j :: P(j).t \rangle = \alpha$. To demonstrate that the system eventually reaches a state where the GVT is equal to the timer value α we need to demonstrate that the difference between α and the GVT decreases continuously. To this end, we need to show that the number of clients whose timer is greater than a given GVT but less than α eventually decreases to zero. This is a consequence of the continuous increase of the GVT and of the fact that the local timer holds its value up to the point where the GVT catches up with it. Formally, this proof requires the application of the induction principle of **leads-to**. The decrease in the distance δ between α and the current GVT is captured by

$$\begin{array}{ll} & P(i).t = \alpha \wedge 0 < (\alpha - \langle \min j :: P(j).t \rangle) = \delta \\ \mapsto & (P(i).t = \alpha \wedge 0 < (\alpha - \langle \min j :: P(j).t \rangle) < \delta) \vee \\ & \alpha = \langle \min j :: P(j).t \rangle . \end{array}$$

The equation above is proven by demonstrating that the number ν of clients at a given current GVT $\bar{\alpha}$ is decreasing, and that the GVT eventually increases, that is

$$\begin{array}{rl} & \langle \min j :: P(j).t \rangle = \bar{\alpha} \wedge 0 < \langle \sharp j :: P(j).t = \bar{\alpha} \rangle = \nu \\ \mapsto & (\langle \min j :: P(j).t \rangle = \bar{\alpha} \wedge 0 < \langle \sharp j :: P(j).t = \bar{\alpha} \rangle < \nu) \vee \\ & \langle \min j :: P(j).t \rangle > \bar{\alpha}, \end{array}$$

which follows from Property 1 proven earlier.

PROOF OF ($\mathcal{L}_1$). We rule out the case when $P(k).t = \langle \min l :: \mathit{Server}.\tau(l) \rangle = \beta$ because it can be demonstrated trivially by application of the implication theorem ($\mathcal{T}_5$). Then, we introduce the invariant

$$\textbf{inv.}\ \mathit{Server}.\tau(k) \leq P(k).t, \qquad (\mathcal{I}_3)$$

which asserts that the timer value mirrored on the server is always at most equal to the original value within the client. This invariant enables us to consider only the case $\langle \min l :: \mathit{Server}.\tau(l) \rangle < \beta$. Hence, we prove

$$\mathsf{oldServerState} \mapsto P(k).t = \langle \min l :: \mathit{Server}.\tau(l) \rangle = \beta,$$

where we introduced the following definition for the sake of readability:

$$\mathsf{oldServerState} \equiv P(k).t = \langle \min l :: P(l).t \rangle = \beta \wedge \langle \min l :: \mathit{Server}.\tau(l) \rangle < \beta.$$

We observe that this obligation can be proven by showing that the number of clients at GVT for which the timer value mirrored on the *Server* is not up-to-date is decreasing. Formally, this corresponds to application of the induction principle of **leads-to**:

$$\begin{array}{rl} & \mathsf{oldServerState} \wedge 0 < \langle \sharp l :: \mathit{Server}.\tau(l) < \beta \rangle = \nu \\ \mapsto & (\mathsf{oldServerState} \wedge 0 < \langle \sharp l :: \mathit{Server}.\tau(l) < \beta \rangle < \nu) \vee \\ & P(k).t = \langle \min l :: \mathit{Server}.\tau(l) \rangle = \beta \end{array}$$

which follows from the lemma

$$\begin{array}{r} P(k).t = \beta \wedge \mathit{Server}.\tau(k) < \beta \\ \textbf{until}\ P(k).t = \mathit{Server}.\tau(k) = \beta. \end{array} \qquad (\mathcal{L}_4)$$

This lemma, whose proof will be given case by case in the following sections, states that the value of a local timer is eventually copied on the *Server*, which guarantees continuous decrease of the metric used for induction.

PROOF OF ($\mathcal{I}_1$). The invariant holds from ($\mathcal{I}_3$) and invariant

$$\textbf{inv.}\ \mathsf{def}(P(k).T) \Rightarrow P(k).T \leq \langle \min l :: \mathit{Server}.\tau(l) \rangle, \qquad (\mathcal{I}_4)$$

which asserts that whenever the GVT estimate is defined within a client, it is at most equal to the minimum among the timer values mirrored on the server.

$P(k).t = \langle \min l :: Server.\tau(l) \rangle = \beta \;\mapsto\; P(k).t = P(k).T = \beta$	$(\mathcal{L}_2)$
$P(k).t = P(k).T = \beta$ **ensures** $P(k).t > \beta$	$(\mathcal{L}_3)$
$P(k).t = \beta \wedge Server.\tau(k) < \beta$ **until** $P(k).t = Server.\tau(k) = \beta$	$(\mathcal{L}_4)$
inv. $\mathsf{def}(Server.T) \Rightarrow Server.T \leq \langle \min l :: Server.\tau(l) \rangle$	(CS_1)
inv. $\mathsf{def}(Server.q(k)) \Rightarrow Server.q(k) \uparrow 2 = \text{CS} \vee$ $Server.q(k) \uparrow 2 = \text{WAIT} \vee \neg\mathsf{def}(Server.q(k) \uparrow 2)$	(CS_2)
inv. $P(k).t = P(k).T = \beta \Rightarrow \langle \min l :: P(l).t \rangle = \beta$	$(\mathcal{I}_2)$
inv. $Server.\tau(k) \leq P(k).t$	$(\mathcal{I}_3)$
inv. $\mathsf{def}(P(k).T) \Rightarrow P(k).T \leq \langle \min l :: Server.\tau(l) \rangle$	$(\mathcal{I}_4)$
inv. $\mathsf{def}(P(k).T) \Rightarrow \neg\mathsf{def}(P(k).RQ)$	(CS_3)
inv. $Server.q(k) \uparrow 1 = \text{SERVER} \Rightarrow Server.q(k) \uparrow 4 = P(k).t$	(CS_4)
$\langle \min l :: Server.\tau(l) \rangle = \beta$ **unless** $\langle \min l :: Server.\tau(l) \rangle > \beta$	(CS_5)

Table 11.1: Lemmas and invariants needed for verification of the Client-Server solution.

11.2.2 Proof of Client-Server Solution

Table 11.1 summarizes the obligations needed to prove the properties for the CS solution. In this section we provide the proof outlines for the lemmas defined in the previous section in the context of the CS solution, together with the proof of some auxiliary obligations. The reader can find the proof outlines for the invariants and the remaining obligations in Appendix A.1.

PROOF OF $(\mathcal{L}_2)$. We rule out the case $P(k).T = \beta$, by application of $(\mathcal{T}_5)$. We introduce the following definitions in order to improve readability:

$$\begin{aligned} \mathsf{minTimer} &\equiv P(k).t = \langle \min l :: Server.\tau(l) \rangle = \beta \\ \mathsf{oldClientT} &\equiv P(k).T < \beta \vee \neg\mathsf{def}(P(k).T) \\ \neg\mathsf{oldServerT} &\equiv Server.T = \beta \vee \neg\mathsf{def}(Server.T) \end{aligned}$$

The first definition states that *P(k)* owns the minimum among the timers in *Server*; the second express the fact that the GVT is out-of-date in client *P(k)*; and the third states that the GVT is not out-of-date in *Server*. Given this, invariant $(\mathcal{I}_1)$ allows us to reformulate the goal of our proof as

$$\mathsf{minTimer} \wedge \mathsf{oldClientT} \;\mapsto\; P(k).t = P(k).T = \beta. \qquad (\text{CS}_6)$$

i.e. if a client *P(k)* owns the timer corresponding to the minimum timer value stored on *Server* but the corresponding GVT estimate $P(k).T$ is behind the timer, the GVT estimate eventually catches up with the timer in *P(k)*. In order for $\mathsf{minTimer}$ to hold, the minimum β over $Server.\tau$ must have been established by a request message from *P(k)*, or from some other client with the same timer value, which updates the value mirrored on *Server*. Due to the interleaving semantics, however, it could happen that, at the moment when this minimum is established in $Server.\tau$, a value referring to a previous value $\bar{\beta} = \langle \min l :: Server.\tau(l) \rangle \neq \beta$ has been already assigned to the GVT estimate $Server.T$. However, invariant (CS_1) guarantees that $Server.T$ can never be assigned a value greater than β.

Thus, we can split (CS_6) into the following two cases

$$\mathsf{minTimer} \wedge \mathsf{oldClientT} \wedge \neg\mathsf{oldServerT} \mapsto P(k).t = P(k).T = \beta \tag{CS$_7$}$$

$$\mathsf{minTimer} \wedge \mathsf{oldClientT} \wedge \mathit{Server}.T < \beta \mapsto P(k).t = P(k).T = \beta, \tag{CS$_8$}$$

which, by application of disjunction, prove (CS_6) and then ($\mathcal{L}_2$). In the remainder, we prove each case separately.

PROOF OF (CS_7). We must show that if the *Server* has already established the correct GVT estimate or it is about to (i.e. its GVT estimate is currently void), while the client still holds an out-of-date GVT estimate, the new one is eventually communicated to the client. We accomplish this by reasoning about the communication taking place between *P(k)* and the *Server*. The only way the client can learn about the new GVT estimate is by receiving a reply to a formerly sent message request. In the worst case, such a request has not been issued yet, i.e. the *Server* does not have a message from *P(k)* in its queue. We show that all the other intermediate states are encompassed in the analysis of communication taking place in this worst case. Thus, we prove that, in the worst case above, a message request containing the timer value is eventually generated by the client, that is,

$$\begin{aligned} & \mathsf{minTimer} \wedge \mathsf{oldClientT} \wedge \neg\mathsf{oldServerT} \wedge \neg\mathsf{def}(\mathit{Server}.q(k)) \\ \mapsto\ & \mathsf{minTimer} \wedge \neg\mathsf{oldServerT} \wedge \\ & \neg\mathsf{def}(\mathit{Server}.q(k)) \wedge P(k).RQ = \langle \text{SERVER,CS,MINSERV},\beta\rangle. \end{aligned} \tag{CS$_9$}$$

It can be proven directly from the text that

$$\begin{aligned} & \mathsf{minTimer} \wedge \neg\mathsf{oldServerT} \wedge \\ & \neg\mathsf{def}(\mathit{Server}.q(k)) \wedge P(k).RQ = \langle \text{SERVER,CS,MINSERV},\beta\rangle \\ \textbf{ensures}\ & \mathsf{minTimer} \wedge \neg\mathsf{oldServerT} \wedge \\ & \mathit{Server}.q(k) = \langle \text{SERVER,CS,MINSERV},\beta\rangle \end{aligned} \tag{CS$_{10}$}$$

that is, such a request is eventually delivered to the *Server*. As a consequence, *Server* eventually accepts the request by forcing it into the WAIT state,

$$\begin{aligned} & \mathsf{minTimer} \wedge \neg\mathsf{oldServerT} \wedge \\ & \mathit{Server}.q(k) = \langle \text{SERVER,CS,MINSERV},\beta\rangle \\ \textbf{ensures}\ & \mathsf{minTimer} \wedge \neg\mathsf{oldServerT} \wedge \mathit{Server}.q(k) \uparrow 2 = \text{WAIT}. \end{aligned} \tag{CS$_{11}$}$$

which, again, can be proven directly from the program text. Finally, we prove that the pending request is eventually satisfied and a reply containing the updated value for the GVT is delivered back to *P(k)*, that is,

$$\begin{aligned} & \mathsf{minTimer} \wedge \neg\mathsf{oldServerT} \wedge \mathit{Server}.q(k) \uparrow 2 = \text{WAIT} \\ \mapsto\ & P(k).t = P(k).T = \beta. \end{aligned} \tag{CS$_{12}$}$$

Application of disjunction and transitivity of **leads-to** to (CS_9), (CS_{10}), (CS_{11}), and (CS_{12}) prove (CS_7). Equation (CS_9) is proven by expanding

$\mathsf{oldClientT}$ into the possible values of $P(k).T$ according to $(\mathcal{I}_1)$, and applying disjunction and transitivity to the following chain of **ensures**:

$$\begin{array}{rl} & \mathsf{minTimer} \wedge \neg\mathsf{oldServerT} \wedge \neg\mathsf{def}(\mathit{Server}.q(k)) \wedge P(k).T < \beta \\ \textbf{ensures} & \mathsf{minTimer} \wedge \neg\mathsf{oldServerT} \wedge \neg\mathsf{def}(\mathit{Server}.q(k)) \wedge \neg\mathsf{def}(P(k).T) \\ \textbf{ensures} & \mathsf{minTimer} \wedge \neg\mathsf{oldServerT} \wedge \neg\mathsf{def}(\mathit{Server}.q(k)) \wedge \\ & P(k).RQ = \langle \text{SERVER,CS,MINSERV},\beta\rangle, \end{array} \qquad (\text{CS}_{13})$$

where the initial state implies that there $P(k)$ has not issued a request yet, in accordance with invariant (CS_3). Similarly, (CS_{11}) can be proven by considering the two values of the GVT estimate $\mathit{Server}.T$ as defined by $\neg\mathsf{oldServerT}$. Finally, disjunction and transitivity applied to the following chain of **ensures**

$$\begin{array}{rl} & \mathsf{minTimer} \wedge \neg\mathsf{def}(\mathit{Server}.T) \wedge \mathit{Server}.q(k) \uparrow 2 = \text{WAIT} \\ \textbf{ensures} & \mathsf{minTimer} \wedge \mathit{Server}.T = \beta \wedge \mathit{Server}.q(k) \uparrow 2 = \text{WAIT} \\ \textbf{ensures} & \mathsf{minTimer} \wedge \mathit{Server}.q(k) = \langle k,\perp,\perp,\beta\rangle \\ \textbf{ensures} & \mathsf{minTimer} \wedge P(k).RQ = \langle k,\perp,\perp,\beta\rangle \\ \textbf{ensures} & P(k).t = P(k).T = \beta, \end{array}$$

which can be derived from the program text, proves (CS_{12}) and complete the proof, together with invariant (CS_2) which constrains the value of $\mathit{Server}.q(k)$ and thus guarantees coverage of all the possible cases in the above obligations.

PROOF OF (CS_8). If *Server* holds an out-of-date value for the GVT, this is eventually cleared and subsequently replaced by a new, up-to-date value corresponding to the minimum timer in $\mathit{Server}.\tau$. At this point, we fall back in the state on the LHS of (CS_7), for which we already shown that such a value is eventually communicated to the client. The GVT estimate on the *Server* is cleared whenever a reply is sent back to the client, hence we prove that a message request is eventually sent to the *Server*:

$$\begin{array}{rl} & \mathsf{minTimer} \wedge \mathsf{oldClientT} \wedge \mathit{Server}.T < \beta \wedge \neg\mathsf{def}(\mathit{Server}.q(k)) \\ \mapsto & \mathsf{minTimer} \wedge \mathsf{oldClientT} \wedge \neg\mathsf{def}(\mathit{Server}.q(k)) \wedge \\ & P(k).RQ = \langle \text{SERVER,CS,MINSERV},\beta\rangle \end{array}$$

The above can be proven with arguments similar to those used for (CS_9). Invariant (CS_1) lets us rewrite the obligation as

$$\begin{array}{rl} & \mathsf{minTimer} \wedge \mathsf{oldClientT} \wedge \mathit{Server}.T < \beta \wedge \neg\mathsf{def}(\mathit{Server}.q(k)) \\ \mapsto & (\mathsf{minTimer} \wedge \mathsf{oldClientT} \wedge \mathit{Server}.T < \beta \wedge \neg\mathsf{def}(\mathit{Server}.q(k)) \wedge \\ & P(k).RQ = \langle \text{SERVER,CS,MINSERV},\beta\rangle) \vee \\ & (\mathsf{minTimer} \wedge \mathsf{oldClientT} \wedge \neg\mathsf{oldServerT} \wedge \neg\mathsf{def}(\mathit{Server}.q(k)) \wedge \\ & P(k).RQ = \langle \text{SERVER,CS,MINSERV},\beta\rangle), \end{array} \qquad (\text{CS}_{14})$$

thus splitting the obligation in two cases. Due to the concurrency in the system, the statements of the clients *P(k)* are interleaved with those of the *Server*. Hence, while client *P(k)* is preparing a request, $\mathit{Server}.T$:

1. may still contain an out-of-date value;
2. may be either undefined or up-to-date, because in the meanwhile the *Server* has sent a reply to some of the other clients or it has established the new GVT estimate as the minimum in $\mathit{Server}.\tau$.

The second disjunct implies the LHS of (CS$_{10}$), then we can apply transitivity thanks to the implication theorem and thus show that it eventually leads to the goal, that is

$$\begin{array}{ll} & \mathsf{minTimer} \wedge \mathsf{oldClientT} \wedge \neg\mathsf{oldServerT} \wedge \neg\mathsf{def}(\mathit{Server}.q(k)) \wedge \\ & P(k).RQ = \langle \text{SERVER,CS,MINSERV},\beta\rangle \\ \mapsto & P(k).t = P(k).T = \beta \end{array} \qquad (\text{CS}_{15})$$

We can exploit this fact through application of the following *cancellation theorem* of **leads-to** [25]

$$\frac{p \mapsto q \vee b,\ b \mapsto r}{p \mapsto q \vee r}. \qquad (\mathcal{T}_{10})$$

In our case, application of the cancellation theorem to (CS$_{14}$) and (CS$_{15}$) yields

$$\begin{array}{ll} & \mathsf{minTimer} \wedge \mathsf{oldClientT} \wedge \mathit{Server}.T < \beta \wedge \neg\mathsf{def}(\mathit{Server}.q(k)) \\ \mapsto & (\mathsf{minTimer} \wedge \mathsf{oldClientT} \wedge \mathit{Server}.T < \beta \wedge \neg\mathsf{def}(\mathit{Server}.q(k)) \wedge \\ & P(k).RQ = \langle \text{SERVER,CS,MINSERV},\beta\rangle) \vee P(k).t = P(k).T = \beta \end{array}$$

Our proof strategy uses the cancellation theorem to perform the verification of the communication steps by ruling out the case where the GVT estimate becomes up-to-date on the *Server*, thus examining always the worst case. In doing this, we proceed along the lines of the proof developed for (CS$_7$), thus reusing the results already achieved. Our next goal is to prove that for the first disjunct the following holds:

$$\begin{array}{ll} & \mathsf{minTimer} \wedge \mathsf{oldClientT} \wedge \mathit{Server}.T < \beta \wedge \\ & \neg\mathsf{def}(\mathit{Server}.q(k)) \wedge P(k).RQ = \langle \text{SERVER,CS,MINSERV},\beta\rangle \\ \mapsto & P(k).t = P(k).T = \beta. \end{array}$$

Similarly to (CS$_{10}$), it can be proven from the program text that

$$\begin{array}{ll} & \mathsf{minTimer} \wedge \mathsf{oldClientT} \wedge \mathit{Server}.T < \beta \wedge \neg\mathsf{def}(\mathit{Server}.q(k)) \wedge \\ & P(k).RQ = \langle \text{SERVER,CS,MINSERV},\beta\rangle \\ \textbf{ensures} & \mathsf{minTimer} \wedge \mathsf{oldClientT} \wedge \mathit{Server}.q(k) = \langle \text{SERVER,CS,MINSERV},\beta\rangle \end{array}$$

which, thanks to invariant (CS$_1$), can be rewritten as

$$\begin{array}{ll} & \mathsf{minTimer} \wedge \mathsf{oldClientT} \wedge \mathit{Server}.T < \beta \wedge \neg\mathsf{def}(\mathit{Server}.q(k)) \wedge \\ & P(k).RQ = \langle \text{SERVER,CS,MINSERV},\beta\rangle \\ \textbf{ensures} & (\mathsf{minTimer} \wedge \mathsf{oldClientT} \wedge \mathit{Server}.T < \beta) \vee \\ & (\mathsf{minTimer} \wedge \mathsf{oldClientT} \wedge \neg\mathsf{oldServerT} \wedge \\ & \mathit{Server}.q(k) = \langle \text{SERVER,CS,MINSERV},\beta\rangle) \end{array}$$

Repeated application of the cancellation theorem and of the proof obligations developed for (CS$_7$), whose details are here omitted, yields to the final goal of proving that

$$\mathsf{minTimer} \wedge \mathit{Server}.T < \beta \wedge \mathit{Server}.q(k) \uparrow 2 = \text{WAIT} \mapsto \quad P(k).t = P(k).T = \beta,$$

that is, even in the case when the *Server* accepts a request while still owning an out-of-date value for the GVT estimate, this eventually leads to communication of an up-to-date value to *P(k)*. We demonstrate this by proving that

$$\begin{array}{ll} & \mathsf{minTimer} \wedge \mathit{Server}.T < \beta \wedge \mathit{Server}.q(k) \uparrow 2 = \text{WAIT} \\ \mapsto & \mathsf{minTimer} \wedge \neg\mathsf{oldServerT} \wedge \neg\mathsf{def}(\mathit{Server}.q(k)) \wedge P(k).T < \beta \end{array} \qquad (\text{CS}_{16})$$

holds, by observing that the RHS corresponds to the LHS of (CS_{13}), for which we already proven that it leads to the goal. This completes the proof together with invariant (CS_2), and disjunction and transitivity of **leads-to**. Property (CS_{16}) is proven by the following chain of **ensures**:

$$\begin{array}{rl} & \mathsf{minTimer} \wedge \mathit{Server}.T < \beta \wedge \mathit{Server}.q(k) \uparrow 2 = \text{WAIT} \\ \textbf{ensures} & \mathsf{minTimer} \wedge \neg\mathsf{def}(\mathit{Server}.T) \wedge \mathit{Server}.q(k) = \langle k,\perp,\perp,\bar{\beta}\rangle \wedge \bar{\beta} < \beta \\ \textbf{ensures} & \mathsf{minTimer} \wedge \neg\mathsf{oldServerT} \wedge P(k).RQ = \langle k,\perp,\perp,\bar{\beta}\rangle \wedge \bar{\beta} < \beta \\ \textbf{ensures} & \mathsf{minTimer} \wedge \neg\mathsf{oldServerT} \wedge \neg\mathsf{def}(\mathit{Server}.q(k)) \wedge P(k).T < \beta. \end{array}$$

If the *Server* owns an out-of-date value for the GVT estimate and there is a request waiting, the only step that can be taken by the *Server* is to clear the GVT value and trasmit the out-of-date GVT estimate. This in turn triggers message delivery to the client and the subsequent voiding of the corresponding element of the message queue of *Server*. However, the important point is that, because the minimum value on $\mathit{Server}.\tau$ is β, whenever the GVT estimate is eventually established, it is given this value, which is eventually communicated to the client, as proven by (CS_7).

PROOF OF ($\mathcal{L}_3$). Invariant (CS_3) guarantees that if the GVT estimate is defined within a client, the latter has no pending request. Thus, if the LHS of ($\mathcal{L}_3$) holds there is no incoming new value for the GVT estimate which could possibly change it, and ($\mathcal{I}_1$) guarantees that $P(k).T$ can never increase past $P(k).t$. Consequently, the LHS of ($\mathcal{L}_3$) is preserved. Finally, the first statement in *P(k)* together with the fairness assumption guarantee that the RHS is established, according to the definitions of f and g.

PROOF OF ($\mathcal{L}_4$). We prove separately the two parts of the **until**. As for the **unless** part, invariant ($\mathcal{I}_4$) guarantees that the following holds

$$\mathit{Server}.\tau(k) < \beta \Rightarrow (\mathsf{def}(P(k).T) \Rightarrow P(k).T < \beta),$$

then $P(k).t$ cannot increase while $\mathit{Server}.\tau(k) < \beta$ holds. Nevertheless, after execution of the first statement in *Server*, the RHS of the **unless** might be established. The **leads-to** is proven with arguments similar to those used to reason about communication in the previous proofs. Informally, starting from the state in the LHS, the client eventually generates a request that carries the new value for the local timer. The message containing this value is subsequently transferred to the message queue of the *Server* by the statements in the **Interactions** section, and then the corresponding value in $\mathit{Server}.\tau$ is updated.

11.3 Additional Interaction Constructs

The client-server example does not exercise all of the features of Mobile UNITY. Next, we discuss some of those unused features in preparation for their use in later examples. For instance, none of the components in the *DSClientServer* system are mobile. Also, the only interactions present are two extra assignment statements. In general, a Mobile UNITY system can contain more powerful interaction clauses, such as the specification of *transient shared variables.* These are variables of one component that are dynamically bound to variables of another component in a location-dependent manner. Any assignments to one of the variables are considered to propagate atomically to the other, while the two variables are bound. This allows the designer to express location-dependent consistency, i.e. when components are "near" each other, a high degree of consistency is maintained due to the availability of a high bandwidth, low latency communication channel. When components move apart, only a low degree of consistency would be possible for good performance in the face of decreased or non-existent bandwidth. For example, assume for a moment that the clients *P(i)* are allowed to migrate around a network of workstations for the purpose of load balancing. The server resides on one of the workstations, and some clients may also be located there. Due to the availability of operating system support for inter-process shared memory, the designers decide that the clients which are co-located with the server may share the message buffers directly with the server rather than using asynchronous message passing. The Mobile UNITY specification of this sharing could be written as:

$$Server.q(i) \approx P(i).RQ \quad \textbf{when } Server.\lambda = P(i).\lambda$$

which states that the request buffer of client i should be shared with $Server.q(i)$ when the two components are at the same place. In the case of a network of workstations, location would be the IP address of the machine, for instance.

While a pair of transiently shared variables are disconnected, they may take on different values, because assignments to one are not immediately propagated to the other. This may present a problem when the variables are later reconnected. If they are to be treated intuitively as one variable, they should immediately take on the same value when they become shared. Mobile UNITY allows for the specification of an **engage** value, which is assigned atomically to both variables immediately upon a transition of the **when** predicate from false to true. We may want to specify that the message buffer takes on the value present at the server when a new process arrives—e.g., because implementation details of inter-process shared memory make it more efficient for the buffers to be allocated in a single block, and thus placed on the server. We would write this as

$$\begin{array}{ll} Server.q(i) \approx P(i).RQ \qquad & \textbf{when } Server.\lambda = P(i).\lambda \\ & \textbf{engage } Server.q(i) \end{array}$$

Similarly, a pair of **disengage** values may be specified that are assigned atomically to the variables when they become disconnected. If, for instance, it is too expensive to copy the contents of the message buffer into the client when it moves to a new workstation, but the contents should be retained at the server, we would specify

$$\begin{array}{ll} Server.q(i) \approx P(i).RQ \qquad & \textbf{when } Server.\lambda = P(i).\lambda \\ & \textbf{engage } Server.q(i) \\ & \textbf{disengage } Server.q(i),\ \bot \end{array}$$

Note that if no **engage** value is specified, the variables remain in an inconsistent state after sharing takes effect until the first assignment is propagated. If no **disengage** value is specified, the variables retain the values they had before the variables are disconnected. Later examples will make extensive use of transient sharing, including one-way sharing, where updates are propagated in one direction but not the reverse. This is expressed as above except with an $\leftarrow$ in place of $\approx$, pointing in the same direction as updates are to be propagated.

To accommodate shared variables, as well as other forms of component interaction, Mobile UNITY makes certain adjustments to the standard UNITY operational model as well as to the proof logic. Because the updates to shared variables must happen in the same atomic step as an assignment, but sharing is specified separately from the (possibly many) assignments that may change the value of a variable, Mobile UNITY has a two-phased operational model where the first phase is an ordinary assignment statement and the second is responsible for propagating changes to shared variables. We call the statements that execute in the second phase *reactive statements*, and they are denoted in the text of a Mobile UNITY program by the use of **reacts-to** in place of **if**. Reactive statements are executed right after the statement they are associated with and may possibly trigger execution of other reactive statements. The reader should keep in mind that transient sharing is really a shorthand notation for a set of reactive statements. Then, particular care must be placed in the case when transitive transient sharing takes place, due to the possibility of circular reactions which may lead the system to deadlock. In general, the Mobile UNITY proof logic copes with these problem by considering the set of reactive statements as a separate program, whose termination has to be proven in order to guarantee correct sharing. However, for the purposes of this work this issue does not arise. Due to the characteristics of mobile code, transient sharing is always pairwise and never leads to circular sharing. In addition, we will always use the high level abstraction provided by transient shared variables without using directly reactive statements. This means that the reactive program is always constituted only by the reactive statements contained in the definition of shared variables, which are well-formed and guarantee termination in absence of circularities. Mobile UNITY also allows for global constraints on the execution of statements, called **inhibit** clauses. These serve to strengthen the guard on a statement and can express scheduling constraints that may not be expressible using only local state information. A third construct, the *transaction*, is used for specifying a sequence of statements that are executed as a unit with respect to other, non-reactive statements. Together, these primitives provide a powerful notation for expressing inter-component interaction in a highly decoupled and dynamic way. However, we found that shared variables suffice for the modeling of interactions taking place in mobile code systems. As a final remark concerned with verification, the properties of the system involving shared variables will be expressed by safety properties. This is meaningful in light of the fact that transient sharing expresses a constraint upon the value assigned to variables when a particular condition holds—usually involving co-location. Further information about Mobile UNITY operational model and proof logic can be found in [69].

Chapter 12

Mobile Code Solutions

In this chapter we present distributed solutions for the problem stated in Chapter 10 using the mobile code paradigms described in Chapter 3. For each solution, an informal description of the Mobile UNITY system is provided, followed by the proof outline of the progress properties stated in Section 10.2. In the following solutions the GVT estimate is always computed by a single component, like in the CS solution, and its value is communicated back to the processes *P(i)*. Thus, we continue to refer to components *P(i)* as *clients* of the process computing the GVT, whether they are mobile or not. Furthermore, as in the CS solution we assume that the value for the simulation mode z is computed statically at initialization time. We will relax this assumption in Section 12.3, where we describe a solution based on the COD paradigm that involves the computation of a dynamically changing simulation mode.

12.1 Mobile Agent

Figure 12.1 shows a Mobile UNITY system designed using the MA paradigm. As in the CS solution, the client processes *P(i)* can increment the timer value, consuming the local estimate for GVT—which prevents further timer increments until a new estimate becomes available. In contrast with the CS solution, no handling of message requests is needed. The location of each client is initialized to a given value, which cannot be changed. The *Server* component, in turn, is initially co-located with an arbitrarily chosen client and changes explicitly its location during execution in order to visit all clients in a round-robin fashion, as shown in Figure 12.2. The *Server* carries with it the global state of all local timers, which is updated with the timer value of a client *P(i)* while the two components are co-located. The *Server* is also responsible for computing a new GVT estimate. However, the update of the global state and the computation of the GVT can happen in any order, because the *Server* cannot depart until both the values are up-to-date. The actions performed by the components above are coordinated by the transient variable sharing defined in the **Interactions** section. The GVT estimate and the local timer belonging to the client *P(i)* are shared with their analogues within the *Server*, as long as the two components

```
System DSMobileAgent
    Program P(i) at λ
        declare
            t,z : integer [] T : integer ∪ {⊥}
        initially
            t = 0 [] T = ⊥ [] λ = Location(i)
        assign
            t,T := f_i(t,T,z),⊥ if def(T)
    end
    Program Server at λ
        declare
            t,T : integer ∪ {⊥} [] τ : array of integer [] pos : clientAddress
        initially
            t,T = 0,0 [] ⟨|| j :: τ(j) = 0⟩ [] λ = Location(pos)
        assign
            τ(pos) := t
            [] T := ⟨min k :: τ(k)⟩
            [] λ,pos := Location(pos + 1 mod N),pos + 1 mod N     if t = τ(pos) ∧
                                                                     T = ⟨min k :: τ(k)⟩
    end
    Components
            ⟨[] i :: P(i)⟩ [] Server
    Interactions
            P(i).T ← Server.T        when P(i).λ = Server.λ
                                     engage Server.T
            [] Server.t ← P(i).t     when P(i).λ = Server.λ
                                     engage P(i).t
end
```

Figure 12.1: Mobile Agent solution for the distributed simulation problem.

are co-located:

$$\begin{array}{ll} P(i).T \leftarrow Server.T & \textbf{when } P(i).\lambda = Server.\lambda \\ & \textbf{engage } Server.T \\ Server.t \leftarrow P(i).t & \textbf{when } P(i).\lambda = Server.\lambda \\ & \textbf{engage } P(i).t \end{array}$$

The last statement in *Server*, which modifies explicitly the location λ of the component, exercises the **engage** clause of the second interaction. Upon departure, absence of a **disengage** clause guarantees that both the client and the *Server* retain their value for the timer and the GVT estimate. Upon arrival at the next location, the **engage** clause specified in the second read-only shared variable definition guarantees that *as soon as* the two components become co-located, they share the same value for the local timer. The **engage** clause in the first interaction guarantees that the GVT value computed by *Server* on the previous client is communicated to *P(i)*. In general, this value is be changed by the recomputation of the minimum of all timers, which takes into account the current value of the local timer in *P(i)*. Nevertheless, if this timer has not changed since the last visit, the corresponding value in τ remains unchanged and the GVT computed on the previous client—and already communicated to *P(i)*—is still valid and does not need to be recomputed. Hence, in this case the third statement in *Server* is enabled and allows *Server* to depart immediately, without waiting for execution of the two other statements. As an aside, note that disengagement of values, movement, and subsequent re-engagement form a single, atomic action. This captures the fact that the agent cannot perform any action while travelling across a communication link.

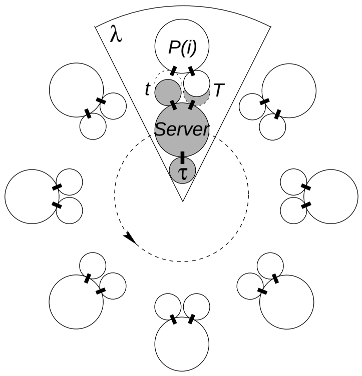


Figure 12.2: In the Mobile Agent solution, the *Server* is able to change its location in order to visit each client in a round-robin fashion.

Verification

The general proof structure provided in Section 11.2.1 is still valid for the MA solution. However, verification must encompass reasoning about the read-only sharing specified in the **Interactions** section. Table 12.1 summarizes the lemmas and invariants needed to prove the MA solution. In this section, we provide the proof outlines for the lemmas used in Section 11.2.1, together with some obligations specific of the MA solution. The remaining obligations are proven in Appendix A.2. Besides the properties described in Section 11.2.1, two additional properties capture the round-robin migration of *Server*. Invariant (MA_1) states that *Server* is always co-located with some client, while lemma (MA_2) guarantees that *Server* is forced to visit all the nodes.

	$P(k).t = \langle \min l :: Server.\tau(l) \rangle = \beta \mapsto P(k).t = P(k).T = \beta$	$(\mathcal{L}_2)$
	$P(k).t = P(k).T = \beta$ **ensures** $P(k).t > \beta$	$(\mathcal{L}_3)$
	$P(k).t = \beta \wedge Server.\tau(k) < \beta$ **until** $P(k).t = Server.\tau(k) = \beta$	$(\mathcal{L}_4)$
	inv. $\langle \exists k : 0 \leq k \leq N-1 :: Server.\lambda = P(k).\lambda \rangle$	(MA_1)
	$Server.\lambda = P(k).\lambda$ **until** $Server.\lambda = P(k+1 \bmod N).\lambda$	(MA_2)
	inv. $Server.T \leq \langle \min l :: Server.\tau(l) \rangle$	(MA_3)
	$\langle \min l :: Server.\tau(l) \rangle = \beta$ **unless** $\langle \min l :: Server.\tau(l) \rangle > \beta$	(MA_4)
inv.	$P(k).t = P(k).T = \beta \Rightarrow \langle \min l :: P(l).t \rangle = \beta$	$(\mathcal{I}_2)$
inv.	$Server.\tau(k) \leq P(k).t$	$(\mathcal{I}_3)$
inv.	$\mathsf{def}(P(k).T) \Rightarrow P(k).T \leq \langle \min l :: Server.\tau(l) \rangle$	$(\mathcal{I}_4)$
inv.	$Server.\lambda = P(k).\lambda \Rightarrow Server.t = P(k).t$	(MA_5)
inv.	$Server.\lambda = P(k).\lambda \Rightarrow$ $Server.T = P(k).T \vee \neg\mathsf{def}(P(k).T)$	(MA_6)
inv.	$Server.\lambda = P(k).\lambda \Rightarrow Server.pos = k$	(MA_7)

Table 12.1: Lemmas and invariants needed for verification of the Mobile Agent solution.

PROOF OF ($\mathcal{L}_2$). Our proof will exploit the *progress-safety-progress theorem* (PSP) [25]

$$\frac{p \mapsto q,\ r \textbf{ unless } b}{p \wedge r \mapsto (q \wedge r) \vee b}, \qquad (\mathcal{T}_{11})$$

which allows for combination of a progress and a safety property in proving a progress property. In our case, the progress property is

$$\mathit{true} \mapsto \mathit{Server}.T = P(k).T = \langle \min l :: \mathit{Server}.\tau(l) \rangle, \qquad (\text{MA}_8)$$

which states that a client always knows eventually the value of a GVT estimate, and the safety property is

$$\begin{array}{r l} & P(k).t = \langle \min l :: \mathit{Server}.\tau(l) \rangle = \beta \\ \textbf{unless} & P(k).t = P(k).T = \beta, \end{array} \qquad (\text{MA}_9)$$

which actually would allow us to reformulate ($\mathcal{L}_2$) in terms of **until** rather than **leads-to**. Application of the PSP theorem to (MA_8) and (MA_9) yields

$$\begin{array}{r l} & P(k).t = \langle \min l :: \mathit{Server}.\tau(l) \rangle = \beta \\ \mapsto & (\mathit{Server}.T = P(k).t = P(k).T = \langle \min l :: \mathit{Server}.\tau(l) \rangle = \beta) \vee \\ & P(k).t = P(k).T = \beta \\ \Rightarrow & P(k).t = P(k).T = \beta \end{array}$$

which proves ($\mathcal{L}_2$) by application of the implication theorem.

PROOF OF (MA_8). This obligation can be regarded as a consequence of transitivity and the induction principle of **leads-to** applied to (MA_2) and

$$\mathit{Server}.\lambda = P(k).\lambda \mapsto \mathit{Server}.T = P(k).T = \langle \min l :: \mathit{Server}.\tau(l) \rangle,$$

which states that co-location of *Server* with a client eventually leads to communication of the GVT estimate. Using invariant (MA_3) we can rule out the case when $\mathit{Server}.T = \langle \min l :: \mathit{Server}.\tau(l) \rangle$ and the LHS of the above hold together, as it follows trivially from the implication theorem ($\mathcal{T}_5$). Then, we prove that

$$\begin{array}{r l} & \mathit{Server}.\lambda = P(k).\lambda \wedge \mathit{Server}.T < \langle \min l :: \mathit{Server}.\tau(l) \rangle \\ \textbf{ensures} & \mathit{Server}.T = P(k).T = \langle \min l :: \mathit{Server}.\tau(l) \rangle, \end{array}$$

which can be done by reasoning only about *Server*. When the LHS holds, the only statements enabled are the first two in *Server*, the guard of the third statement being false. Thus, *Server* movement is prevented and no other statement can modify the value of *Server*.T except for the second one in *Server*. This is eventually executed because of the fairness assumption and establishes the RHS. Application of the disjunction rule ($\mathcal{T}_8$) completes the proof.

PROOF OF (MA_9). We can rule out the case $P(k).T = \beta$ because, analogously to ($\mathcal{T}_5$), implication is a special case of **unless**. Thus, we prove

$$\begin{array}{r l} & P(k).t = \langle \min l :: \mathit{Server}.\tau(l) \rangle = \beta \wedge (P(k).T < \beta \vee \neg \mathsf{def}(P(k).T)) \\ \textbf{unless} & P(k).t = P(k).T = \beta, \end{array} \qquad (\text{MA}_{10})$$

by observing that execution of the first statement in *P(k)* can never affect the value of the local timer in *P(k)*, due to the definition of f and

g. Furthermore, if *P(k)* is not co-located with *Server*, $P(i).T$ cannot be changed to a value different from those in the LHS of (MA$_{10}$), because sharing is not in place and the first statement in *P(k)* can only void the GVT estimate. Finally, according to invariant (MA$_1$), the *Server* must be co-located with some other client which might possibly affect the minimum value in *Server*.τ. However, invariant ($\mathcal{I}_3$) and lemma (MA$_4$), together with the LHS, ensure that no other client may lower the minimum in *Server*.τ. Hence, we can now restrict our analysis to the case where *P(k)* is co-located with *Server*. In this case, the first statement in *Server* does not affect the LHS of (MA$_{10}$) because of invariant (MA$_5$). Execution of the second statement in *Server* may actually establish the RHS, through the read-only sharing expressed by the first interaction and captured by invariant (MA$_6$). Finally, the migration taking place upon execution of the last statement triggers both engagement and disengagement. Engagement may either establish the RHS of (MA$_{10}$), in the case where *P(k)* and *Server* engage and the latter already has an up-to-date value of the GVT, or leave the LHS unaffected, by assigning an out-of-date value to $P(i).T$. Disengagement always leaves the LHS unaffected, because no **disengage** clause is specified, and thus both components retain their current values.

PROOF OF ($\mathcal{L}_3$). The only statements which can affect the LHS of ($\mathcal{L}_3$) are the first statement in *P(k)* and the second statement *Server*. The former actually establishes the RHS, due to the definition of f and monotonicity of g. The latter might in principle affect the value of $P(k).T$ through the sharing expressed by the first interaction. However, the LHS of ($\mathcal{L}_3$), together with invariants ($\mathcal{I}_4$) and ($\mathcal{I}_3$), yields

$$P(k).t = P(k).T = \langle \min l :: \mathit{Server}.\tau(l) \rangle = \beta$$

and hence the LHS of ($\mathcal{L}_3$) is preserved by execution of this statement. Finally, fairness guarantees that the first statement in *P(k)* is executed eventually, thus proving the **ensures**.

PROOF OF ($\mathcal{L}_4$). The **unless** part of the property is proven as in the CS solution. In order to prove the **leads-to** part, we proceed analogously to the proof of ($\mathcal{L}_2$) by applying the PSP theorem. The progress property is

$$\mathit{true} \mapsto P(k).t = \mathit{Server}.\tau(k), \qquad (\text{MA}_{11})$$

which states that a timer value is always eventually communicated to the *Server*, and the safety property is the **unless** part of ($\mathcal{L}_4$). Application of ($\mathcal{T}_{11}$) yields

$$\begin{array}{ll} & P(k).t = \beta \wedge \mathit{Server}.\tau(k) < \beta \\ \mapsto & (P(k).t = \mathit{Server}.\tau(k) \wedge P(k).t = \beta \wedge \mathit{Server}.\tau(k) < \beta) \vee \\ & P(k).t = \mathit{Server}.\tau(k) = \beta, \end{array}$$

from which the goal follows from implication, since the first disjunct in the RHS of the above never holds. Property (MA$_{11}$) is proven by transitivity and the induction principle of **leads-to** applied to (MA$_2$) and

$$\mathit{Server}.\lambda = P(k).\lambda \ \textbf{ensures}\ \mathit{Server}.\tau(k) = P(k).t.$$

This property is proven by observing that, as long as the RHS does not holds, *Server* cannot move and hence the LHS remains unchanged. Finally, invariants (MA$_7$) and (MA$_5$) and the first statement in *Server*, together with the fairness assumption, guarantee that such a statement is executed, thus establishing the RHS of the **ensures**.

PROOF OF (MA$_2$). The **unless** part is proven straightforwardly by observing that no statement affects the LHS except for the last one in *Server*, which actually establishes the RHS. As for the **leads-to** part, inspection of text of

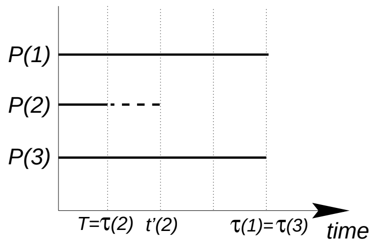


Figure 12.3: A situation where *Server*'s departure is delayed.

the *Server* program evidences that *Server* is allowed to leave only after the local timer of $P(k)$ has been registered in τ and the GVT estimate has been updated accordingly. However, *Server* is not guaranteed to depart right after these two conditions are established together—i.e. the guard of the third statement in *Server* is true. Figure 12.3 describes a situation where *Server* departure is delayed. Here, client $P(2)$ owns the minimum timer, and all the other processes have scheduled their actions far in the future with respect to it. After τ has been updated with the value of the local timer and the new GVT estimate has been computed, $T = \tau(2) = t(2)$ holds. Then, *Server* is allowed to depart. However, at the same time $P(2)$ is allowed to execute its statement and update its timer to $t'(2)$, as shown. Since the other processes have scheduled actions in the distant future, $P(2)$ and *Server* are the only processes enabled to make progress. Depending on the fair, non-deterministic scheduling, either statement can be selected for execution. If the statement of $P(2)$ is selected, its execution changes the value of the local timer, thus disabling the guard of the last statement in *Server* and thus preventing it from departure. In the worst case, if $P(2)$ is "faster" than *Server*, the last statement in *Server* might be enabled several times, each time being subsequently disabled by the timer update in $P(2)$. This situation is guaranteed to cease eventually when the timer in $P(2)$ is greater than the values in τ for the other timers. In this state, when *Server* establishes the GVT estimate, invariant (MA$_3$) guarantees that the GVT estimate is behind the local timer of $P(2)$ and thus, due to the definition of f and g, prevents further update of this timer. Our proof demonstrates that even in the worst case, as the one depicted in Figure 12.3, *Server* eventually departs. The other cases are encompassed in the formal proof for the worst case. Hence, our goal is to show that

$$
\begin{array}{ll}
& Server.\lambda = P(k).\lambda \wedge P(k).t \leq \beta \wedge \langle \min l : k \neq l :: Server.\tau(l) \rangle = \beta \\
\mapsto & (Server.\lambda = P(k).\lambda \wedge Server.T = \langle \min l :: Server.\tau(l) \rangle = \beta \wedge \\
& Server.\tau(k) = P(k).t \wedge P(k).t > \beta) \vee \\
& Server.\lambda = P(k+1 \bmod N).\lambda,
\end{array}
\qquad (\text{MA}_{12})
$$

holds, where the LHS captures the formal definition for the worst case as described earlier. The property above states that a state is reached eventually where either *Server* has departed or the guard of the last statement in *Server* is true and cannot be falsified by an update to $P(k).t$—since it can no longer affect the minimum in $Server.\tau$. From this state, in the RHS of (MA_{12}), application of disjunction, transitivity, and the cancellation theoreom of **leads-to** to the above and to

$$\begin{array}{ll} & Server.\lambda = P(k).\lambda \wedge P(k).t > \beta \wedge \\ & Server.T = \langle \min l :: Server.\tau(l)\rangle = \beta \wedge Server.\tau(k) = P(k).t \\ \textbf{ensures} & Server.\lambda = P(k+1 \bmod N).\lambda \end{array} \qquad (\text{MA}_{13})$$

proves (MA_2). This last property is proven directly from the program text by observing that its LHS cannot be changed by any statement, and it enables the guard of the last statement in *Server*. Then, the fairness assumption guarantees execution of such a statement, which establishes the goal.

PROOF OF (MA_{12}). First, we observe that the following holds from invariant ($\mathcal{I}_3$):

$$\begin{array}{l} \langle \min l : k \neq l :: Server.\tau(l)\rangle = \beta \wedge P(k).t \leq \beta \\ \Rightarrow P(k).t \geq \langle \min l :: Server.\tau(l)\rangle. \end{array} \qquad (\text{MA}_{14})$$

If $P(k).t > \langle \min l :: Server.\tau(l)\rangle$ holds, it can be shown directly from the program text that, thanks to invariant (MA_5), the guard of the last statement is false, and then the *Server* is stuck at the client location. The fairness assumption guarantees that $P(k).t = Server.\tau(k) = \langle \min l :: Server.\tau(l)\rangle$ is established eventually, thanks to the first statement in *Server*. Then, for the sake of readability, we introduce the predicate

$$\begin{array}{rl} \mathsf{farMin} \equiv & \langle \min l : k \neq l :: Server.\tau(l)\rangle = \beta \wedge \\ & Server.\tau(k) = P(k).t = \langle \min l :: Server.\tau(l)\rangle \leq \beta \end{array}$$

and demonstrate (MA_{12}) by proving that

$$\begin{array}{ll} & Server.\lambda = P(k).\lambda \wedge \mathsf{farMin} \wedge \delta = \max(\beta - P(k).t, 0) \\ \mapsto & (Server.\lambda = P(k).\lambda \wedge (\mathsf{farMin} \wedge \delta < \max(\beta - P(k).t, 0)) \vee \\ & (Server.T = \langle \min l :: Server.\tau(l)\rangle = \beta \wedge P(k).t > \beta)) \vee \\ & Server.\lambda = P(k+1 \bmod N).\lambda. \end{array} \qquad (\text{MA}_{15})$$

holds using the induction principle of **leads-to**, applied over the distance between the timer $P(k).t$ and the value β of the next smallest element in $Server.\tau$. The above states that either the distance among the minimum and the next minimum value in $Server.\tau$ is reduced, or the value of the local timer becomes greater than the current minimum value, or *Server* departs. The progress property proving induction is provided by the following:

$$\begin{array}{ll} & Server.\lambda = P(k).\lambda \wedge \mathsf{farMin} \\ \textbf{ensures} & Server.\lambda = P(k).\lambda \wedge \mathsf{farMin} \wedge Server.T = \langle \min l :: Server.\tau(l)\rangle = \beta \\ \textbf{ensures} & ((Server.T = \langle \min l :: Server.\tau(l)\rangle = \beta \wedge P(k).t > \beta) \wedge \\ & Server.\lambda = P(k).\lambda) \vee Server.\lambda = P(k+1 \bmod N).\lambda. \end{array}$$

The first **ensures** is proven by ruling out the case where the RHS already holds, and considering the case when the GVT on *Server* is out-of-date.

In this case, the last statement in *Server* is false and hence is keeping the *Server* co-located with the client. Fairness and the second statement in *Server* prove the property. As for the second **ensures**, we observe that when its LHS holds the guard in the last statement of *Server* is enabled. Due to the fairness assumption, the statement executes eventually, thus determining *Server* departure. However, due to the sharing expressed by invariant (MA_6), the statement in *P(k)* is enabled and executes eventually even if *Server* is already departed, since the GVT estimate is retained according to the **disengage** clause. Since no assumption can be made on the scheduling of these statements, according to the order in which they are executed either the first or the second disjunct in the RHS of the **ensures** is established. Repeated application of this progress property when the first disjunct of (MA_{15}) holds leads either to third disjunct, which actually proves our goal, or to the second one. In this latter case, fairness, the transient sharing captured by invariant (MA_5), and the first statement in *Server* guarantee that the *Server* stays at the client location until the timer value mirrored in *Server*.τ equals the one owned by the client, thus leading to the RHS of (MA_{12}).

PROOF OF (MA_6). The invariant holds initially. If the *Server* is initially co-located with *P(k)*, the GVT estimate owned by the latter is undefined, thus preserving the invariant. If the two components are not co-located, then the LHS of the implication does not hold and thus preserves the invariant. As for the statements in the system, the first statement in *P(k)* preserves the invariant as well, in that it possibly undefines the GVT estimate in *P(k)* or leaves it unaffected. Execution of the second statement when *P(k)* and *Server* are co-located actually guarantees that the GVT estimates in *P(k)* and *Server* assume the same value. Finally, the migration expressed by the last statement in *Server* does not affect the invariant, since upon engagement on *P(k)* the **engage** clause of the first interaction guarantees that equality of the two GVT estimates is established, while disengagement falsifies the LHS of the implication and thus preserves the invariant.

12.2 Remote Evaluation

In the solution shown in Figure 12.4, the client components *P(i)* behave similarly to the ones in the *DSClientServer* system. Besides computing a new local timer, clients can also request the execution of a service that computes the new GVT estimate, provided that the old GVT estimate has been already consumed during an earlier timer update and that a message request has not yet been sent. The corresponding reply message is eventually collected and its fields are checked to verify that the recipient's and receiver's addresses match. In this case, the new estimate becomes defined and available for the assignment that updates the timer, while the message buffer is reset to enable further requests—just like in the CS solution. The key difference between the two solutions is the second parameter in the message request

$$RQ := \langle \text{SERVER}, \text{REV}, \text{MINSERV}, \bot \rangle \ \textbf{if} \ \neg\mathsf{def}(RQ) \wedge \neg\mathsf{def}(T),$$

```
System DSRemoteEvaluation
    Program P(i) at λ
        declare
            t,z : integer [] T : integer ∪ {⊥} [] RQ : request ∪ {⊥}
        initially
            t = 0 [] T = ⊥ [] λ = Location(i) [] RQ = ⊥
        assign
            [] t,T := f_i(t,T,z),⊥                  if def(T)
            [] RQ := ⟨SERVER,REV,MINSERV,⊥⟩         if ¬def(RQ) ∧ ¬def(T)
            [] T,RQ := RQ ↑ 4,⊥                     if RQ ↑ 1 = i
    end
    Program Min(i) at λ
        declare
            t : integer [] T : integer [] τ : (array of integer) ∪ {⊥} [] q : request ∪ {⊥}
        initially
            t,T = 0,0 [] τ = ⊥ [] q = ⊥ [] λ = Location(i)
        assign
            τ(i) := t                     if def(τ)
            [] T := ⟨min k :: τ(k)⟩        if def(τ)
            [] q := ⟨i,⊥,⊥,T⟩              if τ(i) = t ∧ T = ⟨min k :: τ(k)⟩
    end
    Program Server at λ
        declare
            τ : array of integer [] q : array of (request ∪ {⊥}) []
        initially
            ⟨|| j :: τ(j) = 0⟩ [] ⟨|| j :: q(j) = ⊥⟩ [] λ = Location(SERVER)
    end
    Components
            ⟨[] i :: P(i)⟩ [] Server [] ⟨[] i :: Min(i)⟩
    Interactions
            Min(i).t ← P(i).t               when Min(i).λ = P(i).λ
                                            engage P(i).t
            [] Min(i).λ := Server.λ         when Min(i).λ = P(i).λ ∧
                                                 serviceRequest(REV,MINSERV,i)
            [] Min(i).τ ≈ Server.τ          when Min(i).λ = Server.λ
                                            engage Server.τ
                                            disengage ⊥, Server.τ
            [] Min(i).q ≈ Server.q(i)       when Min(i).λ = Server.λ
                                            engage ⊥
            [] P(i).RQ,Server.q(i)[,Min(i).λ] := Server.q(i),⊥[,P(i).λ]   when serviceReady(i)
end
```

Figure 12.4: Remote Evaluation solution for the distributed simulation problem.

specifying that the code for the service named MINSERV must be sent to the *Server*'s location, where its execution exploits bindings with *Server*'s resources. In contrast with the CS solution, the request message is not delivered to the message buffer of the *Server*. Its presence within the client buffer enables subsequent code migration. The code for service MINSERV is described by the program *Min(i)* and is structured in three steps: update of the global state maintaned in $Server.\tau$ with the timer value of the corresponding client *P(i)*, computation of the new GVT, and communication of the results to the client. All these steps can take place only when *Min(i)* and *Server* are co-located. In this situation, the statement

$$\begin{array}{ll} Min(i).\tau \approx Server.\tau & \textbf{when } Min(i).\lambda = Server.\lambda \\ & \textbf{engage } Server.\tau \\ & \textbf{disengage } \bot,\ Server.\tau \end{array}$$

in the **Interactions** section specifies a transient sharing between the representation of the global state τ owned by *Min(i)* and *Server*. Upon departure, the **disengage** clause voids the value of τ in *Min(i)*, thus preventing execution of any statement within this component while it is not at the *Server*'s location.

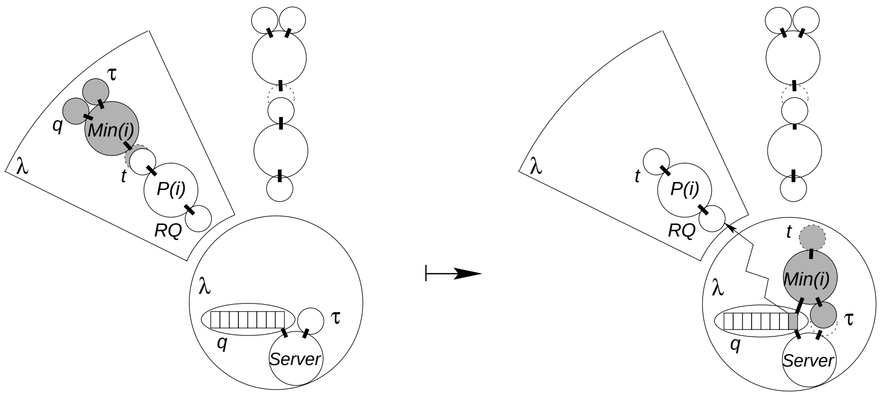


Figure 12.5: A graphical representation of the REV solution.

Upon a subsequent arrival, the τ values owned by the two components are reconciled by the **engage** clause, which assigns to $Min(i).\tau$ the up-to-date τ value kept on *Server*. Updating a client's timer value stored in τ is made possible by the interaction

$$Min(i).t \leftarrow P(i).t \qquad \textbf{when } Min(i).\lambda = P(i).\lambda \quad \textbf{engage } P(i).t$$

which specifies that the timer value in a client *P(i)* is shared with the one in the corresponding *Min(i)*—as long as they are co-located. Upon departure of *Min(i)* from *P(i)*'s location, *Min(i)* retains the current value[1] of *t*, which is used at the *Server* location to update of the global state. The actual code migration happens when

$$Min(i).\lambda := Server.\lambda \qquad \textbf{when } Min(i).\lambda = P(i).\lambda \wedge \mathsf{serviceRequest}(\textsc{rev},\textsc{minServ},i)$$

is eventually executed after a client has sent a message request.

At this point, it is worth noting how, in contrast with the CS solution, the REV request does not contain information that needs to be passed to the *Server*. However, in both solutions the action of putting a message in the request buffer *RQ* can be regarded as modeling the invocation of a communication primitive, which enables actions in the underlying run-time support. In the CS solution, these actions are represented by the transfer of the message to the *Server*; in the REV solution, the actions encompass migration of the *Min(i)* component to the *Server*'s location. Hence, in both solutions the client component is given a uniform interface (the request buffer) to the rest of the system, whose details are handled within the **Interactions** sections—thus modeling the run-time support for communication provided by, say, implementations of RPC and of the REV [101] system. Analogous considerations hold for the output buffer of *Server*. In the REV solution,

$$Min(i).q \approx Server.q(i) \qquad \textbf{when } Min(i).\lambda = Server.\lambda \quad \textbf{engage } \perp$$

[1]Mobile code languages implement parameter passing either explicitly by referring to input parameters and code using RPC-like primitives or implicitly by attaching to the procedure to be executed remotely the portion of data space needed for remote computation [30]. We chose the second alternative, in order to illustrate how to dynamically establish and remove bindings among variables in Mobile UNITY.

models the fact that *Min(i)* is somewhat given access to the communication facilities co-located on the *Server*; the last interaction, in turn, models the actual transfer of information provided by the underlying run-time support. Note that in our REV solution the *Server* does not provide any service, except for offering a name space in which each *Min(i)* can have access to the global state by sharing τ and to the communication facilities by sharing the message buffer q.

Finally, in the last statement of the system the message replies pending in the output queue are sent to the corresponding client. This is accomplished as in the CS solution, except for the assignment

$$[Min(i).\lambda := P(i).\lambda] \quad \textbf{when } \mathsf{serviceReady}(i).$$

This statement is enclosed in square brackets to highlight the fact that it is not a direct consequence of the REV paradigm, yet is needed because of the way Mobile UNITY is currently defined. The REV paradigm involves migration of a copy of a component's code. After that, a message containing the result of the computation on the server is sent back to the client, and what happens to the code remaining on the server is left to the implementation. On the other hand, in the **Component** section of the solution presented we create statically N components which are initialized with their own data and control state. These components are unique in the system, consequently they must return to the client's location in order to become available for another message request—dynamic instantiation of components is not yet available in Mobile UNITY. This issue will be revisited in Section 12.4.

Verification

$P(k).t = \langle \min l :: Server.\tau(l) \rangle = \beta \mapsto P(k).t = P(k).T = \beta$	$(\mathcal{L}_2)$
$P(k).t = P(k).T = \beta$ **ensures** $P(k).t > \beta$	$(\mathcal{L}_3)$
$P(k).t = \beta \wedge Server.\tau(k) < \beta$ **until** $P(k).t = Server.\tau(k) = \beta$	$(\mathcal{L}_4)$
inv. $Min(k).\lambda = P(k).\lambda \vee Min(k).\lambda = Server.\lambda$	(REV_1)
inv. $Min(k).\lambda = P(k).\lambda \Rightarrow \neg\mathsf{def}(Min(k).q)$	(REV_2)
inv. $\mathsf{def}(Min(k).\tau) \Leftrightarrow Min(k).\tau = Server.\tau$	(REV_3)
inv. $Min(k).t = P(k).t$	(REV_4)
inv. $Min(k).\lambda = Server.\lambda \Leftrightarrow Min(k).\tau = Server.\tau$	(REV_5)
inv. $true \Rightarrow \mathsf{def}(Server.\tau)$	(REV_6)
inv. $P(k).t = P(k).T = \beta \Rightarrow \langle \min l :: P(l).t \rangle = \beta$	$(\mathcal{I}_2)$
inv. $Server.\tau(k) \leq P(k).t$	$(\mathcal{I}_3)$
inv. $\mathsf{def}(P(k).T) \Rightarrow P(k).T \leq \langle \min l :: Server.\tau(l) \rangle$	$(\mathcal{I}_4)$
inv. $\neg\mathsf{def}(Min(k).\tau) \Rightarrow \neg\mathsf{def}(Min(k).q)$	(REV_7)
inv. $Min(k).\lambda = Server.\lambda \Rightarrow \neg\mathsf{def}(P(k).T)$	(REV_8)
inv. $Min(k).\lambda = Server.\lambda \Leftrightarrow \mathsf{def}(Min(k).\tau)$	(REV_9)
inv. $true \Rightarrow \mathsf{def}(Min(k).t)$	(REV_{10})
inv. $\neg\mathsf{def}(Server.q(k)) \Leftrightarrow \neg\mathsf{def}(Min(k).q)$	(REV_{11})
inv. $P(k).RQ \uparrow 1 = k \Rightarrow Min(k).\lambda = P(k).\lambda$	(REV_{12})

Table 12.2: Lemmas and invariants needed for verification of the Remote Evaluation solution.

Figure 12.2 shows the lemmas and invariants needed to prove the REV solution. In this section, we provide the proof outlines for some of them, building upon what we already demonstrated in the previous sections. The proof outlines for the remaining obligations are provided in Appendix A.3.

PROOF OF ($\mathcal{L}_2$). The proof of this lemma is similar to the one derived for the CS solution, hence we provide only an informal description for it. We rule out the case $P(k).T = \beta$ by applying the implication theorem. Then, we are left with two cases, according to (REV_1). With the definitions provided in Section 11.2.2, it can be proven that

$$\begin{array}{ll} & \mathsf{minTimer} \wedge \mathsf{oldClientT} \wedge Min(k).\lambda = P(k).\lambda \\ \mapsto & \mathsf{minTimer} \wedge \neg\mathsf{oldServerT} \wedge Min(k).\lambda = P(k).\lambda \wedge \\ & P(k).RQ = \langle \text{SERVER,REV,MINSERV,}\bot \rangle \end{array}$$

with arguments similar to those used in proving (CS_9), together with invariant (REV_2). The presence of a client request triggers migration of the corresponding *Min(k)* process to get co-located with the *Server*, because of the fairness assumption. Once co-located, invariant (REV_3) guarantees that the first two statements are enabled, thus enabling update of the GVT estimate. If the LHS of the lemma holds, actually no one else can execute actions and thus changing the GVT estimate, because $P(k).t$ is the minimum timer. Thus, the *Min(k)* remains co-located with *Server* until the last statement in *Min(k)* is executed. It can be proven, analogously to (CS_{11}), that such statement stores a value in the *Server* message queue, which is eventually communicated to *Min(k)* and there assigned to client's GVT estimate, thus satisfying the goal.

PROOF OF ($\mathcal{L}_3$). This lemma is proven with the same arguments used in Section 11.2 for the CS solution. In fact, it needs only reasoning about a single client, whose behavior is unchanged.

PROOF OF ($\mathcal{L}_4$). The **unless** part of the property is proven as in the CS solution. As for the **leads-to** part, we consider two cases according to invariant (REV_1). Our proof strategy is to show that:

1. if *Min(k)* and *P(k)* are co-located, *Min(k)* eventually migrates to *Server*'s location,

$$\begin{array}{ll} & P(k).t = \beta \wedge Server.\tau(k) < \beta \wedge Min(k).\lambda = P(k).\lambda \\ \mapsto & P(k).t = \beta \wedge Server.\tau(k) < \beta \wedge Min(k).\lambda = Server.\lambda; \end{array} \quad (\text{REV}_{13})$$

2. if *Min(k)* and *Server* are co-located, then

$$\begin{array}{ll} & P(k).t = \beta \wedge Server.\tau(k) < \beta \wedge Min(k).\lambda = Server.\lambda \\ \mapsto & P(k).t = Server.\tau(k) = \beta \end{array} \quad (\text{REV}_{14})$$

holds, expressing the fact that the value of the local timer of *P(k)* is eventually communicated to *Server*.

Disjunction and transitivity of (REV_{13}) and (REV_{14}) complete the proof.

PROOF OF (REV$_{14}$). Invariant (REV$_4$) guarantees that *Min(k)* holds the proper value to be communicated to the *Server*. In addition, invariant (REV$_5$) guarantees that each update made to the copy of the global state belonging to *Min(k)* is seen by *Server* as well. The proofs for invariants (REV$_4$) and (REV$_5$), which both involve reasoning about transient sharing, are given later in the remainder of this section. The implication theorem ($\mathcal{T}_5$), together with the two invariants above, yields

$$\begin{array}{ll} & P(k).t = \beta \wedge Server.\tau(k) < \beta \wedge Min(k).\lambda = Server.\lambda \\ \mapsto & P(k).t = \beta \wedge Server.\tau(k) < \beta \wedge Min(k).\lambda = Server.\lambda \wedge \\ & Min(k).\tau(k) = Server.\tau(k) \wedge Min(k).t = P(k).t \end{array}$$

From the program text, it can be argued that

$$\begin{array}{ll} & P(k).t = \beta \wedge Server.\tau(k) < \beta \wedge Min(k).\lambda = Server.\lambda \wedge \\ & Min(k).\tau(k) = Server.\tau(k) \wedge Min(k).t = P(k).t \\ \textbf{ensures} & P(k).t = \beta \wedge Min(k).\lambda = Server.\lambda \wedge Min(k).t = Min(k).\tau(k) \wedge \\ & Min(k).\tau(k) = Server.\tau(k) \wedge Min(k).t = P(k).t \end{array}$$

which clearly leads to the goal by application of the implication theorem.

PROOF OF (REV$_{13}$). This is proven similarly to (CS$_{13}$), by application of disjunction according to invariant ($\mathcal{I}_2$) and of transitivity to the following chain of **ensures**:

$$\begin{array}{ll} & P(k).t = \beta \wedge Server.\tau(k) < \beta \wedge Min(k).\lambda = P(k).\lambda \\ \textbf{ensures} & P(k).t = \beta \wedge Server.\tau(k) < \beta \wedge Min(k).\lambda = P(k).\lambda \wedge \neg \mathsf{def}(P(k).T) \\ \textbf{ensures} & P(k).t = \beta \wedge Server.\tau(k) < \beta \wedge Min(k).\lambda = P(k).\lambda \wedge \\ & P(k).RQ = \langle \text{SERVER,CS,MINSERV},\beta \rangle \\ \textbf{ensures} & P(k).t = \beta \wedge Server.\tau(k) < \beta \wedge Min(k).\lambda = Server.\lambda. \end{array}$$

PROOF OF (REV$_3$). The invariant holds initially, from the program text. If the LHS of the invariant holds before execution of the first statement, invariant (REV$_9$) guarantees that *Min(k)* and *Server* are co-located. Hence, the sharing expressed by the third interaction is in place, and the invariant is preserved after execution. On the other hand, if the LHS does not hold, the statement is disabled and the invariant is preserved. Migration statements preserve the invariant as well, because engagement and disengagement preserve the invariant either by establishing its RHS or by falsifying its LHS. No other statement can affect the invariant.

PROOF OF (REV$_4$). The only statement that may affect this invariant is the first one in *P(k)*. However, invariant (REV$_8$) let us conclude that, if (REV$_4$) and $Min(k).\lambda = Server.\lambda$ hold before execution of the statement, (REV$_4$) is preserved because the statement is disabled. On the other hand, if (REV$_4$) holds before execution of the statement while $Min(k).\lambda = P(k).\lambda$, then the first interaction takes place as soon as $P(k).t$ is updated, changing the local timer in *P(k)* accordingly. Migration statements do not affect the invariant, although in principle they could involve the read only shared variable in the first interaction. However, the **engage** clause in the first interaction actually establishes (REV$_4$). Furthermore, no **disengage** clause is specified, hence the value of the variables as specified by the invariant before disengagement is retained.

PROOF OF (REV_5). The invariant holds as the conjunction of (REV_3) and (REV_9).

PROOF OF (REV_9). The invariant holds initially, since *Min(k)* and *P(k)* are co-located and τ is undefined in *Min(k)*—both sides of (REV_9) are false. Looking at the program text, the only statements which may affect the invariant are those involving migration of *Min(k)* and the ones modifying $Min(i).\tau$. Let us consider the second statement in the **Interactions** section. If the LHS of (REV_9) holds before execution, the statement is disabled. On the other hand, if the LHS does not hold, $Min(k).\tau$ must be undefined in order to preserve the invariant. Execution of the statement forces *Min(k)* to become co-located with *Server* and causes engagement of the τ variables belonging to *Min(k)* and *Server*. Because of invariant (REV_6), τ becomes defined in *Min(k)*, thus preserving the invariant. Similar reasoning holds for the last statement of the system. If the LHS and the RHS of (REV_9) are both false, invariant (REV_7) prevents execution of the statement and thus preserves the invariant. On the other hand, if both sides of (REV_9) are true, the migration expressed by the bracketed statement take place, thus triggering disengagement of the values of τ. As a consequence, after execution of the statement the LHS of the invariant is false and, according to **disengage** clause of the third interaction, τ becomes undefined in *Min(k)*, thus falsifying the RHS as well and preserving the invariant. Finally, let us consider the first statement in *Min(k)*. If the LHS of (REV_9) does not hold, the RHS is false as well, thus preventing execution of the statement and preserving the invariant. On the other hand, if *Min(k)* and *Server* are co-located, the invariant states that before execution of the statement τ is defined in *Min(k)*. However, execution of the statement must comply with invariant (REV_{10}), and thus preserves (REV_9).

12.3 Code On Demand

We present the solution exploiting the COD paradigm by enhancing the CS solution shown earlier. In the system shown in Figure 12.6, clients *P(i)* are augmented with some statements that enable them to request the code needed to compute dynamically the simulation mode. The requests are issued when a given condition is established, which is modeled by the assignment setting the variable *static* to *false*. This variable is *true* initially, thus the clients initially behave like the ones in *DSClientServer*, which use the initial value $\bar{z}$ for the simulation mode. When *static* becomes *false*, the client is enabled to issue a request

$$RQ := \langle \text{SERVER,COD,DYNMODE},\perp\rangle \ \textbf{if} \ \neg\mathsf{def}(RQ) \wedge \mathit{static} = \mathit{false}$$

in order to make the code for the service DYNMODE available to *P(i)*. This code is contained in program *Dyn(i)*, which simply contains an assignment to update the simulation mode by evaluating the function $d(T)$—provided that the GVT estimate is currently defined in *P(i)*. We assume that, once the simulation mode is computed dynamically, it can no longer be reverted to a statically determined value. Although the latter situation can be modeled, we choose this assumption for the sake of simplicity. The *Server* component is left unmodified with respect to the CS solution, while N components *Dyn(i)* are instantiated in the **Components** section—for reasons similar to those explained for the

System *DSCodeOnDemand*
  Program *P(i)* **at** λ
    declare
      t,z : integer $[]$ T : integer $\cup\{\perp\}$ $[]$ $static$: boolean $\cup\{\perp\}$ $[]$ RQ : request $\cup\{\perp\}$
    initially
      $t = 0$ $[]$ $T = \perp$ $[]$ $\lambda = \mathsf{Location}(i)$ $[]$ $static = true$ $[]$ $RQ = \perp$
    assign
      $[]$ $static := false$ **if** $\mathsf{def}(static)$
      $[]$ $t,T := f_i(t,T,z),\perp$ **if** $\mathsf{def}(T)$
      $[]$ $RQ := \langle \text{SERVER},\text{CS},\text{MINSERV},t\rangle$ **if** $\neg\mathsf{def}(RQ) \wedge \neg\mathsf{def}(T)$
      $[]$ $T,RQ := RQ \uparrow 4,\perp$ **if** $RQ \uparrow 1 = i$
      $[]$ $RQ := \langle \text{SERVER},\text{COD},\text{DYNMODE},\perp\rangle$ **if** $\neg\mathsf{def}(RQ) \wedge static = false$
  end
  Program *Dyn(i)* **at** λ
    declare
      z : integer $[]$ T : integer $\cup\{\perp\}$
    initially
      $\lambda = \mathsf{Location}(\text{SERVER})$
    assign
      $z := d(T)$ **if** $\mathsf{def}(T)$
  end
  Program *Server* **at** λ
    declare
      T : integer $\cup\{\perp\}$ $[]$ τ : array of integer $[]$ q : array of (request $\cup\{\perp\}$)
    initially
      $T = \perp$ $[]$ $\langle \| j :: \tau(j) = 0\rangle$ $[]$ $\langle \| j :: q(j) = \perp\rangle$ $[]$ $\lambda = \mathsf{Location}(\text{SERVER})$
    assign
      $\langle [] j :: \tau(j),q(j) \uparrow 2 := q(j) \uparrow 4,\text{WAIT}\rangle$ **if** $q(j) \uparrow 1 = \text{SERVER} \wedge q(j) \uparrow 2 \neq \text{WAIT}$
      $[]$ $T := \langle \min k :: \tau(k)\rangle$ **if** $\langle \exists j :: q(j) \uparrow 2 = \text{WAIT}\rangle \wedge \neg\mathsf{def}(T)$
      $[]$ $\langle \| j :: q(j),T := \langle j,\perp,\perp,T\rangle,\perp$ **if** $\mathsf{def}(T) \wedge q(j) \uparrow 2 = \text{WAIT}\rangle$
  end
  Components
      $\langle [] i :: P(i)\rangle$ $[]$ $Server$ $[]$ $\langle [] i :: Dyn(i)\rangle$
  Interactions
      $Dyn.\lambda,P(i).static := P(i).\lambda,\perp$ **when** $Dyn(i).\lambda = Server.\lambda \wedge \mathsf{serviceRequest}(\text{COD},\text{DYNMODE},i)$
      $[]$ $P(i).z \leftarrow Dyn(i).z$ **when** $P(i).\lambda = Dyn(i).\lambda$ **engage** $P(i).z$
      $[]$ $Dyn(i).T \leftarrow P(i).T$ **when** $P(i).\lambda = Dyn(i).\lambda$ **engage** $P(i).T$
      $[]$ $Server.q(i) := P(i).RQ$ **when** $\neg\mathsf{def}(Server.q(i)) \wedge \mathsf{serviceRequest}(\text{CS},\text{MINSERV},i)$
      $[]$ $P(i).RQ,Server.q(i) := Server.q(i),\perp$ **when** $\mathsf{serviceReady}(i)$
end

Figure 12.6: Code On Demand solution for the distributed simulation problem.

REV solution. Within the **Interactions** section, the last two statements are unchanged and manage the message exchanges needed to compute the GVT estimate as in the CS paradigm. In turn, the statement

$$Dyn.\lambda,P(i).static := P(i).\lambda,\perp \quad \textbf{when } Dyn(i).\lambda = Server.\lambda \wedge \mathsf{serviceRequest}(\text{COD},\text{DYNMODE},i)$$

satisfies a code request issued by a client *P(i)* by changing the location of the corresponding component *Dyn(i)*. Furthermore, it prevents further changes in the way z is computed by setting *static* to undefined—which permanently disables the statement issuing the request. Finally,

$$P(i).z \leftarrow Dyn(i).z \quad \textbf{when } P(i).\lambda = Dyn(i).\lambda \;\; \textbf{engage } P(i).z$$
$$Dyn(i).T \leftarrow P(i).T \quad \textbf{when } P(i).\lambda = Dyn(i).\lambda \;\; \textbf{engage } P(i).T$$

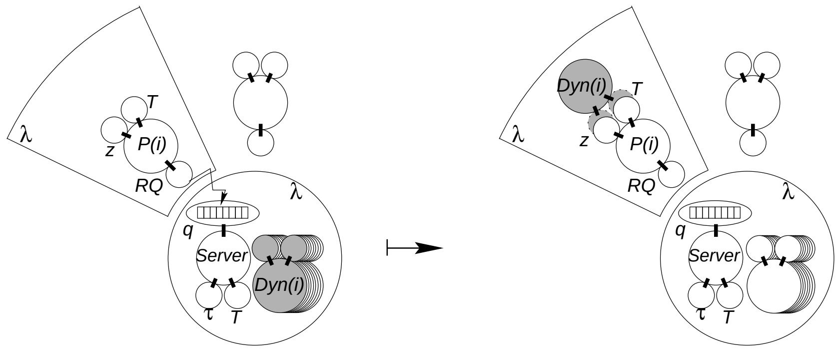


Figure 12.7: A graphical representation of the Code On Demand solution.

specify the bindings established between *P(i)* and *Dyn(i)* when they are co-located. The **engage** clauses initialize the values of z and T in *Dyn(i)* with the corresponding values in *P(i)*. A graphical representation of this system is shown in Figure 12.7. As in the REV solution, we are forced to instantiate statically multiple components from the same program, instead of migrating the code and instantiate components only when and if needed. This and other issues raised by the solutions presented so far are be discussed in the next section. We will not provide any verification of the COD solution, since the computation of z does not affect the correctness of the properties stated in Section 11.2.

12.4 Discussion

Mobile UNITY is a new model of distributed computing specialized for mobile computations, i.e. for systems in which components travel through space, compute in a decoupled fashion, and communicate opportunistically when co-located. Mobile UNITY provides a notation system for capturing mobility and an assertional proof logic. Research on Mobile UNITY has shown that a small number of constructs suffices to express transitive forms of transient data sharing and transient synchronization. Restricted forms of these proposed interaction constructs appear to have efficient implementations and more abstract and powerful interaction constructs can be built from the basic forms. In addition, the proof logic has been tentatively evaluated in the verification of the Mobile IP protocol [71].

Against this background of promising technical developments, this part of the thesis raised a simple question: Can Mobile UNITY model in straightforward manner the kinds of interactions that take place in applications involving mobile code? The question is reasonable in light of the fact that Mobile UNITY makes no explicit distinction between physical and logical movement of components. A positive answer would allow the immediate application of its proof logic to mobile code and may also define a more abstract and objective basis for a different kind of taxonomy of mobile code paradigms. A negative answer would lead to a clarification of possible fundamental differences between the domains of mobile code and mobile components or could reveal possible shortcomings in the way Mobile UNITY was conceived.

The investigative style of the work described here is empirical. We started with established mobile code paradigms and sought out corresponding Mobile UNITY solutions. The decoupled style of computation promoted by Mobile UNITY appears to be a good match for the realities of mobile code. The **Interactions** section was able to encapsulate appropriately the communication taking place between components. Asynchronous data transfer had a direct counterpart in Mobile UNITY and code movement was easily expressed by the same mechanisms by which components change location. Because the only notion of blocking in Mobile UNITY is busy waiting, blocking for responses to requests was naturally captured by tagging relevant variables as not being available (undefined) and strengthening the guards of related statements to check for availability of the data. In several cases we used the fact that the variable was no longer available as the cause for generating a request in the first place—this led to elegant separation between actions embedded in the application program and those supplied by the run-time support. Finally, the transient sharing constructs offered a good solution for the data binding process that needs to take place when a mobile code fragment arrives at a new location. Since the mobile code is treated as a program having its own internal state, the movement of code can be accompanied by data movement. The **engage** feature of transient variable sharing encapsulates the binding process while the **disengage** plays a role in implementing policies that define how much state information may be carried along by a departing code fragment. When a piece of code carries no data state, for instance, the disengagement reinitializes all its shared variables.

In addition to a precise modeling of the characteristics of mobile code, we were also able to reason about such models and prove properties of their expected behavior. This has been accomplished by using the UNITY proof logic, whose corpus of theorems and techniques is still available within Mobile UNITY. The novel notion of sharing has been proficiently taken into account by defining safety properties which embed the constraints expressed by transient sharing among components.

The only possible mismatch identified by this case study has to do with dynamic instantiation of code segments. In the REV solution, for instance, there is no need to "return" the code being evaluated as we do in our example. New fresh copies can be sent each time and several copies may co-exist on different servers. In Mobile UNITY, however, the set of components making up a system is fixed. Further research is needed to evaluate this issue. One solution that requires no change to the Mobile UNITY notation is to create an unbounded set of clones (uniquely indexed) that are placed in a stand-by state until needed. This could have some negative implications on verification and, if not considered carefully, could interfere with the fairness assumption which are at the foundation of the proof logic. Another option involves building upon the experience with Swarm [91, 31], a version of UNITY in which both data and statements are dynamically created and destroyed. The prospect of making changes to Mobile UNITY may also force us to re-examine the issue of what is an appropriate unit of mobility. So far we selected the program to play this role but a finer grained solution at the statement level, for instance, may also be appropriate to consider.

Final Remarks on Formalizing Code Mobility

Mobile UNITY has been the product of three distinct mindsets. First, it was the notion of developing transient and transitive forms of the two UNITY modes of composition, union and superposition. Second, it was the fascination with ad-hoc networks of components moving freely through space, computing in a highly decoupled style, and interacting opportunistically. Neither of the two motivations had anything to do with code mobility. Yet, our empirical evaluation of the model against several established paradigms in mobile code languages shows that the essential features of code mobility have immediate counterparts in Mobile UNITY thus allowing direct applicability of its proof logic to programs involving mobile code. While more work is needed to refine the relation between code and (physical) component movement, this exercise in modeling and formal reasoning provides a strong incentive to seek out solutions and models which are insensitive to the nature of the movement process, be it logical or physical.

Part IV

Exploiting Code Mobility

Chapter 13

Motivations, Background, and Approach

In this part of the thesis we collect various applications of the concepts, techniques, and insights that have been developed so far in this thesis. All the activities described in the following three chapters are still under development.

The first chapter presents a mobile code toolkit that is built upon the considerations developed in the first two parts. In fact, the motivation for it lies in the fact that currently available MCSs do not provide sufficient support for all the strategies of mobile code relocation that have been described in Chapter 2. As we discussed in Part II, using abstractions improperly may generate unnecessary verbosity in the mobile code and thus reduce the effectiveness of the approach. In addition, it also considers other insights concerned with performance provided by our evaluation and strives for a lightweight implementation, in order to minimize the overhead on the network and the computing platform.

The toolkit is being used successfully in the application described in the second chapter of this part of the thesis. As discussed in Chapter 4, there exist very few mobile code applications besides Java applets embedded in browsers. As a consequence, there exist even fewer reports about the benefits of a mobile code design in a real application domain. In the aforementioned chapter we describe the design of a videoconferencing system for active networks. Videoconferencing is a widespread distributed application and yet it poses considerable challenges in terms of performance. Active networks, as discussed in Chapter 4, can be regarded as one of the application areas for code mobility, and aim at changing radically the shape of computer networks by introducing the possibility to send active packets into the network and change dynamically the behavior or network nodes to better suit users' needs. However, like code mobility, applications exploiting active networks are still missing and the work described in this chapter is novel in this respect.

The third chapter presents a system that tries to unveil the relationship that exist between the notion of physical mobility fostered by mobile computing with the notion of logical mobility represented by mobile code. The system treats both kinds of mobility in the same way, and provides coordination among mobile components with the same abstractions. The model is inspired by work on Linda, which is extended with a minimal set of primitives needed to cope with mobility. The formal semantics for the system has been specified using the Mobile UNITY notation already discussed in Part III.

The three chapters provide very diverse perspective on code mobility, being concerned respectively with the development of a novel technology, with the design of mobile code applications, and with the synergies between mobile code and mobile computing with emphasis on formal models. However, it is interesting to note how all of them leverage off of the body of concepts that have been developed in the previous parts of the thesis.

Chapter 14

μCode: A Lightweight and Flexible Mobile Code Toolkit

The evaluation of code mobility described in Part II has provided qualitative and quantitative information about the effectiveness of an approach to application design that exploits mobile code. In particular, the case study in network management presented in Section 8 provided insights about the selection of the technology best suited for implementation of mobile code paradigms. This chapter builds upon the lessons learned in carrying out the aforementioned case study and presents a mobile code toolkit called[1] μCode conceived to be used in conjunction with the Java language and to overcome some of the drawbacks of currently available MCSs. The system is undergoing costant extension. However, the version described here has been used succesfully in the development of the application described in Chapter 15.

This chapter is structured as follows. Section 14.1 analyzes the rationale and motivations for the development of μCode, illustrating the key principles that have driven its design. Section 14.2 introduces some general concepts and provides an overview of the main features of the toolkit. Section 14.3 describes in detail the API. Section 14.4 reports about the architecture of the toolkit and some implementation details. Finally, Section 14.5 contains some concluding remarks and discusses options for future work.

A reduced version of the work presented in this chapter can be found in [86].

14.1 Rationale and Motivation

The experience described in Chapter 8 was focused on network management and on a particular set of technologies. As far as support for mobile code

[1] The μ in μCode can be interpreted in different ways. μ is the symbol for "micro" and thus refers to the fact that the toolkit is lightweight and small. Along this lines, our toolkit can be regarded as providing the "microcode" for systems providing higher levels of abstraction, much like the microcode in a processor. Finally, μ is the Greek letter for "m", that happens nicely to be the initial of "mobile", thus yielding "mobile code" as the complete label.

is concerned, we chose to exploit the Java Aglets API, considered by many researchers to be among the most reliable, supported, and complete mobile code platform currently available. However, this MCS evidenced several drawbacks during our case study. The remarks we make in the following are focused on this particular technology, but most of them apply to many other Java-based MCSs, like Mole or Odyssey.

A first important issue is the lack in flexibility in the strategies provided to relocate code. As we discussed in Section 2.6, Java Aglets provide support for weak mobility in terms of shipping and fetching of stand-alone code. In addition, since they rely on the Java VM, Aglets provides support also for fetching of code fragments. However, the strategies that rule relocation of classes are not completely under the control of the programmer when it specifies that a given aglet has to be moved on another node. In order for this operation to succeed, the class of the aglet must satisfy the constraints illustrated in Table 14.1. However, this constraints are specified *statically* at setup time, by setting properly the environment variables `CLASSPATH` and `AGLET_EXPORT_PATH`, and cannot be easily changed dynamically. Moreover, there is no means to transfer on the destination site all the classes needed by the aglet in one shot—missing classes are dynamically retrieved and linked using the class loader mechanism. An example

	C ∈ CLASSPATH	C ∉ CLASSPATH
C ∈ AGLET_EXPORT_PATH	C cannot be shipped, but can be fetched	C can be shipped and can be fetched
C ∉ AGLET_EXPORT_PATH	C can be neither shipped nor fetched	C can be shipped but cannot be fetched

Table 14.1: How to constrain migration of classes with Java Aglets.

can clarify why this may be a problem. As we discussed in Part II, one of the possible applications of mobile code in general and of a Mobile Agent paradigm in particular is the capability to minimize communication over high-cost links, where the notion of cost may be associated with availability of bandwidth, probability of disconnection, security concerns, or just the actual money cost of using the link. The asset of the MA paradigm lies in the fact that it is possible to perform a task remotely in a disconnected fashion, i.e. without any need for the client side to remain connected for the whole duration of the task. For instance, it is meaningful to send all the classes for the aglet in one shot on a GSM connection at 9,600 baud, in order to minimize communication on the expensive link and leverage off of disconnected operations. However, in the context of the same application it can be also meaningful to send the aglet class only if we are now traversing a local Ethernet, in order to minimize the computational overload at loading time on the remote node and exploit the capability to load classes on demand using the fast link. In other words, *the strategy adopted for the relocation of code is determined by the environment and is not necessarily a property of the code being relocated.* In our example, the same classes are relocated with different strategies according to the characteristics of the communication link the aglet is traversing. Clearly, the impossibility to specify dynamically the constraint on relocation of classes and to limit dynamic linking hampers the exploitation of this important application of the MA paradigm.

Another issue is the level of abstraction provided by the language. Java Aglets, and similarly other Java-based MCSs, provide uniquely the agent abstraction as fundamental in the language, and the whole API is nicely centered around it. However, an application exploiting mobile code does not necessarily need all the power contained in this abstraction. If this is the case, the semantic richness of the agent concept can backfire on the programmer, who is forced to express lower level concepts in a higher level framework. For instance, let us imagine a scenario where a site must be upgraded with new classes *proactively* rather than *reactively* as allowed by the class loading mechanism. This can be achieved by creating an agent on the local site and dispatching it on the destination site, where it installs the class. This is convenient if several sites must be upgraded at the same time since the agent can perform the task autonomously, but seems artificial if only a site needs to be upgraded. In this case, it is desirable to use lower level primitives that deal directly with the relocation of classes and support the shipping and fetching of classes. Similarly, the implementation of a REV paradigm using only the agent abstraction seems unnatural. In many Java-based systems, the only means to accomplish this is to create locally a new agent, dispatch it to destination, and code explicitly within the agent behavior the communication of the results of the remote evaluation. Again, a different set of primitives may be appropriate, even if they are characterized by a lower level of abstraction.

The considerations expressed above raise an issue of communication performance as well. As evidenced in our evaluation, the size of the code being sent on the nodes is a key factor in determining the threshold after which mobile code becomes useful in reducing network traffic. Keeping the code as compact as possible is then one of the goals, and abstractions enforcing unnecessary verbosity are undesired. However, another key factor affecting performance is the overhead introduced by communication protocols. Network and transport level protocols are usually part of the system requirements: an application developed for the Internet is certainly running on top of IP using either UDP or TCP as a transport protocol. However, the application level protocol used to transfer mobile code belongs to the space of choices the API designer has control on. In this respect, different systems use different protocols. For instance, the Mole system relies directly on the serialization provided by Java [107]; the Odyssey system adopts this solution as a default by relying on Java Remote Method Invocation [108], but supports also DCOM and the CORBA-compliant IIOP protocol. As we discussed, Java Aglets exploit a customized Agent Transfer Protocol (ATP) which increases significantly the transmission overhead. Clearly, the overhead of application protocols should be minimized, and in this respect Java serialization provides a good tradeoff between compactness of the encoding and ease of programming and customizability.

Given these fundamental considerations, our toolkit is designed around the following principles:

Minimality The toolkit aims at identifying and providing a minimal set of primitives and abstractions to express several different strategies of code relocation.

Extensibility μCODE can be regarded as a thin layer upon the Java VM, providing support for code mobility. This layer can be either exploited directly by applications or it can provide the building blocks to construct higher-level layers of abstractions.

Flexibility This has in turn different facets:

Control over relocation The toolkit relies on the aforementioned principle that the programmer should be able to select different strategies for the same code in different situations.

Control over the run-time support The run-time support should be under the control of the programmer, and not an external piece of software.

Access to different levels of abstractions The programmer should be able to have different levels of control upon the mobile code, exploiting the high-level constructs provided with the toolkit as well as resorting to the lower-level constructs when some particular or customized functionality is needed.

Non-invasiveness The amount of coding requested to the programmer to deal with code mobility should be minimized. Also, the programmer should not be forced to subclass from the toolkit classes in order to be able to exploit code mobility. Rather, relocation strategies should be applicable directly to the notion of executing unit provided by Java, i.e. directly to a Java thread. Ideally, this fosters exploitation of code mobility even in applications that where not designed with mobility in mind.

Portability Our system is developed for the Java VM, and runs wherever this platform is able to run.

Notably, security is not among the driving principles of our toolkit. Although security is often a major concern in applications exploiting mobile code, for some of them it is not the main issue and performance is instead the key factor. An example is constituted by network management. The vast majority of organizations still rely on SNMPv1, whose security model is limited to a "community name" that is transmitted unencrypted with each SNMP request. In this case, the introduction of mobile code in place of the client-server SNMP interactions does not provide additional breaches in the management systems, since the untrusted code contains the same information provided by the CS interactions. Furthermore, the rather trivial authentication protocol provided by SNMP can be implemented at the application level. Hence, in this cases the overhead imposed on the run-time support by security is undesirable. However, security policies can be implemented either at the application level or by a higher-level mobile code layer.

For the same reasons, our system does not provide any special abstraction to encompass communication among migrating units. In particular, point-to-point mechanisms like those described in Chapter 2 would encompass a naming scheme to identify Java threads, which is presently not provided by Java. However, support for communication can be provided by upper level layers.

14.2 Overview

Each instance of the μCODE run-time support, i.e. each μCODE server, constitutes a computational environment, according to the classification provided in Chapter 2. The fundamental operations provided by the toolkit allows to create or copy a thread from the local to a remote μCODE server, and the relocation of classes from the local to a remote server and vice versa. More precisely, referring to the taxonomy of mechanisms for mobile code technology presented in Chapter 2, the μCODE toolkit provides all the mechanisms we presented for dealing with weak mobility. Hence, it provides mechanisms supporting code shipping and fetching of both code fragments and stand-alone code. Synchronous and asynchronous invocation are supported, as well as mechanisms for both deferred and immediate execution of the mobile code. As for data space management, no dedicated mechanism is presently in place to enable the sharing of resources besides those provided by the Java language, since the toolkit focuses on the relocation of code and threads; on-going work is addressing this issue.

The unit of migration is the *group*. A group is simply a container for classes, objects. In particular, thread objects can be stored in a group as well, although their execution state is lost during transfer. The group abstraction is somehow reminiscent of the briefcase abstraction found in TACOMA [54] or, more closely, of the object-group abstraction found in Sumatra [2]. However, there at least two relevant differences with respect to the approach taken by the designers of Sumatra:

- in Sumatra, the classes needed by objects in an object-group are not transferred together with the group, rather they must be transferred explicitly at destination with a `downloadClass` primitive allowing for shipping of code fragments. This is motivated with the goal of providing the programmer with finer control in determining which of the classes available at destination must be used, and which ones have to be provided by the sender. This is in accordance with the phylosophy of our toolkit, and in fact μCODE provides an analogous primitive `fetchClass` together with the complementary `shipClass`. In addition, we also provide the capability to insert classes needed at destination directly into the group. The added value provided by this choice is improved atomicity. With the decoupled approach, connectivity may be lost in between the shipping of the class and the transmission of the group. With our approach, either the whole group is reconstructed correctly, or none of its components are available at destination.

- in Sumatra, object-groups cannot contain threads. This is a consequence of the fact that Sumatra threads maintain their execution state across migration, since the underlying run-time support is a modification of the Java VM. To preserve portability, our toolkit is built on top of the Java VM and provides only weak mobility, preserving the data state of a thread but not its execution state. Hence, thread objects stored in the group actually lose their flow of control and then become akin to conventional objects, without raising the semantic problems that guided the choice of Sumatra designers.

The fundamental operation provided by the toolkit is the relocation of a group on a different server. Each of the aforementioned operations is implemented by creating a group containing the necessary classes and objects and then delivering it to destination. There, the steps taken by the run-time support are ruled by two classes associated with the group: the handler class and the root class.

The *handler* class is used to instantiate an object responsible for "unpacking" the group and manipulating its contents. μCODE defines a default handler, which is in charge of implementing the operations described above. However, programmers can define their own handlers and thus effectively customizing the way the group contents are handled and even, if necessary, define their own relocation primitives.

The *root* class provides additional information to guide the unpacking of the a group. Classes can be added to a group either individually or collectively by computing the class closure, as defined later. As a consequence, a group can contain several classes and the handler needs information about the "main" class on which the relocation operation must be performed. Then, for instance, the root class enables the programmer to specify which class, among those contained in the group, must be used to spawn a new thread on the destination site. A group can actually contain more than one thread object to be restored at destination. However, only one of them is handled directly by the system. The others must be handled explicitly by a customized handler.

Once unpacked, classes are inserted into a *class space*, which is basically a name space for the classes available on the CE. μCODE defines two kinds of class spaces. Each incoming thread is executed by keeping its classes in a *private class space*. This avoids possible name clashes by guaranteeing that the thread actually uses the classes that have been shipped with the group and not some other class with the same name already present on the CE. If the thread spawns new threads on the destination CE, the class loaded by these threads are kept in the private class space of their parent thread. However, the package offers methods that can be invoked by mobile threads to "publish" their classes in the *shared class space* which is associated with the CE. A class within the shared class space is available to every thread hosted by the CE, and can also be accessed by ship or fetch operations performed remotely by threads on remote CEs.

Class spaces guide the resolution of class names at loading time. When a class name needs to be resolved, the μCODE customized version of the Java class loader is invoked. The class loader looks for the class by performing the following steps until the class is found:

1. Checks whether the class is a system class, i.e. a class defined in the Java API or in the μCODE package. The set of classes considered as belonging to the system is under the control of the programmer, who can change it dynamically if needed. System classes are checked first in order to avoid the situation where an incoming thread redefines their behavior.

2. The class is searched in the private class space of the thread that triggered the loading. Threads that are created by the application in a conventional way, i.e. not as a consequence of a migration, are associated with a private

class space that coincides with the one managed by the default Java class loader.

3. The class is searched in the shared class space.

4. Besides their ability to perform explicit shipping or fetching of classes, mobile threads can be either denyed or granted (at relocation time) the capability to download dynamically classes from a remote site at loading time. If the thread that triggered the class loading is allowed to perform dynamic download, the corresponding class is retrieved from the remote site and loaded. If the class cannot be found on the remote node or the thread is denied dynamic download, a `java.lang.ClassNotFoundException` is raised.

As we mentioned earlier, the programmer can request the insertion in the group of the whole class closure. The *class closure* C^* of a class C is a set of classes defined as follows:

1. $C \in C^*$;

2. $C' \in C^*$ if $\exists \gamma \in C^*$ such that C' is either:

   (a) the superclass of γ;

   (b) the class of a field declared by γ;

   (c) the class of an exception, parameter, or return value for a method or constructor declared by γ;

   (d) an inner class declared by γ;

Thus, in the fragment

```
public Class MyClass extends MySuper implements MyInterface {
  public int MyField f;
  public MyReturn myMethod(MyPar p) throws MyException {
    MyVar v = new MyVar();
    class MyInnerClass {
    }
  }
}
```

the classes and interfaces `MyClass, MySuper, MyInterface, MyField, MyReturn, MyPar, MyException`, and `MyInnerClass` belong to the class closure of `MyClass`, but `MyVar` does not.

The programmer is also provided with methods to insert objects into a group. Differently from class serialization, which is handled explicitly by the μCODE implementation, object serialization relies on the corresponding Java mechanisms, and thus requires the class of the object being serialized to implement the interface `java.io.Serializable`. The Java serialization mechanism relies on the notion of *object closure*. The object closure o^* of an object o is defined as follows:

1. $o \in o^*$;

2. $o' \in o^*$ if $\exists \omega$ such that o' is a field of ω.

If a non-serializable object is added into a group, an exception is raised. The programmer can constrain the amount of the closure being effectively serialized by declaring an object field as `transient`; in this case, the corresponding object is not serialized even if the aforementioned interface is actually implemented by its class.

Finally, the μCODE toolkit provides also a `MobileAgent` abstraction, at least for two reasons. First, although this abstraction can be too rich for some applications involving code mobility, it has been proven useful in many respects has evidenced by the analysis carried out in Part II. Second, it provides a relevant example of how it is possible to exploit the relatively low-level primitives and abstractions provided by the μCODE toolkit to construct easily higher-level abstractions.

14.3 Description of the API

In this section we give a brief description of the API provided by our mobile code toolkit.

14.3.1 The `Group` Class and the `GroupHandler` and `Mobile` Interfaces

As we discussed in the previous section, the `Group` class is central in the μCODE toolkit. The methods of class `Group` are shown[2] in Figure 14.1. The constructor of `Group` takes as arguments the names of the root and handler classes. Class names are specified using the usual Java convention encompassing packages, so that a class name like `Foo.Bar` corresponds to a class `Bar` declared in the package `Foo`. Specifying the package name is optional in general, but may be mandatory in same cases in order to disambiguate classes with names belonging to different packages, or to refer to classes that do not belong to the default package.

The root class can be constituted by any class, and must not necessarily be transmitted within the group; in this case, however, it is up to the programmer to ensure that it is found at destination. Similar considerations hold for the handler class as well, that can coincide with the root class. The handler class can be any class implementing the interface `GroupHandler`, shown in Figure 14.1. The `GroupHandler` interface defines a single method `unpack`, that is called automatically by the run-time support when the group is received. This method is supposed to return an object implementing the interface `Runnable` defined in the Java API, i.e. an object that can be used to spawn a new thread of execution.

The method `addClass` inserts the class corresponding to the given `className` in the `Group` object. If the class already exists it is overwritten. The method `addClassClosure` adds the whole class closure rooted at `className`.

The method `addObjectItem` allows for an object to be inserted in the group. The `label` parameter provides a way to tag objects in order to retrieve them on the remote site using the complementary method `getObjectItem`. The methods

[2]The declarations of exceptions are omitted for the sake of clarity.

```
public class Group {
  public          Group(String rootClass, String handlerClass)

  public void     addClass(String className)
  public void     addClassClosure(String className)
  public void     addObjectItem(String label, Object obj)
  public Object   getObjectItem(String label)

  public void     addMobile(Mobile threadObj)
  public Mobile   getMobile()

  public void     ship(String destination)
}

public interface GroupHandler {
  public abstract Runnable unpack(Group group)
}

public interface Mobile extends Runnable, Serializable {
}
```

Figure 14.1: The `Group` class, and the `GroupHandler` and `Mobile` interfaces.

`addMobile` and `getMobile` constitute a variant of the above. They insert and retrieve in the group an object implementing the interface `Mobile`, shown in Figure 14.1. `Mobile` extends `Runnable` and `Serializable`, and thus describes an object that can contain a thread of control and yet be transferred on a different host. As we mentioned earlier, the execution state of the thread is lost during migration, thus the thread associated to the `Mobile` object can still access the values of the fields of the object but is restarted from the beginning. The method `getMobile` is conceived to be used in the body of the `unpack` method provided by `GroupHandler`. Since no label is provided, there can exist only one mobile thread per group. Other threads can be inserted in the group, but while the mobile thread inserted with `addMobile` is handled automatically by the system if no handler is specified, the other threads must be handled explicitly within an `unpack` method provided with by the programmer.

Finally, the `ship` method provides relocation of the `Group`. The `destination` parameter, specified as a pair `hostname:port`. If only the hostname is specified a default port `1968` is assumed.

14.3.2 The `Server` Class

This class collects all the methods that enable a site to send and receive mobile code. The method `boot()`, at the bottom of Figure 14.2, starts a new instance of the run-time support on the local site. The first constructor creates a server listening on the default port, while the second one enables the programmer to specify a different port, thus allowing for multiple instances of the run-time support co-existing on the same machine and even in the same instance of the Java VM. Nevertheless, the run-time support generated by `boot()`, basically a new thread in the Java VM, is needed only if the site needs to receive and execute incoming mobile code. In the case where the application needs only to send mobile code, *no additional run-time support is needed* besides the Java VM and the classes in the μCODE toolkit, and code mobility is exploited using one of the other methods shown in Figure 14.2.

```
public class Server {
  public final static int NONE;
  public final static int ROOT;
  public final static int FULL;

  public int rSpawnGroup(String destination, Group group,
                         String dynLink, boolean synch)
  public int rSpawnThread(String destination, String className,
                          int classClosure, String dynLink,
                          boolean synch)
  public int rCopyThread(String destination, Mobile threadObj,
                         int classClosure, String dynLink,
                         boolean synch)

  public int fetchClass(String destination, String className,
                        int classClosure, boolean synch)
  public int shipClass(String destination, String className,
                       int classClosure, boolean synch)

  public void boot()
  public void boot(int port)
}
```

Figure 14.2: The `Server` class.

These primitives take a `destination` parameter, which is specified as a pair `hostname:port`. The first three primitives support remote cloning and remote creation of threads. In turn, `fetchClass` and `shipClass` provide the ability to relocate Java classes. `rSpawnGroup` is the most general primitive, and allows to either create or resume remotely a thread, by transferring the corresponding thread object and restarting the corresponding thread of control. The contents of what is actually transferred to the remote site are completely up to the programmer, that specifies them in the `group` parameter using the `Group` methods. Upon receipt at destination, if no custom handler is specified, the default handler looks into the received group for a thread object by performing a `getMobile()`. If the thread object is found, its associated thread is restarted. Otherwise, a new instance of the root class is created and a new thread of control is spawned for it. The `dynLink` parameter enables dynamic linking of classes from a remote site, thus providing direct support for a COD paradigm. This way, the thread can be spawned remotely by providing only a subset of the classes that required for its execution, and demanding to the remote site's class loader the burden to download them dynamically, if and when they are actually needed, from the host specified in the `dynLink` parameter. If the value of this parameter is `null` dynamic linking is prevented. All the class that are not transmitted but are indeed needed for remote creation and execution of the thread must be present at destination or, in alternative, the `dynLink` parameter must be defined in the invocation of the method. In any other case, an exception `java.lang.ClassNotFoundException` is raised on the destination site.

Finally, the `synch` parameter determines whether this operation is performed synchronously or asynchronously. If `synch` is true, the operation is blocking for the caller, that is suspended until a return value is received. If `synch` is false, the method returns immediately after the transmission of the group is completed.

`rSpawnThread` creates a new thread at the specified `destination` and provides some level of automation in the process of building the group. Instead of building explicitly the group to be transferred, the programmer specifies only the `className` of the thread that is going to be spawned. This actually constitutes the root class of the group being sent by the toolkit implementation. In addition, the programmer can also specify whether the *class closure* of the thread class has to be included. The parameter `classClosure` can assume the values `NONE` if no class should actually be transmitted, `ROOT` if only the thread class should be transmitted, and `FULL` if the full class closure of the thread class should be transmitted.

`rCopyThread` clones on a remote site an object present on the local site and implementing the interface `Mobile`. The thread object is passed in the `threadObj` parameter. The copy on the remote site loses any execution state that the associated thread of control might have. The other parameters retain the meaning already discussed for the other methods.

By default, the classes being transferred as a consequence of the invocation of the three operations above are cached on the destination site in the private class space associated with the incoming thread. They can be later moved to the shared class space by invoking the methods provided by `ClassUtil` and described later. This can be done either specifying the relocation within the `unpack` method of a specialized handler or directly into the thread code.

The current implementation of μCODE requires that, in any of the three methods described so far, the root class—corresponding to `className` in `rCopyThread` and `rSpawnThread`—and the possible handler class must have a public, parameterless constructor, that is invoked automatically by the system to generate an instance of such classes.

The methods `shipClass` and `fetchClass` enable relocation of classes across different sites. Both operations can access only the shared space present at the destination site. This is compliant with the semantics defined for private spaces. In any case, since in the toolkit there is no built-in naming scheme to identify threads, there are no means to fetch or ship a class directly to a thread running on a remote site. However, this kind of mechanism can be built on top of μCODE.

The `shipClass` operation copies one or more classes from the local site to the shared class space of the `destination` site. Analogously, the `fetchClass` operation copies one or more classes from the `destination` site shared class space to the private class space of the thread requesting the operation. In both cases, the class(es) remain still available at their source site. The amount of code transferred is ruled by the `classClosure` parameter, with the only difference that the value `NONE` raises a `MobileCodeException`, since the semantics of the operations prescribe that at least one class must always be transferred.

14.3.3 The `MobileAgent` Class

The class `MobileAgent`, shown in Figure 14.3 provides a natural abstraction for implementing a Mobile Agent paradigm with weak mobility. No execution state is preserved: this would require modification of the Java VM, thus losing the portability provided by the language. However, the data state of `MobileAgent` is preserved across migration.

Movement is reduced to a single instruction: the `go` method. The list of parameters is similar to those of methods found in the `Core` class. In particular, a `MobileAgent` instance is allowed to specify a host from which classes can be downloaded dynamically on demand.

```
public abstract class MobileAgent implements Mobile, GroupHandler {
  protected final int go(String destination, int classClosure,
                         String dynLink, boolean synch) {
    String className = this.getClass().getName();
    Group group = new Group(className, className);
    Server s = new Server();

    group.addMobile(this);
    switch (classClosure) {
    case Server.NONE:
      break;
    case Server.ROOT:
      group.addClass(className);
      break;
    case Server.FULL:
      group.addClassClosure(className);
      break;
    }
    s.rSpawnGroup(destination, group, dynLink, synch);
    Thread.currentThread().stop();
  }

  public final Runnable unpack(Group group) {
    return (MobileAgent) group.getMobile();
  }
}
```

Figure 14.3: The implementation of the `MobileAgent` class.

The class `MobileAgent` is particularly interesting since it is an example of how the low-level abstractions provided by `Group` and `GroupHandler` can be used to build higher-level abstractions. Figure 14.3 actually shows the whole implementation of class `MobileAgent`, except for the declarations of the exceptions and other minor details.

The most interesting part is the implementation of the `go` method. Its body first retrieves the class name of the subclass of `MobileAgent` to which the method is actually bound. It is worth noting at this point that `MobileAgent` is defined as abstract because it implements the interface `Runnable`, through `Mobile`. The interface `Runnable` contains an abstract method `run()` that must be redefined by subclasses with the code describing the behavior of the thread associated to them. For this reason, the `go` method can be invoked only by subclasses of `MobileAgent`.

The information about the class name is then used to create an instance of `Group`, that is the unit of mobility being actually sent on the remote node. The method body then adds the thread object to which the `go` is bound to the group, using the method `addMobile()` defined in `Group` and described earlier. When the group is received at destination, the run-time support present at that node will call the implementation of the method `unpack`, shown in Figure 14.3, that retrieves the thread object using the complementary method `getMobile()` and passes it back to the run-time support, which spawns a new thread of execution for it.

After adding itself to the group, the method `go` adds the classes specified by the user using the usual predefined constants. Then, it uses the primitive `rSpawnGroup()` to actually send the group on the destination site. Finally, it stops the current thread, that will be eventually garbaged collected.

14.3.4 Miscellaneous Features

The class `ClassUtil`, shown in Figure 14.4, provides methods to test whether a class is present in either the current thread's private class space or the shared class space, to retrieve a class from a class space (e.g., to create a new instance through the `java.lang.Class.newInstance()` method) and to move classes across the private and shared class spaces. Insertion of a class that is already contained in the class space generates an exception.

```
public final class ClassUtil {
  public static boolean isPrivateClassLoaded(String className)
  public static boolean isSharedClassLoaded(String className)
  public static Class getPrivateClassLoaded(String className)
  public static Class getSharedClassLoaded(String className)
  public static void moveClassFromShared(String className)
  public static void moveClassToShared(String className)
}
```

Figure 14.4: The `ClassUtil` class.

The class `Properties`, shown in Figure 14.5, provides a method `setProperties` that specifies some properties of the toolkit, notably the use of GZIP compression for the output data stream. The run-time support recognizes automatically the format of the input data stream and applies decompression if needed. Finally, the method `addSystemPackage` insert `packageName` among the packages that are considered as belonging to the system and thus are never transmitted across CEs.

```
public class Properties {
  public static void setProperties(boolean compressionON,
                                   boolean debugON,
                                   boolean errorsON)
  public static void addSystemPackage(String packageName)
}
```

Figure 14.5: The `Properties` class.

14.4 Architecture and Implementation

The implementation of μCODE leverages off of some advanced features provided by the Java language.

The transmission of a group is achieved by using Java object serialization and redefining the default behavior in order to provide a customized way to load classes. Object serialization is provided by the `ObjectInputStream` and `ObjectOutputStream` classes. By default, the class objects for the objects being serialized is never transmitted together with them—the underlying assumption is such classes must be present at destination. A class descriptor is stored in the stream, in order to provide the destination VM with sufficient information to loading the appropriate class when restoring the object. In our toolkit, however, the programmer can insert into a group some of the classes that are needed at destination to restore properly the objects in the group. For what explained above, requesting directly the serialization of a `Class` object does not work: in fact, even in this case the actual object is not serialized (since it descends from `Class`) and its descriptor is inserted in the stream instead.

We overcome the problem as follows. The implementation of the `Group` class has the structure shown in Figure 14.6. Classes and objects are stored in the two `Hashtable` objects as specified by the programmer. In particular, classes are stored directly in their bytecode form rather than as class objects. The various fields of the group are transmitted explicitly in the implementation of the `ship` method one after the other.

```
public final class Group extends Object {
  int opCode = Constants.UNKNOWN;
  String rootClassName = null;
  String handlerClassName = null;
  Hashtable classes = null;
  Hashtable objects = null;
  String dynLink = null;
  boolean synch = false;
  transient Class rootClassObj = null;
  transient Class handlerClassObj = null;
  transient String source = null;
}
```

Figure 14.6: The data structure of a `Group`.

On the destination site, the `Group` object is reconstructed by reading the objects in the same order from the stream. Thus, the `classes` hashtable is reconstructed before the `objects` one. This way, we can force the customized class loader to load all the classes that come from the stream. When the `objects` hashtable is deserialized, the classes whose names have been stored in the descriptors must be loaded. The loading process is triggered by the system by executing the `resolveClass` method of `ObjectInputStream`. We redefined this method, forcing its implementation to call the customized class loader provided by μCODE. At this point of the process, the class loader already "knows" about classes retrieved from the stream, and can load them. Otherwise, if the class is not found among those incoming, the behavior described in Section 14.2 applies.

Clearly, class loading plays a key role in the implementation of our system. In particular, it is fundamental to the provision of separate class spaces. This is achieved by exploiting a two-level scheme that *associates a separate class loader to each incoming thread*. The first-level class loader is the one actually called by components in the package—e.g., upon deserialization. It acts basically as a proxy for the actual class loaders, to which it maintains references. At startup, two class loaders are created: a class loader devoted to servicing requests for classes present on the system and one devoted to service requests for the shared class space. When the run-time support is servicing a request that encompasses the creation of a new thread, a new class loader is created to manage class loading for that thread. Due to the semantics of class loading in the Java language, any additional class loading triggered by the thread class or any of the classes loaded for it will go through the same class loader, thus preserving the notion of a privacy of the class space.

Another relevant part of the toolkit is the determination of the class closure. It exploit the features in the Java Core Reflection API [105] and the reflection features already provided in the `Class` class. The private method `getClosure` building the closure first adds to it the class passed as a parameter, and then determines the classes for fields, method parameters, exceptions, and other components considered in the closure, by invoking the methods in `java.lang.reflect` and in `Class`. For instance, the class of the return parameter of a method of the class object `c` is determined by first retrieving the corresponding `Method` object `m` from `c` by invoking `getDeclaredMethod()` on it, and then by retrieving the class object corresponding to the return parameter by executing `m.getReturnType()`. Then, the method `getClosure` invokes recursively itself on the classes retrieved by using reflection.

14.5 Discussion and Future Work

In this chapter we presented a mobile code toolkit designed to provide lightweight, flexible, and extensible support for integrating code mobility in distributed applications developed using the Java language.

The current implementation of μCODE consists of approximately 950 non-commented source statements. This generates a bytecode of approximately 40Kbytes. It has been developed and tested under Sun's JDK 1.1.4 on Sun workstations running Solaris and PCs running Windows95. We have not measurement figures yet, but we believe that the size of the code and the absence of any application level protocol besides object serialization produces an improvement performance in terms of generation of both computational and traffic overload. μCODE has been effectively used for the implementation of the system described in the next chapter. The toolkit has been developed entirely by the author as a proof of concept: currently, it is still under development and extension, and we plan to add several higher-level features to demonstrate the scalability of the approach.

A significant limitation of the toolkit is presently represented by the fact that the determination of the class closure is limited to the information that can be reconstructed through the Java reflection mechanism, and cannot detect the need for classes that are declared in method bodies. The only way to work around this limitation is to resort to direct on-the-fly inspection of the class bytecode, which seems to be a major endeavor with respect to the benefits achievable.

A straightforward extension will be constituted by a package to manage resources, according to the strategies for data space management described in Chapter 2. This will be probably based on a concept of *object space* analogous to the class space concept. The developments in the last version of the Java language, JDK 1.2, and in particular the provision of weak references, i.e. references that can become void without causing exceptions on access, might help in providing mechanisms like handling the data space with relocation by move.

JDK 1.2 will probably bring useful mechanisms also in the context of a possible extension of μCode with a package containing mechanisms devoted to security. The new Java security model fosters a per-thread rather than per-VM security, and thus seems particularly suited to our needs.

Chapter 15

Designing a Videconference System for Active Networks

Concurrently with the developments in code mobility and the associated technology, other researchers have investigated means to introduce flexibility in computer networks by assuming the availability of next-generation network devices whose behavior can be changed dynamically according to users' needs. These *active network*, that we already mentioned in Chapter 4 are receiving a great deal of attention in industry and academia and seem naturally suited to leverage off of the developments in the field of code mobility. In fact, technologies and architectures conceived for building distributed applications exploiting code mobility can be used for supplying dynamically the network devices with application-dependent code changing their behavior.

Networks become active because they take part in the computation performed by the applications rather than being concerned only with the transparent transfer of data. Hence, conventional network functionality will be subsumed in to-be active network devices; a well-defined set of fundamental primitives will enable foreign code to access the device functionality. However, today the field is still quite immature and, besides the lack of active network devices, there is no agreement on which primitives must be provided by them. Moreover, researchers presently interpret the idea of active network at least with two different nuances.

The first, broader, interpretation of the term, embodied for example in the work described in [47, 128], is that the distinction between the nodes inside and those at the edge of a network becomes blurred in terms of their functional characteristics. In this setting, network devices, e.g., routers, can execute directly mobile code of a distributed application, providing it with the capability to access directly functionality and information at lower layers in the network stack. This *programmable switch approach* [110] does not affect the way current lower and mid layer protocols are designed and deployed.

On the other hand, the second interpretation focuses on modifying the heart of network protocols, by extending the control information contained in network packets with code describing how to process the packets at intermediate nodes on the path to destination. This way, protocol deployment is performed on demand, possibly without requiring software preinstallation and upgrade, and is still in the hands of the applications. A packet augmented in this way, also called *capsule* [110], is reminiscent of what the mobile code community calls a mobile agent, i.e. an autonomous unit of mobility containing code and state. Toolkits are being developed [120, 48] to support the creation and deployment of capsules.

The goal of the work described in this chapter is to assess the benefits brought by code mobility in the context of active networks. This is achieved by constructing a videoconference system that is expected to leverage off of the scenario depicted by the aforementioned programmable switch approach. Videoconferencing is an increasingly popular distributed application which poses significant challenges and constraints in terms of performance and thus can be considered a reasonable testbed for our purposes.

The work presented here focuses on the opportunities enabled by code mobility and active networks as far as application *design* is concerned. The paper provides also some details about the implementation of a prototype that is being completed at our university. This are the first steps of our long term research on this subject, which aims at characterizing qualitatively and quantitatively our implementation against conventional ones.

The chapter is structured as follows. Section 15.1 discusses briefly the requirements of a videoconference application, and how they can be satisfied through an architecture exploiting code mobility and an active network. Section 15.2 describes in detail the architecture of our prototype, and provides two variants which feature different degrees of distribution. Section 15.3 provides some details about the on-going implementation. Finally, Section 15.4 contains brief conclusive remarks and discusses future work on the subject.

The work presented in this chapter has been published in [8, 10].

15.1 Videoconferencing on an Active Network

Videoconferencing systems can be split grossly in two categories. In *peer-to-peer* conferencing systems, the participants are connected through a multicast network and the videoconferencing flow generated by each participant is distributed unaltered to the others by leveraging off of multicast. An evident advantage of this first approach is its good scalability. However, conference management, especially as far as security is concerned, is complicated by the characteristics of the architecture. A relevant example of this kind of systems is the tool suite for the Multicast Backbone (MBone).

The alternative, especially popular among commercial systems, employs a *centralized* architecture based on a conference server which receives the conference streams from the active participants and replicates such streams to all the participants. Centralization not only simplifies the problem of secure access, but also enables customizability in that the server can perform additional computation on an incoming videoconference stream before replicating it to the other participants. This way, customizability can be exploited directly by conference users, that achieve control on the quality of service delivered to them. Unfortunately, a centralized approach is affected by the usual drawbacks in terms of scalability and fault-tolerance. An example of centralized system is

CuSeeMe [28]. The H.323 standard [53] defines an architecture that accomodates both categories.

Neither of the videoconference architectures seem suitable for the large scale scenarios that are being envisioned for the Internet. Let us consider for instance broadcasting of a sport event over the Internet. In a multicast network, the conference streams cannot be tailored to the bandwidth available to the users. Whether users are connected on a high-bandwidth LAN or on a dial-up connection, they receive the same stream and thus some of them experience a poor quality of service. On the other hand, a centralized architecture would generate a huge amount of traffic.

Ideally, a videoconferencing system should exhibit features coming from both architectures, thus enabling user customization without preventing the scalability of the architecture. Interestingly, these are, by and large, the same kinds of concerns that motivated active networks and in general the approaches relying code mobility. Our architecture relies on a conference server that we call *reflector*. The aforementioned goal is then achieved, in a novel way with respect to conventional systems, by:

1. enabling the users to "upload" application code into the reflector, thus changing its behavior and *customizing* it to the users' needs;

2. placing the reflector directly on the intermediate nodes of the network, where it can use the information managed by the device to become aware of the status of the network and *adapt* to it;

3. enabling the reflector to *migrate* on a different node as a consequence to adaptation.

The first point provides a degree of customizability that is even higher than the one provided by centralized systems, in that not only the parameters of the server, but also the code that governs its behavior can be changed. The reflector provides a "shell" where each client can plug in dynamically the code describing some customized processing. Thus, for instance:

- The coexistence of different coding algorithms can be allowed. The reflector does not need to be equipped with a plenty of coding algorithms in order to encompass the needs of a wide range of clients. These are linked in the reflector dynamically and on demand.

- The quality of service for the encoded videoconference stream can be changed dynamically to fit the user needs. The desired policy can be arbitrarily complex and can be changed at run-time. For instance, it is possible to specify application-dependent criteria to discard packets in presence of congestion, as suggested in [13].

- The videoconference flows coming to a client from different participants can be treated differently. For example, a participant can request the reflector to give higher priority or higher quality to some of the flows to guarantee a higher quality of service to some flows, and carry the others with a lower quality. The advantage is that flows can be discriminated according to application level information, like the identity of the current speaker.

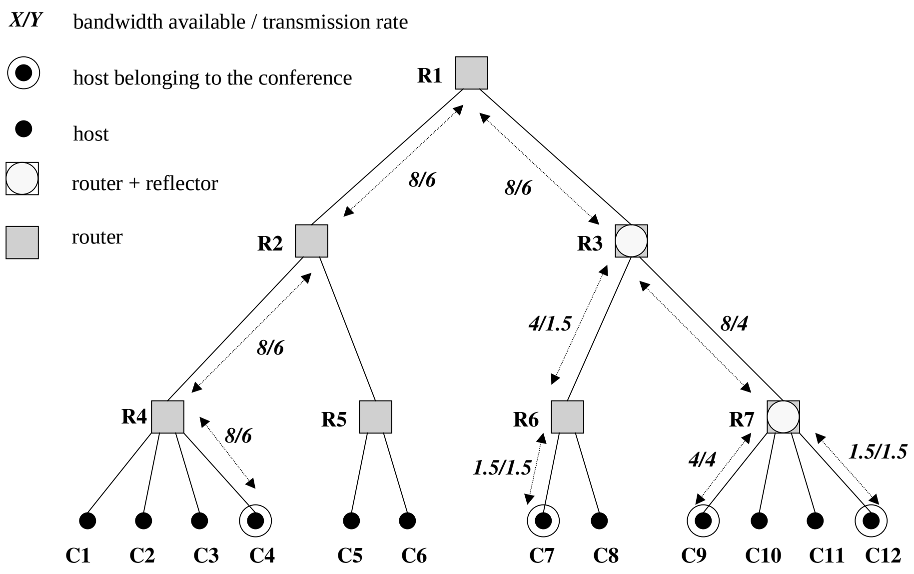


Figure 15.1: Cloning reflectors.

The second point, that actually provides the rationale for exploiting an active network, deserves some elaboration. At the time of writing, network devices equipped with a run-time for execution of mobile code are not widely available. However, many vendors have already announced new releases of their products that feature hardware or software support for the Java language [104]. In order to cope with this problem, that is intrinsic to the whole research area, we adopt the approach followed by many other researchers, i.e. simulate the active device by running the mobile code on a workstation directly connected to the network device.

The last point is at the core of the work described in this chapter. During the analysis of data available on the network device, as soon as the reflector perceives the need to adapt to some change in the conditions of the network a migration is triggered. We devised two architectural solutions to enable adaptivity through migration. In both cases, the criteria to trigger adaptation is embodied in a component of the reflector that is designed to be modular and thus interchangeable, as described in the next section.

In the first solution, the reflector responds by relocating itself in a position in the network that is optimal with respect to some cost function, e.g., distance from the clients. With respect to the classification presented in [37], the reflector relies on a weak form of mobility where its execution state is not preserved across migration: only its code and a portion of its data state is transferred. This solution is suitable for conferences characterized by a limited number of participants or by participant heavily clustered.

In turn, our second architectural solution is conceived for conferences with a great number of participants and employs *cloning* of the reflector, that can be

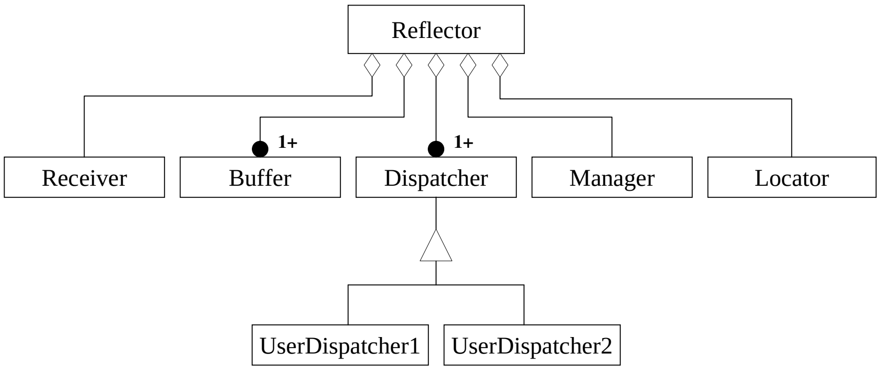


Figure 15.2: The components of the reflector.

regarded as a special case of migration. This way, multiple reflectors are injected into the network upon some special event, e.g., when a new client is added to the participants, and perform successive transformations of the conference streams, providing for improved scalability. For instance, Figure 15.1 shows a network with two reflectors placed on two different routers, R3 and R7 that are responsible for serving hosts C4, C7, C9, and C12. Notably, the reflectors have been placed only in the position of the network where transformation of the videoconference streams is needed.

The next section illustrates the details of both architectures. In principle, the two architectures could be combined effectively in a single system that, upon the occurrence of some user-specified condition, is capable to switch automatically between the two modes of operation. However, currently our prototype implements only the first architecture.

15.2 The Architecture

The design described in this section aims at modularity and reconfigurability, in order to leverage off of the opportunities provided by code mobility in changing dynamically the code associated with a component. Hence, we specified the design of our prototype using the OMT [96] object-oriented notation, that made it easier to implement it subsequently in Java. The full OMT design is available in [73].

15.2.1 Reflector

The Reflector is composed of five classes, as specified in Figure 15.2. Receiver, Buffer, and Dispatcher provide the "workpower" of the reflector and deal with receiving and trasmitting the data packets that constitute the conference stream. The classes Manager and the Locator components are the "brain" of the reflector.

Locator is devoted to monitor information available on the device the reflector is residing on and can trigger migration or cloning to adapt to changes in the network. The Manager is the control component that governs the behavior of the reflector. All the aforementioned components are described in the following.

Receiver

Instances of this class are responsible for handling the input videoconference streams. Each instance of the Receiver contains a separate thread of control that listens on a socket port used to collect the multimedia stream from all the clients. The Receiver filters the packets according to the sender, as specified by the IP protocol, and stores the packets in the corresponding element of a buffer pool. Videoconference streams often embed audio and video components. These are actually received on two distinct sockets, and thus actually two instances of Receiver are spawned for each reflector. The Receiver is a key element as far as performance is concerned. Thus, keeping it simple is desirable to achieve good performance.

Buffer

Instances of this class contain circular lists of packets. The number of buffers contained in the reflector is usually twice the number of clients currently connected to it, in order to separate the audio streams from the video streams. Each Buffer object is accessed concurrently by Dispatcher instances, that can read the packets stored in it by the Receiver object in charge of the buffer. Each buffer provides methods to return the packet stored at a given position as well as to determine the index of the packet stored most recently. Clearly, the length of the buffer is a key parameter that must be set according to the maximum delay tolerated by each client and the bit rate.

Dispatcher

Instances of Dispatcher have a separate thread of control and are responsible for replicating input streams as output for the clients, possibly after some additional processing. The class Dispatcher, in Figure 15.2, provides a default implementation for the dispatcher that retransmits always the most recent packets present in the audio and video buffers. Clients can supply their own classes that are specialized from Dispatcher, in order to redefine the default behavior and provide for additional, customized processing and provide further adaptation to the network conditions. The customized processing may be as simple as selective discarding of frames in order to adapt to change in traffic, or as complex as recoding of the stream, e.g., to convert it from MPEG to H.261. This scheme can exploit effectively layered encoded video. The selection of a customized Dispatcher object may take place at setup time, but it can also be performed during the conference, by exploiting mobile code mechanisms.

Figure 15.3 illustrates how it is possible to leverage off of customized Dispatcher objects in order to filter out differently the same conference streams. In the figure, the first Dispatcher object privileges the streams coming from client A, while the second treates both client in the same way, although with reduced bandwidth.

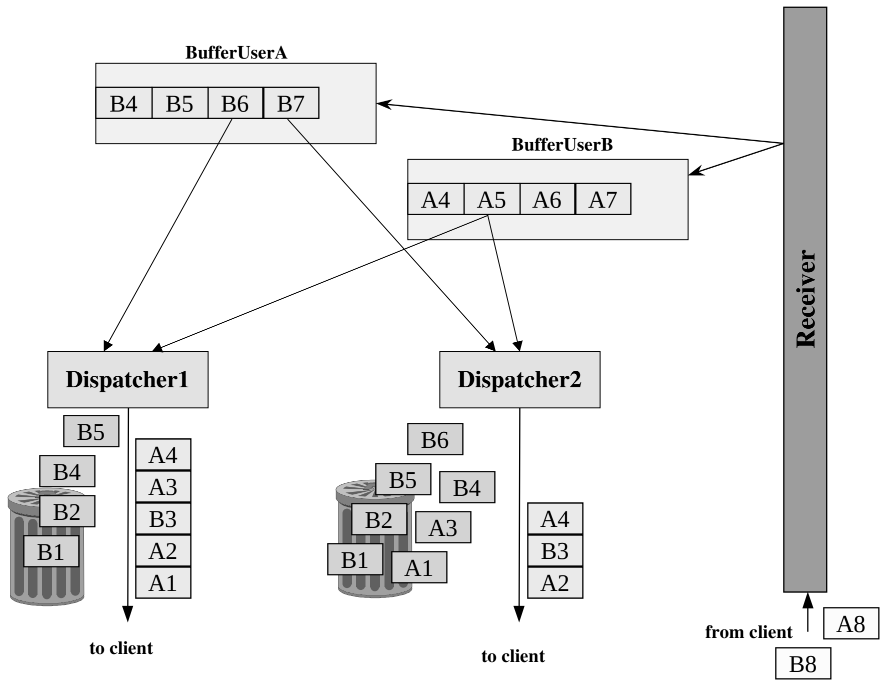


Figure 15.3: Exploiting customized Dispatcher objects.

Locator

The object of this class has a separate thread of control and is responsible for gathering information on the status of the network as perceived through the network device on which the Reflector is currently running. Then, it can react to changed network condition by suggesting either migration or cloning of the reflector. The new location chosen, and communicated to the Manager that triggers the actual relocation, is supposed to optimize a given cost function.

For instance, a cost function could minimize a weighted combination of relevant indexes like traffic on selected links, overall traffic on the network, maximum or average delay experienced by conference participants, and money cost. The data used by the Locator to perform its decision can be obtained either by monitoring the network or directly from the databases of the network node on which it is running. In the first case, the Locator could, for example, probe the round trip delay of the path to clients using the `ping` mechanism. However, a better solution that leverages off of the availability of local interaction with the network device, would be to exploit directly the information contained in the topological database of a router, the status of its queues, the traffic statistics of its interfaces.

The amount of intelligence embedded in the Locator and the computational overhead due to the optimization it performs are implementation dependent.

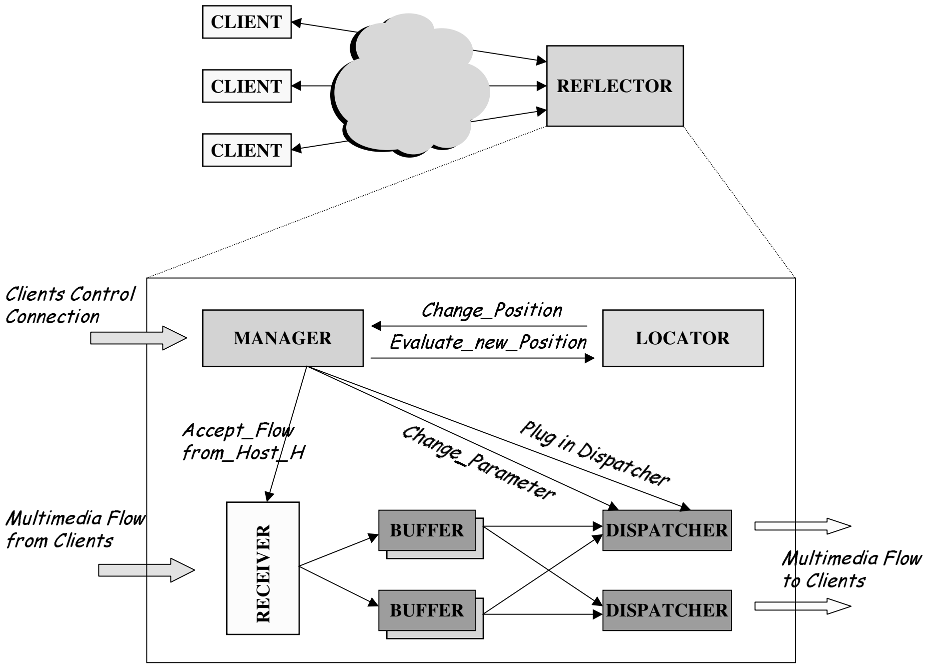


Figure 15.4: Managing the videoconference.

Customized Locator classes can be created to cope with different videoconferencing scenarios, and substituted to the default one. Also, the Locator can be reconfigurable by applications and users that can tune its parameters and thresholds to better suit their needs. In particular, the strategy for relocation, i.e. migration vs. cloning, can be specified.

Manager

As shown in Figure 15.4, the object instance of Manager coordinates the operations of the reflector and is in charge of communications with clients. Typical tasks of the Manager include joining new members to the conference, accepting incoming code for the dispatcher, dealing with security issues, and taking care of the migration or cloning of the reflector.

The Manager handles join requests from new clients. This are matched against a access control list provided by a privileged client, the *conference owner*, and subsequently accepted or refused. If the request is accepted, the new client may possibly send a Dispatcher class that is used to generate a new thread in the reflector.

The Manager keeps a control connection to each client. This connection is used for sending feedback to the clients about the operations of the reflector and to receive requests from the clients, e.g., for changing transmission parameters. These requests are interpreted and then performed by invoking the appropriate methods on the Dispatcher.

The control connection, in particular, is used to inform the client during relocation of the reflector, described in the next section, in order to notify the clients so that they can redirect their flows to the new location.

15.2.2 Migrating the Reflector

The movement of the reflector should be as transparent as possible to users in order to avoid an even temporary service disruption. This is especially true if the locator component is aggressive and movements are frequent. Temporary service disruption could take place in the time frame starting when the reflector is no longer operational in the old location and is not yet operational in the new location. In order to avoid this situation, migration takes place in two phases:

1. The manager object generates a clone of the reflector, which is sent on the network node chosen by the locator. As soon as the new reflector starts its execution on the new node, it notifies the manager of the old reflector, that can in turn notify the clients of the new location of the reflector. After the clients receive this notification they start transmitting their flow to the new location. The execution state of the reflector is lost, and only a portion of the data space is carried with it. In particular, the buffers are not transferred at destination, as they contain volatile information that is better handled at the old reflector. In contrast, the client profiles are transferred with the reflector clone, in order to maintain the information about the client currently connected. It is worth noting how effectively there exists a time frame when two reflectors exist in the network.

2. When all the clients have redirected their flows to the new reflector, i.e. when it is receiving streams from all of them, the new reflector sends back a control message to the manager object of the original reflector, killing the corresponding process running at the old location.

15.2.3 Conference Clients

The minimum setup for a conference includes an instance of Reflector and two clients. Each client can be either *active* or *passive*, i.e., it can receiving the conference flow from the reflector and either originating or not its own flow. Clients are associated with an access control list that is used to provide secure access to the conference. It identifies the conditions to be satisfied by a client for joining a conference, as well as its capabilities in interacting with the reflector, e.g., the capability to provide a customized dispatcher.

A conference starts when the access control lists of the first two clients are compatible. In this case, one of the two clients acts as the conference owner and inject a Reflector object into the network. From this point on, normal operations take place. Further communication with the reflector involves tuning of the conference parameters or notification upon relocation of the reflector.

15.2.4 Using a Capsule to Join the Conference

In normal operations, the beginning of a session is triggered by a client that "calls" another one. The identity of the latter, which is the conference owner, must be known a priori by some sort of off-line announcement[1]. After the two have agreed to start a conference, further participants could join the conference in a similar fashion by calling the conference owner. This approach does not scale to large distribution events, like an Internet TV broadcast, because the conference owner would be overwhelmed by join requests. Moreover, after the conference is started it is the reflector that owns all the information about access control and it should be responsible of handling new join request, even though its position is changing over time and its address cannot be advertised off-line when the conference is announced.

A solution to this problem involves joining the conference by injecting into the network an active packet, i.e. a capsule. An instance of capsule is injected in the network when the client requests a join, and travels autonomously towards the conference owner, effectively following an MA paradigm. At each node, the capsule object verifies whether a Reflector instance is running on that node. If this is the case, the reflector handles the join request by examining the access control information carried by the capsule and notifies directly the sender about acceptance or rejection of the request. In the case where the capsule object does not step into the reflector on its way to the conference owner, it will reach the latter. Then, the conference owner can either forward the capsule to the a Reflector instance, since this information is known to it, or handle the request as described earlier if the session has not started yet. In both cases, the request is handled and the conference is started, provided that the access control lists match.

The approach can be further extended by allowing a joining client to send the capsule to any participant to the conference. This increases the reliability of the system since the possibility of joining the conference is not conditioned by the status of the conference owner.

15.3 The Prototype

A prototype of the videoconference system described here has been implemented using the Java language. The support for mobility has been provided by the μCODE toolkit described in Chapter 14, that was developed concurrently with the system to provide a flexible mobile code infrastructure with limited overhead.

The client implementation makes use of the public domain tools `vic` and `vat` to capture and playback the video and audio streams, respectively. In addition, a console written in Java allows the user to communicate with the reflector and set the parameters of the session. Currently, we are implementing an integrated version of the client using the Java Media Framework [106]. This new version is aimed at integrating the console with the visualization and playback

[1]Consider as an example the MBone conference announcements which are distributed through `sdr` and contain the multicast address of the conference. In the framework of our conferencing system the announcement contains the address of the client which generated the announcement.

of the conference streams, in order to present the user with a single and uniform interface.

Presently, the implementation of the reflector does not include any authentication and acces control features as we decided to focus on the impact of customizability and mobility. Also, the extended architecture encompassing cloning is not yet implemented.

15.4 Discussion and Future Work

The work described in this chapter investigates how a mobile code design can be applied to an application for active networks. The selected application, videoconference, benefits of the advantages provided by code mobility by exhibiting the capability to adapt to network conditions by exploiting mobility through a MA paradigm, and providing flexibility through the COD paradigm.

The approach is inspired by the work described in [89] concerning a chat server that migrates to adapt to the position of the participants. That work focuses on the benefits of the network aware positioning of the server with respect to a traditional fixed displacement. In our work the potential of code mobility and active networks have been exploited also in terms of customizability which can be strategic in the videoconference application domain. Also, the aforementioned work was mainly concerned with mechanisms needed to probe and monitor resources. In our work, we concentrated on the exploitation of mobility; however, devising efficient strategies for relocation to be embedded in the locator object is the subject of future research. This encompasses individuating and defining which are the mechanisms that must be provided at the application level and which can be assumed to be provided by the network device—still an open issue in active network research.

Finally, another line of evolution for what described here is the quantitative evaluation of the benefits provided by our design with respect to traditional ones, along the lines of what described in Part II.

Chapter 16

Coordinating Mobile Components Through Tuplespaces

The importance of computer networks has reached a point where users demand computing facilities that encompass ubiquitous access to the network, independent from the users' physical locations, as we mentioned in the introduction to this thesis. This style of computation, often referred to as *mobile computing*, is currently enabled by wireless communication technology. The network scenario enabled by mobile computing ranges grossly between two extremes. At one extreme, the availability of a fixed network is assumed, and some of its conventional facilities are exploited by the mobile infrastructure. This is the case of the Mobile IP [85] protocol. At the other extreme, the mobile network has no connection with a fixed network and all the network facilities, e.g., routing, must be implemented by relying only on the availability of mobile hosts. This is the assumption made by the research area concerned with *ad hoc* networks [55]. The characteristics of the physical communication media, in terms of limited bandwidth and frequent disconnection, bring into mobile computing a decoupled and opportunistic style of computation. Computation is decoupled in that it should proceed even in the presence of disconnection—a rather common event in this domain. Computation is opportunistic in that it exploits connectivity whenever it becomes available. This peculiar style of computation demands a new approach to the crafting of distributed application.

As for mobility, mobile computing can be regarded as representing a form of *physical mobility* where the hosts roam across the physical space and modify the topology of the network; this is complementary to code mobility, that provides a form of *logical mobility* where the entity being migrated is the code running on the hosts, which are usually not allowed to move.

In both the domains, the notion of location seems to play a key role. Different locations provide a different context and, in turn, different services. Thus, for instance, a laptop may access a printer only if it is in proximity of it. Analogously, relocation of code may take place to access remote resources that are not present on other nodes. Despite these apparent similarities, understanding the relationships and the synergies between these two forms of mobility is still the subject of research.

In this work we present a system, called LIME, which is intended to provide application designers and implementors with powerful yet minimal abstractions to deal indifferently with physical or logical mobility—or both at the same time. In our system, physical mobility is concerned with the movement of mobile hosts, such as laptop or PDAs. Logical mobility is concerned with the movement of *mobile agents*, i.e. processes that are able to migrate from one host to another by preserving their code and state. The underlying assumption in our work is that both mobile hosts and mobile agents can be regarded as derivations of a generic concept of mobile component, whose instances can be *coordinated* in the same way.

The work described in this chapter is the subject of on-going work with Prof. Gruia-Catalin Roman, Amy L. Murphy, and Gustavo Rosini at Washington University in St. Louis, USA.

16.1 Overview

The abstractions provided by LIME rely on the notion of a *tuple space* as described in the Linda model [40]. In Linda, coordination is achieved through communication taking place in a tuple space globally shared among several components that, independently from their actual location, can access the tuple space by inserting, reading, or withdrawing tuples containing information. This model is said to be decoupled in space and time, in that the components do not need to be in communication at the same time and can reside anywhere in the distributed system—the tuple space provides for decoupling. As we mentioned earlier, decoupling is intrinsic to mobility, which makes the Linda model a natural choice for our system.

Every mobile component is permanently associated with its own tuple space, called *interface tuple space* (ITS). Conceptually, the ITS contains tuples representing information that the component wants to share with other components. (It is conceivable to associate also a *private tuple space* to each component for storing tuples of information that should not be shared, but we disregard this issue for the time being.) The global tuple space is built dynamically by merging the ITSs of the various components. This is performed according to two tiers of abstraction, encompassing physical and logical mobility, as illustrated in Figure 16.1.

All the mobile agents residing on a host share a global tuple space that is built dynamically by merging the ITS of all the agents currently co-located on that host. Each agent, however, still transparently accesses the global tuple space through its own ITS. Upon departure of the mobile agent, its ITS is disenganged and removed from the machine to follow the agent, and the global tuple space on the source laptop is updated accordingly by the system. Similar consideration holds for engagement of tuple spaces upon arrival of a mobile agent. Thanks to the decoupling provided by the notion of tuple space, the reconfiguration of its contents is transparent to the other agents sharing it, as discussed further in the remainder of the paper. This global tuple space described so far constitutes the first tier, and provides coordination to co-located mobile agents by enabling

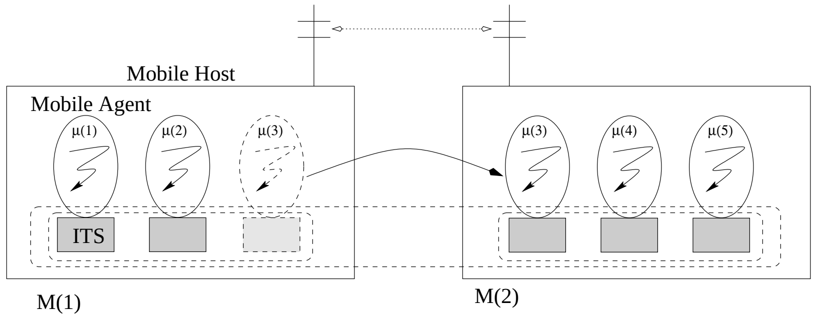


Figure 16.1: Physical and logical mobility, a two tier structure.

them to share information.

The second tier of abstraction looks at the same system at a different level of granularity. Like mobile agents, mobile hosts can be regarded as being associated with an ITS. The ITS of a mobile host is constituted by the global tuple space constructed from the ITSs of the mobile agents located on the hosts. Thus, all mobile hosts share a global tuple space that is constructed by merging the ITSs of all the mobile hosts. Only this time, engagement and disengagement of the ITS is ruled by connectivity. Whenever a disconnection occurs, the tuple space of the mobile host is disengaged from the global tuple space.

Hence, LIME can be used as a coordination infrastructure in different ways. When only the first tier is used, it provides support for coordination of mobile agents, and tuple spaces on different hosts are disjoint while the tuple spaces of co-located agents are merged. When only the second tier is used, LIME provides coordination among applications running on the mobile hosts. In this case, the ITS of each host is generated by one or more stationary agents residing on it. Finally, the two tiers can be used together, providing for coordination of physical and logical mobile components at the same time.

The final goal of LIME is to provide an extensible middleware in the form of a library for the programmer. An implementation of LIME for the Java language is currently under development. The abstractions and operations provided by LIME strive for minimality and composability, thus enabling the programmer to build higher level abstractions on top of our system.

In this work, we describe the abstractions and operations provided by LIME. In order to define precisely their meaning, and to avoid restraining the description to a given programming language, we define a formal semantics for LIME. The semantics is described using the Mobile UNITY already discussed in Part III.

The paper is structured as follows. Section 16.2 provides the reader with the necessary background on the Linda model. Section 16.3 is the core of the paper. It presents our system by proceeding through layers of increasing complexity, showing how it is possible to extend Linda incrementally in order to obtain the LIME operations as a composition of simpler primitives. For each layer, we give the formal semantics of the operations introduced using the Mobile UNITY notation. Section 16.4 sketches architectural and implementation issues involved in the on-going development of the system. Finally, Section 16.5 draws some conclusions and describes directions for future work.

16.2 Linda

Linda has been proposed at the beginning of the past decade [40] as a new model of communication among concurrent processes. The fundamental abstraction provided to each process is a shared *tuple space* that acts as a repository of elementary data structures—the *tuples*. Tuples contain the actual information being communicated, and can be read, inserted, and removed from the tuple space by the processes. Tuples are anonymous, and selection for reading or removal takes place through pattern matching on the contents of the tuple.

The communication model of Linda is peculiar under many respects. It is *generative*, because the tuple generated by a process has an independent existence in the tuple space until another process withdraws the tuple from the tuple space. Communication is *implicit*. After a tuple is inserted in the tuple space, it is not bound to any particular process. Thus, differently from other well-established paradigms of communication like message passing, processes do not address explicitly the other party involved in communication. Another way to regard this aspect is in the fact that communication is *content-based* rather than name-based, in that the binding between the information being communicated and its intended recipient is not established statically using some naming scheme, rather it is determined dynamically and non-deterministically by the content of the information. In a sense, tuples embody a notion of generalized structured name. Finally, communication in Linda is decoupled in *time* and *space*. Decoupling in time refers to the fact that sender and receiver do not need to be in communication in order to exchange information. Tuples are stored in the tuple space and can be retrieved later, even if the process that produced the tuple has already terminated its execution. Decoupling in space refers to the fact that a tuple in the tuple space is available to a number of processes that can be dispersed on the nodes of a distributed system—their actual location is abstracted into the tuple space and thus hidden from the tuple producer.

This notion of decoupling is appealing, as it separates clearly the behavior of the individual processes from the communication needed to coordinate their actions. This idea has encountered wide acceptance in many scientific communities ranging from economics to artificial intelligence, and is at the core of a new interdisciplinary research area that investigates technologies and methodologies for the *coordination* of components in complex systems [67, 39]. In particular, as far as software engineering is concerned, the decoupling between components and their coordination realized by Linda can be regarded as fostering the distinction between components and their interconnection which is at the root of research in languages and methods for software architectures [99].

Due to the emphasis on the coordination among components rather than on the description of their behavior, Linda was not conceived to be a standalone language. The implementations of Linda are usually constituted by a run-time support and a library that are tailored to a specific language, which is used to program the behavior of the processes. Thus, implementation of

Linda/C, Linda/Eiffel, Linda/Pascal exist. Efficient implementations of Linda have been developed for a variety of parallel and conventional hardware, and are being exploited in many different real world applicative domains. Thus, the Linda model has been succesful not only in stimulating a considerable body of research in many sectors of computer science, but also in providing viable solutions for real problems.

In the work we are going to describe, we chose to rely on the Linda model due to its minimality and to the decoupling in time and space it provides. Ideally, minimality corresponds to ease of implementation, while decoupling is a key characteristic of applications involving physical and logical mobility. The next section illustrates how we developed this intuition by preserving the original phylosophy of Linda.

16.3 From Linda to LIME

This section provides an incremental development path from a traditional Linda model to LIME, Linda in a Mobile Environment. First we present formal semantics for the standard Linda operations **out**, **in**, and **rd**. Next we introduce a new Linda primitive, **change**, to atomically alter a tuple without removing it from the tuple space. Following this, we introduce one of the operations from Mobile UNITY and describe its usefulness in the Linda environment. Finally we introduce fundamental concepts in mobility such as location, engagement, and disengagement and show how the previous ideas are combined and expanded to yield a new model for both physical and logical mobility.

16.3.1 A Formal Semantics for Linda

In Linda, a tuple space is a multiset of tuples that can be accessed concurrently by several processes, often called *workers*. Tuples are a list of typed parameters, such as ⟨"foo", 9, 27.5⟩. Linda provides operations for inserting to, removing from, and reading tuples in a tuple space. The corresponding operations are **out**, **in**, and **rd**. Linda implementations typically provide also an **eval** operation which is used primarily to delay the evaluation of a tuple parameter. For our purposes, we use hereafter only **out**, **in**, and **rd**.

Tuples are added to a tuple space by performing an **out**(t) operation on it. After its execution, the tuple t is available to any subsequent operation on the tuple space. The update of the tuple space is performed atomically, and can never fail. In addition, **out** is non-blocking, i.e. the process invoking the **out**(t) does not wait for the update of the tuple space to complete.

Tuples can be removed from a tuple space by executing **in**(p). The resulting tuple is assigned to same variable belonging to the worker that invoked the **in**. The argument p is often called a *template*, and its fields contain either *actuals* or *formals*. Actuals are values; the parameters of the previous tuple are all actuals, while the last two parameters of ⟨"foo", ?x, ?y⟩ are formals. Formals are like "wild cards", and are matched against actuals when selecting a tuple from the tuple space. The template above matches the tuple defined earlier, provided that x and y are declared with a type compatible with that of the corresponding actuals. Tuples can contain formals as well. The matched tuple

Program *ProducerConsumer*
  declare
    T : set of Tuple $[\!]$ p,q : Tuple $\cup \{\perp\}$ $[\!]$ x,i_C,i_P : integer
  initially
    $T = \{\}$ $[\!]$ $p,q = \perp,\perp$ $[\!]$ $i_c,i_p = 0{,}0$
  assign
    $p := \langle i_P,\mathsf{Produce}()\rangle$ **if** $p = \perp$
    $[\!]$ $\mathbf{out}(T,p) \parallel p,i_P := \perp,i_P + 1$ **if** $p \neq \perp$
    $[\!]$ $q,i_C := \mathbf{in}(T,\langle i_C,?x\rangle),i_C + 1$ **if** $q = \perp$
    $[\!]$ $\mathsf{Consume}(q) \parallel q := \perp$ **if** $q \neq \perp$
end

Figure 16.2: Modeling a Linda producer and consumer in UNITY.

and the template must have the same arity. If more than one tuple exists matching a template, the one returned to **in** is selected non-deterministically. The **in** operation is blocking, i.e. if no matching tuple is available in the tuple space the process performing the **in** is suspended until a matching tuple becomes available.

Tuples can also be read from the tuple space using the **rd** operation. The execution of a **rd**(p) proceeds identically to the **in**, except for the fact that the tuple matched and delivered to the worker that executes the operation is copied rather than withdrawn from the tuple space. Similarly to the **in** operation, **rd** is blocking.

The semantics of these operations are defined in slightly different ways within the various Linda dialects. Hence, an immediate need at this point is to define precisely the semantics we choose for our system. Also, as we pointed out before, Linda is not a language per se, and the details of the interface between the host language and Linda operations vary according to the implementation. However, in this work we are interested in the Linda model and not in the technicalities of a particular implementation. Thus, besides defining precisely the semantics of Linda operations we need also to avoid the details of the host language to capture the essence of Linda operations.

The formal tool we rely on to accomplish this goal is the UNITY notation described in [25] and already discussed in Part III. Its notation and logic provide an adequate level of formality for our description. Also, UNITY does not rely on any programming language, and provides a minimal set of concepts and notation for describing concurrent programs, thus providing the adequate level of expressiveness for our description. In the remainder of this section we recall the UNITY notation by modeling a producer-consumer system that exploits a shared tuple space for communication. In this system, described in Figure 16.2, Linda operations must be considered in the light of the informal semantics we gave earlier, and treated like ordinary UNITY statements. Then, we show how their semantics can be expressed in UNITY.

In the producer-consumer example of Figure 16.2 the tuple space T is defined as a set of tuples. This is not in conflict with the notion of a tuple space constituted by a multiset of tuples: we adopt this solution only for mathematical convenience in defining the semantics of our system. We assume that each tuple t, besides the fields specified by the programmer, is uniquely identified by the content of an additional field hidden to the programmer. This way, a tuple space can be treated more conveniently, and the tuple space available to the programmer can be seen as a projection of the tuple space used to define the semantics. p and q are singleton tuples, used as parameters for tuple space operations. We assume that a sound definition for the type Tuple exists. The variable x is an integer that is used to define a template for selecting the tuples to be consumed. The variables i_P and i_C are integers that are used to index the tuples in order to ensure that they are consumed in the same order they are produced. In our example, the tuple space is initially empty, while p and q are initially undefined, as specified in the **initially** section.

The first statement in the **assign** section of the program models the production of a new tuple in the tuple space. It assigns to the variable p a pair constituted by an index i_P and the result of some function $\mathsf{Produce}()$ that we assume embodies the behavior of the producer and whose details we leave unspecified. This assignment is guarded in order to prevent the generation of a new tuple p when this already exists, although it has not yet been inserted into the tuple space. The parallel composition operator $\|$ allows for synchronous execution of multiple statements. The second statement actually performs such an insertion using the **out** operation, and at the same time clears the input tuple p and updates the tuple index. The statement on the right hand side of the parallel operator is a multiassignment, and can be regarded as a notational shortcut to express parallel composition of multiple assignments. Hence, the statement $p,i_P := \perp, i_P + 1$ is equivalent to $p := \perp \;\; \| \;\; i_P := i_P + 1$. The third and fourth statements are the analogue of the first two statements for the consumer.

As we anticipated earlier, in the program shown in Figure 16.2 Linda operations are considered as ordinary UNITY statements. However, they must be regarded more properly as *macros* defined for notational convenience, whose meaning is actually represented in terms of the basic UNITY statements, which we define in this last part of the section.

Relying on the declarations provided for variables T and p, the semantics of the **out** operation can be expressed naturally as

$$\textbf{out}(T,p) \triangleq T := T \cup \{p\},$$

that is, the tuple p is simply inserted in the set of tuples representing the tuple space T. By including T as a parameter, we allow each program to have multiple tuple spaces, such as both the interaction and private tuple spaces mentioned earlier. Execution of statements in UNITY is atomic, hence an **out** defined as above actually blocks the process until completion. However, no guard is present on the assignment implementing the **out**, thus it always completes.

The semantics for **in**, and similarly **rd**, is slightly more complicated because it must account for the non-deterministic selection of the matching tuple. The template p can be thought of as a predicate specifying the pattern matching that must take place between the template and a candidate tuple. To express the fact that a tuple θ matches a template p, we use the notation $\mathcal{M}(\theta,p)$. Then, we can give the semantics of **in** as

$$\begin{array}{ll} t := \textbf{in}(T,p) \triangleq & \langle \| \; \theta : \theta = \theta'.(\mathcal{M}(\theta',p) \wedge \theta' \in T) :: T,t := T - \{\theta\},\theta \\ & \quad \textbf{if} \; \langle \exists \, \theta : \theta \in T :: \mathcal{M}(\theta,p) \rangle \rangle \end{array}$$

where $t : \mathsf{Tuple}$. The statement is guarded to guarantee that it is executed only

Program *ProducerConsumer-OneSpaceBuffer*
declare
T : set of Tuple $[]$ p,q : Tuple $\cup \{\perp\}$ $[]$ x,i_C,i_P,i'_P : integer
initially
$T = \{\langle\text{'}synch\text{'},0\rangle\}$ $[]$ $p,q = \perp,\perp$ $[]$ $i_C,i_P,i'_P = 0,0,0$
assign
$p := \langle i_P, \mathsf{Produce}()\rangle$ **if** $p = \perp$
$[]$ $\mathsf{rd}(T,\langle\text{'}synch\text{'},?i'_P\rangle)$
$[]$ $\mathsf{out}(T,p) \parallel p,i_P := \perp,i_P+1$ **if** $p \neq \perp \wedge i'_P = i_P$
$[]$ $q,i_C := \mathsf{in}(T,\langle i_C,?x\rangle),i_C+1 \parallel$
$\mathsf{change}(T,\langle\text{'}synch\text{'},?i_C\rangle,\langle\text{'}synch\text{'},?i_C+1\rangle)$ **if** $q = \perp$
$[]$ $\mathsf{Consume}(q) \parallel q := \perp$ **if** $q \neq \perp$
end

Figure 16.3: Synchronizing producer and consumer with a **change** operation.

when a matching tuple is present in the tuple space. The notation

$$\theta = \theta'.(\mathcal{M}(\theta',p) \wedge \theta' \in T)$$

is inspired by the notion of *non-deterministic assignment* as defined in [5]. In a non-deterministic assignment $x := x'.Q$, x is assigned a value x' selected non-deterministically among those satisfying condition Q. In our case, we use a similar notation to select non-deterministically a tuple θ' to the tuple θ used to quantify the three-part notation. A singleton tuple is returned, hence in this case the parallel operator in the three-part notation serves the purpose to create a quantified statement, and does not express parallel execution of multiple statements. The statement then executes removing the tuple from the set in T and copying its value in t, which models the variable actually returned to the program invoking the **in**. In the UNITY model, each single statement can be regarded as a separate thread of control. Hence, the statement takes into account blocking in that it never executes until a tuple matching its template appears in the tuple space.

The semantics of **rd** follows directly from the one defined for **in**, and the only difference is that the matching tuple θ is not removed from the tuple space:

$$t := \mathsf{rd}(p) \triangleq \langle \parallel \theta : \theta = \theta'.(\mathcal{M}(\theta',p) \wedge \theta' \in T) :: T,t := T,\theta \;\; \textbf{if}\; \langle \exists \theta : \theta \in T :: \mathcal{M}(\theta,p)\rangle\rangle.$$

16.3.2 Updating Tuples Atomically

As a first extension to the basic Linda model, we introduce the **change** operation. Basically, **change** provides a mechanism to perform "in-place" atomic updates of a tuple. Updates may encompass changing the value of a field, changing an actual into a formal, and adding or removing fields of the tuple. The syntax for this operation uses a two part notation that specifies the template used to select the tuple to be changed and a replacement for it. Thus, **change** $(T,p,\mathcal{F}(p))$ means that one of the possibly many tuples matching the template p is replaced atomically by the tuple obtained by applying the function $\mathcal{F}$ to the fields of p. If no such tuple exists, the **change** operation blocks.

For example, $\mathsf{change}(T,\langle\text{'}label'\text{,}?x\rangle,\langle\text{'}label'\text{,}?x+1\rangle)$, increments the value of the second field of a tuple chosen non-deterministically (from tuple space T) among those whose first field is '$label'$ and second field is an integer, assuming the declarations in the producer-consumer example. It is worth noting how the variables in the two patterns are considered bound in the same scope. Also, since the **change** operation can declare new fields fresh identifiers can be used when specifying the update pattern, together with proper declarations. For instance, **change**(T,$\langle$'$label'$,?$x\rangle$,$\langle$'$label'$,?$x+1$,?$y\rangle$), y : integer extends the previous tuple with an additional integer field.

From the discussion so far, it follows that the semantics of **change** consists of the combination of an **in** and an **out** performed in sequence and atomically:

$$\textbf{change}(T,p,\mathcal{F}(p)) \triangleq\ T := (T - \{\theta\}) \cup \{\mathcal{F}(\theta)\}\ \textbf{if}\ \langle \exists\, \theta : \theta \in T :: \mathcal{M}(\theta,p)\rangle$$

To illustrate a possible use of the **change** operation, we return to the producer-consumer and effectively turn the tuple space into a one place buffer by adding a synchronization tuple that stays permanently in the tuple space. Figure 16.3 shows that the producer reads the value in the synchronization tuple. When this value and the producer's index value are equal, the producer knows that the last tuple it put out into the shared tuple space has been consumed. Therefore, the **out** action is enabled and a new data tuple can be put into the space. The consumer has the responsibility to update the synchronization tuple each time it removes a tuple. This is accomplished using the **change** operation. Although it is possible for the consumer to simply do an **in** followed by an **out** with a new index value, we do not have the notion of a transaction, or sequence of actions, in the UNITY model, and therefore we would have to add unnecessary complexity to guard the statements such that no consumer operations occurred while the synchronization tuple was being held by the consumer. Further, by using the **change** operation we are assured that the producer never blocks on the **rd** of the synchronization tuple.

16.3.3 Reacting to Events

The other operation we introduce before defining the full LIME system is the reactive statement **reacts-to** from Mobile UNITY[1]. The reactive statement extends UNITY with the capability to specify actions that must be executed immediately after a given condition occurs rather than by relying on fair interleaving, e.g., to model interrupts or event driven computations. A program can specify several reactive statements. In order to guarantee termination and preserve fairness, the reactive statements of a program are combined to form a single reactive program that is executed to fixed point after the execution of every conventional statement in the program. It is the responsibility of the programmer to ensure fixed point is actually reached for the reactive program. The syntax of a reactive statement, s **reacts-to** p, augments that of a conventional statement, s, with a reactive clause that strengthens its guard, p. The introduction of reactive statements affects the UNITY logic, and proper inference rules are needed to prove the correctness of reactive programs. The interested reader can find details about this subject in [70].

Elaborating again on the producer-consumer example, we return now to the bounded buffer but grant the producer the capability to send a high priority

[1] This is the reason why we **reacts-to** with a font different from the one used for **in**, **rd**, **out**, and **change**, to distinguish primitives that are defined in Linda from those that belong to Mobile UNITY.

Program *ProducerConsumer–WithPriority*
  declare
    T : set of Tuple $[]$ p,q,r : Tuple $\cup \{\perp\}$ $[]$ x,i_C,i_P : integer
  initially
    $T = \{\}$ $[]$ $p,q,r = \perp,\perp,\perp$ $[]$ $i_C,i_P = 0,0$
  assign
    $p := \langle i_P,$Produce()$\rangle$ **if** $p = \perp$
    $[]$ **out**(T,p) $\|$ $p,i_P := \perp,i_P + 1$ **if** $p \neq \perp$
    $[]$ **out**(T,$\langle$*'priority'*,Produce()$\rangle$)
    $[]$ $q,i_C :=$ **in**(T,$\langle i_C,?x\rangle$),$i_C + 1$ **if** $q = \perp$
    $[]$ Consume(q) $\|$ $q := \perp$ **if** $q \neq \perp$
    $[]$ $r :=$ **in**(T,$\langle$*'priority'*,$?x\rangle$) **reacts-to** $\langle \exists\, \theta : \theta \in T :: \mathcal{M}(\theta,\langle$*'priority'*$,?x\rangle)\rangle \wedge r = \perp$
    $[]$ Consume(r) $\|$ $r := \perp$ **reacts-to** $r \neq \perp$
end

Figure 16.4: Adding priorities with the **reacts-to** operation in the producer-consumer example.

tuple that is processed immediately and separately from the low priority ones. The producer can output the high priority tuple at any time. Since it models some special event distinct from normal processing it is placed directly in the tuple space rather than in the buffer p. Also, this tuple does not receive an index value. The reactive behavior of the consumer process is modeled by two **reacts-to** statements. The first statement reacts to the presence of a tuple tagged with the *'priority'* label in the first field, and withdraws the tuple storing it in the variable r, normally undefined. The second statement reacts to the presence of a value in r, that means that the processing of a high priority tuple is active, by consuming the tuple and restoring r to undefined. Although the two statement are distinct, they are executed as a separate UNITY program after each non-reactive statement in Figure 16.4. It is easy to see that the reactive program so constructed always reaches fixed point. Also, the inspection of the program text reveals that the reactive program can start only after a high priority tuple has been placed in the tuple space; in the other cases the guards of the reactive statements prevent their execution, by implying that fixed point is immediately reached.

16.3.4 LIME: Linda in a Mobile Environment

The features provided by the core Linda model as presented in Section 16.3.1 already provide for coordination among components that execute concurrently. However, the concept of tuple space fostered by Linda is a static one; even in a distributed setting, Linda relies on the availability of a persistent, globally accessible tuple space that resides permanently on a pool of constantly interconnected machines. The approach we describe in this section makes different assumptions that are motivated by the peculiar characteristics of mobility.

A fundamental assumption is that connectivity cannot be taken for granted. Some studies [126] have conceived fault-tolerant implementations of Linda; they aim at increasing availability of tuples during temporary absence of connectivity by employing replication schemes. Nevertheless, even in this case disconnection is considered to be an accidental property of the system rather than one of its intrinsic properties as in mobile computing. Mobile hosts do not disconnect only due to increased noise or limited bandwidth; most often, disconnection is forced explicitly by the user, e.g., because he wants to quit the session and move physically to another location or in order to save battery power or reduce communication costs over cellular lines. Mobile hosts are completely independent. For example, because they are operated by occasional users, they can come in contact and never meet again. In this highly dynamic scenario where mobile hosts form transient communities, the idea of a static, persistent, and globally visible tuple space becomes less appealing.

The approach taken by LIME is to permanently associate with each mobile component an *interface tuple space* (ITS) which is accessed using the conventional Linda operations **in**, **rd**, and **out** described in Section 16.3.1. The semantics of these operations is unaltered; simply, the tuple space they operate on is the component's ITS rather than a globally known one. Whenever the mobile component moves, the corresponding ITS is moved as well. Section 9 illustrated how the dynamic composition of the ITSs provides a definition for two tiers of abstraction in LIME. The composition of the ITSs of mobile agents co-located on a host constitutes the first tier that present each mobile agent with a unique tuple space global to the mobile host. This tuple space, in turn, constitutes the ITSs of the mobile host that can be further composed with the ITSs of all the mobile hosts directly or indirectly connected with it, thus providing the second tier of abstraction. The implementation of the LIME system enables the programmer to use either or both tiers of abstraction. If both tiers are used, each mobile component accesses through its own ITS the union of the tuples contained in the ITSs of all the mobile components currently connected, as defined formally in the following.

Thus, in LIME there is no tuple space that exists a priori and is statically visible to every component. Rather, the contents of the ITS are reconfigured dynamically and automatically by the system according to the current composition of the community of mobile components. In a sense, LIME takes the notion of decoupling proposed by Linda further, by effectively decoupling the mobile components from the global tuple space coordinating them.

A question arises at this point about the criteria needed to rearrange the ITS of a mobile component when it disengages from the global tuple space. Before disconnection or departure, the ITS of the component provides access to the union of the ITS of all the components currently connected; thus, the question becomes determining which tuples are kept in the ITS of the departing component. In principle, three alternatives are available. A mobile component could keep all the tuples currently in the ITS. However, this is likely to accumulate continuously large quantities of tuples every time the ITS is re-engaged with those of other components. Even worse, if the same community of components is rejoined right after a transient disconnection, this solution would raise nontrivial problems in the treatment of duplicate and possibly inconsistent tuples. The second alternative is constituted by a mobile component that does not take any tuples with it upon disengagement. However, this would be a rather weak model. This means essentially that components do not have a means to retain relevant information in their ITS and possibly make it available to other components at a different location. Taking some of the tuples is the criteria we choose, but then the question becomes how how to select the tuples that must remain with the ITS.

The criteria is provided by another fundamental concept in our system:

location. For mobile hosts, location in space determines the availability of connectivity. For mobile agents, a location defines the computational environment where the component can be executed. In both cases, locations determine the context for the computation and the services that can be exploited by the mobile component. Thus, since in LIME tuples take a fundamental role in the computation, it is meaningful to associate a location with tuples and provide mechanisms for their relocation. In general, LIME strives for minimality in the mechanisms and abstractions it supports. Locations are not an exception in this respect. In LIME, the location of a tuple is constituted by ITS of a mobile agent, and thus can be safely expressed by the identifier of the mobile agent. The programmer can build more complex schemas on top of this notion of location by exploiting the other features of LIME, as discussed later in this section.

In order to handle relocation of tuples upon disengagement, we enforce a given tuple structure that reserves two fields to handle location. One of the fields is managed only by the system and contains the current location of the tuple. The other field is indicated by the application and contains the final location of the tuple, i.e. where the tuple should be placed eventually. This occurs when the tuple is produced, and is specified by using a variant of the **out** operation that is annotated with the final location. Thus, **out**$[\lambda]$ outputs a tuple that is destined to be placed at location λ. When an agent performs **out**$[\lambda]$, the tuple is always inserted first in the agent's ITS. If the intended recipient λ of the tuple is currently not reachable, the tuple remains in this ITS. Otherwise, if and when the intended recipient becomes reachable the tuple is delivered to its ITS. Applications are not forced to specify the final location, and can still use the conventional Linda operation **out**. In this case, the tuple remains with the mobile agent that created it until it is removed from the tuple space.

The modifications to Linda mentioned so far do not affect the decoupling in time and space and yet provide the user with the capability to retain some degree of control on where the information is placed. For instance, this control may be useful to applications for leveraging off of knowledge of network topology or about the other components' behavior in order to improve efficiency. Also, communication is still implicit even if a final location is specified in the tuples; selection of a tuple is still based on pattern matching independent of the location of the tuple. It is worth noting at this point that specifying a final location does not enforce a notion of ownership. The final location does not constrain in any way the execution of **in** and **rd**. "Misplaced" tuples still belong conceptually to a globally shared tuple space, and anybody can withdraw or read a tuple independently of the final responsible associated to it. Thus, it may be the case that, even if the final responsible is not presently connected, another agent removes the tuple from the space. This is consistent with the Linda model, where no notion of ownership is associated with tuples. Our notion of responsibility is only a clue given to the system to provide relocation of tuples in the best position. If strong guarantees about the delivery of a tuple must be defined, i.e. a notion of tuple ownership is needed, this can be implemented as an additional layer on top of what is described here, in particular by restricting the **in** and **rd** operations to tuples owned by a particular agent.

The semantics of **out**$[\lambda]$ provide support for the decoupled and opportunistic communication among mobile components that is intrinsic to mobile computing.

However, LIME provides location-enhanced versions of the other Linda operations, **in**$[\lambda]$ and **rd**$[\lambda]$. These operations, in turn, break the Linda property of implicit communication in that they access only the ITS specified by λ or, in other words, looking only at tuples in the global tuple space that have λ as the actual in the current location field.

In the remainder of this section we define formally the concepts mentioned so far.

Characterizing Mobile Components

In order to treat mobile components formally, we need a notation to distinguish them. Also, since we want to describe the mobility of components through physical and logical locations, we need to characterize location as well. Clearly, the identity of a mobile component is separated from its location: multiple components can be co-located and yet they are different.

We represent a mobile host as a pair $M = \langle x,i \rangle$, x representing a physical location and i a globally unique identifier. Informally, the first component defines "where" the mobile host is and the second defines "who" it is. At the level of detail we are concerned within this model, the details of the types for x and i are irrelevant. We can safely think of x as being a pair of coordinates in some reference system and of i as the IP address of the mobile host.

Mobile agents are represented similarly with a pair $\mu = \langle M,j \rangle$. The first field is a mobile host M that effectively defines the location for a mobile agent. The second field is a globally unique identifier j.

For convenience, hereafter we use the notation $M(i)$ to refer to a mobile host M such that $M \uparrow 2 = i$, and $\mu(j)$ to refer to the agent μ for which $\mu \uparrow 2 = j$.

Defining Connectivity

Connectivity has different meanings according to the abstraction tier we are considering. For mobile hosts, connectivity is mapped directly onto the availability of a communication link between two hosts. In our model, we capture this by defining a relation κ such that $M(i)\ \kappa\ M(j)$ means that the mobile hosts with identifiers i and j are in *direct* communication. To model intuitively the characteristics of the mobile computing domain, κ is defined as a reflexive, transitive, and commutative relation.

As for mobile agents, two agents are in *direct* communication when they are on the same host or when the host they reside upon are in direct communication. To define this property of mobile agents we define a relation γ based on the previous definition of κ:

$$\mu(i)\ \gamma\ \mu(j) = (\mu(i) \uparrow 1)\ \kappa\ (\mu(j) \uparrow 1)$$

The reflexivity of κ accounts for the fact that $\mu(i)$ and $\mu(j)$ could be on the same mobile host. Notably, the relation is defined, for notational convenience in the formulae to follow, on the domain of agent identifiers.

Finally, connectivity does not necessarily imply direct communication. In mobile computing, two mobile host that are not in communication range can still be connected through a third host that is in communication range with both of them and is able to route correctly their packets. Indirect, transitive communication is expressed naturally by defining $\Gamma \triangleq \gamma^*$, the transitive closure of γ. It enables us to express concisely the fact that two mobile agents $\mu(i)$ and $\mu(j)$ are indirectly connected through an unspecified chain of mobile hosts by writing that $i\,\Gamma\,j$.

As an aside, it is interesting to note that, although we use these mathematical abstractions to represent connectivity, we should still view them as UNITY variables whose modification we do not model in our system.

Specifying the Location of Tuples

In order to cope with disconnection, tuples are tagged with two additional fields dealing with their location. Thus, a tuple $t = \langle f_1, f_2, \ldots, f_n\rangle$ as specified by an application with $\textsf{out}[\lambda](T,t)$ is actually considered by the system as $t' = \langle \omega, \lambda, f_1, f_2, \ldots, f_n\rangle$. The *current location* ω is the identifier of the mobile agent whose ITS currently contains the tuple, while the *final location* λ is the identifier of the mobile agent that is intended to be eventually responsible for the tuple. When the tuple is output by ω by performing $\textsf{out}[\lambda]$ the tuple is inserted in the ITS of ω and its current owner is changed accordingly. Formally, this means that the definition of $\textsf{out}[\lambda]$ is[2]:

$$\textsf{out}[\lambda](\mu(\omega).T,t) \triangleq \mu(\omega).T := \mu(\omega).T \cup \{\langle \omega, \lambda, t\rangle\},$$

where we use the notation $\mu(\omega).T$ to specify the ITS of the mobile agent $\mu(\omega)$ performing the operation. In turn, the definition of $\textsf{in}[\lambda]$ and $\textsf{rd}[\lambda]$ is obtained by strengthening properly the guards of the original operations. For instance, the semantics of $\textsf{in}[\lambda]$ is:

$$\begin{aligned} t := \textsf{in}[\lambda](\mu(\omega).T,p) \triangleq \quad & \langle \| \ \theta : \theta = \theta'.(\mathcal{M}(\theta',\langle\lambda,?\nu,p\rangle) \wedge \theta' \in T) :: T,t := T - \{\theta\},\theta \\ & \textbf{if } \langle \exists\, \theta : \theta \in T :: \mathcal{M}(\theta,\langle\lambda,?\nu,p\rangle)\rangle\rangle, \end{aligned}$$

if ν is declared to be a location.

Relocating Tuples

Due to the decoupled and opportunistic style of computation fostered by LIME, if $\textsf{out}[\lambda](\mu(\omega).T,t)$ is issued and the agent λ is presently not connected, i.e. $\omega\,\Gamma\,\lambda$ does not hold, the tuple remains with ω. As soon as λ becomes connected, however, the tuple is transferred into its ITS and the current location changed accordingly to λ. This behavior of the system is described using the **change** and **reacts-to** operations defined in the previous sections:

$$\begin{aligned} &\langle [\!]\, \omega :: \mu(\omega).T := \langle \text{set}\, t : t \in \mu(\omega).T :: \textsf{change}(\langle\omega,\lambda,t\rangle,\langle\lambda,\lambda,t\rangle) \textbf{ if } \mu(\omega)\ \Gamma\ \mu(\lambda)\rangle\rangle \\ &\textbf{reacts-to } \textit{true} \end{aligned}$$

As discussed in Section 16.3.3, reactive statements are run all together as a single reactive program after the execution of any conventional statement. Hence, the above possibly updates the ITS of any agent after the execution of any conventional statement. In particular, it takes care of changing properly the responsibility of a tuple upon engagement of two tuple spaces, or when an **out** is performed on a tuple that has a final responsible different from the current one, and the two are connected, possibly transitively. Changing the final location

[2] If $t = \langle f_1, f_2, \ldots, f_n\rangle$, for the sake of readability we use the notation $\langle x,t\rangle$ to express $\langle x, f_1, f_2, \ldots, f_n\rangle$.

in our implementation actually corresponds to tuple movement, as discussed in the following section. Thus, by using the **reacts-to** operation, we guarantee that all tuples that need to move will move atomically in both of these cases.

Engaging and Disengaging

Mobile components become connected and disconnected dynamically. Thus, changes in the Γ relation trigger engagement or disengagement of the ITSs of connected mobile components, and consequent reorganization of the shared tuple space.

Upon engagement, the ITSs of all the mobile components that are currently connected are merged together. Because of the unique tuple identifier imposed by the model, when the tuple spaces are merged there is no danger of creating erroneously multiple copies of a single tuple, even if the two mobile components participating in the sharing were already transitively connected prior to the establishment of this connection.

Upon disengagement, conceptually the system partitions the shared tuple space in all the ITS of the mobile agents currently connected and then merges them to form the new shared tuple space based on the new connectivity of the graph. Thus, a useful concept to define is the set of tuples belonging to a particular mobile agent. For this definition we use the $\downarrow$ projection operation and specify the mobile agent ω we are interested in. The tuples in a tuple space T that belong $\mu(\omega)$ are those for which the current owner is equal to ω, the identifier of the agent:

$$T \downarrow \omega \triangleq \langle \mathrm{set}\, t : t \in T :: t \uparrow 1 = \omega \rangle$$

Given these definitions, we are ready to define the engagement and disengagement of tuple spaces of mobile components:

$$\begin{array}{ll} \mu(i).T \approx \mu(j).T & \textbf{when } \mu(i)\ \gamma\ \mu(j) \\ & \textbf{engage } \mu(i).T \cup \mu(j).T \\ & \textbf{disengage } \langle \cup\, \omega : \mu(\omega)\ \Gamma\ \mu(i) :: \mu(\omega).T \downarrow \omega \rangle, \\ & \qquad\qquad\quad \langle \cup\, \omega : \mu(\omega)\ \Gamma\ \mu(j) :: \mu(\omega).T \downarrow \omega \rangle \end{array}$$

This expression describes the fact that the two variables corresponding to the ITS of the two agents are shared, i.e. any changes in one of them are immediately visible to the other. Sharing is in place when the predicate on the right hand side holds, i.e. when the two agents $\mu(i)$ and $\mu(j)$ are connected. The **engage** clause determines the initial value of the shared variable, thus effectively reconciling the contents of the two ITS of the agents to provide access to the union of the tuples, and is executed as soon as the predicate becomes true. The **disengage** clause, in turn, is executed when the predicate is falsified and thus the two agents become disconnected; it represents the atomic splitting and re-merging of the ITSs in order to regenerate the shared tuple space.

Migrating Mobile Components

For a mobile agent, location is effectively modeled as part of its state. Thus, migration of a mobile agent $\mu = \langle M, j\rangle$ is expressed by an assignment to its location, that is constituted by a mobile host M. On the other hand, migration

```
System MobileProducerConsumer
    Program P at λ
        declare
            T : set of Tuple ▯ p : Tuple ∪ {⊥} ▯ i : integer
        initially
            T = {} ▯ p = ⊥
        assign
            p := ⟨i,Produce()⟩              if p = ⊥
          ▯ out(p) ∥ p,i := ⊥,i + 1         if p ≠ ⊥
    end
    Program C at λ
        declare
            T : set of Tuple ▯ p : Tuple ∪ {⊥} ▯ x,i : integer
        initially
            T = {} ▯ p = ⊥
        assign
            p,i := in(i,?x),i + 1        if p = ⊥
          ▯ Consume(p) ∥ p := ⊥          if p ≠ ⊥
    end
    Components
            P ▯ C
    Interactions
            P.T ≈ C.T           when P γ C
                                engage P.T ∪ C.T
                                disengage ⟨∪ π : π ∈ {P,C} ∧ π Γ P :: π.T ↓ π⟩,
                                          ⟨∪ π : π ∈ {P,C} ∧ π Γ C :: π.T ↓ π⟩
          ▯ ⟨▯ π : π ∈ {P,C} :: π.T := ⟨set t : t ∈ π.T :: change(⟨π,π′,t⟩,⟨π′,π′,t⟩ if π Γ π′)⟩⟩
                                reacts-to true
end
```

Figure 16.5: Mobile producers and consumers with Linda and Mobile UNITY.

of a mobile host remains outside the scope of this description. As a mobile host moves, the value of the relation κ changes, which in turn affects the connectivity of the software agents through γ.

The Producer-Consumer Example Revisited

Figure 16.5 describes the producer-consumer example in a mobile setting. The Mobile UNITY statements that define the semantics of the LIME system are presented together with the application dependent statements describing our example, thus effectively forming a full specification of our application example while running on top of LIME.

Thanks to the modularization constructs of Mobile UNITY, the producer and the consumer are now represented directly as separate components in the system. The behavior of LIME is represented in the **Interactions** section, that provides for coordination of the components and can be regarded as the specification of the LIME run-time support.

16.4 Architecture

Having formally defined a model of mobile coordination, we now turn our attention to reality and describe our current implementation progress and plans. Overall, our intent is to provide the mobile application programmer with a library of Java tools to enable the rapid development of reliable mobile applications incorporating both physical and logical mobility.

We describe the architecture of our system in three stages. First, we illustrate the architecture of a single mobile host where multiple agents are run concurrently and coordinated using the shared tuple space. Then, we present a distributed scenario where the network topology is fixed and engagement and disengagement may take place only due to the migration of mobile agents. Finally, we describe the most complex scenario where engagement and disengagement, as well as the availability of a communication link, are determined by the movement of mobile hosts in the physical space.

16.4.1 Coordinating Stationary Agents on a Single Host

The architecture of a single, stationary host is depicted in the left hand side of Figure 16.6. The run-time support for our system creates a *tuple space (TS) manager* object on the host. The TS manager is responsible for providing the agents running on the host with the abstraction of a shared tuple space. Essentially the TS manager builds this virtual tuple space by maintaining an array of pointers to the individual agents, and transitively to their tuple spaces.

Each operation on the tuple space by every agent is sent through the TS manager, which in turn accesses the individual tuple spaces to service the query. In this manner, we achieve serialization of operations. When a new agent is created, it must register its ITS with the TS manager and begin to send all of its operations through the manager. Similarly, when an agent terminates, it announces this to the TS manager, which removes the pointer to the terminating agent. Agent creation and deletion are primitive forms of engagement and disengagement; however we leave the discussion about the movement of tuples based on current and final location until later in this section.

The functionality of the **out** operation is fairly straightforward because the operation is non-blocking, while implementations of the **in** and **rd** operations must take into account their blocking nature, and be able to hold the request until it can be satisfied. To this end, the TS manager maintains a registry for the templates of such pending requests. Because the operations are blocking, the agents are suspended until a matching tuple is found. Because all **out** operations go through the TS manager, before a tuple is sent to the shared tuple space it is checked against all registered patterns. If an **in** matches, then the agent is awakened and the tuple is taken immediately to satisfy this request. In the case of a **rd**, the tuple is put into the tuple space, and the agent is awakened. If no match is found, the tuple simply goes into the tuple space. It should be noted that it is possible to service multiple **rd** requests with one tuple, however only one **in** request can receive a tuple.

16.4.2 Coordinating Mobile Agents on Fixed Hosts

Next we move to a setting with multiple agents running on different hosts, but fix the network topology, therefore focusing our attention on the distributed nature of the system rather than on the connectivity changes. Figure 16.6 shows this view, however the connection depicted by κ is fixed. Our first extension in this design is to add a *connection manager*. This new manager has two primary responsibilities: maintain information concerning other agents at remote hosts,

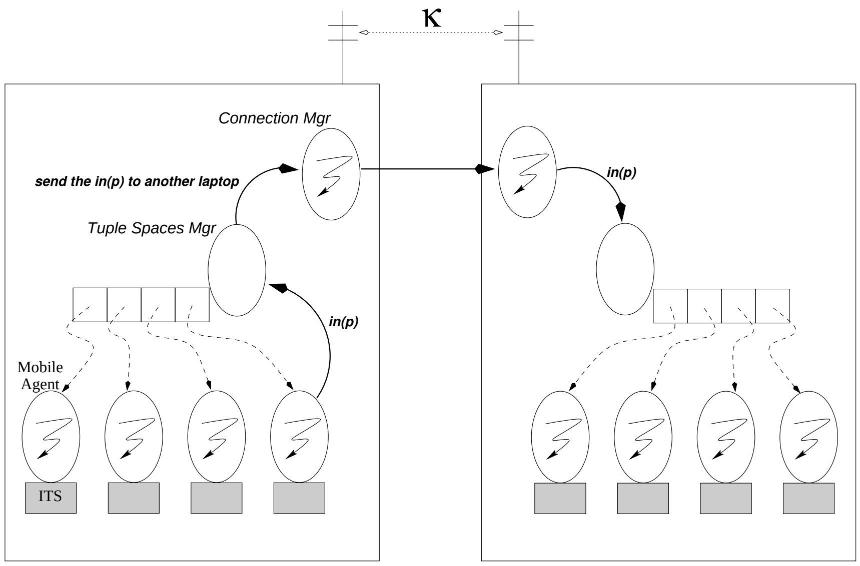


Figure 16.6: A high-level view of the architecture.

and serve as the message passing mechanism to send queries and tuples to other hosts and agents.

Even though the host network is fixed, we still allow agents to be created and destroyed, and to move within this network. Therefore, the connection manager must maintain an up-to-date list of which agents are at which host. To accomplish this, we require that when an agent registers with the TS manager, the connection manager is also informed. The connection manager then communicates this new information with each of the hosts it knows about. Because we assume a fixed set of hosts, this communication is trivial.

In addition to maintaining the agent list, the connection manager interacts with the tuple space manager to service queries to the remote hosts and agents. An **out**[λ] request for a remote tuple space should directly access the connection manager for agent λ and complete the tuple transfer. As before, the non-blocking nature of **out** makes it more straightforward than **in** and **rd**. For these blocking operations, we must have some mechanism to request a matching tuple from the remote tuple spaces, and block if none is found. To accomplish this, the connection manager contacts each host simultaneously with the query. If the remote host cannot satisfy the query, it puts the query into its registry of unfulfilled queries and continues to check all incoming tuples for a match. If a matching tuple is found, the remote tuple space sends an affirmative response to the requesting connection manager announcing that it has a tuple. The requesting connection manager collects these responses, selects one at random and sends it an accept. When the accept is received remotely, if the tuple is still present at the agent, a reliable connection is established between the originating host and the host with the tuple, and the tuple is transferred. If the tuple has been consumed before the accept arrives, the host either looks for another matching tuple, or replies to the originator with a negative response. Because the originator has a list of other possible tuples, it runs the same protocol for each of these until a tuple is retrieved. It is important at this point to send a reject message to all hosts so that the pattern can be removed from the query registry. A similar mechanism can be developed for the **rd** operation. In systems with small tuples or which require fast access response times, it is possible to bypass part of the request protocol and instead of sending a positive response, send the actual tuple. Because the underlying tuple space is not affected by the **rd** this procdure will not create inconsistencies. It is still important to send the cancel messages so that the registry can be cleaned up. This optimization is not worthwhile, however, in a system with large tuple sizes or where there are expected to be a large number of matching tuples.

With the **in** example, we demonstrated how the underlying tuple space can change in the middle of an operation and how we can adjust for this at a higher level by using the connectivity manager. This is one example of how the atomicity assumptions present in the model for the operations are implemented. Another important atomic operation that must be considered is the reactive statement that moves tuples based on their current and final location fields. As mentioned previously, the **out** operation for a connected agent is a simple process because the connection manager maintains the list of available agents, and a reliable connection can be established to send the tuple. However, when a new agent arrives and registers, it is possible that tuples will have to be transferred both to and from it, and to respect the atomicity of the model, all tuples must move before the engagement is complete. However, we have a slight problem because so far we have no restrictions on when an agent can unregister with the tuple space manager. Therefore, an agent could leave the system after its tuples had been transferred to the new agent, but before the new agent's tuples were transferred to the leaving agent. Even though we use reliable data transmission for the tuples, this partial tuple transfer is undesirable. Therefore, we introduce the notion of announced disconnection, requiring all agents to announce their intention to disconnect before disconnecting, and before actually leaving the system, to complete any transactions they are part of. Therefore, the case described above is not possible.

Although the network is fixed, we still need a mechanism to contact each of the remote connection manager for the various operations. For this, we employ a combination of reliable TCP connections over IP, and unreliable UDP connections using multicast. Basically, we use TCP whenever we need guaranteed data transfer to keep the tuple space consistent, and multicast when we need to get information to all hosts. For instance, when informing the other hosts of the arrival of a new agent, the connection manager simply multicasts the announcement to its list of hosts. Because we want the connection managers to know about all agents, we add a reliability mechanism and require that the hosts acknowledge the announcement. If an acknowledgment is not received from one of the hosts, the connection manager can contact it directly with the announcement. Multicast can also be employed when sending the **in** and **rd** requests. If no responses are received within a reasonable timeout, the connection manager rebroadcasts the request. Hopefully this will take care of the cases where the multicast message was dropped on the first transmission, but will arrive on one of the later transmissions. All messages must be tagged with an identifier so that the remote hosts can ignore duplicates. When a tuple is being transferred, we require the connection manager to use TCP to directly connect with the remote host. Therefore, both **out**[λ] and the final tuple transfer for the **in** and **rd** will use TCP. Although it is not strictly necessary to use reliable transfer for the **rd** operation, we do so for simplicity.

16.4.3 Coordinating Mobile Agents on Mobile Hosts

Next we take the small step to move from the fixed network to the mobile network. One of the most important changes that we must make is the updating of the connection manager. For this, we access the low level network protocols of wireless systems which effectively broadcast a periodic 'hello' message. These messages can be used to determine which hosts are available. When a new host is available, it sends a multicast message similar to that described above for agent registration. The multicast message contains the agents on the host and the host identifier. Because multicast is unreliable, we can extend it with additional mechanisms to ensure all hosts are reached. Presumably all previously connected hosts knew of the existence of the other connected hosts. Therefore, their response to the multicast message can contain not only their list of agents, but also the list or simply the number of other hosts the new host can expect to hear from. If any are missed, a unicast message can be sent to the missed agent and the engagement can continue. All tuple transfers are required to use a reliable transport protocol such as TCP.

Despite the reliable protocol, as before, we require all agents to announce their disconnection; effectively restricting the movement of the mobile host during times of tuple transfers. This restriction is reasonable because not only will most transfers be small, but we make one more simplifying assumption about the graph to guarantee all tuple transfers will take place only between directly connected hosts.

An as yet unsolved problem is routing in an ad hoc network. In order to run protocols such as TCP, we need to have a route established in order to get from one node, through another, to the final destination. Because these protocols do not yet exist, our implementation starts with a simple setting in which we require a direct connection among all machines. In other words, we do not require routing, only a single hop, point-to-point connection. This model accommodates a situation where business people meet in an airport, or salespeople meet in a conference room. Our hope is that eventually ad hoc routing will mature to the point where protocols are released and can be easily integrated into our framework.

16.5 Discussion and Future Work

In this chapter we presented a two tier model that encompasses physical and logical mobility and uses a shared tuple space to coordinate mobile hosts and mobile agents. Thus, it investigates with a pragmatic approach the relationships existing between these two forms of mobility.

The notion of tuple space comes directly from the Linda model, for which we defined the formal semantics using UNITY. The Linda model seems particularly suited to mobility in that it fosters a model of computation that is decoupled in time and space. In Lime, Linda is extended with a minimal set of abstractions and mechanisms to cope with mobility. The capability to change atomically the contents of a tuple, the capability to react to events, and a predefined structure for tuples that takes into account their current and final location is all we needed to devise a formal semantics for our system using Mobile UNITY.

The final goal of Lime is to provide developers with means for designing and implementing applications encompassing mobility more quickly and more reliably, by exploiting a rather abstract view of the underlying system and leveraging off of its precise definition, which leaves room also for formal verification.

In this work, we also described how the real system can be realized, taking into account to some extent the constraints imposed by implementation. A prototype of the system built in Java is presently under development, whose effectiveness will be verified in real applications.

Final Remarks on Exploiting Code Mobility

In this part of the thesis we applied what we developed in the previous part of the thesis to different subjects, namely, the development of a new mobile code toolkit, the design of distributed applications exploiting mobile code, and formal models and programmer support to cope with both physical and logical mobility.

Further work is needed on all of these issues, as they are all still on-going and some of them are in their early stage of development.

Conclusions

Code mobility is a new and promising research field that, by and large, is still looking for a commonly accepted background of concepts, methodologies, models, and techniques. The development of MCSs, and in particular of the Java language, has raised a lot of interest in this field. Nevertheless, it has also generated confusion, as the interest of market and media fostered a technology oriented approach to the research area. As a consequence, few efforts have been directed towards uncovering and formalizing the fundamental concepts in this field, as well as providing evidence for the soundness of the approach.

This thesis has sought out a research path through the area that addresses the aforementioned problems. The first step of our research has been concerned with developing an understanding of the domain. The many MCSs proposed in literature use different terms and concepts, they are conceived with different applications in mind, and provide several different mechanisms and abstractions. On the research side, this hampers comparison and communication of the results. On the development side, it makes it difficult to understand whether code mobility is useful in a given application domain and in this case to select the technology that is more appropriate for the given task. To deal with the complexity of this situation, we proposed a conceptual framework that is structured along the dimensions of technologies, architectures, and applications. In our experience, this framework has proven itself useful. It has guided further developments described in this thesis, providing a reference in classifying new approaches and methodologies, as well as in devising experimental architectures and applications involving mobile code.

The second step in our research has been concerned with the evaluation of code mobility. Mobile code can be accepted widely only if convincing evidence for the benefits it can provide is demonstrated. We attacked the problem by focusing on an applicative domain that is often referred to as ideal for code mobility: network management. First, we analyzed qualitatively which are the benefits provided by code mobility, using our taxonomy of concepts. Then, we built a model of network management applications and evaluated quantitatively different alternatives, providing also some experimental evidence verifying the theoretical results. This enabled us to determine precisely under which conditions mobile code is useful to reducing the amount of management traffic generated and the load on the management station, one of the main problems of current centralized protocols.

Building on the knowledge acquired in developing our conceptual framework, and more confident with the usefulness of a mobile code approach, we moved then into formalizing the concepts of code mobility. The immaturity of the research area is revealed in the fact that the first formal models of concurrency encompassing mobility have been proposed only very recently. We approached the problem by employing a notation and logic that was originally conceived for describing a mobile computing scenario encompassing physical mobility of hosts. This notation has been used to express the key abstractions and mechanisms defined in our taxonomy in the context of a sample problem, and the associated logic has been exploited to illustrate how to verify formally the correctness of a mobile code design.

As our fourth step, we seeked for application of the core of concepts, techniques, and insights developed so far. This encompasses the most recent activities developed in the thesis. On one hand, we concentrated on the technological issues. The aforementioned evaluation in network management and in other mobile code applications, together with the taxonomy of concepts developed in our framework convinced us that current mobile code technology does not provide the necessary flexibility. Then, we developed the first version of a mobile code toolkit that supports all the relocation strategies we identified in the first step of our research and that tries to minimize the tradeoffs for network and computational overhead. This toolkit has been used successfully in an experience where we designed and built a prototype of a videconferencing system for active networks. Finally, we are in the process of developing a system that is supposed to provide a set of minimal abstractions that can be used to build applications that encompass indifferently physical and logical mobility. This activity has leveraged off of the aforementioned formalization activity in that the formal semantics of the system is specified with the same notation employed there.

Clearly, although this thesis spans several layers in the research area, further work is needed in order to refine what we proposed and investigate new developments. For instance, MCSs found in literature are constantly improved and enhanced, and new abstractions are added to them. Moreover, our evaluation of code mobility showed how the characteristics of the technology is fundamental in determining the threshold after which mobile code becomes useful. Thus, a quantitative characterization of mobile code technology seems a natural evolution of our research. Also, applications are very important in developing a more solid understanding of the issues in this research field, as they allow to highlight the technical problems as well as to seek for unforeseen opportunities for mobile code. In this respect, the uncovering of relationships between physical and logical mobility seems particularly intriguing. On-going and future work will concentrate on all these aspects.

Nevertheless, we hope and believe that the work described in this thesis could contribute to create a mature and comprehensive background for the evolution of the research area concerned with code mobility.

Appendixes

Appendix A

Proof Outlines

We provide here the proof outlines for the obligations used but not proven in the previous sections.

A.1 Client-Server

Invariant ($\textsc{cs}_2$) can be proven directly from the program text.

PROOF OF ($\mathcal{I}_2$). The invariant holds initially, because the value of each timer and GVT estimate is initially zero. To prove the invariant we need to prove that

$$\begin{array}{l}\textbf{inv.}\ P(k).t = P(k).RQ \uparrow 4 = \beta \wedge P(k).RQ \uparrow 1 = k \Rightarrow \\ \qquad \langle \min l :: P(l).t \rangle = \beta \end{array} \qquad (\textsc{cs}_{17})$$

In order to prove this invariant, we observe that $P(k).t$ is not allowed to change because of invariant ($\textsc{cs}_3$), which can be rewritten as

$$\textbf{inv.}\ \mathsf{def}(P(k).RQ) \Rightarrow \neg\mathsf{def}(P(k).T),$$

and thus proves that when the LHS of ($\textsc{cs}_{17}$) holds the first statement is always disabled. Furthermore, ($\textsc{cs}_{17}$) is proven by the following invariants

$$\begin{array}{l}\textbf{inv.}\ P(k).t = \mathit{Server}.q(k) \uparrow 4 = \beta \wedge \mathit{Server}.q(k) \uparrow 1 = k \Rightarrow \langle \min l :: P(l).t \rangle = \beta \\ \textbf{inv.}\ P(k).t = \mathit{Server}.T = \beta \wedge \mathit{Server}.q(k) \uparrow 2 = \textsc{wait} \Rightarrow \langle \min l :: P(l).t \rangle = \beta \\ \textbf{inv.}\ P(k).t = \langle \min l :: \mathit{Server}.\tau(l) \rangle = \beta \wedge \mathit{Server}.q(k) \uparrow 2 = \textsc{wait} \Rightarrow \\ \qquad \langle \min l :: P(l).t \rangle = \beta \end{array}$$

which can be proven directly from the program text, and observing that

$$\textbf{inv.}\ P(k).t = \beta \wedge \langle \min l :: \mathit{Server}.\tau(l) \rangle = \beta \Rightarrow \langle \min l :: P(l).t \rangle = \beta$$

holds because of ($\mathcal{I}_3$).

PROOF OF ($\mathcal{I}_3$). The invariant holds initially, because the value of each local timer and each element of $\mathit{Server}.\tau$ is initially zero. To prove that ($\mathcal{I}_3$) holds, we observe that the first statement in *Server* is the only one that modifies $\mathit{Server}.\tau$, and that this is done according to invariant ($\textsc{cs}_4$).

PROOF OF ($\mathcal{I}_4$). The invariant holds initially, because each element of $Server.\tau$ and each GVT estimate is initially zero. The first statement in *P(k)* does not affect the invariant, as if the LHS is false before execution of the statement this is disabled, otherwise the LHS becomes false after execution of the statement. To prove that the invariant holds before and after the execution of the other statements which might affect the invariant we can use the invariants

$$\begin{aligned}&\textbf{inv.}\ P(k).RQ \uparrow 1 = k \Rightarrow P(k).RQ \uparrow 4 \leq \langle \min l :: Server.\tau(l)\rangle\\ &\textbf{inv.}\ Server.q(k) \uparrow 1 = k \Rightarrow Server.q(k) \uparrow 4 \leq \langle \min l :: Server.\tau(l)\rangle\\ &\textbf{inv.}\ \mathsf{def}(Server.T) \Rightarrow Server.T \leq \langle \min l :: Server.\tau(l)\rangle,\end{aligned}$$

together with the lemma (CS_5), which states that the minimum value in $Server.\tau$, if changed, can only increase monotonically.

PROOF OF (CS_1). The invariant holds initially, and is proven from the program text, taking advantage of lemma (CS_5).

PROOF OF (CS_3). The invariant can be rewritten as

$$\textbf{inv.}\ \mathsf{def}(P(k).RQ) \Rightarrow \neg\mathsf{def}(P(k).T)$$

In order to prove the invariant above, the last interaction requires us to prove that

$$\textbf{inv.}\ Server.q(k) \uparrow 1 = k \Rightarrow \neg\mathsf{def}(P(k).T)$$

which can be proven directly from the program text by observing that

$$\begin{aligned}&\textbf{inv.}\ Server.q(k) \uparrow 1 = k \Rightarrow \neg\mathsf{def}(Server.q(k) \uparrow 2), \text{ and}\\ &\textbf{inv.}\ P(k).RQ \uparrow 1 = k \Rightarrow \neg\mathsf{def}(P(k).T)\end{aligned}$$

hold.

PROOF OF (CS_4). The invariant holds initially. Then, we observe that the only statement that could possibly invalidate the invariant is the first interaction, which copies the value of $P(i).RQ$ into $Server.q(i)$. However, we demonstrate that

$$\textbf{inv.}\ P(k).RQ \uparrow 1 = \text{SERVER} \Rightarrow P(k).RQ \uparrow 4 = P(k).t$$

holds, so that the value communicated to *Server* is still equal to $P(i).t$. This is proven observing the statements modifying $P(i).RQ$, namely, the first statement in *P(i)* and the second interaction, preserve the invariant, and that no statement can modify the value of the local timer except for the first one in *P(i)*. Moreover, it can be demonstrated from the program text that

$$\textbf{inv.}\ P(k).RQ \uparrow 1 = \text{SERVER} \Rightarrow \neg\mathsf{def}(P(k).T),$$

holds, which guarantees that such a statement is disabled when the LHS holds.

PROOF OF (CS_5). We observe that no statement modifies the LHS of the **unless**, except for the first one in *Server*. Under the guard of this statement, invariant (CS_4) guarantees that $Server.\tau(l)$ is updated to the current value of $P(l).t$. Then, the new value for the minimum is

$$\min(P(l).t, \langle \min m : m \neq l :: Server.\tau(m)\rangle) \geq \beta$$

because of invariant ($\mathcal{I}_3$).

A.2 Mobile Agent

Invariants ($\mathcal{I}_2$), (MA$_1$), (MA$_3$), and (MA$_7$) can be proven directly from the program text.

PROOF OF ($\mathcal{I}_2$). The invariant holds initially. Then, the statements that may affect the invariant, namely, the one in *P(k)* and the second and third in *Server*, either leave the variables involved unaffected or falsify the LHS, thus preserving the invariant.

PROOF OF ($\mathcal{I}_3$). The invariant holds initially. Then, the first statement of *P(i)* satisfies the invariant, since $P(i).t$ can only increase, due to the definition of f and g. Furthermore, execution of the first statement in *Server* actually establishes the invariant. No other statement can affect the invariant.

PROOF OF ($\mathcal{I}_4$). The invariant holds initially. The first statement in *P(k)* falsifies the LHS of the invariant, thus preserving it. Then, we observe that the behavior of *Server* may affect the invariant only when the *Server* and *P(k)* are co-located. Then, invariants (MA$_3$) and (MA$_6$) guarantee that the invariant is preserved. The migration statement in *Server* does not affect the invariant neither on engagement, for which the aforementioned argument holds, or disengagement, where values are retained thus preserving the invariant.

PROOF OF (MA$_4$). We observe that the only statement that could affect the lemma is the first one in *Server*. However, invariant ($\mathcal{I}_3$) guarantees that either the LHS of the **unless** is unaffected or its RHS is established.

PROOF OF (MA$_5$). This invariant is proven with the same argument used for the verification of (MA$_6$) in Section 12.1.

A.3 Remote Evaluation

Invariants (REV$_1$), (REV$_6$), (REV$_{10}$) and (REV$_{12}$) can be proven directly from the program text.

PROOF OF ($\mathcal{I}_2$). The invariant holds initially, because the value of each timer and GVT estimate is initially zero. As in the corresponding proof in Appendix A.1, proving ($\mathcal{I}_2$) involves proving

$$\textbf{inv.}\ P(k).t = P(k).RQ \uparrow 4 = \beta \wedge P(k).RQ \uparrow 1 = k \Rightarrow \langle \min l :: P(l).t \rangle = \beta.$$

and observing that the first statement in *P(k)* is disabled when the LHS of the above holds, as a consequence of invariant (REV$_5$). Then, the above is proven by observing that

$$\textbf{inv.}\ P(k).t = Min(k).q \uparrow 4 = \beta \wedge Min(k).q \uparrow 1 = k \Rightarrow \langle \min l :: P(l).t \rangle = \beta$$

can be proven directly from the program text together with invariant (REV$_2$) and (REV$_{11}$), and also

$$\textbf{inv.}\ Min(k).t = Min(k).T = \beta \Rightarrow \langle \min l :: P(l).t \rangle = \beta$$

can be proven from the program text. The above, together with (REV$_4$), proves ($\mathcal{I}_2$).

PROOF OF ($\mathcal{I}_3$). The invariant holds initially, from program text. The are only two statement that may affect the invariant. If *Min(k)* and *P(k)* are co-located, the first statement in *P(k)* may execute. However, the invariant is preserved according to the definition of f and g. If the two components are not co-located, invariant (REV_8) guarantees that such a statement is disabled. In this case, the first statement in *Min(k)* is enabled, and it actually establishes the invariant because of (REV_3) and (REV_4).

PROOF OF ($\mathcal{I}_4$). The invariant is proven with the same arguments used in Appendix A.1, although the lemma

$$\langle \min l :: \mathit{Server}.\tau(l)\rangle = \beta \ \textbf{unless} \ \langle \min l :: \mathit{Server}.\tau(l)\rangle > \beta$$

is proven differently. The only statement that may affect the lemma is the first in *Min(k)*, which is enabled only when *Min(k)* and *Server* are co-located, as stated by (REV_9). Execution of this statement, together with invariant ($\mathcal{I}_3$) and (REV_5), establishes

$$\min(P(l).t, \langle \min m : m \neq l :: \mathit{Server}.\tau(m)\rangle) \geq \beta$$

as in the CS solution, which satisfies the lemma.

PROOF OF (REV_2). The invariant holds as the conjunction of (REV_1), (REV_7), and (REV_9).

PROOF OF (REV_7). The invariant holds initially, from the program text. The first statement and the third statement in *Min(k)* are enabled only when the LHS of the invariant does not hold. Execution of the first statement, due to (REV_{10}), keeps the LHS false, and thus preserves the invariant. Execution of the third statement does not affect the LHS at all, and thus the invariant is preserved straightforwardly. Migration statements may affect the invariant. Execution of the second statement in the **Interactions** section establishes $\mathsf{def}(\mathit{Min}(k).\tau)$ due to the **engage** clause in the third interaction and to (REV_6). Finally, execution of the last interaction establishes the invariant. The migration statement is executed synchronously with the statement voiding $\mathit{Server}.q(i)$: disengagement of this variable and $\mathit{Min}(k).q$ causes the latter to remain undefined after migration. At the same time, the **disengage** clause in the third interaction voids the value of $\mathit{Min}(k).\tau$.

PROOF OF (REV_8). The invariant holds initially, since *P(k)* and *Min(k)* are initially co-located and hence the LHS of the implication is false. Then, there are only two statements which may affect the invariant. If the LHS holds before execution of the first statement in *P(k)*, then the statement is disabled and the invariant preserved. If the LHS holds before execution of the third statement in *P(k)*, invariant (REV_{12}) guarantees that the statement is disabled, preserving the invariant.

PROOF OF (REV_{11}). The invariant holds initially, from the program text. The first statement in *Min(k)* may affect the invariant. However, if *Min(k)* is not co-located with *Server*, invariant (REV_9) guarantees that this statements is disabled. On the other hand, if the two components are co-located, execution of the statement assigns a defined value to $\mathit{Min}(i).q$

and then to *Server*.$q(i)$, being sharing in place through the fourth interaction. Migration statements may also affect the invariant. The second statement in the **Interactions** section actually establishes the invariant through the **engage** clause of the fourth interaction. The last statement in the system establishes the invariant as well, since it assigns an undefined value to *Server*.$q(i)$, which is still shared with *Min*$(i).q$, and causes migration of *Min(k)*; since no disengagement clause is specified for the fourth interaction, the two variables retain an undefined value.

Appendix B

Additional References on the World-Wide Web

Agent Tcl	`http://www.cs.dartmouth.edu/~agent/`
Ara	`http://www.uni-kl.de/AG-Nehmer/Ara/`
Facile	`http://www.ecrc.de/research/projects/facile/`
Java	`http://java.sun.com/`
Java Aglets	`http://www.trl.ibm.co.jp/aglets`
M0	`http://cuiwww.unige.ch/Telecom-group/msgr/home.html`
Mole	`http://www.informatik.uni-stuttgart.de/ipvr/vs/projekte/mole.html`
Obliq	`http://www.luca.demon.co.uk/Obliq/Obliq.html`
Safe-Tcl	`http://sunscript.sun.com`
Sumatra	`http://www.cs.umd.edu/users/saltz/`
TACOMA	`http://www.cs.uit.no/DOS/Tacoma/`
Telescript	`http://www.genmagic.com/Telescript`
Odyssey	`http://www.genmagic.com/agents`
Voyager	`http://www.objectspace.com/Voyager`
Concordia	`http://www.meitca.com/HSL/Projects/Concordia`
Messengers	`http://www.ics.uci.edu/~bic/messengers`

Bibliography

[1] Abowd, G., Allen, R., and Garlan, D. Using Style to Understand Descriptions of Software Architecture. In *Proc. of SIGSOFT'93: Foundations of Software Engineering* (Dec. 1993).

[2] Acharya, A., Ranganathan, M., and Saltz, J. Sumatra: A Language for Resource-aware Mobile Programs. In Vitek and Tschudin [116], pp. 111–130.

[3] Adobe Systems Incorporated. *PostScript Language Reference Manual.* Addison-Wesley, 1985.

[4] Amadio, R. An Asynchronous Model of Locality, Failure, and Process Mobility. In Garlan and Métayer [39], pp. 374–391.

[5] Back, R., and Sere, K. Stepwise Refinement of Parallel Algorithms. *Science of Computer Programming 13*, 2-3 (May 1990), 133–180.

[6] Bal, H. *Programming Distributed Systems.* Prentice Hall, 1991.

[7] Baldi, M., Gai, S., and Picco, G.P. Exploiting Code Mobility in Decentralized and Flexible Network Management. In Rothermel and Popescu-Zeletin [94], pp. 13–26.

[8] Baldi, M., Malenchino, V., Picco, G.P., and Risso, F. Servizi e Flessibilità nello scenario ICT. Tech. Rep. DTR97.1115, Centro Studi e Laboratori Telecomunicazioni S.p.A. (CSELT), Dec. 1997. In Italian.

[9] Baldi, M., and Picco, G.P. Evaluating the Tradeoffs of Mobile Code Design Paradigms in Network Management Applications. In *Proc. of the 20^{th} Int. Conf. on Software Engineering* (1998), R. Kemmerer, Ed. To appear.

[10] Baldi, M., Picco, G.P., and Risso, F. Designing a Videoconference System for Active Networks. Tech. rep., Politecnico di Torino, Feb. 1998. Submitted for publication.

[11] Baumann, J., Hohl, F., Radouniklis, N., Rothermel, K., and Strasser, M. Communication Concepts for Mobile Agent Systems. In Rothermel and Popescu-Zeletin [94], pp. 123–135.

[12] Bharat, K., and Cardelli, L. Migratory Applications. Tech. Rep. 138, Digital Equipment Corporation, Systems Research Center, Feb. 1996.

[13] Bhattacharjee, S., Calvert, K., and Zegura, E. On Active Networking and Congestion. Tech. Rep. GIT-CC-96/02, Georgia Institute of Technology, 1996.

[14] Bic, L., Fukuda, M., and Dillencourt, M. Distributed Computing Using Autonomous Objects. *IEEE Computer* (Aug. 1996).

[15] Birrell, A., and Nelson, B. Implementing Remote Procedure Calls. *ACM Trans. on Computer Systems 2*, 1 (Feb. 1984), 29–59.

[16] Boggs, J. IBM Remote Job Entry Facility: Generalize Subsystem Remote Job Entry Facility. IBM Technical Disclosure Bulletin 752, IBM, Aug. 1973.

[17] Borenstein, N. EMail With A Mind of Its Own: The Safe-Tcl Language for Enabled Mail. Tech. rep., First Virtual Holdings, Inc, 1994.

[18] Cai, T., Gloor, P., and Nog, S. DataFlow: A Workflow Management System on the Web using transportable Agents. Tech. Rep. TR96-283, Dept. of Computer Science, Dartmouth College, Hanover, NH, 1996.

[19] Cardelli, L. A language with distributed scope. *Computing Systems 8*, 1 (1995), 27–59.

[20] Cardelli, L., and Gordon, A. Mobile Ambients. Available at `http://www.luca.demon.co.uk/Papers.html`, July 1997.

[21] Carzaniga, A., Picco, G.P., and Vigna, G. Designing Distributed Applications with Mobile Code Paradigms. In *Proc. of the 19th Int. Conf. on Software Engineering (ICSE'97)* (1997), R. Taylor, Ed., ACM Press, pp. 22–32.

[22] Case, J., Fedor, M., Schoffstall, M. L., and Davin, C. Simple Network Management Protocol. RFC 1157, May 1990.

[23] Case, J., McCloghrie, K., Rose, M., and Waldbusser, S. Structure of Management Information for version 2 of the Simple Network Management Protocol. RFC 1902, Jan. 1996.

[24] Chandy, K., and Misra, J. Distributed Simulation: A Case Study in Design and Verification of Distributed Programs. *IEEE Trans. on Software Engineering 5*, 5 (Sept. 1979), 440–452.

[25] Chandy, K., and Misra, J. *Parallel Program Design: A Foundation.* Addison-Wesley, 1988.

[26] Chang, E. Echo Algorithms: Depth Parallel Operations on General Graphs. *IEEE Trans. on Software Engineering* (July 1982).

[27] Chess, D., Grosof, B., Harrison, C., Levine, D., Paris, C., and Tsudik, G. Itinerant Agents for Mobile Computing. *IEEE Personal Communications* (Oct. 1995). Also available as IBM Technical Report.

[28] Cornell University. CU-SeeMe. Web page `http://cu-seeme.cornell.edu`.

[29] Cugola, G., Ghezzi, C., Picco, G.P., and Vigna, G. A Characterization of Mobility and State Distribution in Mobile Code Languages. In Mühlaüser [75], pp. 309–318.

[30] Cugola, G., Ghezzi, C., Picco, G.P., and Vigna, G. Analyzing Mobile Code Languages. In Vitek and Tschudin [116], pp. 93–111.

[31] Cunningham, H., and Roman, G.-C. A UNITY-style Programming Logic for a Shared Dataspace Language. *IEEE Trans. on Parallel and Distributed Systems 1*, 3 (July 1990), 365–376.

[32] De Nicola, R., Ferrari, G., and Pugliese, R. Locality Based Linda: Programming with Explicit Localities. In *Proc. of the 7th Int. Joint Conf. on the Theory and Practice of Software Development (TAPSOFT'97)* (Lille, France, Apr. 1997), M. Bidoit and M. Dauchet, Eds., vol. 1214 of *LNCS*, Springer.

[33] Falcone, J. A Programmable Interface Language for Heterogeneous Distributed Systems. *ACM Trans. on Computer Systems 5*, 4 (July 1987), 330–351.

[34] Farmer, W., Guttman, J., and Swarup, V. Security for Mobile Agents: Authentication and State Appraisal. In *Proc. of the 4th European Symp. on Research in Computer Security* (Rome, Italy, Sept. 1996), Springer, Ed., vol. 1146 of *LNCS*, pp. 118–130.

[35] Forman, G., and Zahorjan, J. The Challenges of Mobile Computing. *IEEE Computer 27*, 4 (1994), 38–47.

[36] Fournet, C., et al. A Calculus of Mobile Agents. In *Proc. of the 7th Int. Conf. on Concurrency Theory (CONCUR'96)* (Pisa, Italy, Aug. 1996), vol. 1119 of *LNCS*, Springer, pp. 406–421.

[37] Fuggetta, A., Picco, G.P., and Vigna, G. Understanding Code Mobility. Tech. rep., Politecnico di Milano, July 1997. Accepted for publication in *IEEE Trans. on Software Engineering.*

[38] Fukuda, M., Bic, L., Dillencourt, M., and Merchant, F. Intra- Inter-Object Coordination with MESSENGERS. In *1st Int. Conf. on Coordination Models and Languages (COORDINATION'96)* (1996).

[39] Garlan, D., and Métayer, D. L., Eds. *Proc. of the 2nd Int. Conf. on Coordination Models and Languages (COORDINATION '97)* (Berlin, Germany, Sept. 1997), vol. 1282 of *LNCS*, Springer.

[40] GELERNTER, D. Generative Communication in Linda. *ACM Computing Surveys 7*, 1 (Jan. 1985), 80–112.

[41] GENESERETH, M., AND KETCHPEL, S. Software Agents. *Comm. of the ACM 37*, 7 (July 1994).

[42] GHEZZI, C., AND VIGNA, G. Mobile Code Paradigms and Technologies: A Case Study. In Rothermel and Popescu-Zeletin [94], pp. 39–49.

[43] GOLDSZMIDT, G., AND YEMINI, Y. Distributed Management by Delegation. In *Proc. of the 15th Int. Conf. on Distributed Computing* (June 1995).

[44] GRAY, R. Agent Tcl: A transportable agent system. In *Proc. of the CIKM Workshop on Intelligent Information Agents* (Baltimore, Md., Dec. 1995).

[45] GRAY, R., KOTZ, D., NOG, S., RUS, D., AND CYBENKO, G. Mobile agents for mobile computing. In *Proc. of the 2nd Aizu Int. Symp. on Parallel Algorithms/Architectures Synthesis* (Fukushima, Japan, Mar. 1997).

[46] HARRISON, C., CHESS, D., AND KERSHENBAUM, A. Mobile Agents: Are they a good idea? In Vitek and Tschudin [116], pp. 25–47. Also available as IBM Technical Report.

[47] HARTMAN, J., MANBER, U., PETERSON, L., AND PROEBSTING, T. Liquid Software: A New Paradigm for Networked Systems. Tech. Rep. 96-11, Dept. of Computer Science, Univ. of Arizona, Tucson, AZ, June 1996.

[48] HICKS, M., KAKKAR, P., MOORE, J., GUNTER, C., AND NETTLES, S. PLAN: A Programming Language for Active Networks. Tech. rep., Univ. of Pennsylvania, USA, Nov. 1997.

[49] HOGG, J. Island: Aliasing Protection in Object-Oriented Languages. In *Proc. of OOPSLA '91* (1991).

[50] HOHL, F. An Approach to Solve the Problem of Malicious Hosts in Mobile Agent Systems, 1997.

[51] IMIELINSKY, T., AND BADRINATH, B. Wireless Computing: Challenges in Data Management. *Comm. of the ACM 37*, 10 (1994), 18–28.

[52] ITU-T. Information Technology—Open Systems Interconnection—The Directory: Authentication Framework. ITU-T Recommendation X.509, Nov. 1993.

[53] ITU-T RECOMMENDATION H.323. *Visual telephone systems and equipment for local area networks which provide a non-guaranteed quality of service*, Nov. 1996.

[54] JOHANSEN, D., VAN RENESSE, R., AND SCHNEIDER, F. B. An Introduction to the TACOMA Distributed System—Version 1.0. Tech. Rep. 95-23, Dept. of Computer Science, Univ. of Tromsø and Cornell Univ., Tromsø, Norway, June 1995.

[55] JOHNSON, D. Routing in Ad Hoc Networks of Mobile Hosts. In *Proc. of the Workshop on Mobile Computing Systems and Applications* (Santa Cruz, CA, 1994), IEEE, pp. 158–163.

[56] JONES, K. Internet's SNMP and ISO's CMIP Protocols for Network Management. *Int. Journal of Network Management* (Sept. 1994), 130–137.

[57] JUL, E., LEVY, H., HUTCHINSON, N., AND BLACK, A. Fine-grained Mobility in the Emerald System. *ACM Trans. on Computer Systems 6*, 2 (Feb. 1988), 109–133.

[58] KINIRY, J., AND ZIMMERMAN, D. A Hands-On Look at Java Mobile Agents. *IEEE Internet Computing 1*, 4 (1997), 21–30.

[59] KNABE, F. *Language Support for Mobile Agents.* PhD thesis, Carnegie Mellon Univ., Pittsburgh, PA, USA, Dec. 1995. Also available as Carnegie Mellon School of Computer Science Technical Report CMU-CS-95-223 and European Computer Industry Centre Technical Report ECRC-95-36.

[60] KNUDSEN, P. Comparing Two Distributed Computing Paradigms—A Performance Case Study. Master's thesis, Univ. of Tromsø, 1995.

[61] LANGE, D. Java Aglets Application Programming Interface (J-AAPI). IBM Corp. White Paper, Feb. 1997.

[62] LANGE, D., AND CHANG, D. IBM Aglets Workbench—Programming Mobile Agents in Java. IBM Corp. White Paper, Sept. 1996.

[63] Lea, R., Jacquemont, C., and Pillevesse, E. COOL: System Support for Distributed Object-Oriented Programming. *Comm. of the ACM 36*, 9 (Nov. 1993), 37–46.

[64] Limongiello, A., Melen, R., Roccuzzo, M., Scalisi, A., Trecordi, V., and Wojtowicz, J. ORCHESTRA: An Experimental Agent-based Service Control Architecture For Broadband Multimedia Networks. GLOBAL Internet '96, Nov. 1996.

[65] Maes, P. Agents that Reduce Work and Information Overload. *Comm. of the ACM 37*, 7 (July 1994).

[66] Magedanz, T., Rothermel, K., and Krause, S. Intelligent Agents: An Emerging Technology for Next Generation Telecommunications? In *INFOCOM'96* (San Francisco, CA, USA, Mar. 1996).

[67] Malone, T., and Crowston, K. The interdisciplinary study of coordination. *ACM Computing Surveys 26*, 1 (Mar. 1994), 87–119.

[68] Martin-Flatin, J.-P., and Znaty, S. Annotated Typology of Distributed Network Management Paradigms. In *Proc. of the 8^{th} IFIP/IEEE Int. Workshop on Distributed Systems Operations and Management (DSOM'97)* (Oct. 1997). To appear.

[69] McCann, P., and Roman, G.-C. Mobile UNITY: A Language and Logic for Concurrent Mobile Systems. Tech. Rep. WUCS-97-01, Dept. of Computer Science, Washington Univ. in St.Louis, Dec. 1996. Submitted for publication. `http://www.cs.wustl.edu/~techrep/reports/1997/wucs-97-01.ps.Z`.

[70] McCann, P., and Roman, G.-C. Compositional Programming Abstractions for Mobile Computing. Tech. Rep. WUCS-97-45, Dept. of Computer Science, Washington Univ. in St.Louis, Oct. 1997. To appear in *IEEE Trans. on Software Engineering*.

[71] McCann, P., and Roman, G.-C. Mobile UNITY Coordination Constructs Applied to Packet Forwarding for Mobile Hosts. In Garlan and Métayer [39], pp. 338–354.

[72] Merz, M., and Lamersdorf, W. Agents, Services, and Electronic Markets: How do they Integrate? In *Proc. of the Int'l Conf. on Distributed Platforms* (1996), IFIP/IEEE.

[73] Millet Sorolla, M. Realizzazione di un'applicazione su rete attiva. Master's thesis, Politecnico di Torino, Italy, Feb. 1998. In Italian.

[74] Milner, R., Parrow, J., and Walker, D. A Calculus for Mobile Processes I. *Information and Computation 100*, 1 (Sept. 1992), 1–40.

[75] Mühlaüser, M., Ed. *Special Issues in Object-Oriented Programming: Workshop Reader of the 10^{th} European Conf. on Object-Oriented Programming ECOOP'96* (July 1996), dpunkt.

[76] National Bureau of Standards. *Data Encryption Standard*, Apr. 1977. FIPS Publication no. 46.

[77] Necula, G. Proof-Carrying Code. In *Proc. of the 24^{th} ACM Symp. on Principles of Programming Languages* (Paris, France, Jan. 1997).

[78] Nuttall, M. Survey of systems providing process or object migration. Tech. Rep. Doc 94/10, Dept. of Computing, Imperial College, May 1994.

[79] Object Management Group. *CORBA: Architecture and Specification*, Aug. 1995.

[80] OSI. ISO 9595 Information Technology, Open System Interconnection, Common Management Information Protocol Specification, 1991.

[81] Ousterhout, J. *Tcl and the Tk Toolkit.* Addison-Wesley, 1995.

[82] Ousterhout, J., Levy, J., and Welch, B. The Safe-Tcl Security Model. Tech. rep., Sun Microsystems, Nov. 1996.

[83] Park, A., and Leuker, S. A Multi-Agent Architecture Supporting Services Access. In Rothermel and Popescu-Zeletin [94], pp. 62–73.

[84] Peine, H., and Stolpmann, T. The Architecture of the Ara Platform for Mobile Agents. In Rothermel and Popescu-Zeletin [94], pp. 50–61.

[85] Perkins, C. IP mobility support. RFC 2002, 1996.

[86] Picco, G.P. μCode: A Lightweight and Flexible Mobile Code Toolkit. Tech. rep., Politecnico di Torino, Feb. 1998. Submitted for publication.

[87] Picco, G.P., Roman, G.-C., and McCann, P. Expressing Code Mobility in Mobile UNITY. In *Proc. of the 6^{th} European Software Engineering Conf. held jointly with the 5^{th} ACM SIGSOFT Symp. on the Foundations of Software Engineering (ESEC/FSE '97)* (Zurich, Switzerland, Sept. 1997), M. Jazayeri and H. Schauer, Eds., vol. 1301 of *LNCS*, Springer, pp. 500–518.

[88] Picco, G.P., Roman, G.-C., and McCann, P. Reasoning about Code Mobility with Mobile UNITY. Tech. Rep. WUCS-97-43, Washington University, St. Louis, MO, USA, Dec. 1997. Submitted for journal publication.

[89] Ranganathan, M., Acharya, A., Sharma, S., and Saltz, J. Network-Aware Mobile Programs. In *Proc. of the USENIX 1997 Annual Technical Conf.* (Anaheim, CA, USA, Jan. 1997). Version available as Univ. of Maryland Dept. of Computer Science Technical Report CS-TR-3659.

[90] Rivest, R. The MD5 Message Digest Algorithm. RFC 1321, Apr. 1992.

[91] Roman, G.-C., and Cunningham, H. Mixed Programming Metaphors in a Shared Dataspace Model of Concurrency. *IEEE Trans. on Software Engineering 16*, 12 (Dec. 1990), 1361–1373.

[92] Roman, G.-C., McCann, P., and Plun, J. Mobile UNITY: Reasoning and Specification in Mobile Computing. *ACM Trans. on Software Engineering and Methodology 6*, 3 (July 1997), 250–282.

[93] Rosenblum, D., and Wolf, A. A Design Framework for Internet-Scale Event Observation and Notification. In *Proc. of the 6^{th} European Software Engineering Conf. held jointly with the 5^{th} ACM SIGSOFT Symp. on the Foundations of Software Engineering (ESEC/FSE '97)* (Zurich, Switzerland, Sept. 1997), M. Jazayeri and H. Schauer, Eds., vol. 1301 of *LNCS*, Springer, pp. 344–360.

[94] Rothermel, K., and Popescu-Zeletin, R., Eds. *Mobile Agents: 1^{st} International Workshop MA '97* (Apr. 1997), vol. 1219 of *LNCS*, Springer.

[95] Rozier, M., Abrossimov, V., Armand, F., Boule, I., Gien, M., Guillemont, M., Herrmann, F., Kaiser, C., Leonard, P., Langlois, S., and Neuhauser, W. Chorus Distributed Operating Systems. *Computing Systems 1* (Oct. 1988), 305–379.

[96] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W. *Object-Oriented Modeling and Design.* Prentice Hall, 1991.

[97] Schneier, B. *Applied Cryptography—Protocols, Algorithms, and Source Code in C*, 2^{nd} ed. John Wiley & Sons, Inc., 1996.

[98] Schönwälder, J. Network Management by Delegation From Research Prototypes Towards Standards. In *Proc. of the 8^{th} Joint European Networking Conf.* (May 1997).

[99] Shaw, M., and Garlan, D. *Software Architecture: Perspective on an Emerging Discipline.* Prentice Hall, 1996.

[100] Stamos, J., and Gifford, D. Implementing Remote Evaluation. *IEEE Trans. on Software Engineering 16*, 7 (July 1990), 710–722.

[101] Stamos, J., and Gifford, D. Remote Evaluation. *ACM Trans. on Programming Languages and Systems 12*, 4 (Oct. 1990), 537–565.

[102] Steiner, J., Neuman, B., and Schiller, J. Kerberos: An Authentication Service for Open Network Systems. In *Proc. of the Winter USENIX Conf.* (Feb. 1988), pp. 191–201.

[103] Strasser, M., Baumann, J., and Hohl, F. Mole—A Java Based Mobile Agent System. In Mühlaüser [75], pp. 327–334.

[104] Sun Microsystems. The Java Language: An Overview. Tech. rep., Sun Microsystems, 1994.

[105] Sun Microsystems. *Java Core Reflection*, Jan. 1997. Available at `ftp://ftp.javasoft.com/docs/jdk1.1/java-reflection.pdf`.

[106] Sun Microsystems. *Java Media Framework*, Jan. 1997. Available at `http://java.sun.com/products/java-media/jmf`.

[107] Sun Microsystems. *Java Object Serialization Specification*, Feb. 1997. Available at `http://java.sun.com/products/jdk/rmi/serial`.

[108] SUN MICROSYSTEMS. *Java Remote Method Invocation Specification*, Feb. 1997. Available at `http://java.sun.com/products/jdk/rmi`.

[109] TARDO, J., AND VALENTE, L. Mobile agent security and Telescript. In *Proc. of IEEE COMPCON'96* (Feb. 1996).

[110] TENNENHOUSE, D., SMITH, J., SINCOSKIE, W., WETHERALL, D., AND MINDEN, G. A Survey of Active Network Research. *IEEE Communications 35*, 1 (Jan. 1997), 80–86.

[111] THIEL, G. Locus operating system, a transparent system. *Computer Communications 14*, 6 (1991), 336–346.

[112] THOMSEN, B., LETH, L., PRASAD, S., KUO, T.-M., KRAMER, A., KNABE, F., AND GIACALONE, A. Facile Antigua Release programming guide. Tech. Rep. ECRC-93-20, European Computer Industry Research Centre, Munich, Germany, Dec. 1993.

[113] TSCHUDIN, C. *An Introduction to the* `MO` *Messenger Language*. Univ. of Geneva, Switzerland, 1994.

[114] TSCHUDIN, C. OO-Agents and Messengers. In *ECOOP'95 Workshop W10 on Objects and Agents* (Aug. 1995).

[115] VIGNA, G. Protecting Mobile Agents through Tracing. In *Proc. of the 3^{rd} Int. Workshop on Mobile Object Systems (MOS'97)* (1997), J. Vitek and C. Tschudin, Eds.

[116] VITEK, J., AND TSCHUDIN, C., Eds. *Mobile Object Systems: Towards the Programmable Internet*, vol. 1222 of *LNCS*. Springer, Apr. 1997.

[117] WALDBUSSER, S. Remote Network Monitoring Management Information Base. RFC 1757, Feb. 1995.

[118] WALDO, J., WYANT, G., WOLLRATH, A., AND KENDALL, S. A Note on Distributed Computing. In Vitek and Tschudin [116]. Also available as Sun Microsystems Laboratories Technical Report TR-94-29.

[119] WELLENS, C., AND AUERBACH, K. Towards Useful Management. *The Simple Times 4*, 3 (July 1996).

[120] WETHERALL, D., GUTTAG, J., AND TENNENHOUSE, D. ANTS: A Toolkit for Building an Dynamically Deploying Network Protocols. Tech. rep., MIT, 1997. Submitted for publication to IEEE OPENARCH'98.

[121] WHITE, J. Telescript Technology: The Foundation for the Electronic Marketplace. Tech. rep., General Magic, Inc., 1994. White Paper.

[122] WHITE, J. Telescript Technology: Mobile Agents. In *Software Agents*, J. Bradshaw, Ed. AAAI Press/MIT Press, 1996.

[123] WILHELM, U. Cryptographically Protected Objects. Tech. rep., Ecole Polytechnique Fédérale de Lausanne, Switzerland, 1997.

[124] WONG, D., PACIOREK, N., WALSH, T., DICELIE, J., YOUNG, M., AND PEET, B. Concordia: An Infrastructure for Collaborating Mobile Agents. In Rothermel and Popescu-Zeletin [94], pp. 86–97.

[125] WOOLDRIDGE, M., AND JENNINGS, N. Intelligent Agents: Theory and Practice. *Knowledge Engineering Review 10*, 2 (June 1995).

[126] XU, A., AND LISKOV, B. A design for a fault-tolerant, distributed implementation of Linda. In *FTCS 19 Digest of Papers. The 19^{th} Int. Symp. on Fault-Tolerant Computing* (Chicago, IL, June 1989), IEEE Computer Society Press, pp. 199–206.

[127] YEMINI, Y. The OSI Network Management Model. *IEEE Communications* (May 1993), 20–29.

[128] YEMINI, Y., AND DA SILVA, S. Towards Programmable Networks. In *IFIP/IEEE Int. Workshop on Distributed Systems: Operations and Management* (L'Aquila, Italy, Oct. 1996).

[129] YEMINI, Y., GOLDSZMIDT, G., AND YEMINI, S. Network Management by Delegation. In *Proc. of the 2^{nd} Int. Symp. on Integrated Network Management* (Apr. 1991).

[130] ZIMMERMAN, P. *PGP User's Guide*, Mar. 1993.