Expressing Code Mobility in Mobile UNITY

Gian Pietro Picco[1,2], Gruia-Catalin Roman[2], and Peter J. McCann[2]

[1] Dipartimento di Automatica e Informatica, Politecnico di Torino,
C.so Duca degli Abruzzi 24, 10129 Torino, Italy
picco@polito.it

[2] Department of Computer Science, Washington University,
Campus Box 1045, One Brookings Drive, Saint Louis, MO 63130-4899, USA
[roman | mccap]@cs.wustl.edu

Abstract. Advancements in network technology have led to the emergence of new computing paradigms that challenge established programming practices by employing weak forms of consistency and dynamic forms of binding. Code mobility, for instance, allows for invocation-time binding between a code fragment and the location where it executes. Similarly, mobile computing allows hosts (and the software they execute) to alter their physical location. Despite apparent similarities, the two paradigms are distinct in their treatment of location and movement. This paper seeks to uncover a common foundation for the two paradigms by exploring the manner in which stereotypical forms of code mobility can be expressed in a programming notation developed for mobile computing. Several solutions to a distributed simulation problem are used to illustrate the modeling strategy for programs that employ code mobility.
Keywords. Code mobility, UNITY, Mobile UNITY, coordination, mobile computing, mobile code languages.

1 Introduction

Code mobility is defined informally as the capability, in a distributed application, to dynamically reconfigure the binding between code fragments and the location where they are executed [2]. *Mobile code languages (MCLs)* [12,11] provide specialized abstractions and run-time support capabilities designed to support various forms of code mobility. They assume that hosts and communication links are part of a highly dynamic global computing platform, where the application code can move freely among the computing nodes. This *network centric* style of computing is at the center of the emerging mobile code languages.

For a comprehensive survey of MCLs, the reader is directed to [5] which reviews a number of existing languages and attempts to extract their essential features. The unit of mobility, that in [5] is called *executing unit*, is implemented differently in different languages, but can be thought of as a process in an operating system or a thread in a multithreaded environment. *Strong mobility* allows executing units to move their code and their *execution state* to a different site. *Weak mobility* allows an executing unit at a site to be bound dynamically to freshly initialized code coming from a different site.

By and large, these developments 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, MCLs 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 paper.

The model we use in our study is called *Mobile UNITY* [7,10], an extension of work by Chandy and Misra on UNITY [4]. 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.

The remainder of this paper is structured as follows. Section 2 describes a simplified version of a distributed simulation problem and introduces standard UNITY via a centralized solution. Section 3 presents a 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. Section 4 introduces basic concepts relevant to code mobility, provides some background information, and presents mobile code solutions to the distributed simulation problem. They illustrate our strategies for modeling mobile code in Mobile UNITY and are modeled after the taxonomy of mobile code design paradigms found in [2]. Finally, Section 5 explores some of the issues raised by this investigation.

2 A Distributed Simulation Problem

In this section we present an example that will be used for illustration purposes throughout the remainder of the paper and 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 [3]. 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 section, 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.

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 definition of the function $f_i(t(i), T, z)$ where $t(i)$ denotes the value for a local timer, T denotes the GVT value, and z identifies the simulation mode, e.g., a parameter of the physical components. The definition captures 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.
3. The value of a local timer can never be behind the GVT. For such cases the function f is undefined.

In this section, we discuss a solution for the distributed simulation problem that uses the UNITY notation described in [4]. In the UNITY program shown in Fig. 1, the **declare** section contains variable declarations. The array t contains the value of the local timer for each process i. T stores the current value for the GVT and z represents the simulation mode. The **initially** section contains a set

[1] The three-part notation $\langle$**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 $min\{t(i) \mid 0 \leq i \leq N-1\}$.

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

Fig. 1. Standard UNITY solution for the distributed simulation problem.

of predicates, separated by the symbol $[]$, which define the allowed set of initial states. Uninitialized variables will assume an arbitrary value 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, $\bar{z}$. 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, otherwise it 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 all the $t(i)$ 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. The details about the behavior of processes are irrelevant. In this solution, we abstract them into a function $SimulationWork(t)$, which will be 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.

3 A Client-Server Solution in Mobile UNITY

Although the solution presented in Fig. 1 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 this section 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 Section 4 to describe the distributed solutions exploiting code mobility.

In the *Client-Server (CS)* paradigm, a server component exports a set of services. The client component, at some point in its execution, lacks some of the resources needed to proceed with its computation. The resources are located on the server host and they are accessed by the client by interacting with the server through message passing. The interaction specifies the service that needs to be invoked on the server in order to access the resources.

As in our earlier centralized solution, we will 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 Fig. 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 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 2 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)*, will serve 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 $\mathsf{Location}()$ 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 mod-

```
System DSClientServer
    Program P(i) at λ
        declare
            t, z : integer [] T : integer ∪ {⊥} [] RQ : request ∪ {⊥}
        initially
            t, z = 0, z̄ [] 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
```

Fig. 2. Client-Server solution for the distributed simulation problem.

els actual migration of the component through some physical or logical address space.

Throughout this paper, we will leave the type of λ unspecified. The type may be determined by the characteristics of a particular problem domain and needs to be specified only to the extent necessary to the respective application. We will 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, although we will sometimes leave off the program name when the context is clear.

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 Fig. 2 allow for communication between each client and the

$$
\begin{array}{l}
\mathsf{clientAddress} = \langle Set\ n : 0 \le n \le N-1 :: n \rangle \\
\mathsf{address} = \{\{\textsc{server}\} \cup \mathsf{clientAddress}\} \\
\mathsf{opName} = \{\textsc{cs},\textsc{rev},\textsc{cod}\} \\
\mathsf{opStatus} = \{\textsc{wait}\} \\
\mathsf{request} = \langle \mathsf{address},\ \mathsf{opName} \cup \mathsf{opStatus} \cup \{\perp\},\ \mathsf{serviceName} \cup \{\perp\},\ \mathsf{integer} \rangle \\
\mathsf{serviceRequest}(j : \mathsf{opName}, k : \mathsf{serviceName}, i : \mathsf{clientAddress}) \equiv \\
\qquad P(i).RQ \uparrow 1 = \textsc{server} \wedge P(i).RQ \uparrow 2 = j \wedge P(i).RQ \uparrow 3 = k \\
\mathsf{serviceReady}(i : \mathsf{clientAddress}) \equiv Server.q(i) \uparrow 1 = i
\end{array}
$$

Fig. 3. Macro definitions used in the example systems. Allowed values for each data element are represented as sets.

server. We assume that the index i is instanced 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:

$$
\begin{array}{ll}
Server.q(i) := P(i).RQ & \textbf{when } \neg\mathsf{def}(Server.q(i)) \wedge \\
& \qquad \mathsf{serviceRequest}(\textsc{cs}, \textsc{minServ}, i)
\end{array}
$$

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.

The meaning of the macros used in the guards of the interactions is shown in Fig. 3. These macros will be used throughout the remaining examples. A request is a tuple consisting of four elements. We denote the tuple by enclosing it in angle brackets and separating its four elements by commas. We use the field index operator $\uparrow$ to access individual fields of a record, 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, with the meaning defined later. This field may also represent 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 across all clients. 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 will take 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 2, except that now the variable 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 will compute this value dynamically. Otherwise, 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 is a predicate that 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 is a predicate that 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 *P(i)* processes the reply by updating the local GVT estimate $P(i).T$ and clearing the request buffer.

The client-server example does not exercise all the features of Mobile UNITY. 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. 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. 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, for instance, that the message buffer will take 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

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

Similarly, a pair of **disengage** values may be specified that are assigned atomically to the variables when they become disconnected:

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

Note that if no **engage** value is specified, the variables will remain in an inconsistent state after sharing takes effect until the first assignment is propagated. If no **disengage** value is specified, the variables will 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. 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**. 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, although for the purposes of this paper, we will express our solutions using only shared variables. The reader should keep in mind that transient sharing is really a shorthand notation for a set of reactive statements. Further information, including a proof logic that has been developed to match the new operational model, can be found in [7].

4 Mobile Code Design Paradigms

The idea behind code mobility is not new, as witnessed by the work of Stamos et al. [6] and Black et al. [1]. Nevertheless, these technologies were conceived mostly to provide operating system support on a LAN, while MCLs explicitly target large scale distributed systems—like the Internet.

On the Internet, client-server is the most used paradigm for the development of applications. In this paradigm, the application code is statically bound to the client and server hosts and the binding cannot be changed during the execution of the distributed application. Each interaction between the client and the server makes use of the communication infrastructure through message passing or some higher level mechanism like remote procedure call (RPC). Mechanisms that actually hide the location of components from the application programmer are also being considered, e.g., CORBA [9].

By contrast, in MCLs component locations are not hidden. Location is explicitly handled by the programmer who is able to specify *where* the computation of a given code fragment must take place. This capability leads to new paradigms for the design of distributed applications where the interaction between client and server is no longer constrained to exchanging simple, non-executable data via the network. For example, a portion of the client may move in order to bypass the communication infrastructure and to achieve local interaction with the server. This may improve performance by reducing latency and may increase dependability by avoiding problems inherent to partial failures.

The essential features of the interaction patterns found in MCLs can be characterized by considering the kinds of pairwise interactions that are possible between two software design components located on different hosts. In [2], Remote Evaluation (REV), Mobile Agent (MA), and Code on Demand (COD) are identified as the key mobile code design paradigms. A schematic view of all the paradigms appears in Fig. 4.

In the following solutions the GVT 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. We will relax this assumption when we describe a solution based on the COD paradigm.

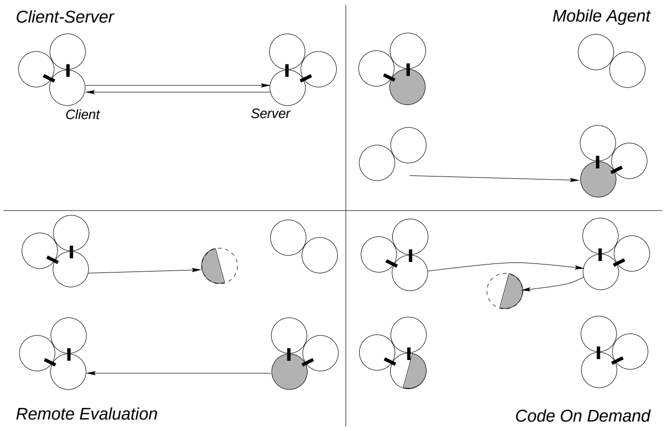

Fig. 4. Mobile code design paradigms. Components are identified by circles, and bindings between components are identified by black rectangles connecting circles. A component can be thought of as being composed of code, data state (holding data local to the component), and control state (holding information about its run-time execution). Grayed components participate in migration. Half-grayed dashed circles represent migration of a portion of a component, i.e., the code, or code plus the data state. Such portions can be used to create a new component, like in REV, or to augment an already existing component, like in COD. Arrows indicate network traffic, i.e., calls, return values, and code movement.

4.1 Remote Evaluation

The *Remote Evaluation* paradigm can be regarded as a variation of the CS paradigm where the server component offers its computational power and resources, like the server in a CS paradigm, but does not provide any application specific service. In the REV paradigm the client provides the code constituent for a component that will be instantiated on the server and bound there to the resources it needs to access. Eventually, a result will be sent back to the client component via a message, like in a CS paradigm. Hence, the REV paradigm leverages off the flexibility provided by the server, instead of relying on a fixed functionality. The REV paradigm is inspired by work on the REV [6] system, which extends remote procedure call with one additional parameter containing the code to execute on the server. Among recent MCLs, the paradigm is supported natively in TACOMA, Agent Tcl, Facile, and M0.

In the solution shown in Fig. 5, the client components *P(i)* behave similarly to the ones in the *DSClientServer* system. Besides computing a new local timer,

```
System DSRemoteEvaluation
    Program P(i) at λ
        declare
            t, z : integer [] T : integer ∪ {⊥} [] RQ : request ∪ {⊥}
        initially
            t, z = 0, z̄ [] 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, ⊥ [] 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
```

Fig. 5. Remote Evaluation solution for the distributed simulation problem.

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

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

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

$$Min(i).\tau \approx Server.\tau \qquad \textbf{when } Min(i).\lambda = Server.\lambda \quad \textbf{engage } Server.\tau \quad \textbf{disengage } \bot,\ Server.\tau$$

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. 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)* will retain the current value[2] of t, which will be used at the *Server* location to update of the global state. The actual code migration happens when

$$Min(i).\lambda := Server.\lambda \quad \textbf{when } \mathsf{serviceRequest}(\text{REV}, \text{MINSERV}, i) \wedge Min(i).\lambda = P(i).\lambda$$

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 implementations of RPC and of the REV [6] system. Analogous considerations hold for the output buffer of *Server*. In the REV solution,

[2] 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 [5]. We chose the second alternative, in order to illustrate how to dynamically establish and remove bindings among variables in Mobile UNITY.

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

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

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

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 5.

4.2 Mobile Agent

In the *Mobile Agent* paradigm, the client needs to access some of the resources that are located on the server and, like in the REV paradigm, it owns the necessary know-how to use such resources. In contrast with the REV paradigm, the client sends a whole component that is already being executed on the client host, together with its data and control state. Bindings to resources on the client host are voided and replaced by the new bindings to resources on the server host. The component, once arriving on the server host, resumes execution as if no migration took place. Typically, this step is repeated many times by the same component, which consequently is able to visit a number of hosts on behalf of the client without requiring interaction with it. The MA paradigm is supported natively by languages exploiting strong mobility, like Telescript and Agent Tcl.

Figure 6 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

```
System DSMobileAgent
    Program P(i) at λ
        declare
            t, z : integer [] T : integer ∪ {⊥}
        initially
            t, z = 0, z̄ [] 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 = ⊥, ⊥ [] ⟨|| 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
```

Fig. 6. Mobile Agent solution for the distributed simulation problem.

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. 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 value. 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 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. 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 will be changed by the recomputation of the minimum of all timers, which will take 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.

4.3 Code On Demand

The *Code On Demand* paradigm is gaining in popularity mainly due to the success of the Java language. In this paradigm, a component on a host performs some kind of computation on its local resources. When it recognizes that a portion of the know-how needed to perform the computation is lacking, the know-how is retrieved from some host on the network. The retrieved code augments the one already present in the client component and new bindings may be established on the client host. The COD paradigm is natively supported in Java through the class loader feature, and in Tcl derivates through the `unknown` function. In both cases, the programmer can determine the actions to be performed by the run-time support whenever a name cannot be resolved locally. In particular, these actions may encompass retrieval of the corresponding code from a remote site.

We present the solution exploiting the COD paradigm by enhancing the CS solution shown earlier. In the system shown in Fig. 7, 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

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

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

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

```
System DSCodeOnDemand
   Program P(i) at λ
      declare
         t, z : integer [] T : integer ∪ {⊥} [] static : boolean ∪ {⊥} [] RQ : request ∪ {⊥}
      initially
         t, z = 0, z̄ [] T = ⊥ [] λ = Location(i) [] static = true [] RQ = ⊥
      assign
         [] static := false                          if def(static)
         [] 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
         [] RQ := ⟨SERVER,COD, DYNMODE, ⊥⟩           if ¬def(RQ) ∧ static = false
   end
   Program Dyn(i) at λ
      declare
         z : integer [] T : integer ∪ {⊥}
      initially
         λ = Location(SERVER)
      assign
         z := d(T)      if def(T)
   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 [] ⟨[] i :: Dyn(i)⟩
   Interactions
         Dyn.λ, P(i).static := P(i).λ, ⊥              when Dyn(i).λ = Server.λ ∧
                                                            serviceRequest(COD, DYNMODE, i)
      [] P(i).z ← Dyn(i).z                            when P(i).λ = Dyn(i).λ
                                                      engage P(i).z
      [] Dyn(i).T ← P(i).T                            when P(i).λ = Dyn(i).λ
                                                      engage P(i).T
      [] 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
```

Fig. 7. Code On Demand solution for the distributed simulation problem.

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,

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

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)*. As in the REV solution, we are forced to instantiate statically multiple components from the same program, instead of migrating the code and instantiating components only when and if needed. This and other issues raised by the solutions presented so far will be discussed in the next section.

5 Conclusions

Mobile UNITY is a new model of distributed computing specialized for mobile computation, i.e., for systems in which components travel through space, compute in a decoupled fashion, and communicate opportunistically when colocated. 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 suffice 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 Mobile UNITY proof logic has been tentatively evaluated in the verification of the Mobile IP protocol [8]. Against this background of promising technical developments, this paper raised a simple question: Can Mobile UNITY model in straightforward manner the kinds of interactions that take place in applications involving mobile code?

The investigative style of this paper is empirical. We started with established mobile code paradigms and sought out corresponding Mobile UNITY solutions. As expected, 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 being unavailable (undefined) and strengthening the guards of related statements to check for availability of data. In several cases we used the fact that a variable was no longer available as the trigger 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 maybe 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.

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.

Acknowledgements This paper is based upon work supported in part by the National Science Foundation under Grant No. CCR-9217751. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the National Science Foundation. Gian Pietro Picco was partially supported by Centro Studi e Laboratori Telecomunicazioni S.p.A., Italy.

References

1. A. Black, N. Hutchinson, E. Jul, and H. Levy. Fine-Grained Mobility in the Emerald System. *ACM Transactions on Computer Systems*, 6(1), February 1988.
2. A. Carzaniga, G.P. Picco, and G. Vigna. Designing Distributed Applications with Mobile Code Paradigms. In R.N. Taylor, editor, *Proceedings of the 19th International Conference on Software Engineering*, 1997.
3. K.M. Chandy and J. Misra. Distributed Simulation: A Case Study in Design and Verification of Distributed Programs. *IEEE Transaction on Software Engineering*, 5(5):440–452, September 1979.
4. K.M. Chandy and J. Misra. *Parallel Program Design.* Addison-Wesley, 1988.
5. G. Cugola, C. Ghezzi, G.P. Picco, and G. Vigna. Analyzing Mobile Code Languages. pages 93–111. In [12].
6. J.W. Stamos and D.K. Gifford. Remote Evaluation. *ACM Transactions on Programming Languages and Systems*, 12(4):537–565, October 1990.
7. P.J. McCann and G-.C. Roman. Mobile UNITY: A Language and Logic for Concurrent Mobile Systems. Technical Report WUCS-97-01, Dept. of Computer Science, Washington University, St.Louis, 1996.
8. P.J. McCann and G-.C. Roman. Mobile UNITY Coordination Constructs Applied to Packet Forwarding for Mobile Hosts. Technical Report WUCS-96-15, Dept. of Computer Science, Washington University, St.Louis, 1996. To appear at the 2nd *Intl. Conf. on Coordination Models and Languages*, Berlin, September 1-3, 1997.
9. Object Management Group. *CORBA: Architecture and Specification*, August 1995.
10. G-.C. Roman, P.J. McCann, and J.Y. Plun. Mobile UNITY: Reasoning and Specification in Mobile Computing. Technical Report WUCS-96-08, Dept. of Computer Science, Washington University, St.Louis, 1996. To appear in *ACM Transactions on Software Engineering and Methodology.*
11. K. Rothermel and R. Popescu-Zeletin, editors. *Mobile Agents*, volume 1219 of *Lecture Notes on Computer Science*. Springer-Verlag, 1997.
12. J. Vitek and C. Tschudin, editors. *Mobile Object Systems: Towards the Programmable Internet*, volume 1222 of *Lecture Notes on Computer Science*. Springer-Verlag, 1997.