PEERWARE: Core Middleware Support for Peer-to-Peer and Mobile Systems

Gianpaolo Cugola and Gian Pietro Picco
Dipartimento di Elettronica e Informazione
Politecnico di Milano
Piazza Leonardo da Vinci, 32
20133 Milano, Italy
{cugola, picco}@elet.polimi.it

ABSTRACT

The pervasiveness of computer networks, together with the availability of wireless links, are steering distributed systems towards scenarios where computing is increasingly decentralized, decoupled, and dynamically reconfigurable. The popularity of and demand for applications that exploit mobile and peer-to-peer interactions is a symptom of such change. Nevertheless, by and large these applications are being built in an ad hoc manner, and often with architectures that, by sticking to the traditional client-server paradigm, do not fully capture and support the peculiar requirements of the new scenario.

In this paper, we present a new middleware, called PEERWARE, whose design is geared towards peer-to-peer and mobile systems. The paper is a presentation of the model underlying PEERWARE, followed by a discussion of its architectural implication, and by a description of the current implementation efforts.

1. INTRODUCTION

The last few years have witnessed a growing interest in paradigms of computing that are increasingly decentralized and exhibit high demands for flexibility and reconfigurability. This has brought to the frontline of research several problems that intrinsically involve a large number of components that are physically distributed and operate autonomously. The growing interest for coordination languages, multiagents architectures, and middleware is a result of this situation.

More recently, the problem of supporting large-scale and highly decentralized distributed computing has been further complicated by the evolution of communication technology. Wireless communication allows for new modes of exploiting the network. Users are able to connect to a fixed network (e.g., the enterprise-wide intranet, or even the Internet) without the need of being physically connected through cables, and they can remain connected even while in movement. In addition to this scenario, usually referred to as *base station mobility*, in more radical mobility settings users can completely bypass the fixed network and exploit opportunistically formed network structures, typically called *ad hoc networks* [16], where communication is enabled at all levels uniquely by the mobile hosts.

In both domains, that is, large-scale, highly decentralized computing and mobile computing, it is becoming evident how the traditional client-server paradigm is no longer sufficient. By imposing a tight coupling between clients and servers, and by relying on the permanent availability of the latter, this paradigm does not accommodate well the requirements of scalability and flexibility posed by the new distributed scenarios, and may be impractical in a mobile scenario where the physical and logical structure of the network is made extremely fluid by connectivity changes [20].

In this context, the peer-to-peer paradigm has been lately revamped by a flurry of research and commercial systems. Napster [1], Gnutella [2], and Freenet [3] are examples of applications that, by enabling distributed applications where each user shares transparently with all the connected peers information residing on the local host, have spurred a new round of hype about "the next Internet".

Beyond the hype, however, there are reasons that motivate a shift from client-server to peer-to-peer. In a peer-to-peer environment, users directly host the resources they want to share globally, without any need for publishing them on some server. Information and services are no longer gathered in a single point of accumulation on the network; instead, each peer is responsible for a subset of the global services. At the same time, the aforementioned applications are symptomatic and supportive of modes of exploiting the network that, rather than forcing the user to find the server to be contacted, allow her to query the global space of information at once. This is indeed a key feature in a heavily decentralized architecture, and happens to satisfy the requirement for a pattern of interaction Internet users experience on a daily basis, any time they use a search engines to find a specific Web server.

These motivations for peer-to-peer become even stronger when mobility, and especially ad hoc networking, is part of the picture. In such scenario, peer-to-peer interactions are simply the way of life, since they constitute the fundamental mode of communication forced by the network environment. Similarly, the ability to query the global space of informa-

Submitted to ESEC/FSE 2001.

tion at once assumes an even greater role, since the set of connected peers changes dynamically, and any kind of reference to another peer may quickly become obsolete and non-available.

It is interesting to note that, thus far, development of these applications is largely ad hoc, to the point that some of the aforementioned applications, that indeed provide the user with a peer-to-peer view, are actually implemented by exploiting the centralized client-server approach behind the scenes. No concept of *middleware*, i.e., of an application programming interface supporting abstractions that simplify the chore of developing applications in a specific domain, as yet appeared for peer-to-peer systems.

Similar considerations hold for the mobile computing domain, where the available middleware is either application-specific (e.g., Coda [14], or Bayou [21]), or geared towards the networking layers (e.g, Limbo [9] or Mobiware [8]).

Hence, the problem we address in this paper is: Can we simplify the development of peer-to-peer and mobile applications with middleware support? And, if yes, what are the fundamental abstractions this middleware should provide?

This paper is a reflection over these issues, concretized in the presentation of the first version of the model of a new middleware, called PeerWare, which directly supports the adoption of a peer-to-peer architecture for distributed applications. The first and most important application domain in which PeerWare is expected to be used is that of distributed applications involving mobile hosts, but we also consider more traditional scenarios, like the Internet, which introduce and stress different requirements, like scalability.

PeerWare is evidently influenced by our previous experience with the Jedi [10] and Lime middleware [15, 18]. Both these experiences were focused on supporting mobility, albeit under different forms. In the case of Jedi, a publish/subscribe middleware, the components were allowed to disconnect from the Jedi event dispatcher and to reconnect from a different location. Instead, Lime was especially conceived to support both the logical mobility of agents and the physical mobility of hosts in an ad hoc network, supporting coordination through a dynamically reconfigured Linda-like tuple space. PeerWare aims at taking the best from these two approaches and at integrating them, together with new ideas, under a unifying framework, with the overall goal of seeking simplicity and minimality of the resulting application interface.

The paper is structured as follows. Section 2 presents the requirements for PeerWare, the intended scenario of application, and the design goals we established for it. Section 3 provides the reader with some background by illustrating some approaches from which PeerWare borrows ideas. Section 4 discusses the strengths and weaknesses of these approaches, and sets the scenes for the presentation of the PeerWare model in Section 5, together with its architectural implications and a description of the current prototype. Section 6 summarizes the main contributions of this work, also by comparing it with similar research efforts. Finally, Section 7 hints at future work on PeerWare, together with some concluding remarks.

2. APPLICATION DOMAINS AND GOALS

PeerWare is currently being developed by targeting the needs of two projects we are involved in, which will give us the opportunity to verify the soundness of our middleware primitives against the requirements of real-world applications. Together, the requirements of these application cover most of the requirements for peer-to-peer and mobile computing. For this reason, this section illustrates in some detail the characteristics of these applications.

The first testbed application for PeerWare is a platform supporting distributed and mobile teamwork, currently being developed within the project MOTION[1], funded by the European Community. The goal of the project is to build a framework of teamwork services that can be tailored to the specific needs of companies to build an enterprise-wide infrastructure for cooperation. The target distributed teamwork applications must support potentially large groups of people, typically distributed over the enterprise network or even the Internet, which cooperate towards some goal possibly according to some business process. The platform must support searches for documents and people belonging to the organization, as well as the ability to subscribe to changes occurring in such information. Mobility is part of the requirements for this project, to enable users to take advantage of mobile hosts and wireless links for connecting to a fixed network and get involved in the cooperation process.

Hence, base station mobility is predominant in this scenario, and disconnections can be assumed to happen in a controlled way. Moreover, a peer-to-peer perspective naturally supports such an environment where the space of information of relevance for the application is actually built out of the resources individually made available by each user.

The other testbed application is instead concerned with the problem of providing support for operations in a disaster recovery scenario, within a research project currently under evaluation by the Italian Research Council. Disaster recovery is a typical example of an application demanding extreme forms of mobility like ad hoc networking. People operating on a disaster scene (e.g., an earthquake or a flooding) typically cannot rely on any form of fixed network, that is often destroyed. They must communicate by taking advantage only of their own devices, and typically cooperate by exchanging information about the field of operation, and by sending and receiving notifications about relevant events.

In this context, the middleware must take into account mobility in its fullest implications: the base station scenario is no longer applicable, and disconnections are frequent due to the fact that users actually move while communicating on the field of operation.

As mentioned in the previous section, the requirements of flexibility, scalability, and support for a dynamically changing environment in which new nodes connect and disconnect at run-time we identified here, intuitively clash with the notion of centralized and globally known servers that characterize the traditional client-server architectures. Conversely, they naturally lead to a peer-to-peer architecture in which the different pieces composing the application act as autonomous components that communicate and coordinate through the services provided by the middleware which, in turn, must be able to operate even in absence of any centralized and fixed support.

Hence, our main goal for PeerWare was to design a peer-to-peer middleware that provides abstractions that appropriately encompass the kind of reconfigurability required by the aforementioned application domains.

[1]IST-1999-11400, `www.motion.softeco.it`.

In doing this, we were driven by two objectives. The first was performance. We aimed at this goal by privileging choices in the definition of the PeerWare model that would reasonably lead to an efficient implementation of the middleware. The second objective was minimality. We aimed at this goal by trying to identify a minimal set of concepts and abstractions that other could exploit to build middleware providing higher-level abstractions.

3. THE ROOTS OF PeerWare

The conception and development of PeerWare has been influenced by several state-of-the-art approaches, concepts, and ideas concerned with the development of distributed middleware. While the footprints of these approaches can be clearly identified in PeerWare, the latter is not merely an assembly of previously developed ideas. Instead, our goal is to integrate the best of the different perspectives and cover the widest range of concerns under a single, unifying model.

In this section, we discuss these approaches in some detail, to provide the reader with the necessary background. The next section will provide a critique of these approaches, and set the scenes for the presentation of PeerWare.

3.1 Publish/Subscribe Systems

In publish/subscribe systems, also known as event-based systems, autonomous components interact through *event notifications*, often simply called *events*. Each component may notify a change in its state, or in the state of the environment it interacts with, by *publishing* an event E. Components may *subscribe* to one or more classes of events, thus expressing their interest in receiving them. All the components that subscribed to a class of events that includes E receive a copy of such event. At the architectural level, publish/subscribe systems usually include an *event dispatcher* a special component in charge of managing event distribution by collecting event subscriptions and distributing events to all the subscribers. Examples of systems exploiting the publish/subscribe paradigm are graphical user interfaces, embedded applications whose control flow is regulated by interrupt requests, component-based frameworks like Sun's JavaBeans [4], and several classes of distributed applications. Usually, a distributed, publish/subscribe application is built around a publish/subscribe middleware. It provides an implementation (some times distributed) for the event dispatcher, an event model, and a language to express event subscriptions[2].

Thus, publish/subscribe middleware embodies a communication and coordination model that is inherently asynchronous; multicast, because event notifications are sent to all the interested components; anonymous, because the identity of the sender is hidden to the receiver; implicit, because the set of recipients of each event notification is chosen implicitly, based on subscriptions, and cannot be changed by the sender; and stateless, because event notifications are not persistently stored by the system, rather they are sent only to components that subscribe before the event is sent.

3.2 Linda

Linda [13] can be regarded as a coordination model and an architectural style for building both centralized and distributed applications. In an application based on the Linda model the various components interact by writing and reading elementary data items, ordered sequences of typed fields called *tuples*, to and from a persistent, globally shared data space, the *tuple space*. Each component can drop, withdraw, or read a tuple in the tuple space by invoking, respectively, the `out`, `in`, or `read` primitive. The `in` and `read` primitives accept a pattern as a parameter and return a matching tuple, chosen non-deterministically among all the matching tuples present in the tuple space. Matching is based on the arity of the tuples and on the type of their fields. Furthermore, if no matching tuple is currently available during a `in` or `read` call, the caller is suspended until a matching tuple is inserted, thus allowing synchronization among components. In some variants of the model, non-blocking versions of the access primitives, called *probes*, are also provided.

Linda implementations have been traditionally used in the context of parallel computation. Recently, a number of Linda-based middleware (e.g., [5, 6]) have been conceived for distributed computing. Nevertheless, such middleware is typically just a client-server implementation enabling remote access to a centralized tuple space.

Communication in Linda can be both unicast and multicast, depending on the use of `in` and `read`; it allows synchronization among components, as well as asynchronous communication among them; anonymous; implicit, because writers cannot choose the readers of the tuple they write; and state-based, because tuples added to the tuple space remain there until some other component removes them.

3.3 Mobile Code

Code mobility[3], that is, the ability to relocate at run-time the software components of a distributed application, has become popular in recent years. While the idea has been typically associated to the Java language and its provision of mechanisms to program the dynamic loading of classes, the very concept of code mobility is rooted at the architectural level rather than at the technological one. When code mobility is employed, location becomes a first class concept, and designers may play with components location, by choosing how code and computation must move at run-time. The benefits expected from this expressive power are twofold:

- Designers are allowed to move the application knowledge close to resources, thus reducing bandwith requirements of distributed applications. On the other hand, it has been argued [12] that the effective benefits of this strategy strongly depend on the characteristics of the application and of the environment where the application runs.

- Designers are granted an unprecedented degree of flexibility in customizing applications. Code may be shipped or fetched at run-time to customize applications as required. In this case, the benefits are obvious and less arguable.

As we will explain better in Section 5, in PeerWare we used mobile code as a tool to obtain both benefits, but we focused on the second one. By taking advantage of mobile

[2]For a comparison of several publish/subscribe middleware see [11].

[3]The reader interested in learning more about technologies, architectures, and applications of code mobility, is redirected to [12].

code we have been able to provide a very small set of primitives that can be customized by the programmers to provide the required coordination semantics.

3.4 Global Virtual Data Structures

The concept of *global virtual data structure* (GVDS) [17] is a generalization of the LIME [15, 18] coordination model.

In LIME, each component holds a local tuple space, which is transiently shared with the tuple spaces of the connected components. By accessing its own local tuple space, each components has effectively access to the global tuple space whose content is defined by the union of the tuple spaces belonging to the connected components. In other words, actions that are perceived as local actually have a global effect.

With the term GVDS we refer to a meta-model of communication for mobile environments, centered around the idea of supporting coordination among a set of mobile hosts through a data space that is transiently shared and dynamically built out of the data spaces provided by each component in range. This data space is global because it includes all the data contained in all the local data spaces of the connected components, and it is virtual since it does not exist physically as a single entity, like the local ones. Instead, it is an "illusion" dynamically generated by the underlying middleware upon invocation of the local access primitives, and according to changes in connectivity. Hence, the GVDS paradigm naturally enables a *context-aware* style of coordination where, in every moment, the context is described by the content of the GVDS, which reflects the state of accessible components.

Observe that the GVDS paradigm is a meta-model of communication and coordination since it does not specify how the GVDS is structured (e.g., it could be a data matrix organized in such a way that each host holds part of the matrix, or a tree with each sub-tree held by a different component, and so on) and which primitives are provided to effectively access the GVDS. In particular, the set of primitives to access the GVDS may be chosen to hide the portion of the context containing information about the system configuration (e.g., the location of a given piece of data), or to reveal it. Moreover, primitives can then be tailored to the scenario of mobility taken into consideration for the specific incarnation of the meta-model, thus allowing developers to choose the desired compromise between expressiveness, context-awareness, and efficiency of the resulting middleware.

4. TOWARDS A NEW MIDDLEWARE

Each of the aforementioned approaches essentially covers a facet of the solution towards the problem of supporting peer-to-peer and mobile applications and, at the same time, exhibits some drawbacks.

The publish/subscribe paradigm is well suited for distributed and mobile applications that need purely message-based communication and that are largely leveraging the ability to react to changes in the state of the other components. Given the high degree of reconfiguration undergoing in the scenarios we target, this feature is surely a desirable one. Moreover, the different publish/subscribe middleware available today show that this paradigm is amenable to be implemented in a very efficient and scalable way by distributing the event dispatcher. On the other hand, by adopting purely asynchronous, stateless communication primitives only, publish/subscribe middleware offers a rather limited support to coordination and synchronization.

Conversely, the Linda paradigm, centered around a persistent and globally accessible, shared data space, very easily supports state-based coordination and synchronization among components but, on the other hand, it offers only limited support to a reactive model of execution.

In PEERWARE, we essentially rejoin the two perspectives under a single coordination model that ties together event notification and the notion of a shared data space holding the global context for the computation. Event definition, rather than being completely arbitrary, is always tied to portion of the state contained in the data space. Furthermore, applications are able both to query the global data space and to subscribe to events occurring in it.

In this perspective, the notion of GVDS provides the conceptual framework that allows to cope with the reconfiguration needs of physical mobility and peer-to-peer systems by defining a global data space that can be transparently and dynamically broken apart and reconstructed.

Together with this form of support for reconfiguration at a physical level, mobile code adds the capability for dynamic reconfiguration at the logical level. By enabling dynamic relocation of code, PEERWARE allows programmers to redefine at run-time the coordination primitives used to interact with the shared data space, and thus provide another level of flexibility.

To our knowledge, such a synergic combination of the aforementioned approaches is novel, and it has the potential to open up opportunities for the development of a new breed of middleware technology.

5. PEERWARE

In this section we first describe the communication and coordination model adopted by PEERWARE, which is strongly rooted on the idea of GVDS. Then we provide some architectural considerations that may guide the implementation of a middleware that implements this model. Finally, we describe our first prototype for such a middleware.

5.1 The Model

PEERWARE is centered around the notion of a global virtual data structure built out of the local data structures contributed by each user. From the point of view of the user accessing this GVDS, the content of the data structure is automatically and dynamically reconfigured according to changes occurring in the system, typically induced by changes in connectivity among components. This approach is very similar to the one exploited in LIME [15, 18], PEERWARE exploits a richer data structure that, together with the choices made about the operations allowed on it, is intended to give more expressive power to the application programmer and to open up opportunities for optimizations at the run-time layer.

5.1.1 Data as Forests of Trees

The data structure managed by PEERWARE is organized as a graph composed of *nodes* and *documents*, collectively referred to as *items*. More formally, a *data structure* is a tuple:

$$DS = \langle N, D, \Gamma, L, \Lambda \rangle$$

where:

- N is a set of *nodes*;
- D is a set of *documents*;
- $\Gamma \subseteq N \times N \cup N \times D$ is the *containment relation*, such that $n\Gamma i$ iff node n contains item i;
- L is a set of *labels*;
- $\Lambda : N \rightarrow L$ is a *labelling function*, which associates each node with a (non necessarily distinct) label. It is a total function.

Nodes are essentially containers of items, and are meant to be used to structure and classify the documents managed through the middleware. To enforce this meaning, Γ must satisfy the following properties:

$$\forall n_1, n_2 \in N \ (n_1 \Gamma^* n_2 \Rightarrow \neg(n_2 \Gamma^* n_1))$$
$$\forall n_1, n_2, n_3 \in N \ (n_1 \Gamma n_3 \wedge n_2 \Gamma n_3 \Rightarrow n_1 = n_2)$$
$$\forall d \in D \ \exists n \in N \ (n \Gamma d)$$

where Γ^* is the transitive closure of Γ, defined as

$$\forall n_1, n_2 \in N \ (n_1 \Gamma^* n_2 \Leftrightarrow n_1 \Gamma n_2 \vee \exists n \in N \ (n_1 \Gamma^* n \wedge n \Gamma n_2))$$

This basically means that nodes are structured in a forest of trees, with distinct root, which most likely represent different perspectives on the documents contained into the data structure. For instance, one could have an "Administration" tree, a "Current Projects" tree, and so on. Within this graph, each node is linked to at most one parent node and may contain different children nodes. Conversely, standalone documents are forbidden; documents are linked to at least one parent node and do not have children. Hence, a document may be contained in multiple nodes.

As for labels, two nodes may have the same label, as long as they are not both roots and are not directly contained into the same node. Formally this means that Λ must comply to the following rules:

$$\forall n_1, n_2 \in R \ (n_1 \neq n_2 \Rightarrow \Lambda(n_1) \neq \Lambda(n_2))$$
$$\forall n, n_1, n_2 \in N \ (n \Gamma n_1 \wedge n \Gamma n_2 \wedge n_1 \neq n_2 \Rightarrow \Lambda(n_1) \neq \Lambda(n_2))$$

where $R \subseteq N$ is the set of roots in N:

$$R = \{n \in N \mid \nexists n' \in N (n' \Gamma n)\}$$

This organization of nodes is very useful to express complex data classification schemes. For instance, the document you are reading could be found under the "Software Engineering" node but also under the "Distributed Systems"

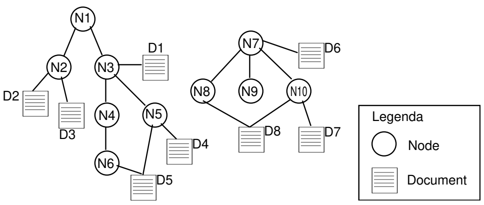

Figure 1: An example of the global virtual data structure managed by PeerWare.

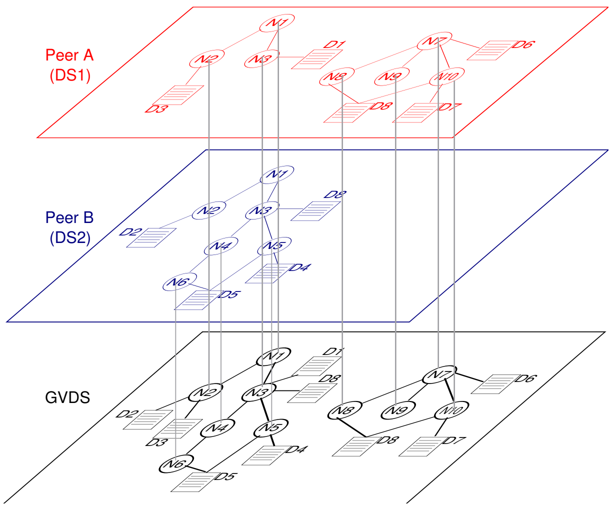

Figure 2: An example of the global data structure that results when two users are connected.

node, both part of the tree rooted at the "Information Technology" node. The resulting structure, depicted[4] in Figure 1 is similar to a standard file system with multiple roots where directories play the role of nodes, files are the documents, and Unix-like hard links are allowed only on documents.

5.1.2 *Sharing the Data under a* GVDS

Each component, which we will call from now on simply *peer*, using PeerWare is associated with a *local data structure DS*, whose content is assumed to be stored locally to the peer. As a special case, this data structure may be empty (e.g., in case of lightweight peers running on a PDA).

At any time, the local data structures held by the peers connected to PeerWare are made available to the other peers as part of the *global virtual data structure* managed by PeerWare. This gvds has the same structure of the local data structure (i.e., it complies with the above definition of DS) and its content is obtained by "superimposing" all the local data structures belonging to the peers currently connected, as shown in Figure 2.

More formally, let $DS_1 \ldots DS_k$ be the local data structures held by the set of k peers that are connected through PeerWare at a given instant in time:

$$DS_i = \langle N_i, D_i, \Gamma_i, L_i, \Lambda_i \rangle \quad i = 1 \ldots k$$

and let $GVDS$ be the gvds managed by PeerWare:

$$GVDS = \langle N, D, \Gamma, L, \Lambda \rangle$$

Beside the properties defined above for generic data structures DS, the following must be satisfied by $GVDS$:

$$D = \bigcup_{i=1\ldots k} D_i$$
$$L = \bigcup_{i=1\ldots k} L_i$$

[4] Observe that, to keep the example simpler, we used different labels for each node, but this is not a requirement. Notice also that to distinguish documents we labelled them with a name, which is not part of the model.

and N, Γ, and Λ have to be defined in such a way that it must be possible to define a *projection function* Π:

$$\Pi : \bigcup_{i=1\dots k} N_i \rightarrow N$$

which must be total, surjective, and must satisfy the following properties:

$$\forall i \in [1,k]\ \forall n_i \in N_i\ (\Lambda(\Pi(n_i)) = \Lambda_i(n_i))$$
$$\forall i \in [1,k]\ \forall n_i \in N_i\ (n_i \in R_i \Leftrightarrow \Pi(n_i) \in R)$$
$$\forall i \in [1,k]\ \forall n_i, n'_i \in N_i\ (n'_i \Gamma_i n_i \Leftrightarrow \Pi(n'_i)\Gamma\Pi(n_i))$$
$$\forall i \in [1,k]\ \forall n_i \in N_i\ \forall d \in D_i\ (n'_i \Gamma_i d \Leftrightarrow \Pi(n_i)\Gamma d)$$

where $R_i \subseteq N_i$ is the set of roots of DS_i, and $R \subseteq N$ is the set of roots of *GVDS*.

In practice this means that N, Γ, and Λ must be chosen in such a way that not only they must satisfy the constraint of a generic data structure DS, but it must also be possible to define a function Π that projects the content of *homologous nodes* into a single node of the GVDS which holds the same position of the projected nodes and has the same name. Homologous nodes are nodes with the same name and holding the same position in trees belonging to different peers. For instance, in Figure 2 the node N_2 on peer A contains a single document D_3, while the homologous node on peer B contains a single document D_2. However, the content of N_2 made available through the GVDS includes both D_2 and D_3.

Changes in connectivity among peers determine changes in the content of the global data structure constituting the GVDS, as new local data structures may become available or disappear. Nevertheless, the reconfiguration taking place behind the scenes is completely hidden to the peers accessing the GVDS, which need only to be aware of the fact that its content and structure is allowed to change over time. Moreover, it will also become clear in the remainder of the section how such conceptual reconfiguration can actually be implemented with only a minimal degree of reconfiguration taking place at the run-time layer.

5.1.3 *Operating on the* GVDS

Now that the data structure for the GVDS has been defined, the next question on the table is the definition of the operations that are allowed on such data structure.

It is important to note at this point how PEERWARE draws a sharp line between the operations that can be performed globally, i.e., on the whole GVDS, and those that instead can be performed only on the local data structure. This choice is different from the one taken by LIME, where the programmer always accesses a single data structure representing the GVDS, and is given syntactic means to restrict the scope of operations to localize their effects. Our claim is that a separation between local and global scope made at the model level is more natural, in that it mirrors the fact that only the local data structure is a *concrete* data structure, while the GVDS is a *virtual* data structure. While hiding this difference provides an elegant uniformity to the model, it may also hide the fundamental difference between local and remote effects of the operations. This, as pointed out in [22], may induce the programmer to use the operations in rather ineffective ways, and may heavily complicate the incarnation of the model into a middleware. We will elaborate further on these considerations in the next section.

The operations defined only on the local data structure are:

- createNode(n,n_f) and removeNode(n). Allow manipulation of the schema of the local data structure, by creating the new node n into an existing node n_f or by removing the node n.

- placeIn(d,n) and removeFrom(d,n). Allow manipulation of the content of the local data structure, by inserting (removing) a given document d into (from) a given node n.

- publish(e,i). Generates a notification that a given event e has occurred on a given item i.

Instead, the following are the operations available on both the local and the global data structures:

- I = execute($\mathcal{F}_N, \mathcal{F}_D, a$). Executes an arbitrary action a on the projection of the data structure identified by $\mathcal{F}_N$ and $\mathcal{F}_D$, by performing the following steps:

  1. The node filter function $\mathcal{F}_N$ is applied to the nodes of the data structure to return the set M_n of matching nodes.
  2. The document filter function $\mathcal{F}_D$ is then applied to all the documents included in the nodes in M_n, to return the set M_i of matching items.
  3. M_i is then handed to the action a, a code fragment that, starting with these items and, possibly by operating again on the data structure, yields a set I of items.
  4. I is then returned to the caller.

- subscribe($\mathcal{F}_N, \mathcal{F}_D, \mathcal{F}_E, c$). Allows a peer to subscribe to the occurrence of an event matching the event filter function $\mathcal{F}_E$ and being published within the projection of the data structure identified by the filter functions $\mathcal{F}_N$ and $\mathcal{F}_D$. When the event occurs the code fragment c is executed locally to the caller.

- I = executeAndSubscribe($\mathcal{F}_N, \mathcal{F}_D, \mathcal{F}_E, a, c$). Executes an arbitrary action a on the projection of the data structure identified by $\mathcal{F}_N$ and $\mathcal{F}_D$, similarly to execute. Also, in the same atomic step, it subscribes for events that match $\mathcal{F}_E$, and occur within the same projection, by specifying the code fragment c that must be executed locally to the caller, when one of such events occurs.

Despite the fact that the signature of these operations is identical for both local and global data structures, their effect is limited in scope by the nature of the data structure they are applied to. Moreover, also the semantics of the operations is affected by this choice. In particular, the semantics of a global operation can be regarded as being equivalent to a distributed execution of the corresponding operation on the local data structures of the peers currently connected.

This latter remark, however, raises another issue. The definition of the operations, and especially of those acting on the GVDS, must take into account the tradeoffs between expressive power, efficiency, and implementability. In particular, since the intended domain of application of PeerWare is the distributed and mobile setting, special attention must be paid to the issue of atomicity. Thus far, we said nothing about the atomicity of the operations, implicitly assuming it is somehow guaranteed. While this is a reasonable assumption for the operations defined on the local data structure, this may become an impractical assumption in a distributed setting. For this reason, we define variants of the global operations, namely $\overline{\text{execute}}$, $\overline{\text{subscribe}}$, and $\overline{\text{executeAndSubscribe}}$, that do not provide any guarantee about global atomicity, and guarantee only that the execution of the corresponding operations on each local data structure, that we said to be an integral part of the global execution, is correctly serialized (i.e., it is executed atomically on each local data structure).

It is interesting to note how the number of operations introduced by PeerWare is rather small. This is the consequence of a precise design choice, in that PeerWare is designed to provide a *minimal* core of primitives and run-time infrastructure that, although useful when employed directly, is conceived to be the base for middleware providing higher-level abstractions.

This design criteria explains also the role of the execute primitive, which is essentially a sort of meta-operation demanding to the action, provided by the caller, the actual semantics for accessing the data structure. We envision a scenario where a set of predefined actions are available as a library, e.g., operations to read one or more documents from the local data structure, operations to add metadata associated to documents, or to filter documents in ways that are application dependent. These operations may vary from implementation to implementation, and even from application to application. Actions are then a way to implement these operations, and ensure their execution in the context of a single atomic access to each local data structure. The execute operation essentially provides a basic mechanism to enable a global execution of such actions over the GVDS. For the same reason, it must be underlined that actions are not necessarily drawn from a predefined pool associated with the target local data structures. Instead, they can be defined by the caller, that is effectively allowed to change dynamically the high level coordination primitives by which it accesses the GVDS. This latter remark opens up architectural solutions that heavily exploit the notion of mobile code, on which we elaborate in the next section.

Clearly, the aforementioned notion of action provides the programmer with tremendous flexibility—and danger—in the way the local and global data structures can be accessed. For instance, in principle actions may have side effects that are outside the scope of the local data structure (e.g., send an e-mail, or invoke a method on a co-located object). Still, execute provides a level of flexibility that is appropriate for the designer of higher-level middleware or application tiers.

Similar considerations hold for the executeAndSubscribe primitive, that nevertheless extends execute with the ability to "hook" on some information, by allowing the realization of schemes providing strong consistency on such information by retrieving some data and monitoring events occurring on them. For instance, a programmer might want to retrieve the content of a node and be notified if any new document appears in that node, e.g., to build a graphical browser of the global data structure.

Observe that the same behavior cannot be obtained by simply invoking execute followed by subscribe. In fact, given the inherently distributed and asynchronous nature of the system, a peer could publish a relevant event right in between the execute and the subscribe. Such event would not be captured by the subscription, and the notification would never show up, thus leading to an inconsistent state.

5.2 Towards a Middleware: Architectural Considerations

Given our ultimate goal of delivering a new middleware, the model we presented thus far is meaningful only if an implementation providing reasonable efficiency can be conceived and realized in practice. In this section, we discuss what are the implications of our model in terms of deriving a distributed architecture and, ultimately, a middleware implementation supporting it. The next section will elaborate further on this latter aspect, illustrating our on-going prototyping activities.

The PeerWare model naturally suggests a middleware implementation that is intrinsically peer-to-peer, where each peer hosts a repository that contains its local data structure. An operation on the GVDS, e.g., a global execute, is then performed by disseminating on the connected peers the request for a local invocation of the corresponding primitive, and sending the results back to the caller. Hence, each peer needs to host a run-time support to manage the routing of system messages, like event notifications and requests for execute operations.

Nevertheless, the model does not prescribe anything about *how* such routing must be performed, e.g., what is the topology of the network interconnecting the peers, and what algorithms are used to perform routing on top of it. As such, the model leaves us free to experiment with several architectural alternatives, essentially driven by the network environment the middleware will operate in, and by non functional requirements like performance, scalability, and flexibility in supporting mobile hosts.

Thus, for instance, we are currently investigating two different implementations of PeerWare, one aimed at a fixed network environment, where connectivity among peers is determined by the explicit action of logging in and out of the system, and one aimed at an ad hoc network environment, where instead logical connectivity among peers is determined by the physical connectivity among their hosts. Clearly, the two implementations are likely to differ widely in the routing algorithms employed, due to the very different assumptions that can be made about the environment. For instance, to arrange the peers in a logical network that is hierarchical, and to route messages following this hierarchy may be a simple and efficient strategy for the fixed network case. On the other hand, this solution might be impractical in an ad hoc network, where the ever-changing topology would force a continuous reconfiguration of the hierarchy. A graph-like logical network would probably be more appropriate in this case, as it provides improved reliability by allowing multiple linkgs among peers. Nevertheless, it is our contention that application programmers will be shielded by these differences and will be able to use a single, unifying programming model in both scenarios.

On the other hand, the PeerWare model includes several choices that have been made on purpose to open up opportunities to improve efficiency and scalability of any Peer-

Ware implementation, independently from the underlying architecture. This is already evident in the choice of the gvds, whose hierarchical nature happens to provide a natural way to restrict the scope of the operations performed over the gvds, and thus to allow optimizations of the processing involved. For instance, the distribution of requests for an execute could be somehow "steered" only towards the peers that actually contain the nodes that are targeted by this operation. Analogously, the containment relation among nodes may enable implementations where subscriptions to events occurring in a node n are not propagated in the system if there is already an outstanding subscription for the parent node p. Similarly, this relation would enable a sort of event multiplexing for the case where a peer has two outstanding subscriptions, one for events occurring on n and one for events occurring on p. If an event that matches both subscriptions occurs on n, rather than sending it twice along the path from the event source to the destination, a single event could be transferred to destination and then demultiplexed at the target peer, thus potentially saving bandwidth in an application where this pattern occurs frequently.

This last consideration is related with another relevant issue that needs to be resolved before implementing an actual middleware, namely, the expressiveness of the languages used to specify the filters $\mathcal{F}_N$, $\mathcal{F}_D$, and $\mathcal{F}_E$. Here, the trade-offs are between the expressive power placed in the hands of the programmer and the burden of added complexity and overhead placed on the middleware run-time support.

Since each node n is uniquely characterized by the sequence of labels of the nodes in the path from the root of the tree holding n to n itself, a minimalist choice for the node filter $\mathcal{F}_N$ would be to allow the programmer to specify only one such sequences of labels, thus identifying a single node. Nevertheless, this would prevent some of the aforementioned optimizations and, for many applications, complicate excessively the life to the programmer. At the other extreme, one could choose to exploit an extremely powerful language based on regular expressions, to allow programmers to filter out different nodes in a single step (e.g., filtering all the subnodes of a given node).

Things become even more complex in the case of event filters and data filters, since their characteristics largely depend on the nature of events and documents, which are rightfully left unspecified by the model definition. As an example, if events were uniquely characterized by a name the event filter could be specified as a regular expression. Conversely, in the case of complex events characterized by several typed fields, one could imagine to exploit an extremely powerful filtering language. This approach is similar to the one taken by JMS [7], which prescribes the use of a subset of SQL-92. Similarly, if documents were as simple as tuples, the data filter could specify a Linda-like pattern matching on tuple fields. On the other hand, if documents were as complex as XML documents, then filters could be as complicated as an XML query.

Observe that, in the case of data filters, the capability to specify arbitrary actions in the operations on the gvds introduces an additional degree of freedom, by allowing the middleware designers to keep a very simple language for data filters, thus streamlining the distributed processing, and exploit actions to perform finer-grained filtering taking place while co-located with the actual data.

This last consideration highlights a peculiar characteristic of the PeerWare model. The mechanism of actions not only allows programmers to define dynamically the exact behavior of the primitives through which they access the gvds, but also allows computation to be moved close to resources, thus opening up interesting opportunities to efficiently implement complex operations over documents. As mentioned in the previous section, at the architectural level this involves the use of mobile code technology to implement the shipping and fetching of the code of actions.

Another peculiarity of the PeerWare model is the strong separation between local and global primitives. This allows programmers to easily adopt a location-aware model of coordination that could be pushed to its extreme if they were provided with a mechanism to access the local data structures of remote hosts. This mechanism has been rightfully left out of the PeerWare model, since it is focused on efficiently managing the gvds as a whole. Consequently, an important architectural issue becomes if and how the local primitives have to be made visible to remote peers. Several strategies are possible. On one extreme, one could imagine a pure gvds-based implementation of the PeerWare model that does not offer any mechanism to access the local data structures of remote peers. Even in this case, the PeerWare model would enable a certain form of location-aware programming through the use of actions, which are executed on remote peers where data is found and are able to access the local data structure of that peers. On the other extreme, a middleware adopting the PeerWare model could provide full access to the local data structure of remote peers.

5.3 Implementation Strategy

Our strategy towards implementing PeerWare follows an incremental approach, starting with a rather constrained application environment where the concepts and mechanisms of PeerWare are being checked for soundness and completeness against the experience of developing toy examples, demos, and real-world applications, and then evolving towards more challenging scenarios.

The first stage of this strategy involves building an implementation of PeerWare that satisfies the needs of users connected through a (medium-scale) fixed network, whose connectivity is ruled by the explicit action of logging in and out of the system, as well as mobile users whose connectivity is under control (e.g., users that are not physically moving while working). In this scenario, the fixed network provides a backbone of permanently active PeerWare hosts, which taking care of processing and routing the control messages related to requests for operations, as well as subscriptions to and notifications of events. As leaves of this backbone, other PeerWare hosts may be permanently or discontinuously attached. In particular, attached to the backbone is a dynamic fringe of mobile hosts, whose connectivity is enabled by wireless devices. This scenario encompasses peer-to-peer applications involving nomadic users, who dynamically collaborate by sharing documents and other information, like in the case of mobile teamwork applications described in Section 2.

The architecture of this first prototype relies heavily on Jedi [10], a traditional, albeit distributed, publish/subscribe core, which employs an event dispatching strategy based on a hierarchical arrangement of dispatchers, and already provides some support for dealing with disconnections happening in a controlled way. Jedi events and subscriptions are used to implement PeerWare events and subscriptions, but also to implement the routing of execute and executeAndSubscribe requests. In particular, each time a new node n is created on the local data structure of some peer, a special subscription is issued, holding the full path of n. Every call to a global execute or executeAndSubscribe whose node filter matches n is implemented by sending a special Jedi event that matches the subscription above. This way, it is possible to take advantage of Jedi mechanisms to route events as a mean to route execute and executeAndSubscribe calls only towards peers that hold matching nodes. Data filters are then executed locally. As for the support for mobile code, which is necessary to implement the notion of actions, it is provided by μCode [19], a lightweight and flexible toolkit for fine-grained code mobility.

Observe also that, given the requirements of our testbed application domains and the need of focusing on performance and scalability, we chose to implement only the non-atomic version of the global primitives. We plan to continue working on this first implementation to add more features in the future.

6. DISCUSSION AND RELATED WORK

The coordination paradigm adopted by PeerWare, which allows a set of components to share information and to react to changes occurring to this information, is very similar to that of Lime. On the other hand, several aspects contribute to differentiate the two middleware.

First of all, Lime does not distinguish between local and global primitives. This provides an elegant model of communication but does not help Lime users in adopting a location-aware style of programming. As pointed out in [22], the difficulty resulting from distinguishing between local and global actions often precludes the possibility of adopting the most efficient design solution. Moreover, the need for location-aware primitives has resulted in Lime in the introduction of the notion of "misplaced tuples". Lime components may specify a different component as the destination of a tuple that is being added to the global tuple space. If the two components are connected at the time the operation is issued, the result of the call is that the tuple is added to the local tuple space of the caller, and then moved to the destination tuple space in a single atomic step. Conversely, if the destination component is not connected at the time the operation is issued, the tuple is added to the tuple space of the caller and it is marked as "misplaced". When the destination component connects, an engagement protocol is run that moves the misplaced tuple into its local tuple space. To guarantee the consistency of the global tuple space in the presence of changes in the set of connected components, the engagement protocols runs as a distributed transaction that moves all the misplaced tuple to their expected destination. It is evident how this approach impacts on the scalability of the middleware.

As for events, Lime allows components to react only to the insertion of a new tuple. The event model of PeerWare is much more complex and results in greater expressive power. Moreover, Lime provides the concept of *strong reaction*. Strong reactions are supposed to fire atomically with the insertion of a new tuple, and run until fixed point is reached. Clearly, this mechanism provides expressiveness, but it may be impractical if applied in a distributed setting. For this reason, Lime actually constrains strong reactions to run only within the scope of a single host, where it is exploited mainly to support coordination among mobile agents.

Finally, at the level of the data model, the notions of node and tree of nodes provided by PeerWare enable a mechanism of scoping for both events and queries that increase the expressiveness of the model and the potential scalability of the middleware. Lime has nothing similar and the tuple space is a flat data space, which does not provide any mechanism to limit the scope of the different primitives used to access it.

Another class of potential competitors of PeerWare is that of publish/subscribe middleware. With respect to these environments, we observe that PeerWare implements most of the features they traditionally offer. Like most advanced publish/subscribe middleware, PeerWare fully exploits content based subscriptions through event filters. Moreover, it adds the concept of node as a tool to classify events and to reduce the scope of subscriptions, thus potentially increasing the scalability of the middleware[5].

Additionally, in traditional publish/subscribe middleware events are usually published as a result of changes in the internal state of one of the middleware clients. This state is not managed by the middleware itself, and consequently it is not immediately accessible to other clients. In most cases, this forces programmers to put in place application-level mechanisms to make this state visible to other clients, or it forces them to add to the event itself all the information needed to rebuild the relevant parts of this state. Conversely, PeerWare allows events to be directly associated to changes in any item that is part of the gvds managed by PeerWare itself. Since PeerWare peers may directly access this gvds, it is possible to reduce the information carried by events and to easily exploit complex interaction patterns.

As for scalability, one could argue that the primitives added by PeerWare to allow peers to access the gvds could reduce the middleware scalability. This is not the case thanks to the fact that PeerWare primitives have been designed from the beginning by keeping scalability in mind. Moreover, PeerWare publish/subscribe primitives are not influenced by other primitives and even considering these other primitives alone, as described in previous sections, they can be easily implemented by using a pure distributed, asynchronous approach. A practical proof of this claim is that the current prototype of PeerWare has been implemented on top of Jedi, a publish/subscribe middleware that has been explicitly designed to offer good scalability, without having to introduce any constraint in the way Jedi routes events and subscriptions.

As a last consideration, it is useful to compare PeerWare with some of the peer-to-peer environments that recently attracted the attention of the community of Internet users. In particular, here we refer to file-sharing applications like Napster [1], Gnutella [2], and Freenet [3] that adopt a peer-to-peer style of coordination among their users to allow them to share documents. First of all, it is relevant to observe that this application domain is quite different from the one for which PeerWare is intended. As an example, file sharing over the Internet usually does not require searches to be executed over the entire set of documents available. It is

[5]This mechanism is similar to that of topics in JMS [7].

considered perfectly acceptable that a search does not return any matching file even if matching files exist. This characteristics is exploited by the above mentioned systems to put in place special mechanisms to reduce the network traffic generated by searches and to improve scalability. Conversely, the PeerWare model does not consider this behavior acceptable. It is possible to imagine an implementation of PeerWare specially tailored for this application domain that relaxes the semantics of the global primitives to exploit similar approaches to reduce the scope of searches (i.e., of the execute primitive). Moreover, the natural scoping mechanism of PeerWare based on the concept of nodes could be used as an alternative, or complementary, mechanism to obtain similar results.

7. CONCLUSIONS AND FUTURE WORK

In this paper we presented PeerWare, a middleware for distributed, peer-to-peer and mobile applications. PeerWare adopts a model of coordination based on the notion of global virtual data structure in which each peer shares a set of data with the other peers, thus contributing to create the GVDS that is managed by the middleware. Together with this model of communication based on data sharing, PeerWare offers full support to an event-based communication style. The two models are strongly integrated, and take benefit of the same concepts to limit the scope of both operations to access the GVDS and events occurring on items in it. At the design level, we choose to provide a minimal set of primitives to access the GVDS and publish and subscribe to events, whose expressive power is enriched through the use of mobile code. In this sense, PeerWare is a core middleware on top of which application-specific primitives may be implemented.

Here, we presented also some architectural considerations that guided the implementation of PeerWare, and we described the results of our initial effort in providing a first prototype that follows these design guidelines.

One of our long-term research goals is actually to use PeerWare as both a conceptual and a "real" tool we can shape from time to time to adapt to various scenarios and to experiment with novel implementation strategies. In particular, we started working on a second prototype of PeerWare, especially tailored for ad hoc networking and we plan also to study the feasibility of a special implementation of PeerWare tailored to the application domain of file sharing over the Internet.

Finally, we plan to refine the model, and provide a complete formalization of both the data structure managed by PeerWare and the primitives it provides, with the goal of identifying possible weakness of the approach or the potential for improvements.

Acknowledgments

We would like to thank Carlo Ghezzi, Ouejdane Mejri, and Mattia Monga for the valuable comments they made on this document. Finally, we would like to thank Mirko Cesarini and Francesco Bardelli for their work on the prototype.

8. REFERENCES

[1] http://www.napster.com.
[2] http://www.gnutella.org.
[3] http://freenet.sourceforge.net.
[4] http://java.sun.com/products/javabeans.
[5] http://www.sun.com/jini/specs/js-spec.html.
[6] http://www.almaden.ibm.com/cs/TSpaces.
[7] http://java.sun.com/products/jms.
[8] O. Angin, A. Campbell, M. Kounavis, and R. Liao. The Mobiware Toolkit: Programmable Support for Adaptive Mobile Netwoking. *IEEE Personal Communications*, Aug. 1998.
[9] G. Blair, N. Davies, A. Friday, and S. Wade. Quality of Service Support in a Mobile Environment: An Approach Based on Tuple Spaces. In *Proc. of the 5th IFIP Int. Wkshp. on Quality of Service*, May 1997.
[10] G. Cugola, E. Di Nitto, and A. Fuggetta. Exploiting an event-based infrastructure to develop complex distributed systems. In *Proc. of the 19th Int. Conf. on Software Engineering (ICSE98)*, 1998.
[11] G. Cugola, E. Di Nitto, and A. Fuggetta. The jedi event-based infrastructure and its application to the development of the opss wfms. *IEEE Transactions on Software Engineering*, To appear.
[12] A. Fuggetta, G.P. Picco, and G. Vigna. Understanding Code Mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, May 1998.
[13] D. Gelernter. Generative Communication in Linda. *ACM Computing Surveys*, 7(1):80–112, Jan. 1985.
[14] J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. *ACM Trans. on Computer Systems*, 10(1):3–25, 1992.
[15] G.P. Picco, A. Murphy, and G.-C. Roman. Lime: Linda Meets Mobility. In D. Garlan, editor, *Proc. of the 21st Int. Conf. on Software Engineering (ICSE'99)*, pages 368–377, Los Angeles, CA, USA, May 1999. ACM Press.
[16] M.Corson, J.Macker, and G.Cinciarone. Internet-Based Mobile Ad Hoc Networking. *Internet Computing*, 3(4), 1999.
[17] A. Murphy. *Enabling the Rapid Development of Dependable Applications in the Mobile Environment.* PhD thesis, Washington University in St. Louis, MO, USA, Aug. 2000.
[18] A. Murphy, G.P. Picco, and G.-C. Roman. Lime: A Middleware for Physical and Logical Mobility. In *Proc. of the 21st Int. Conf. on Distributed Computing Systems (ICDCS-21)*, May 2001. To appear.
[19] G. P. Picco. μCode: A Lightweight and Flexible Mobile Code Toolkit. In *Proc. of Mobile Agents: 2nd Int. Workshop (MA'98)*, volume 1477 of *LNCS*, pages 160–171, Stuttgart (Germany), Sept. 1998. Springer.
[20] G.-C. Roman, G. Picco, and A. Murphy. Software Engineering for Mobility: A Roadmap. In A. Finkelstein, editor, *The Future of Software Engineering*, pages 241–258. ACM Press, 2000.
[21] D. Terry et al. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. *Operating Systems Review*, 29(5):172–183, 1995.
[22] J. Waldo, G. Wyant, A. Wollrath, and S. Kendall. A Note on Distributed Computing. In J. Vitek and C. Tschudin, editors, *Mobile Object Systems: Towards the Programmable Internet*, volume 1222 of *LNCS*. Springer, Apr. 1997.