# 1

# On Global Virtual Data Structures

*Gian Pietro Picco (Politecnico di Milano, Italy),*
*Amy L. Murphy (University of Rochester, NY, USA)*
*Gruia-Catalin Roman (Washington University in Saint Louis, MO, USA)*

**Abstract**

In distributed computing, global information is rarely available and most actions are carried out locally. However, when proving system properties we frequently turn to defining abstractions of the global state, and when programming we often find it convenient to think of a distributed system as a global centralized resource. In this paper we build upon this observation and propose the notion of *global virtual data structures* as a model for building a new generation coordination models and middleware that allows programmers to think of local actions as having a global impact and places upon the underlying system the burden of preserving this appearance. The model itself is inherently peer-to-peer, lending itself toward applications which are largely decentralized, and built out of autonomous components.

## 1.1 INTRODUCTION

Distributed computing is no longer the exception, but rather the rule. More often than not, the loss of network connectivity severely cripples if not completely destroys the ability to continue processing. As networks continue to increase in size, pervasiveness, and complexity, it becomes more difficult to both conceptualize the distributed system and understand the consequences of interacting with various parts. Further complicating these issues is the existence of multiple users, simultaneously interacting with the same system, making changes and viewing the changes made by others.

Moreover, distributed computing is increasingly moving from a rather static and controlled architecture, stigmatized by the client-server paradigm, towards more autonomous, decentralized, and dynamic settings. For instance, Two environments exposing such dynamicity are those defined by peer-to-peer networks and by mobility. In peer-to-peer systems, the members of the community are constantly changing, and each peer is individually responsible for the data which they are sharing with the rest of the system and for the propagation of queries operating on the data. Mobility adds an additional level of complexity by allowing hosts to move through space, altering the shape of the underlying communication framework along with the changes in connected components as hosts move in and out of wireless communication range. Another level of dynamicity comes from the applications themselves which must change to adapt to the changing topologies and available data. Such applications may be formed out of logically mobile components which can relocate to better utilize local resources or perform load balancing. All of these dynamic aspects inherent to these environments increase the effort in application development.

In this work we present an abstract model which, unlike other models, does not identify a single view of distributed environments, rather it lays a foundation for building a broad range of models. Our work is grounded in a study of coordination models for distributed computing, which separates the computation, or the task-specific programming, from the communication, or the interaction among processes. Distributed coordination models consider the need to take local decisions while still conceptualizing the effects of these actions at the global level. Typical coordination models attempt to simplify the programming task by hiding the distribution of the environment, providing operations to users which have the same effect regardless of the location of the information being acted upon. Our work distinguishes itself by seeking not to hide the inherent distribution, rather to take advantage of the distribution, enabling a richer set of operations and more flexible user interactions.

Despite this added complexity, our driving design strategy is summarized by the desire to coordinate distributed applications by thinking globally while acting locally. Put simply, the user works with an abstract global view of the system, and while all operations are issued in the same manner as local operations, their effect is defined with respect to this global view. An important feature of our abstract model is its applicability to a wide range of distributed environments (from large, wired, wide area networks to wireless, ad hoc environments) and a diversity of application domains (from data-oriented to dynamic, configuration-centered systems).

Our model, termed *global virtual data structures* (GVDS), is essentially a coordination model. For the programmer of an individual software component, GVDS manifests itself in the form of a common data structure accessible via a standard set of operations. This data structure, however, is not real but only a reflection of the global state of the system further constrained by its current configuration, i.e., by a set of rules that establish how much of the global state is actually accessible at each point in time and how much accuracy this representation carries with it. Of particular interest is the ability to define the projection of accessible data with respect to the component issuing the operations. In other words, it is possible for two connected components to have different views of the data structure depending on user-defined constraints. The choice of abstract representation and the relation between the global state and its virtual local view are the differentiating factors among different instantiations of the GVDS concept.

The paper is structured as follows. Section 1.2 aims at defining precisely, albeit informally, the notion of GVDS. Section 1.3 presents some existing models and systems that incarnate this concept. Section 1.4 highlights the main assets of GVDS. Section 1.5 discusses the design challenges and alternatives of a system that adopts a GVDS perspective and highlights the research opportunities involved. Section 1.6 elaborates further on the notion of GVDS, discussing its relationship with other approaches and systems, and identifying research challenges. Finally, Section 1.7 ends the paper with some concluding remarks.

## 1.2 CONCEPT DEFINITION

The problem of providing distributed access to information through a programmable medium can be regarded as a coordination problem. Coordination is a programming paradigm that seeks to separate the definition of components from the mechanics of interaction. In traditional models of concurrency, processes communicate with each other via messages or shared variables. The code of each component is explicit about the use of communication primitives and the components are very much aware of each other's presence. Actually, communication fails when one of the parties is missing. By contrast, coordination approaches promote a certain level of decoupling among processes and the code is usually less explicit about mechanics of the interactions. Ideally, the interactions are defined totally outside the component's code.

Coordination is typically achieved through implicit communication among the components. Linda [9] is generally credited with bringing coordination to the attention of the programming community, but other models (e.g., those based on CHAM [4]) take a similar perspective. Agents exchange information indirectly through manipulation of a shared data structure, with no need for the parties involved to know about each other. Usually, additional decoupling between the communicating parties is provided by the fact that some degree of persistency of the data structure is allowed, hence the communicating parties are freed from the constraint of being available for communication at the same time and/or at a well-known location. In traditional coordination models the data structure that represents the coordination medium is assumed to be *indivisible, persistent, and globally available* to each of the coordinated component.

Clearly, these assumptions clash with the highly dynamic scenario we are targeting in this paper. Models based on a GVDS retain a coordination perspective, fostering decoupling of behavior and interaction, but reconcile it with the requirement of dynamicity by lifting precisely the aforementioned assumptions.

In a GVDS, the data structure serving as a coordination media is distributed according to some well-defined rule among the coordinated components. Each component is then associated with a fragment of the overall data structure, that can be manipulated through the operations defined on it. Distribution may involve the schema of the data structure (e.g., in the case of a tree where subtrees are assigned to different components) or the content of the data structure (e.g., different subsets of a set). Hence, the coordination data space is no longer indivisible, rather it is, by definition, made of separate parts.

Nevertheless, the coordination space is still perceived as a single entity, since the local data owned by the components are transiently and dynamically shared under a single, global data structure—the GVDS. However, sharing occurs only among those components that are mutually reachable. By ignoring connectivity constraints, combining the data of all these separate parts forms a single, global data structure. However, because of constraints of the dynamic environment, not all parts are simultaneously available. Therefore, this global data structure is *virtual*, meaning no single component can view it as a whole, but all components know of its existence. The part of the GVDS which is concretely visible to an individual component is defined by the contents of the data owned by the individual components that are reachable. Typically, physical connectivity among components is assumed as the rule determining the maximal local view. While this a reasonable and useful assumption, and only a precondition for more elaborate rules. For instance, components belonging to the same network partition, determined according to physical connectivity in a mobile ad hoc network, may actually be considered belonging to two different partitions under a GVDS where partitions are determined according to security domains. In any case, the data structure exploited for coordination is no longer globally available, rather it is available only within a partition. Moreover, it is not persistent either, since partitions are in general allowed to change dynamically.

Interestingly, access to this shared data structure is granted through the operations defined on the local data structure. Hence, from the point of view of components, the distinction between local and global data becomes blurred, and a single way of interacting with the rest of the system is provided. Moreover, it is worth noting that the notion of GVDS defines a model of coordination that is intrinsically peer-to-peer and component-centric. In other models, the data structure that holds the information relevant to coordination is totally external to the coordinated components, whose contributed data are essentially indistinguishable. Instead, the notion of GVDS clearly separates not only the behavior of the component from the interaction, but also the data provided by the component from that present in the rest of the system.

The notion of GVDS we just defined informally can then be characterized by the following fundamental properties. A GVDS is:

- *Distributed.* Parts of the data structure are owned by different components, that store them locally.
- *Constructive.* A component's local view of the GVDS is formed through some combination of the data which is on reachable components.
- *Uniformly accessed.* All the components access the GVDS through the same set of operations.
- *Parochial.* Operations appear to be performed locally, even though they may access distant data or have global effects.

These properties are concisely summarized by the name "global virtual data structure", meaning that the union of the system-wide data forms a global data structure. Operations have potentially global effect, but no single component can access the entire structure, making the data structure virtual from the perspective of the component.

The notion of GVDS represents a meta-model for coordination. It provides a conceptual framework that can be leveraged off to define coordination models addressing specific needs. Specific instantiations of the GVDS concept may differ in several respects. For instance, not only the data structure of choice and the corresponding operations may be different, but also the rules for dividing the data structure and merging it back may vary. Appropriate selection among these and other alternatives are the key to a successful use of the GVDS concept. Moreover, the resolution of the design tradeoffs involved are likely to be the source of a new way of looking at old research problems, if not a source of new ideas.

Before examining these design alternatives and research challenges, we now turn our attention to some existing models and systems based on the notion of GVDS, to show some possible instantiations of this meta-model. We will come back to the issue of how to design a coordination model that exploits GVDS in Section 1.5.

## 1.3 GVDS INCARNATIONS

In this section, we describe a few existing systems that are based on the notion of GVDS. This description serves the purpose of making our discussion of GVDS more concrete by looking at existing models and systems, and of exemplifying some the choices that can be made when defining a specific instantiation of the GVDS meta-model.

### 1.3.1 Lime

LIME (Linda in a Mobile Environment) [14, 12, 15] is a coordination model and middleware designed to support the development of applications involving physical and logical mobility. Many of the ideas put forth in this paper are inspired by our own experience in designing and implementing LIME.

In LIME, the data structure chosen for the GVDS is a Linda-like tuple space. Tuples can be inserted in the tuple space using out, read using rd, and withdrawn using in. Tuple spaces are permanently associated to the *agents* in the system, which can be thought as components with a thread of control. Agents can be mobile, and can move across *hosts* carrying along their tuple space, called the *interface tuple space*, during migration. Hosts can be themselves mobile, and yet retain communication through wireless links.

The transient sharing provided by the GVDS perspective becomes key in rejoining physical and logical mobility under a single coordination approach. As shown in Figure 1.1, sharing occurs at two levels, based on different notions of connectivity. When two or more mobile agents are co-located on a given host, their interface tuple spaces become transiently shared. The resulting tuple space is called a *host-level tuple space*. Similarly, when two or more mobile hosts are in communication range, their host-level tuple spaces become transiently shared under a single *federated tuple space*.

The meaning of Linda operations is redefined in LIME to take into account transient sharing. Operations no longer manipulate a globally available and persistent tuple space, rather they are defined on an agent's interface tuple space. However, because of transient sharing, they effectively operate on the whole federated tuple space, provided that other agents are connected. Hence, the idea of transiently shared tuple spaces reduces the details of distribution and mobility to changes in what is perceived as a local tuple space. This view is powerful as it relieves the designer from specifically addressing configuration changes, but sometimes applications may need to address explicitly the distributed nature of data for performance or optimization reasons. For this reason, LIME extends Linda operations with location parameters, expressed in terms of agent or host identifiers, that restrict the scope of operations to a given projection of the transiently shared tuple space.

The $\mathsf{out}[\lambda](t)$ operation extends out by allowing the programmer to specify that the tuple $t$ must be placed within the tuple space of agent $\lambda$. "Misplaced" tuples, i.e., tuples destined to an agent $\lambda$ that is currently not connected, are kept in the caller's tuple space until $\lambda$ becomes part of the system. This way, the default policy of keeping the tuple in the callers context until it is withdrawn can be overridden, and more elaborate schemes for transient communication can be developed. Location parameters are also used to annotate

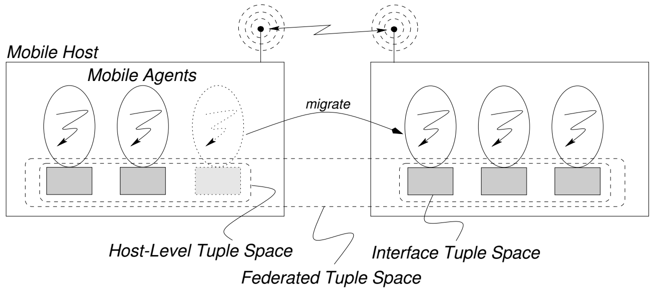


*Fig. 1.1* Combining the tuple spaces of the individual components (mobile agents) to form the federated tuple space.

the other operations to allow access to a slice of the current context. For instance, $\mathsf{rd}[\omega, \lambda](p)$ looks for tuples matching $p$ that are currently located at $\omega$ but destined to $\lambda$.

Lime extends Linda by introducing also a notion of *reaction*, motivated by the fact that, in the dynamic environment defined by mobility, reacting to changes is a big fraction of application design. A reaction $R(s,p)$ is defined by a code fragment $s$ specifying the actions to be performed when a tuple matching the pattern $p$ is found in the tuple space. Details about the semantics of reactions can be found in [14, 12]. Here, it suffices to note that two kinds of reactions are provided. Strong reactions couple in a single atomic step the detection of a tuple matching $p$ and the execution of $s$. Instead, weak reactions decouple the two by allowing execution to take place eventually after detection. Strong reactions are useful to react locally to a host, while weak reactions are suitable for use across hosts, and hence on the federated tuple space.

The Lime model has been embodied in a middleware available as open source at `lime.sourceforge.net`). We have successfully built a variety of applications with Lime [15], ranging from a data-centered multi-player jigsaw assembly program to a system-configuration centered spatial game involving teams of players moving through physical space and interacting when connectivity is enabled. Lime is also being exploited in an industrial collaboration in the context of automotive applications.

### 1.3.2 PeerWare

PeerWare [8], under development at Politecnico di Milano, is a middleware that aims at providing core support for systems exploiting a peer-to-peer architecture, hence including mobile systems. While this goal is very similar to the one of Lime, the two systems differ in the choices made while instantiating in their own model the GVDS meta-model.

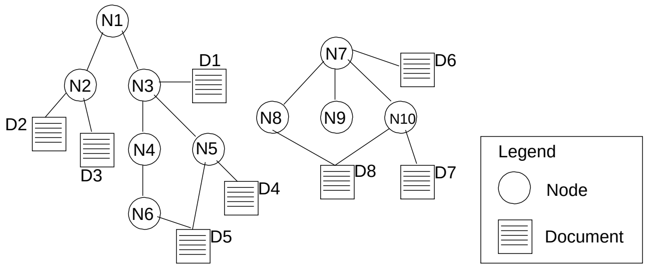


*Fig. 1.2* The data structure provided by PEERWARE.

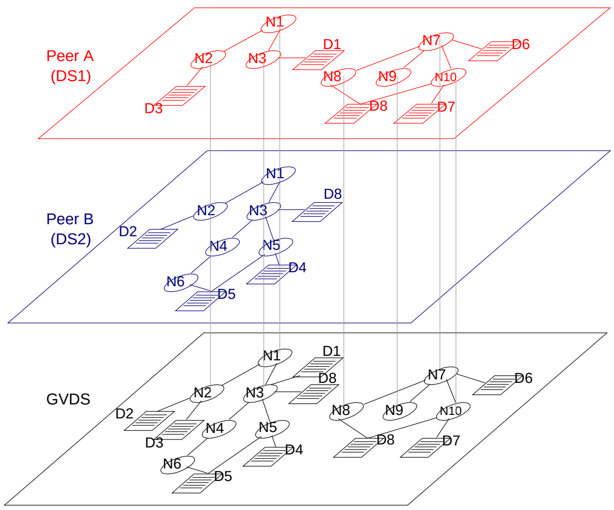


*Fig. 1.3* Building the GVDS in PEERWARE.

The data structure chosen in PEERWARE is a tree or, more precisely, a forest of trees. Each component of the system, called *peer* in PEERWARE, hosts several trees with distinct roots. The *nodes* of each tree (see Figure 1.2) are essentially containers for *documents* holding the actual data belonging to the application context. When connectivity among peers is established, the GVDS that is dynamically reconstructed is defined as follows. All the *homologous* nodes, i.e., nodes with the same name and holding the same position in trees belonging to different peers, are represented by a node in the GVDS having the same name and position as the concrete ones. The content of such a node in the GVDS is the union of the documents contained in each homologous node. A pictorial representation of this process is provided in Figure 1.3.

PEERWARE distinguishes between the local data structure and the global one, by providing two different sets of operations. The rationale behind this choice is to make the distinction between local and global access, as well as its implications, explicit to the programmer. This is a different approach with respect to LIME, where the GVDS subsumes the local data structures, and syntactic means are used to restrict scope.

Hence, a set of operations is defined only on the local data structure. These operations are concerned with local data modification, such as creation and removal of nodes and documents. Notably, PEERWARE does not provide any feature for modifying remote data, while this is available in LIME in the form of out[$\lambda$]. In addition, a publish primitive is provided that, similarly to publish-subscribe systems [16], allows the generation of an event occurring on a document or node.

Another set of operations is defined both on the local data structure and the GVDS. The operations allowed on the GVDS are very limited in number, but rather expressive in power. The basic access primitive is execute, which accepts as parameters a set of *filters* and an *action*. The filters are expressions that allow one to restrict the set of documents and nodes (collectively referred to as *items*) the execute operates on. In turn, the action specifies the sequence of operations that must be performed on such items before being returned to the caller. The simplest action is clearly an empty one. In this case, the effect of the action is simply to return the set of items selected by the filters, unchanged. In general, however, the programmer has great flexibility in defining the behavior of actions, thus effectively redefining the semantics of access to the data structure. For instance, actions may process further the filtered items (e.g., by performing additional filtering or changing the items' properties) and may also access again the data structure (e.g., to remove the selected items). Essentially, execute provides a core mechanism to disseminate, among the currently connected peers, a request for local execution of application code accessing the local data structure. The execution of a single action is atomic on each receiving peer (i.e., actions coming from different peers do not interfere), but not across several peers.

Besides execute, PEERWARE provides also a subscribe primitive, whose semantics is similar to that of publish-subscribe systems, but operates on the *state* of the system, rather than on generic application events. Thus, subscribe allows a peer to register interest for events occurring on a subset of the data structure content, as specified by a filter. Finally, the last primitive provided is executeAndSubscribe which, as the name suggests, is essentially a combination of the previous ones. This primitive essentially allows the programmer to "hook on" a given set of data by performing an execute and, in the same atomic step, a subscribe. This allows the programmer to implement schemes with a consistency that is stronger than the one that could be achieved by using execute and subscribe separately, since this latter solution would allow events occurring in between the two operations to be lost.

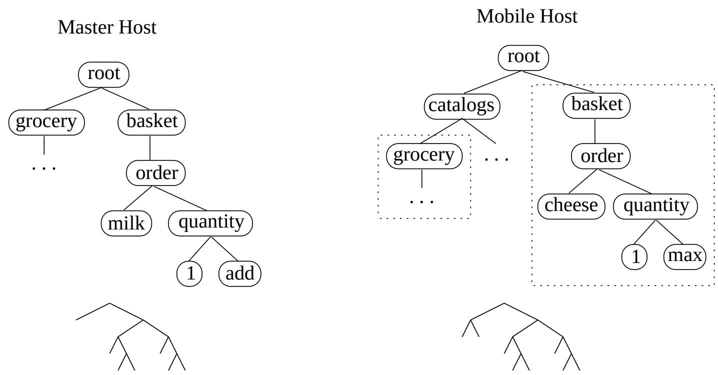


*Fig. 1.4* Two disconnected XMIDDLE hosts and their XML tree view of the data. The mobile host mounts the directories for the catalog and the basket (outlined with dotted lines), and makes updates while offline (adding an order). After reconnection of the hosts, the views of the trees are updated, and form the abstract view shown below each XML tree. Note the asymmetry between the master host's data structure and that of the mobile host.

PEERWARE is currently being exploited as the middleware layer in the MOTION[1] project, funded by the European Union. The goal of the project is to build a framework of teamwork services supporting cooperation among geographically distributed and mobile users in an enterprise-wide infrastructure.

### 1.3.3 XMiddle

The final example we consider is the XML-based middleware XMIDDLE [11] under development at University College London. The main principles behind this work are simple connectivity rules and powerful, high-level data operations. In XMIDDLE, the data structure is a set of XML documents whose content can individually be viewed as trees. Documents are combined by mounting one tree at a well defined point in another tree, thus essentially embedding one local document inside another or indicating a mount point for a remote document, much the way remote file systems are mounted in operating systems. Figure 1.4 shows a simple example of two hosts, where the mobile host mounts two subtrees of the master. The resulting local data structures are neither symmetric, nor are connections transitively shared.

Two of the more powerful features of XMIDDLE are its support for data replication and offline operations. While connected, the data of two components are kept synchronized by pushing updates along the pairwise connections.

[1] MObile Teamwork Infrastructure for Organisations Networking, IST-1999-11400, `www.motion.softeco.it`.

When disconnected, changes are still allowed to the replicated portions of the tree, but may also lead to inconsistencies of the data. These inconsistencies are reconciled upon reconnection using a set of application defined reconciliation rules which are embedded in the XML document itself. In the example of Figure 1.4, the reconciliation rules for orders are to add or take the max of the quantity values. Another option is to leave both conflicting elements and notify the user.

The XMIDDLE system has been successfully used to implement an electronic shopping system where multiple users in the same family can mount the same catalog of items and collectively build a shopping list in a distributed fashion. The replication allows changes in both the catalog and shopping list to propagate among members of the family and the reconciliation rules address any conflicts which may arise from multiple individuals adding the same item to the list.

## 1.4 THE ASSETS OF A GVDS APPROACH

A natural question to pose at this point is what are the key advantages from a coordination model adopting a GVDS-based perspective. From the illustration of the GVDS concept and the brief overview of the models exploiting it, it appears that in coordination models exploiting the notion of GVDS, the association between the coordination context contributed by a given component and the component itself is made explicit. Hence, the information provided by a given component is no longer dispersed in some coordination "soup" external to the components. Moreover, this association between a component and the data it is willing to share for the sake of coordination models more naturally what is locally available to the component and what instead needs to be looked for in the rest of the system.

Hence, GVDS fosters a coordination style where:

- the resulting model is truly distributed, and the coordination focus is shifted from an external coordination data space back to the components;

- the sharp decoupling between communication and behavior, typical of coordination, is retained: a globally available coordination data space is still available, only its contents are now automatically and dynamically reconfigured according to changes in the system;

- coordination is achieved through local actions that have a global effect.

The conjecture is that these characteristics are going to simplify the task of building—and reasoning about—applications that are built out of autonomous components whose relationships are dynamically and frequently reconfigured. Clearly, a lot about the truth of this latter statement depends on how the notion of GVDS is instantiated into a specific model. The alternatives and tradeoffs available are the subject of the next section.

## 1.5 FROM THE META-MODEL TO MODELS AND SYSTEMS

As we mentioned earlier, we regard the concept of GVDS defined informally in Section 1.2 as a meta-model of coordination. The quick survey of models and systems we carried out in Section 1.3 seems to suggest that the design space available is rather broad. In this section we examine more closely the space of design alternatives, by setting ourselves free from the details concerned with specific systems. The outcome of this investigation is a preliminary set of issues that we deem relevant in the design of a GVDS-based model and/or system, based on our own experience to date. This list of issues is by no means meant to be exhaustive. As a matter of fact, in defining the content of this section we are essentially defining the agenda for a new area of research on coordination models.

*Data structure.* Obviously, the choice of the data structure is the first one that needs to be faced. This choice is fundamental in all GVDS-based models, and is usually determined by the overall requirements or design criteria for the model. For instance, the tree structure of PEERWARE naturally suited the requirements of collaborative applications. In XMIDDLE, the choice was driven by the need to merge XML files.

However, the choice of the data structure has an impact that goes well beyond the suitability of the data structure for the particular requirements at hand, affecting non-functional dimensions like performance or security. For instance, LIME exploits a data structure that is essentially flat, although multiple federated tuple spaces can coexist, distinct through names. Instead, PEERWARE may leverage off the hierarchical structure of trees to reduce the amount of traffic generated by queries. Similar considerations apply if security is at stake. A tree structure naturally lends to a security architecture where each tree node is a protection domain, potentially inheriting (and modifying) access rights from the containing tree. Hence, different parts of the same data structure could be made visible to different components. In LIME, protection domains would need to be mapped on tuple spaces, thus providing a single, all-or-nothing mechanism for access control.

*Operations.* A data structure is intimately connected with the operations that are allowed on it, which constitute another fundamental decision in defining a GVDS-based model. Query operations, allowing to retrieve one or more elements from the data structure, are often a natural choice, in that they simplify the task of querying at once the local data associated with a potentially large number of distributed components. Nevertheless, query operations (like an rd or in in a tuple space, or the selection of an element in a matrix) define a "pull" approach, where a component must proactively retrieve the data it needs. In scenarios where the system configuration is continuously changing, and where the benefits of a GVDS approach are more evident, it may be preferable to complement, or even substitute, proactive operations with reactive operations, thus fostering a "push" approach where the burden of delivering the relevant data to the component is placed on the system. This observation is actually what motivates the presence of reactions in LIME and of publish-subscribe features in PEERWARE. Our experience in building applications with LIME [15] shows how the reactive features of LIME are used much more than the proactive ones in the development of applications for mobile computing.

Clearly, other approaches are possible, like primitives to navigate through graph-like data structures distributed across multiple sites, similarly to what is provided by distributed database management systems supporting the network data model or the object-oriented data model.

*Local vs. global.* Another relevant design issue about operations is whether the distinction between local and global operations should be apparent from the model (and hence the system) or not. In a "pure" GVDS, there is no distinction between what is local and what is global. Operations are simply performed through a handle to the data structure, and their effect is determined by the input parameters and by the current configuration of the system. However, such a radical location-transparent approach may have significant drawbacks. As noted in literature [17], masking the distinction between local and remote may have a detrimental effect on the overall performance of the system. In LIME, the design tradeoff between location-aware and location-transparent data access is tackled by annotating operations with location parameters that allow the programmer to restrict the scope of queries. In PEERWARE, the distinction is made even more evident by providing two separate interfaces for accessing the local data structure and the global one. The problem is not raised only by operations for data retrieval. For instance, the ability to ship a tuple to a different agent provided by the LIME out[$\lambda$] forces the system to buffer tuples and ship them around as soon as the target component shows up. This feature complicates slightly the implementation of the system. At the same time, it is one of the most expressive features of LIME, raising considerably the programming abstraction level. In PEERWARE, writing data to a remote component is assumed to be handled externally to the model, thus greatly simplifying the implementation at the price of expressiveness and conceptual uniformity.

*Rules for dynamic sharing.* The last topic is strongly related to another cornerstone of the design of a GVDS model, namely, the definition of the rules governing transient sharing. Three issues are relevant: the conditions under which transient sharing is enabled, the rules defining the shape of the global data structure resulting from sharing, and the amount of computation that is possibly carried out when a change in the configuration, enabling or disabling the sharing of one or more components, occurs.

The first issue is typically tied to some notion of connectivity. For instance, in LIME two notions of connectivity are exploited. Two mobile agents are "connected" (and hence share their tuple spaces) when they are co-located on a host; two mobile hosts are "connected" when they are in range. Clearly, the two can be combined. Weaker notions of connectivity are also possible, e.g., encompassing the ability to communicate through a sequence of hosts that are not all connected at the same time, along the lines of [7]. XMIDDLE enables transient sharing by considering only pairwise connectivity.

While physical connectivity is an intuitive way to define the condition under which sharing may occur, this notion is more properly regarded as a baseline for more sophisticated triggering rules. For instance, transient sharing may be subject not only to the ability to communicate, but also to the satisfaction of security requirements, e.g., belonging to the same administrative domain. A generalized notion of connectivity can be defined, that extends the physical one with application-dependent conditions.

The second aspect related with the details of transient sharing is the definition of the rules determining the perceived content of the shared structure. Clearly, this issue is intimately tied to the specifics of the data structure. The systems we surveyed in Section 1.3 define the shared data structure either as a union or as a composition of the local ones. Names play a key role in defining how the shared data structure is constructed. Both LIME and PEERWARE enable sharing of data on a per-name basis, but while in LIME names are singletons, in PEERWARE they have a structure reflecting the tree. While this fact may appear as a petty technicality at the model level, allowing structured names may have a significant impact at the implementation level, as we discussed earlier. In contrast, XMIDDLE exploits trees in a different way, by defining the shared tree through composition of the local subtrees.

Another relevant issue is the degree of symmetry of sharing. In LIME and PEERWARE sharing is symmetric, i.e., all the components involved in the transient sharing perceive the same data structure. Instead, in XMIDDLE sharing is asymmetric, since each of the components involved in a pairwise sharing may compose the tree in a different way. In addition, physical or logical constraints may introduce asymmetry in the definition of transient sharing. For instance, unidirectional wireless links may enable only one of the components engaged in a pairwise connection to output data in the data structure belonging to the other, but not vice versa. Analogously, security constraints may allow one of the components to access data from the other, but not vice versa.

The third and last aspect directly related to transient sharing is the amount of computation occurring when sharing is triggered. The dominant tradeoff involved is likely to be the one between expressive power and performance. As we mentioned earlier, in PEERWARE no user-specified computation is triggered upon engagement (although control information about the tree structure needs to be exchanged behind the scenes in the implementation). In LIME, the provision of primitives that allow transparent relocation of tuples forces the model to take into account the reconciliation of the misplaced tuples, i.e., tuples generated when the target component was not yet part of the system. In XMIDDLE, sharing involves replication, to increase data availability during disconnection. However, replication is carried out entirely by the system and is not made explicit at the model level.

*Other Issues.* This last remark raises a number of interesting questions. For instance, is it possible and reasonable to specify a replication behavior by exploiting the very primitives provided by the model at hand? For instance, preliminary experience with LIME showed that its reactive features enable the creation of a higher-level veneer on top of LIME dealing with replication. The concept could be pushed even further, allowing the programmer to specify several veneers that enhance the system by providing some amount of self-reconfiguration or reflection.

Of course, in designing a GVDS-based model of coordination, a number of other well-known concerns must be tackled, like the choice between synchronous and asynchronous primitives and the degree of atomicity they exhibit. The latter one is likely to be the most critical in defining a successful model. Ideally, operations carried out on the GVDS should have the same semantics as the same operations carried out on a data structure that is local to the caller. In practice, however, this is often not achievable. Striking a good balance between expressiveness and scalability is particularly delicate in this case.

## 1.6 DISCUSSION

In this section, we elaborate further on the concept of GVDS we put forth in this paper, by highlight some of its relationships with other models and systems, and by identifying open research issues and challenges.

### 1.6.1 Intellectual Connections

Coordination models have been categorized along the lines of data-oriented or process-oriented [13] depending on whether it is changes in the data which drive the application, or events raised by the components which are the focal point. The GVDS meta-model encompasses both with the main emphasis placed on the data structure and its basic operations. However, as has been noted, reactive constructs similar to the events of a process-driven model are sometime key to application development in dynamic environments and are clearly supported in the GVDS model. As we have revisited the Linda coordination model to form LIME, other coordination models may be extended to the unique environment of GVDS applications by studying them in this new light.

It is interesting to note how the GVDS concept to some extent subsumes more traditional models. For instance, the provision of location-aware primitives may subsume the traditional client-server model as a special case of GVDS. In LIME, operations annotated with location information allow for the retrieval of data from a specific host or agent, in a way that closely resembles a client-server interaction. Also, while a GVDS encompasses a coordination data space that can be very dynamic and characterized by transient sharing, in doing so it includes also scenarios where the configuration of the system is partially, or completely, static. Hence, for instance, the transiently shared tuple space defined in LIME "defaults" to a Linda tuple space in the case where all agents and hosts are stationary and connected. This fact can be exploited in practice in applications that need to have some data held persistently: at deployment time, some of the nodes may be designated to be stationary and hence providing a persistent base for the rest of the system, as they will always be able to share their content with any non-stationary component in range.

Also, some systems that are usually not labeled as GVDS-based exploit an architecture that can be framed under this concept. This is the case of systems supporting peer-to-peer file sharing over the Internet, such as Napster [1], Gnutella [2], or Freenet [3]. In these systems, users share the content of a fraction of their file system with other users. Typically, the content is available only when the user is connected, although caching schemes (such as those exploited by Freenet) allow the creation of distributed replicas. To each user, the set of files present in the repository of connected users looks effectively like a global repository—which can be regarded as a GVDS. The users are allowed to perform queries to search for a given file in this global structure, and requests are automatically routed to the users in the system. In these applications, the scope of queries is typically determined by considerations related to performance rather than expressiveness: queries typically have a time-to-live that effectively restricts their extent.

The notion of GVDS is somehow reminiscent of the notion of *distributed shared memory*, in which multiple processes map the same piece of memory and interact through basic read, write and lock operations. The typical emphasis of these systems is on supporting heterogeneous architectures and multiple languages. Recent research has focused on increasing performance by allowing the application programmer to tune the level of consistency required of the data [18, 6], where increased coherence requirements result in higher overhead as compared to weaker coherence models. In general, distributed shared memory systems assume a well known mechanism to find the server of a memory segment, and further assume connectivity between the client and server for the duration of the interaction. Thus, they do not seamlessly address many of the issues of dynamic environments we describe as typical GVDS scenarios.

Some systems have begun to address mobility as one step toward supporting dynamic environments. For example, the Coda file system [10] allows users to cache files and work offline, performing reconciliation when the user reconnects to the master file server. The Deno distributed object system [5] supports object replication and a distributed voting scheme for managing disconnected object updates. While these systems do support mobility, they do not address all of the issues of a GVDS, and ignore high level coordination operations over the managed data. Still, some of the mechanisms exploited by these systems are likely to inspire the implementation of some features of GVDS-based systems.

### 1.6.2 Research Challenges

We see GVDS as a meta-model, a conceptual framework guiding the definition of novel coordination models and systems. As we already mentioned, the notion of GVDS we shaped in this paper raises a number of open issues, that can be regarded as defining a novel line of research. They include:

- *What is a good balance to strike among design alternatives?* In Section 1.5 we highlighted some of the dimensions determining the move from the GVDS meta-model into a coordination model, and how these decisions affect the tradeoffs between expressiveness and performance. Clearly, the identification of these tradeoffs is a key to the success of GVDS-based models, and especially systems.

- *What are the implications of* GVDS *for security?* The architecture implied by a GVDS is intrinsically radically decentralized and open. At least at the model level, no central point of control is assumed, and in addition components may come and go at will. Finally, the characteristics of implicit communication, typical of coordination approaches, make it more difficult to define security policies based on authorization. Hence, it reasonable to ask whether the background of security techniques, e.g., involving access control or authentication, are sufficient to deal with this scenario, or whether new models (e.g., of trust) are needed. Interestingly, the problem as defined is analogous to the problem of ensuring security and trust in a peer-to-peer network.

- *Can the choice of the data structure be separated from the distribution and communication aspects?* Is it reasonable to define a sort of GVDS "schema language", analogous to what is done in DBMS systems, allowing the programmer to define the shape of the distributed data structure, and keep this definition decoupled from the details of distribution? And, if yes, to what extent?

- *Is there a theory of* GVDS*?* Is it possible to capture the notion of GVDS in a formal framework, possibly separating the data issues (e.g., the "schema" of the data structure) from the distribution and communication issues? And, once this is done, how can this framework be (re)used to refine the meta-model into an actual model?

- *What is the impact of* GVDS *on formal reasoning and verification?* We hinted earlier at the conjecture that the notion of GVDS, by shifting the focus from the coordination infrastructure back to the coordinated components, provides a more natural way to prove properties about the overall system. However, this is only a conjecture, and needs to be verified on the field. Moreover, what kind of formalism or notation is appropriate to reason about GVDS-based systems?

## 1.7 CONCLUSIONS

Distributed systems exhibit increasing degrees of decentralization, autonomy, and dynamic reconfiguration. In this paper, we defined the concept of *global virtual data structure*, a novel coordination approach that fosters a component-centered perspective by privileging distributed interaction through local actions that have a global effect. By shifting the focus from the coordination infrastructure back to the components, and yet retaining a coordination perspective that separates behavior from communication, it is our contention that the notion of GVDS is the right match for the fluid environment defined by current trends in distributed computing.

It is our hope that the ideas presented in this paper will stimulate interest among researchers and cooperation in finding answers to the issues above, thus leading to the definition of a new generation of coordination models and middleware targeting highly dynamic environments.

*Acknowledgments* This paper is based upon work supported in part by the National Science Foundation (NSF) under grants NSF CCR-9970939 and EIA-0080124, and by the Italian government under the MIUR project SAHARA.

# *References*


1. `http://www.napster.com`.
2. `http://www.gnutella.org`.
3. `http://freenet.sourceforge.net`.
4. G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96:217–248, 1992.
5. U. Cetintemel, P.J. Keleher, and M. J. Franklin. Support for speculative update propagation and mobility in deno. In *Proceedings of the IEEE Intl. Conf. on Distributed Computing Systems (ICDCS)*, Mesa, AZ, USA, 2001.
6. D. Chen, C. Tang, X. Chen, S. Dwarkadas, and M.L. Scott. Beyond S-DSM: Shared state for distributed system. In *Proceedings of the International Conference on Parallel Processing (ICPP)*, August 2002.
7. X. Chen and A. Murphy. Enabling disconnected transitive communicaiton in mobile ad hoc networks. In *Workshop on Principles of Mobile Computing, colocated with PODC'01*, pages 21–27, Newport, RI (USA), August 2001.
8. G. Cugola and G.P. Picco. PEERWARE: Core middleware support for peer-to-peer and mobile systems. Technical report submitted for publication. Available at `www.elet.polimi.it/~picco`.

9. D. Gelernter. Generative Communication in Linda. *ACM Computing Surveys*, 7(1):80–112, Jan. 1985.

10. J.J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. *ACM Trans. on Computer Systems*, 10(1):3–25, 1992.

11. C. Mascolo, L. Capra, S. Zachariadis, and W. Emmerich. XMIDDLE: A data-sharing middleware for mobile computing. *Int. Journal on Wireless Personal Communications*, April 2002.

12. A.L. Murphy, G.P. Picco, and G.-C. Roman. LIME: A Middleware for Physical and Logical Mobility. In F. Golshani, P. Dasgupta, and W. Zhao, editors, *Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS-21)*, pages 524–533, May 2001.

13. G.A. Papadopoulos and F. Arbab. Coordination models and languages. In *761*, page 55. Centrum voor Wiskunde en Informatica (CWI), ISSN 1386-369X, 31 1998.

14. G.P. Picco, A.L. Murphy, and G.-C. Roman. LIME: Linda Meets Mobility. In D. Garlan, editor, *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*, pages 368–377, Los Angeles, CA, USA, May 1999. ACM Press.

15. G.P. Picco, A.L. Murphy, and G.-C. Roman. Developing Mobile Computing Applications with LIME. In M. Jazayeri and A. Wolf, editors, *Proceedings of the 22th International Conference on Software Engineering (ICSE 2000)*, pages 766–769, Limerick (Ireland), June 2000. Formal research demo.

16. D.S. Rosenblum and A.L. Wolf. A Design Framework for Internet-Scale Event Observation and Notification. In *Proc. of the 6th European Software Engineering Conf. held jointly with the 5th Symp. on the Foundations of Software Engineering (ESEC/FSE97)*, LNCS 1301, Zurich (Switzerland), September 1997. Springer.

17. 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*, pages 49–66. Springer, April 1997.

18. H. Yu and A. Vahdat. The costs and limits of availability for replicated services. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (SOSP)*, October 2001.