WASHINGTON UNIVERSITY
SEVER INSTITUTE OF TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE

ENABLING THE RAPID DEVELOPMENT OF DEPENDABLE APPLICATIONS IN THE MOBILE ENVIRONMENT

by

Amy L. Murphy

Prepared under the direction of Professor Gruia-Catalin Roman

---

A dissertation presented to the Sever Institute of
Washington University in partial fulfillment
of the requirements for the degree of

Doctor of Science

August, 2000

Saint Louis, Missouri

WASHINGTON UNIVERSITY
SEVER INSTITUTE OF TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE

ABSTRACT

ENABLING THE RAPID DEVELOPMENT OF DEPENDABLE APPLICATIONS IN THE MOBILE ENVIRONMENT

by Amy L. Murphy

ADVISOR: Professor Gruia-Catalin Roman

August, 2000
Saint Louis, Missouri

Recent technological trends in the miniaturization of computing devices and the availability of inexpensive wireless communication have led to an expansion of effort in mobile computing. In this environment, interactions are transient, computations become highly decoupled and rely on weak forms of data consistency, disconnections are frequent and unpredictable, and component behavior is sensitive to changes in location, context, quality of service, or administrative domain. The goal of this thesis is to develop an environment that will facilitate rapid development of dependable mobile applications executing in both physical and logical mobile environments. Our focus is the development of abstractions that simplify the programming task. We present two design strategies to achieve this goal, the first of which focuses on algorithm development to support abstractions for inherently difficult problems in mobility while the second describes a technique to develop high level coordination constructs to support transient interactions among components.

to my family

Contents

List of Figures

Acknowledgments

As with any non-trivial effort, this dissertation was not completed in isolation, but with the support of many people. First, I would like to thank my advisor Catalin Roman. For five years he has expected much from me, pushing me to do my best and giving just the right balance between encouragement and red ink. For teaching me the art of research and instilling in me a desire to continue, I thank you.

Next I thank Gian Pietro Picco. The many tireless hours of Java programming and the innumerable dinners around the city changed my graduate experience for the better. I hope this fruitful collaboration will continue.

George Varghese participated in the algorithm development presented in this thesis and was an inspiring classroom teacher. I learned much from all my interactions with him. I also wish to thank my doctoral committee, namely Sally Goldman, Mark Jakiela, Catalin Roman, Tuomas Sandholm, and George Varghese. This work was financially supported by Washington University and NSF grants CCR-9624815 and CCR-9970939.

Others who contributed to the technical content of this thesis include Jason Ginchereau, Brian Mesh, and Bryan Payne, three talented undergraduate students I had the honor of working with.

While pursuing this degree, many, many hours were spent in the computer science department. I honestly believe that one of the greatest assets of this department the support that it gives us both in official matters and in personal sanity. For both, I wish to thank the office staff, especially Myrna Harbison, Peggy Fuller, Jean Grothe, and Sharon Matlock, as well as the other students who make Friday afternoon Happy Hours successful, especially Tilman Wolf who started it all. Special thanks also go out to Girish Chandranmenon, Del Hart, and Pete McCann.

Outside the university, I have relied on the support of a variety of people. To all the members of the Traditional Karate Institute and especially my sensei, Mr. Campbell, I owe a debt of personal gratitude. Through interactions with them I have found a balance in my life and leave St. Louis a better person. Thanks also to my friends, Sinique Betancourt and Juan Caicedo. I could not have made it through the interview season without their support.

On a personal note, part of this thesis goes to Jay Thompson and Colleen Dodd, two friends whose short lives touched mine.

Finally, my thanks go to my family, to whom this thesis is dedicated. My parents, Tom and Kay, have supported me throughout my life, encouraging me to achieve everything I was capable of and not to give up. My brother, Kevin, has become a close friend and confidant, giving me both an occasional kick in the behind or a shoulder to lean on, whichever was most appropriate. To my sister-in-law, Karla, and my grandmother, Connie Murphy, I am thankful for their company through this experience.

Thank you!

Amy L. Murphy

Washington University in Saint Louis
August 2000

Chapter 1

Introducing Mobility

Mobility entails the study of systems in which components change location, in a voluntary or involuntary manner, and move across a space that may be defined to be either logical or physical. By definition, systems of mobile components are distributed systems, and while distributed computing has been carefully studied for decades, mobility poses new challenges that have not previously been addressed.

The development of compact computing devices such as notebook computers and personal digital assistants allow people to carry computational power with them as they change their physical location in space. The number of such components is steadily increasing. One goal, referred to as ubiquitous computing [86], is for these devices to become seamlessly integrated into the environment until we are no longer explicitly aware of their presence, much the way that the electric motor exists in the world today. Part of enabling this vision is coordinating the actions of these devices, most likely through wireless mediums such as radio or infrared.

Logical mobility, or the movement of code and state through a fixed infrastructure of servers, is emerging as a powerful design abstraction for distributed systems. The pervasiveness of the Java programming language and its portability have led to a wealth of mobile agent systems. Demonstration purpose applications built on top of these systems range from logical agents managing physical objects in a kitchen [56] to agents managing the placement of a video conferencing server to minimize bandwidth consumption [10].

Developing applications in the mobile environment is a difficult task. Many existing applications restrict themselves to addressing a specific aspect of mobility in a highly specialized environment, such as disconnected operation in the Coda filesystem [40] or using agents to perform remote queries on a database as in the Oracle Agent System [63]. Development of these systems requires highly specialized knowledge of low level networking as well as details of the application domain.

Our goal is to enable the development of more general applications by providing flexible abstractions which can be applied in a variety of settings. Our success in this area comes from an integrated research approach that involved analyzing the needs of mobile applications, formulating models to describe the key concepts of our approaches, specifying formally these models, implementing the abstractions, and returning to the development of applications to evaluate our results.

1.1 Contributions

In this thesis we present two new design strategies which describe the development process for abstractions in the mobile environment, specifically identifying useful abstractions and guiding the implementation of these abstractions. Application of these strategies has produced a number of application programmer interfaces (APIs) for abstractions of several complex problems of mobility.

The first design strategy applies to the base station model of mobility. This encompasses nomadic computing (e.g., physically mobile components moving in a cellular network) and mobile agent computing (e.g., logically mobile code and state moving among a set of hosts). Abstractly, a base station system can be modeled as a graph of nodes (representing the cells or hosts) and channels providing the communication among these fixed components. We model mobile hosts that are in communication with a base station and mobile agents that are executing on a host as part of the state of the node and model migration as the traversal of a link by the mobile component.

- The goal of our first strategy is to develop new algorithms to support abstractions for difficult problems in the base station model of mobile computing. The approach defines how algorithms from standard distributed computing can be adapted directly to the mobile environment by noting the similarities between the abstract mobile model and the network model of distributed computing, and by *treating the mobile unit as a persistent message moving in the network*. Successful application of this strategy involves matching algorithms from distributed computing to the abstractions they can implement in the mobile computing environment.

  This approach has two major benefits. First, by grounding new developments on well understood algorithms from distributed computing, we are able to directly apply the proven properties of the distributed algorithms to the mobile environment and therefore increase the dependability of our algorithms with little effort. Second, this straightforward approach leads to the rapid development of algorithms in the mobile environment which solve difficult problems in mobile computing. These algorithms can then serve as a foundation for extensions which address additional problems of mobility.

- The specific abstraction we focus on is reliable message delivery to both physical and logical mobile components. The Chandy-Lamport distributed snapshot yields a mobile algorithm for both reliable unicast and multicast. The Dijkstra-Scholten diffusing computation is adapted to track the movements of mobile components and a reliable message delivery scheme is build on top of the tracking mechanism.

  In both cases, the developed algorithms describe the underlying implementation to the abstraction of reliable message delivery which is presented to application programmers.

Our next contribution comes from focusing on ad hoc networks where no infrastructure exists to support communication among physically mobile hosts. Instead, hosts communicate directly with one another and the distance between hosts determines connectivity. A system is typically composed of multiple communities of hosts with connectivity available within the community but no communication from one community to another. Changes in connectivity and corresponding changes in available resources make this a challenging environment for application design.

- Our strategy describes the design for new high-level coordination abstractions, generically referred to as *global virtual data structures*. The abstraction presented to the application programmer is simply a local data structure whose content changes according to connectivity. Conceptually each component stores a piece of a global data structure, when components are within communication range these pieces are transiently shared and accessible to other components. Interaction with the data structure occurs exclusively by executing operations on the local data structure, however, transient sharing enables transparent interaction with other mobile components.

  One of the features of this approach is its ability to facilitate the development of applications which never explicitly access remote data. We term this *context-transparent interaction*, where the data is part of the current context in which a mobile component finds itself. The distribution and changes to the data structure are hidden from the application programmer by the abstraction itself. Alternately, *context-aware interaction* can easily be provided as an extension to the basic model by explicitly introducing the notion of location.

- We have successfully applied this strategy in the development of LIME which provides the simple mobile coordination abstraction through transiently shared Linda tuple spaces, enabling application programmers to clearly separate the concerns of computation from the communication among hosts. The implementation of LIME in the form of middleware presents the same interface and semantics as the model, simplifying the implementation process. Mobile application developers utilizing the LIME concepts

need not concern themselves with any of the low level details of communication or changing connections, as all of these are handled within the implementation of the LIME middleware.

Work with the LIME system has shown it to be a clean conceptual tool for introducing programmers to the concepts of mobility. Several applications have been built on top of the middleware, demonstrating its usefulness in a variety of mobility scenarios.

1.2 Dissertation Overview

This thesis is organized into two main parts, framed by an introduction to the current trends in mobility in Chapter 2 and conclusions and directions for future work in Chapter 12. The two main parts introduce and explore the new design strategies for abstraction development outlined in the previous section of this chapter.

Algorithms to support abstractions. Chapters 3 through 6 deal with algorithm development to support abstractions in the base station model of mobility. Chapter 3 defines in detail our design strategy and then introduces reliable message delivery as the problem we address in the following chapters. Chapters 4 to 6 describe the message delivery algorithms derived by applying this strategy.

Chapters 4 and 5 both develop algorithms based on Chandy-Lamport snapshots. Chapter 4 focuses on physical mobility, explaining the necessary details to implement the algorithm in a network similar to the cellular telephone infrastructure. Chapter 5 approaches the algorithm development from the perspective of logical mobility and presents several extensions to the basic algorithm to make it more flexible and applicable to a variety of logical mobility scenarios.

Chapter 6 shows how the Dijkstra-Scholten diffusing computation model can be adapted to track the movement of mobile units. It then builds reliable message delivery abstractions on top of the tracking functionality.

Global virtual data structures. Chapters 7 through 11 address the development of high-level abstractions for ad hoc mobility. Chapter 7 begins by presenting the design strategy for global virtual data structures while Chapters 8 through 11 present the development of LIME, a single global virtual data structure based on the Linda tuple space model. LIME (Linda in a Mobile Environment) serves as a proof of concept for the global virtual data structure design strategy.

The development of LIME demonstrates our integrated design strategy by showing the informal model definition, Chapter 8, formal specification in Mobile UNITY, Chapter 9, and implementation as middleware, Chapter 10. Our experience has been very promising

and the LIME model has been applied to a variety of mobile applications. Chapter 11 presents several extensions to the basic model which further broaden its applicability.

Chapter 2

Perspective on Mobility

Mobile computing reflects a prevailing societal and technological trend towards ubiquitous access to computational and communication resources. The communication industry is actively pursuing mobility opportunities by investing in new wireless technologies (e.g., wireless LAN [48]), by cooperating in the establishment of interoperability standards (e.g., IEEE 802.11b High Rate standard), and by forming powerful consortia. The IETF Mobile Ad Hoc Networks (manet) Working Group [44] is considering standardization efforts based on IP technology. The list of consortia includes: 3G.IP [2] which focuses on high bandwidth wireless technology using W-CDMA; Bluetooth [25] which uses frequency hopping and is designed to provide low-cost support for small groups of co-located devices; and others which promise home networks (via a Shared Wireless Access Protocol or SWAP) or Web page delivery to low-bandwidth devices (via a Wireless Application Protocol or WAP).

Of course, wireless communication is not the same thing as delivery of data to a mobile unit. The latter presupposes the ability to find the current location of the unit and to continue to send data as the unit moves from one place to another. Cellular phone systems accomplish this through a combination of broadcasts (to notify the unit about the incoming call) and hand-off protocols (to maintain the connection in the face of movement). In the Internet setting, special protocols, such as Mobile IP [65], have been designed to enable packet delivery while a mobile unit is away from its home base. The next version of IP (IPv6) is anticipated to provide still better support for transparent packet delivery to mobile units away from their home networks [66]. Efforts are also under way to respond to the special needs of ad hoc networks. Rapidly changing topology renders impractical many well-established routing strategies such as link-state and distance-vector. New variants are being proposed and evaluated. They include Temporally Ordered Routing Algorithm (TORA) [64], Dynamic Source Routing (DSR) [15], and Ad hoc On demand Distance Vector (AODV) [67]. A common feature among all three is their reactive nature, i.e., routing information is built in response to the demand to communicate among specific hosts. The

provision of multicast services is still another area receiving much attention in the mobile setting.

The impact of mobility on systems research and development is manifest mostly at the algorithm and middleware levels. Algorithms from the area of traditional computing do not address the complexities of the distributed mobile environment. New algorithms must be developed to provide the same functionality available in distributed environment to the mobile environment. Additionally, new styles of algorithm development must be explored to exploit the unique characteristics of the mobile environment. Middleware is emerging as one of the most fertile areas of systems research in mobility as it allows developers to take advantage of the deployed software infrastructure while providing clean high-level programming abstractions in languages already available today. Middleware hides the protocol layer but makes explicit the key concepts involved in the development of mobile applications, e.g., the management of location data, event notification, quality of service assessment, adaptability, etc. Middleware can be specialized for logical or physical mobility or may combine the two in a single cohesive package.

This chapter presents our perspective on the current status of mobility research, exploring the application characteristics that shape research directions (Section 2.1), models which focus on the fundamental concepts of mobility and lay the foundation for development efforts (Section 2.2), algorithm research which centers on discovering difficult problems that are frequently encountered during design and on formulating and analyzing basic solutions to such problems (Section 2.3), and finally on the kinds of middleware that integrate these concepts and ultimately help to realize mobile applications (Section 2.4).

2.1 Applications

Current trends in computing technology include the manufacturing of increasingly smaller, more powerful, and more portable computing devices. A glance around any airport terminal shows that notebook computers are pervasive among business travelers. Common usage of these computers is for tasks that require no interaction with outside resources, also referred to as disconnected operation. The Coda filesystem [40], for instance, supports this by allowing users to specify a set of files to be hoarded on disconnection. On reconnection, any update conflicts within this set of files must be explicitly handled by the user.

Another common task for mobile users is access to remote resources such as the Internet or company database systems. Recently Palm Computing released the Palm VII personal digital assistant with built in wireless capabilities for accessing the Internet [1]. By simply raising an external antenna, a connection is made to the nationwide private 3Com network. No wireless ethernet or cellular modem is necessary. Cellular telephones with limited Internet access are also becoming commonplace. Although the user interface

is limited by screen size and resolution, the ability to access information is key. To access a corporate database from a mobile device, Oracle provides support for three common database operations [63]. First, users are able to manage a database remotely. Second, partial database replication allows mobile devices to carry a piece of the data and have constant access, possibly out of date with the original. Third, by using a mobile agent paradigm, mobile users can pose queries while disconnected, an agent collects these queries and when a connection is available to the database the agent moves to the server. The user can then disconnect while the queries are being processed, and when the connection is reestablished, the agent moves back to the mobile host where the results are accessible.

Smaller devices, such as active badges [85], provide several interesting application scenarios. If a badge is associated with an individual, when that user moves to a new room, the environment in that room can automatically adjust to predefined user preferences. Alternately, badges can be attached to equipment and be used to locate those objects as they are moved into different laboratories throughout an office complex. These systems rely on an infrastructure to track and make available such information.

Another mobility scenario, different from the client-server model, describes a group of individuals coordinating on a project in an environment without network support. For example, laptops carried to a program committee meeting should be able to interact to construct a short-lived network during a plenary session or allow division into multiple independent networks to support individual working groups. Similarly, the participants in a conference may form an ad hoc group with the need to share information such as business cards, schedules, session notes, etc.

Global positioning systems are becoming popular devices in many automobiles and, while the design of these devices does not require access to remote data, once wireless access becomes readily available, new kinds of applications may be considered. For example, cars moving in opposite directions could share information about road conditions on recently traveled roads. Those moving in the same direction may be able to coordinate for extended periods of time on a variety of tasks. Another interesting possibility is the placement of information kiosks at key places throughout a city or countryside. These kiosks could provide location specific information such as tourist information, available to the automobiles over a low power wireless link while themselves being connected to a fixed network.

Specialized computing devices can contribute to the emergence of yet other interesting applications. In a teaching laboratory, multiple devices can coordinate to assist a student with an experiment by providing instructions, performing computations, and collecting and displaying information from multiple instruments in a single place. At a smaller level weaving processors into clothing enables wearable computing, and thus a more natural way to carry and access computation power while moving. Tiny devices, such as those proposed by the Dust project [38], have potential as sensing devices spread throughout a

room or desktop. Although some of these scenarios may appear to be the stuff of science fiction, both society and technology are moving in this direction. Of course many technical challenges must be addressed before they become reality and they place new requirements on the enabling technologies.

One of the first concerns a developer must address is defining the user perception of the application with respect to the degree to which mobility is exposed at the application level. If the user is location-aware, one must face the related question of how the user is made aware of this property. In a mobile filesystem such as Coda, the user explicitly specifies information to be hoarded on disconnection and explicitly resolves conflicts on reconnection. Alternately, when accessing location dependent information, such as a query for *local* resources in Odyssey [80], the user should be able to make a generalized query and have the system perform the specific resolution. In robot scenarios with autonomous movement, it may be reasonable to hide the absolute location and expose only relative positions among components, thus allowing each component to think of itself as the center of the universe.

Variability in quality of service parameters is another factor that may contribute to the user's perception of location and movement. As a user moves, possible bandwidth degradation requires some form of adaptation in the behavior of the application. Odyssey provides a nice illustration of this feature by allowing control over the fidelity of data on the fly. In a video session, for example, frame rate and frame quality provide two tuning parameters. In general, applications must offer a variety of adaptive parameters that affect the presentation style and make other adjustments reflecting different levels of knowledge about the overall configuration and available information.

Similar situations are encountered when entering and leaving administrative domains especially when they have diverse levels of security. From the user perspective, the amount of personal information to be shared must vary depending on context. The ability to express and alter both individual security policies and security demands of a domain is important to many mobile applications. This ties in the issue of open environments in which applications on mobile components must be able to interact with other mobile components about which no prior knowledge exists and, similarly, with applications never before encountered. While it is possible to prohibit such interaction entirely, it is much more preferable to provide mechanisms that are capable of discovering beneficial modes of interaction in new circumstances. The ability to adapt to an open environment must be weighed against the associated costs. Openness, for instance, may compromise security while excessive generality may require too many resources.

Assessing the capabilities of the environment is also important for effective performance of an application. Mobile devices range from relatively high-power portable notebook computers to low-power personal digital assistants with limited display and computation;

communication capabilities may include powerful base stations enabling full connectivity among all mobile components or may be limited to ad hoc environments in which repartitioning and changes in connectivity pattern are frequent. Finally, the speed and pattern of movement can also exhibit great variability. This variety of environmental conditions makes application development challenging, but the ability to accommodate increases the potential degree of penetration by mobile applications in the society at large.

2.2 Models

In this section, we focus our attention on models that entail an explicit notion of space and components that move through it. A component may be either a code fragment that is given the ability to roam the address spaces of a computer network or a physical device moving through the real world. Abstraction often blurs the distinction between logical and physical mobility thus allowing us to formally specify and reason about arbitrary components moving across a broad range of conceivable spaces. By and large, models tend to subsume physical into logical mobility, as the latter exhibits characteristics that have no direct physical counterparts, e.g., the ability to spawn remotely a new mobile unit. As one might expect, models vary greatly in the way they answer questions such as who is allowed to move, where it can go, and how context changes caused by movement are managed. The choice of unit of mobility is central to any model of mobility since it shapes to a large extent the way in which the other two questions are addressed. The treatment of location is indicative of the model's perception of space. The handling of contextual changes reflects the component's perception of the coordination mechanisms that tie components into a system. Ultimately, the assumptions and choices a model makes relative to these particular concerns differentiate it from other models of mobility.

The *unit of mobility* represents the smallest component in the system that is allowed to move. A typical choice is to make the unit of mobility coincide with the unit of execution. This approach fits well a mobile device that moves in physical space as well as a mobile agent that migrates among network hosts. The vast majority of models share this choice, e.g., higher-order extensions of π-calculus [79], Ambients [17], and Mobile UNITY [50], to name only a few. However, the reality of middleware and applications for logical mobility suggests that finer-grained units, weaker than full-fledged execution units, are pervasive in every day practice. Among various design paradigms for code mobility [28], for instance, code on demand is probably the most widely used at this time. In this style of logical mobility, the unit of execution does not actually move. Its behavior is dynamically augmented by foreign code that becomes linked when a particular trigger condition occurs. Evidently, this fine-grained perspective provides a new degree of freedom in describing how a distributed system gets reconfigured by exploiting mobility among its components. The unit of execution is no

longer tied to a host and neither are the unit's constituents tied to it. From this perspective, the ability to move a unit of execution as a whole (commonly called a mobile agent) may be regarded as a special case of a more general framework in which single code fragments and/or their corresponding states can change location. Not surprisingly, this notion has a direct counterpart in physical mobility, where the *alter ego* of code and state are the applications and the data they use on some device.

So far, despite its theoretical and practical relevance, fine-grained mobility received only limited attention in the formal models community. A commonly used approach is to view the code and the state associated with an executing unit as degenerate cases of the unit, e.g., state may be carried by a unit in which the code is missing or has no effect on the computation. Because code and state are not treated as first-class units of mobility, this approach is not sufficiently expressive, e.g., it cannot capture code assemblies still under construction. To our knowledge, the only model that addresses fine-grained logical mobility explicitly is the one presented by Mascolo, et. al [46]. In that work, this idea is pushed to an extreme by investigating a model where the unit of mobility is as small as a single variable or statement in a programming language. This radical perspective, readily encompassing more common situations where the unit of mobility is as coarse as a class or an object, is expected to provide new insights in the design of programming languages that foster high degrees of reconfigurability.

Location identifies the position of a mobile unit in space. This view of location is tied to the intuitive notion of mobility and distinguishes models of interest to us in this work from other highly dynamic models that equate mobility with a more general notion of change. In π-calculus [54], for instance, there is no notion of location built into the model, and yet the structure of the system can change dynamically. Processes exchange communication channels (represented by names) and, in some extensions [79], even processes. This provides the expressive power needed to describe systems whose structure evolves but fails to treat location as a first class concept. It is important for a model to be capable of dealing with location throughout the software development lifecycle, starting from the definition of the environment where mobility occurs, through designing and reasoning about a mobile application, and down to the tools provided to programmers. For this reason, numerous researchers are investigating calculi [17, 27, 60, 6] which extend π-calculus with some notion of location and also approaches that are not based on process algebras but on state transitions and logic [50].

The type of location is affected by the choice of unit of mobility. For instance, location could be represented by Cartesian coordinates for a mobile device, by a host address for a mobile agent, or by a process identifier in the case of a code fragment. For this reason, some models avoid specifying the details of location altogether and focus on how to effect movement and on how to detect and handle location changes and their consequences.

This is precisely the case of Mobile UNITY [75], where location is modeled explicitly as a distinguished variable that belongs to the state of a mobile component. Changes in its value correspond to changes in the position of the component. Other models start with different assumptions and impose a predefined structure on the space (typically hierarchical). Such is the case with MobiS [45] where locations are nested spaces containing tuples, which in turn may contain code as well as data with migration taking place upwards in the hierarchy of spaces. Ambients [17] provides a richer model where locations are ambients containing processes or other ambients. The boundary of an ambient, however, can be reconfigured dynamically to change the overall system structure. These latter approaches combine the notion of location, which only abstracts the notion of position in space, and the notion of context described below.

Context represents the peculiar and novel aspect of mobile computing, to the point that some researchers characterize mobility as "context-aware computing." The context of a mobile unit is determined by its current location which, in turn, defines the environment where the computation associated with the unit is performed. The context may include resources, services, as well as other components of the system. Conventional computing tends to foster a static notion of context, where changes are absent, small, or predictable. In a mobile setting, changes in location may lead to sudden changes in the context a unit perceives. Moreover, these changes are likely to be abrupt and unpredictable. A handheld wireless device carried across the floors of an office building has access to different resources (e.g., printers or directory information) on each floor; a mobile agent migrating on different servers may use different sets of services on each of them; in a fine-grained model, a statement with free identifiers may be bound to different variable instances each time it is linked into a different unit of execution.

The ability to detect whether the context has changed, e.g., whether a given unit is now part of the context, is often a precondition for the ability to react to such a change. Timely reaction is often a requirement, because some actions may be enabled for a limited time after an event occurs (e.g., after two mobile agents become co-located, or after the noise level on a wireless link goes beyond a given threshold). Letting the component interested in handling an event probe for its occurrence proactively may not be acceptable, due to the potentially high number of conditions to be verified and of parties involved. Instead, a reactive approach may be more appropriate, allowing the interested component to provide a specification of the event condition and of the actions that should handle of it. The portion of context considered for evaluating the enabling condition and the degree of reactivity (i.e., the degree of atomicity of the reaction with respect to the event occurrence) is what discriminates among these models. At one extreme, event-based systems [76, 21] consider only the *occurrence* of events that are filtered through a given specification. The corresponding reaction is guaranteed to execute eventually. At the other extreme, there

are models [69] where the enabling condition is a particular *state* of the system (i.e., of the context), and the reaction to a state change is completed before any other state change is performed. The question about what degree of atomicity and style of reaction is more reasonable for mobility is still an open one in the research community.

The manner in which we deal with the context is greatly affected by whether it is distributed or localized. In logical mobility, for instance, the context is typically localized within the boundary of a host. A code fragment is moved onto a different host in order to exploit some resource or service provided locally. Network communication is exploited only during the migration process. In contrast, physical mobility seems to require a distributed notion of context. Mobile hosts construct the context through wireless communication and the resources and services that contribute to defining the context are provided by the other components and are accessed in a distributed fashion. While it might be reasonable to look at both logical and physical mobility under the same modeling lens, their nature is intrinsically different. The extent to which it is reasonable to treat both forms of mobility as one remains an open question that demands careful consideration. Where is the threshold separating the realm of logical mobility from the one of physical mobility?

Formal models enable precise description of the semantics of existing languages and systems and formal reasoning about their correctness. In the novel field of mobility, models appear to assume an increased level of significance. Models must be used as intellectual tools to uncover the conceptual grounds of mobility and, armed with the power of abstraction, highlight parallels and differences among the various forms of mobility as well as conventional distributed computing. Mobility may even throw a different light on the role of reasoning and correctness proofs. Reasoning about locations could be exploited not only to determine the correctness of a system, but also to optimize its configuration. For instance, by analyzing formally the patterns of migration of a group of mobile agents, proper placement of code could be determined in advance in order to minimize remote dynamic linking.

In the past the impact of models was felt most directly through the development of new languages and associated tools. This is no longer true today. Novel mobile applications with great intellectual and commercial success are likely to benefit much more from the development of appropriate middleware than from any advances in language technologies. As such, we see middleware as the conduit through which research on models for mobile computing will exercise its greatest influence of software engineering practice.

2.3 Algorithms

The algorithms we employ reflect the assumptions we make about the underlying systems. As the shift to mobile computing is taking place, it is natural to expect that new algorithms would need to be developed. Location changes, frequent disconnections, resource variability, power limitations, communication constraints, dynamic changes in the connectivity pattern, all contribute to a demand for new algorithm design strategies. Given the diversity of mobile systems, the range of options is enormous and indeed research on mobile algorithms spans a broad spectrum. Some of the work, however, reflects what one might consider short-term technological limitations that will eventually be overcome or do not enjoy universal applicability. Power consumption falls in this category. Research on energy efficient algorithms is interesting but not necessarily fundamental. Even the concern with quality of service, particularly in multimedia applications, is probably not of the essence. Such research fits best in the category of system infrastructure design rather than algorithms. Of course, some specific elements of these problem areas may survive the process of abstraction and make their way into fundamental algorithms. For instance, algorithms involving asymmetric communication channels may end up being studied because it takes less power to listen to a signal than to broadcast it. Ultimately, it is the treatment of space and coordination again that shape the landscape of mobile algorithms.

The ability of a mobile component to move through space requires new algorithms to control and manage information about its location and that of other components. Spatial knowledge is important in many applications involving independent purposeful movement, cooperative activities, or involuntary movement. In settings where components have control over their own location, forming and maintaining geometrical shapes proves useful. For example in the task of robot exploration of an open field for unexploded ordnance [52], the ability to follow a leader through a known safe path is one useful application of a group movement strategy. Similarly, clustering around or encircling an object can be used to identify an object's boundaries, protect other group members from danger, or protect the object itself. Both of these are examples of geometrical global invariants which have been specified and achieved by describing algorithms to effect local, independent movements.

Less specifically tied to geometry is the necessity in a sparse network to maintain connectivity among all components. Maximizing functions such as total covered area or distance between the farthest components guide individual movements while keeping the group goals. In highly populated networks with possibly millions of nodes, connectivity is almost guaranteed, but organization is critical. Hierarchical structures that mimic the organization of the human body from cells into organs, and organs into a functioning whole offer immediate applications to scoping issues, communication capability, and possible movement patterns [20]. While these examples tend to highlight opportunities in ad hoc mobility, nomadic computing and logical mobility also demand the ability to leverage off knowledge

about component locations. This is usually accomplished by keeping track of where mobile units are located on location servers that are queried for up to date information. Variations in the assumptions made about the number and placement of servers, and in update and query procedures are likely to lead to a rich set of algorithmic studies of practical significance [71]. Other sources of potentially interesting algorithms may be the result of exploiting metrics over the space and relative distances. Distance information, for instance, is commonly utilized in route optimization.

Other aspects of mobility entail more of a coordination perspective on algorithm development. Mobile components often work together to perform collective tasks which need to be monitored and controlled. Although many of these task oriented algorithms have been solved for traditional distributed computing, the reality of voluntary disconnection of mobile components demands the redesign of these algorithms with mobility in mind. For example, a traditional distributed snapshot relies on the availability of communication between neighboring nodes. In a mobile system, not only do neighbor sets change, but disconnections often prohibit communication with some components for extended periods of time. Global checkpointing [4], causal event ordering [70], leader election, and termination detection [47] are other examples of algorithms which are meaningful to mobile distributed processing and must be revisited to account for disconnections. Transactions involving mobile components must be reexamined to address the movement of components, location dependent queries, and data delivery to future locations [24].

Strategies used in the development of algorithms for mobility vary widely. In the presence of a fixed support infrastructure, the most common strategy is to push computation and communication away from the mobile components and wireless links and onto the infrastructure [9]. For example, in the case of checkpointing, while storage on physically mobile devices may be limited and even inaccessible due to disconnection, the state of the mobile components can be stored at a fixed node and communicated to other nodes along a fixed, higher bandwidth communication medium.

When a network infrastructure does not exist or the network has no inherent structure of its own, an artificial structure can be imposed over the components, grouping them for communication concerns or creating a hierarchy for management. The ability to maintain and rely on this structure depends on the patterns of movement of the mobile components. Different situations call for a variety of patterns ranging from general connectivity constraints such as eventual transitive communication between all pairs of components, to physical movement characteristics such as a predetermined path or direction of movement. These general patterns can be exploited by any fundamental algorithms.

Other strategies try to exploit the advantages of known algorithm design paradigms and re-adjust them for mobility. For example, randomized algorithms can be used to generate probabilistic results when component reconnection is uncertain. Alternately, if connectivity is guaranteed to be reestablished, disconnection may be viewed in a manner similar to a network fault. In this case, fault tolerant algorithms and self-stabilizing techniques can be applied. Epidemic algorithms may prove to be the key to distributing information to components when connectivity is available.

The availability of a standard and well-understood set of algorithms, supported through formal models and middleware, is a measure of the field's level of maturity but also an asset for the developers. Experience with distributed computing has shown that problems that may appear to be simple have very subtle solutions prone to error. This is likely to continue to be the case in the area of mobile computing.

2.4 Middleware

Middleware supports the software development task by enhancing the level of abstraction associated with the programming effort. Middleware adds mechanisms and services that are much more specialized than those provided by the operating system, within the context of established languages without modifying their syntax and semantics. Recent years have seen a flurry of middleware developments for distributed systems. It is then reasonable to expect that a new generation of middleware specialized for mobility will follow suit. Despite the similarities between logical and physical mobility, research on middleware tends to treat the two forms of mobility very differently. Besides factors that have to do with separation of the related research communities, a compelling reason for this situation rests with the different roles logical and physical mobility play with respect to application development.

Logical mobility is essentially a new *design* tool for the developers of distributed applications. The ability to reconfigure dynamically the binding between hosts and application components provides additional flexibility and, under given conditions, improved bandwidth utilization. On the other hand, physical mobility poses new *requirements* for distributed applications, by defining a very challenging target execution environment. These different roles are mirrored in the characteristics of the corresponding middleware. Middleware for logical mobility is centered around new abstractions that enable code and state relocation, whereas middleware for physical mobility often tends to minimize differences with respect to non-mobile middleware, by relegating, as much as possible, the differences into the underlying runtime support. In the remainder of this section, we report about the state of the art in the field and highlight some of the open research issues.

Traditionally, middleware for *physical mobility* has been application centered. For instance, the Bayou [84] system provided the core functionality needed to build database

applications that can handle disconnection through reconciliation and data hoarding. This approach was symptomatic of an interpretation of mobile computing as a very specialized and rare form of computing that could be accommodated with application specific support and by exposing as little as possible of its characteristics to the user. Although this view may still hold true for many applications, with the rise of mobility as the base of future computing, general purpose middleware becomes more of a necessity. Hiding mobility becomes more difficult, if at all meaningful, and a new core of abstractions that extend distributed middleware with support for mobility must be devised.

With regard to physical mobility, the challenge for mobile middleware is to devise mechanisms and constructs to allow detection of changes in location, to specify what belongs to the context of the computation, to relate changes in location to context modifications, and to determine how the computation itself is affected by changes in the context.

Many issues related to tracking the dynamics of location and context require tight interaction with the underlying operating system and device. Of particular significance is the availability of mechanisms that enable detection of connectivity, of variations in the quality of service of communication, of the appearance of new mobile hosts within communication range, and of battery power status. All these considerations are of paramount importance for the core of mobile applications and constitute a major point of departure from distributed computing, where the need for primitives that dig so deeply into the underlying machine is more the exception than the rule. For the time being, availability of such mechanisms and primitives is heavily constrained by the lack of appropriate programming interfaces at the underlying wireless device level.

Similar constraints exist for detecting changes in the location of a mobile device. Location management is a novel and interesting requirement of mobile middleware, one that is likely to become more and more important as experience with a wide range of truly mobile applications becomes available. Managing the location of a mobile host may assume many different nuances. It is desirable to have mechanisms that allow the programmer to determine where the host currently is and to maintain a history of the visited locations. Furthermore, it is natural to think about their integration with mechanisms that allow reactive modification of the context. It should be possible, for instance, to have location changes trigger specialized computations in order to reconcile data or to determine the role the mobile host must assume upon entering a new administrative domain. Location may be absolute or relative to that of other neighbors. In both cases, primitives are needed to define a notion of space and the associated notions of position and distance. It should be noted that relative locations pose demanding requirements on location management, as they presuppose the ability to track continuously the movements of a given set of mobile hosts. In a world of autonomous mobile entities, tracking services may become fundamental to enable cooperation when decoupled computation is not possible.

A different set of issues that middleware for mobility must consider are actually well known in distributed computing, but need to be redefined in the new context. Service lookup belongs to this category. In distributed computing, the problem of discovering available services is often solved by forcing service providers to register with a server. In many popular architectures, e.g., Jini [49], the server is essentially centralized and more sophisticated schemes that take into account mobility being hand-coded on top of Jini. Instead, mobility scenarios often require constructs that allow the programmer to perform service lookup without any knowledge about the configuration of the current context.

A well known alternative to centralized service discovery is the use of an event dispatching mechanism, which provides also for reactive capabilities. Although most commercially available event dispatching systems are indeed centralized, there is a significant body of research on distributed events growing both in industry and academia. Mobility complicates further the picture of dispatching events in a distributed fashion. Hierarchical configurations of dispatchers, like those proposed in [21], are no longer suitable when confronted with the fluid configuration of mobile hosts. Disconnection translates to the impossibility of delivering an event to a subscriber for a given time interval, thus raising the problem of how to reconcile the view of the subscriber upon reconnection. If events generated during disconnection are discarded, the subscriber may miss relevant events; if, on the other hand, events are queued and transmitted to the subscriber, the overhead of this bulk transmission may be prohibitive. Finally, delivering an event to a mobile unit may become a problem itself, even in presence of a fault-free network. Other issues that need to be revisited for mobility include mechanisms for security and access control, as well as support for transactions.

Early approaches to *logical mobility* started out as what nowadays would be called middleware. For instance, the REV system [82] provided an extended version of remote procedure call where the client could specify the code of the procedure to be executed, and the Emerald [37] system provided an object-oriented layer on top of an operating system that handled transparent object migration. By contrast, recent approaches to logical mobility focused initially on the design of new languages or on the extension of already existing languages with primitives expressly conceived for handling logical mobility. This is the case of Telescript [87] and Facile [41], among the others. The creation of a brand new language was justified by the absence, in traditional languages, of hooks into the runtime support to enable relocation of code and state. The fact that today these systems, that nevertheless influenced heavily subsequent developments, are relegated to a totally marginal role is a symptom of the current trend dominated by systems based on the Java language. Java provides some of the runtime hooks, notably the ability to reprogram dynamic linking, combined with a degree of portability and security that, although not optimal, is still higher than what many other platforms provide.

However, current middleware for logical mobility is falling short of expectations. On one hand, there are mobile agent systems, i.e., systems providing as the main abstraction a unit of mobility coincident with the unit of execution. Despite the initial excitement about this notion of mobile agents, technology did not meet the expectations. Most existing systems provide basically the same abstractions with the same limitations. In many respects, rather than building mobile agent systems as a facility that can interoperate with mainstream distributed middleware, many systems reimplement support mechanisms like events, dispatching, directory services, transactions, messaging. This could be justified by the challenges logical mobility poses on the implementation of such services (very similar to those present in physical mobility). Yet, in many instances the tough problems are left unsolved and the mobility of agents is curtailed (impacting negatively on the very reason for the existence of mobile agent system). The key observation that logical mobility is just another design tool, and it should be made available to the programmer in combination, and not in alternative, to distributed middleware is not acknowledged by these systems. Notable exceptions, representative of very different design strategies, are Voyager [39], a distributed middleware that provides object mobility as one of the many features of a full-fledged platform, and μCODE [68], a minimal, lightweight support for mobile code providing abstractions that enable the relocation of any mixture of code and state, thus encompassing also the notion of mobile agent.

At the other extreme there are systems that exploit logical mobility by choosing a unit of mobility smaller than the unit of execution, typically the Java class. In contrast to the notion of mobile agent, this finer-grained logical mobility is finding its way into popular distributed middleware like Java/RMI and Jini [49]. In these systems, logical mobility is exploited for the sake of improved flexibility. While the benefits of static type checking are retained through the notion of a mutually agreed service interface between client and server, the implementation of such service may be changed dynamically by using subtyping and code mobility. The problem with this form of middleware, however, is that it exploits only a minimal fraction of the power provided by logical mobility. Only the code on demand paradigm [28] is supported; other paradigms, like mobile agent or remote evaluation, that have been proven useful [10], must be hand-coded. No relocation of state is allowed, except for the ability to copy the entire closure of an object that is being passed as a parameter of a remote invocation.

Contrary to popular belief, building support for relocation of code and state is not a monumental endeavor, especially using the Java language which already provides many of the necessary building blocks. The real issue is the design of the constructs that are made available to the programmer and their underlying conceptual model. Researchers have only begun to scratch the surface of discovering the level of flexibility provided by logical mobility. The next challenge is to provide support for varying grains of mobility,

mechanisms allowing different rebinding strategies, and different architectural styles for relocation—all in a single, uniform programming interface.

Coordination, by abstracting away from the behavior of the mobile units and focusing on high level communication protocols, may provide a way to rejoin the logical and physical mobility in a single, uniform framework. In particular, systems based on tuple spaces provide a suitable and direct abstraction for an unstructured (and thus general) representation of the context where a mobile computation is performed. This way, coordination middleware does not impose specific data structures to represent the constituent of the context, instead, it provides basic mechanisms that rule the access, modification, and consistency of such data structures.

It is interesting to note that the advantages of coordinating distributed components through a Linda-like model are well recognized also by the industry, where companies like IBM and Sun compete with their Java-based implementations of a tuple space called T Spaces [33] and JavaSpaces [35], respectively. The two systems have slightly different implementations but a very similar philosophy. The degree of distribution is still extremely limited, as these systems essentially provide remote access to a centralized tuple space which acts as a tuple server providing shared access to clients. No support for disconnection is provided, and the presence of a centralized, well-known server almost instantly rules out applicability to an ad hoc network setting. Logical mobility is more of an hindrance than an asset for these systems, as downloading of tuple code is not handled automatically. The Linda model is coupled with a primitive event system that augments the expressive power, but its policies and guarantees are not easily adaptable by the user in need of specialized and reactive cooperating behavior.

Some academic systems push further the coordination perspective by providing systems that tie together Linda with mobility. For instance, the MARS and TuCSoN systems [16] provide the notion of a reactive tuple space. Changes in the tuple space content trigger reactions that modify the tuple space.

By and large, these coordination approaches tend to adopt a coarse grain perspective, providing support for coordination of mobile agents and mobile hosts. Nevertheless, mobile code could be exploited as a means to modify dynamically the behavior of such mobile components, e.g., by employing schemes where tuples actually contain code, as in the MobiS model [45], and providing reactive rules for their dynamic linking and execution.

Independently of the slant towards coordination, however, middleware systems are ultimately generated through a design mindset and, as such, they are the result of compromises resulting from proper evaluation of tradeoffs. A first relevant tradeoff is about how much power should be put in the hands of the programmer. Middleware platforms nowadays tend to provide extremely rich interfaces, i.e., powerful and expressive constructs, at the cost of increased complexity, poor conceptual cohesion, and high performance overhead.

The other tradeoff is between horizontal coverage for a broad range of scenarios and configurations (e.g., a platform providing abstractions that span from the fixed to the ad hoc setting) in contrast with a vertical coverage of specific scenarios (e.g., providing support only for palmtop devices in a nomadic setting). Identification of the proper balance between these opposing forces, combined with effective and validated support to real world applications, is what will ultimately determine the emergence of a new generation of mobile middleware.

2.5 Concluding Remarks

Mobility is rapidly emerging as an important research area with concerns ranging from low-level media issues to high-level application concerns. In this chapter we focused on the more abstract issues, outlining several current research projects and presenting several perspectives to guide continued research in areas we believe will be critical to enabling future developments in mobility. It is also clear that a successful research effort should at least be aware of, if not inclusive of both systems and theory.

Chapter 3

Message as a Mobile Unit: A design strategy for the development of base station mobility abstractions

The typical model of distributed computing treats a network as a graph in which vertices represent processing nodes and edges denote communication channels. Faults may render parts of the network inoperational either temporarily or permanently. Despite faults, the overall structure is considered to be static. One way to introduce mobility in a similar model is to treat the nodes as mobile support centers coordinating multiple radio base stations. Mobile units are allowed only to connect and disconnect from mobile support centers through communication with the base stations. The result is a fixed core of static nodes and a fluid fringe consisting of mobile units. The obvious connection to traditional distributed computing and an extensive investment in current network technologies helped this model, commonly referred to as nomadic computing, become dominant in mobile computing today [8, 36]. This model also applies to logical mobility where the nodes represent the serves willing to host mobile agents and the edges represent channels along which mobile agents man migrate.

While mobility demands a novel perspective on distributed computing, it is equally imperative to investigate any essential features of mobility that are already present in our current view of distributed computing. In fact, the term "mobility" has been used already to refer to processes that migrate across the network [31, 72]. We suggest yet another way of thinking about mobility in the context of the traditional fixed graph structure. The basic idea is to treat mobile units as roving messages which preserve their identity as they travel across the network. Many practical applications may be suitable for this kind of

modeling. A cellular telephone, for instance, travels from one cell to the next. While operating inside one cell, the phone may be viewed as residing at a node inside the support network; similarly, the handover protocol (triggered by the detection of signal degradation) may be modeled as the traversal of a channel between two nodes representing the individual cells. Voice transmissions among two phones are also modeled as messages. Logical mobility is naturally represented by this model with the agents taking the role of persistent messages moving among hosts.

Our interest in this model rests with *its ability to facilitate the application of established distributed algorithms to problems in mobile computing.* The resulting algorithms can then be used to support the design of higher level abstractions, which can be provided to the application programmer to aide in application development.

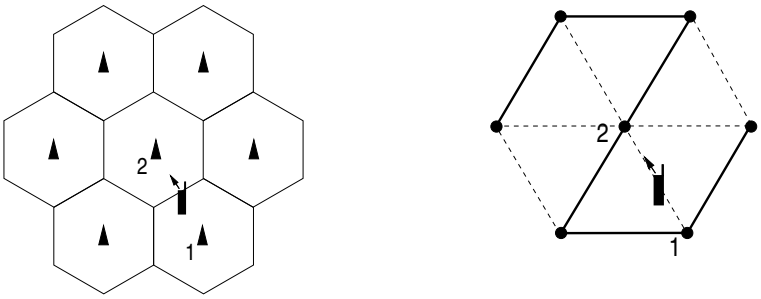
By viewing the mobility infrastructure as a graph of nodes and channels, we have a model similar to the model of networked distributed system where the nodes are processes and the communication links between them are channels. An immediate observation is that many common distributed algorithms can be executed in the mobile setting with few changes. It has been noted that because of the unique properties of mobility such as limited bandwidth and disconnection, it is not practical to do such a direct translation. We propose a fundamentally different direction. Previous work has focused on moving distributed algorithms into the mobile domain to solve the same types of problems, while we adapt distributed algorithms to solve issues unique to the mobile environment. The execution of the algorithm does not change, but the semantics of the algorithm do.

In the following chapters we make this strategy concrete by showing how it can be applied to the development of algorithms for reliable message delivery to mobile units in both the physical nomadic computing model and logical mobility. Specifically this is accomplished by adapting the Chandy-Lamport distributed snapshot algorithm [18] and the Dijkstra-Scholten model for diffusing computations [23] to mobility. The remainder of this chapter describes the details of the nomadic and logical mobility models (Section 3.1), provides a precise definition and the motivation for providing algorithms for reliable message delivery (Section 3.2), and shows several previous message delivery approaches and where many fail to provide reliability (Section 3.3).

3.1 Model

Nomadic mobility. The cellular telephone design provides the foundation for the model of nomadic mobility we adopt. Figure 3.1(a) shows a typical cellular telephone model with a single mobile support center (MSC) in each cell. The MSC is responsible for communication with the mobile units within its region and serves as a manager for handover requests when a mobile moves between MSCs. Figure 3.1(b) shows how the cellular telephone model is

transformed into a graph of nodes and channels where the nodes represent the individual cells and the channels represent the ability of a mobile unit to move from one cell to another. For simplicity, we assume that the resulting network is connected, in other words, a path exists between every pair of nodes.



(a) Cellular system (b) Graph model

Figure 3.1: (a) Cellular system with one MSC per cell. All MSCs are assumed to be connected by a wired network. (b) Abstract model of a cellular system, as a graph of nodes and channels. Solid lines form a spanning tree.

We also assume that a mobile unit moving between two MSCs can be modeled as being on a channel identical to messages in transit. In this manner, we no longer differentiate between physical movement and wired communication. It is reasonable to ask what happens when messages and mobile units are found on the same channel. We make the assumption that all channels preserve message ordering, i.e., they are FIFO channels. The details of how to achieve this in a real setting are discussed in Chapter 4.

Logical mobility. The logical mobility model we work with is the typical network graph where the nodes represent the servers willing to host agents and the edges represent directional, FIFO channels along which agents can migrate and messages can be passed. The FIFO assumption will be discussed in more detail in Chapter 5.

We assume a connected network graph (i.e., a path exists between every pair of nodes), but not necessarily fully connected (i.e., a channel does not necessarily exist between each pair of nodes). Communication between each pair of nodes is assumed to be standard, bounded asynchronous message passing. In a typical IP network, all nodes are logically connected directly. However, this is not always the case at the application level, as shown in Figure 3.2. There, a set of subnetworks are connected to one another through an IP network, but an agent can enter or leave a subnetwork only by passing through a gateway server, e.g., because of security reasons.

We also assume that the mobile agent server keeps track of which agents it is currently hosting, and that it provides some basic mechanism to deliver a message to an agent,

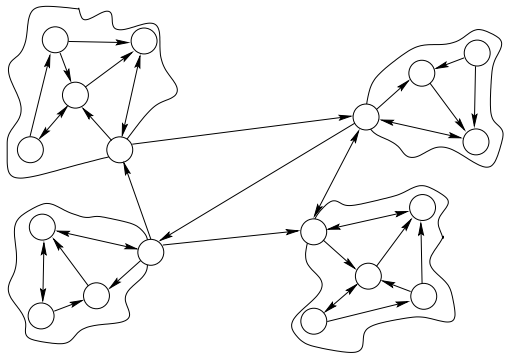


Figure 3.2: A connected network with connected subnetworks. Agents can enter and leave the subnetworks only by going through the gateway servers.

e.g., by invoking a method of the agent object. Finally, we assume that every agent has a single, globally unique identifier, which can be used to direct a message to the agent. These latter assumptions are are already satisfied by the majority of mobile agent platforms.

3.2 Problem Definition: Message Delivery

The problem we are interested in is the delivery of messages among pairs of either physically or logically mobile units. A mobile unit can send and receive messages only when it is present at some node in the fixed network, a situation that models the existence of an established connection between a mobile unit and a support center or a mobile agent executing on a host. When a mobile unit is on a channel, it may be viewed as being temporarily disconnected from the network and, therefore, unable to communicate.

In typical distributed systems, message passing communication is handled by the routers in the fixed network which track of the relative location of all attached computing devices based on IP address. This model cannot be extended to mobile computing because the location of the mobile units changes with respect to the fixed routers. However, the ability to send and receive messages is critical to many applications.

The typical use of the logical mobile agent paradigm is for bypassing a communication link and exploiting local access to resources on a remote server [28]. Thus, one could argue that, communication with a remote agent is not important and a mobile agent platform should focus instead on the communication mechanisms that are exploited locally. Nevertheless, there are several common scenarios which exploit communication with or among remote agents, some of which are related to mobile agent management. Imagine a "master" agent spawning a number of "slave" mobile agents which are subsequently injected in the network to perform a cooperative computation, e.g., find a piece of information. At some point, the master agent may want to actively terminate the computation of the slave agents, e.g., because the requested information has been found by one of them and thus it is

desirable to terminate the agent in order to prevent unnecessary resource consumption. Or, it may want to change some parameter governing the behavior of the agents in response to a change in the context that determined their creation. Alternately, the slave agents may want to detect whether the master agent is still alive by performing some sort of orphan detection, which requires locating the master agent if this is itself allowed to be mobile.

Other examples arise because mobile agents are just one of the paradigms available to designers of a distributed application. Within the context of the same application, a mixture of mobile agent and message passing can be used to achieve different functionalities. For instance, a mobile agent could visit a site and perform a check on a given condition. If the condition is not satisfied, the agent could register an event listener with the site. This way, while the mobile agent is visiting other sites and before reporting its results, it could receive notifications of state changes in the sites it has already visited and decide whether a second visit is warranted.

In both physical and logical mobility, a desirable requirement for any communication mechanism is reliability. Programming primitives that guarantee that the data sent effectively reach the communication target, without requiring further actions by the programmer, simplify greatly the development task and lead to applications that are more robust. In conventional distributed systems, reliability is typically achieved by providing some degree of tolerance to faults in the underlying communication link or in the communicating nodes.

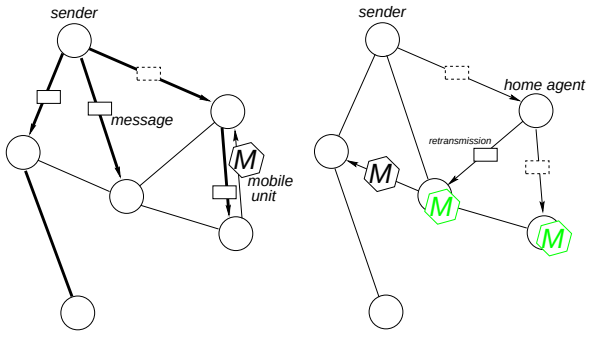
Nevertheless, fault-tolerance techniques are not sufficient to ensure reliability in systems that exhibit mobility. Because mobile units are move freely from one node to another according to some unknown migration pattern, delivery of data is complicated. It is difficult both to determine where the mobile unit is, and to ensure that the data effectively reaches the mobile unit before it moves again. If this latter condition is not guaranteed, data loss may occur. Thus, the challenge to reliable communication persists even under the assumption of an ideal transport mechanism, which itself guarantees only the correct delivery of data from node to node despite the presence of faults. *It is the sheer presence of mobility, and not the possibility of faults, that undermines reliability.*

Problem definition. The reliable message delivery problem can now be formulated as follows: Given a fully connected graph with FIFO channels and guaranteed message delivery between nodes, a message located at one node, and a mobile unit for which the message is destined, develop a distributed algorithm that guarantees single delivery of the message, and leaves no trace of the message, at either a node or a mobile unit, within a bounded time after delivery. The solution should have a tight bound on the storage time for any given message at a node.

Because mobile units do not communicate directly with one another, the network must provide a mechanism to transmit the message. The original message is assumed to be in the local memory of some node, presumably left there by the mobile unit which is the source of the message. Since a mobile unit is not required to visit all nodes to gather its messages, the message cannot remain isolated at the node on which it is dropped off, but instead must be distributed through the network. The specifics of this distribution mechanism are left to the algorithm.

3.3 Previous Work on Message Delivery

Typical message delivery schemes suffer from the fundamental problem that a mobile unit in transit during the delivery can easily be missed. To illustrate this issue, we discuss two strawman approaches to message delivery: broadcast and forwarding.



(a) Spanning tree broadcasting. (b) Forwarding.

Figure 3.3: The problem: Missing delivery in simple broadcast and forwarding schemes.

A simple broadcast scheme assumes a spanning tree of the network nodes which any node can utilize to send a message. This source node broadcasts a copy of the message to each of its neighbors, which broadcast the message to their neighbors, and so on until the leaf nodes are reached. This, however, does not guarantee delivery of the message. In the case when a mobile unit is traveling along a channel in the reverse direction with respect to the propagation of the message, as depicted in Figure 3.3(a), or more generally when the agent moves from the region of the spanning tree ahead of the message propagation to a region behind the message propagation, the agent and the message will cross in the channel, and delivery will never occur.

A simple forwarding scheme maintains a pointer to the mobile unit at a well-known location, referred to as the *home agent* in the Mobile IP protocol [65] where this idea enables physical mobility of hosts. Upon migration, the mobile unit must inform the home agent of its new location in order to enable further communication. However, any messages sent between the migration and the update are lost, as the mobile unit moved before the

message reached the destination. Even if retransmission to the new location is attempted, the mobile unit can move again, running away from the message and effectively preventing guaranteed delivery, as depicted in Figure 3.3(b). Furthermore, forwarding has the additional drawback that it requires communication to the home agent every time the mobile unit moves. In some situations of logical mobility, this defeats the purpose of using mobile agents by reintroducing centralization. For instance, in the presence of many highly mobile agents spawned from the same host, this scheme may lead to considerable traffic overhead around the home agent, and possibly to much slower performance if the latency between mobile and home agent is high. In the physical mobility environment, the rapid movement necessary to avoid a forwarded message seems unlikely. However, one of the trends in mobility is to reduce the size of the cell (e.g., nanocells) to increase the frequency reuse. As cell sizes decrease, the time that it takes to traverse a cell similarly decreases.

In the logical mobility environment, currently available mobile agent systems implement communication by relying on well-known and conventional facilities, such as message passing or remote procedure call. These mechanisms are often blindly borrowed from distributed systems research and exploited with little or no adaptation to the mobile setting. While the problem of guaranteeing data delivery is only seldom acknowledged, the solutions employed usually require knowledge about the location of the mobile agent. Mobile agent location is typically obtained either by overly restricting the freedom of mobility or by assuming permanent connectivity—assumptions that in many cases defeat the whole purpose of using mobile agents.

The OMG MASIF standard [55] specifies only the interfaces that enable the naming and locating of agents across different platforms. The actual mechanisms to locate an agent and communicate with it are intentionally left out of the scope of the standard, although a number of location techniques are suggested which by and large can be regarded as variations of broadcast and forwarding. Some agent systems, notably Aglets [42] and Voyager [61], employ forwarding by associating to each mobile component a proxy object which plays the role of the home agent. Some others, like Emerald [37], one of the early approaches to object migration, use forwarding and resort to broadcast when the object cannot be found. Others, e.g., Mole [11], assume that an agent never moves while engaged in communication; if migration of any of the parties involved take place, communication is implicitly terminated. Mole also exploits a different forwarding scheme which does not keep a single home agent, rather it maintains a trail of pointers from source to destination for faster communication. However, this is employed only in the context of a protocol for orphan detection [12]. Finally, some systems, e.g., Agent Tcl [32], provide mechanisms that are based on common remote procedure call, and leave to the application developer the chore of handling a missed delivery.

While the *unicast* problem of delivering a message to a *single* recipient is important, in recent years the *multicast* problem of delivering a message to *multiple* recipients has also become crucial. Multicast support through the MBONE has become a standard part of the Internet [26] and is finding wide use for conferencing (e.g., tools like VIC and VAT [51, 34]) and video distribution. In the context of Mobile IP, two possible approaches are available. The mobile unit can register to receive multicast packets through its home agent, in which case, the home agent must encapsulate every multicast message and unicast it to the foreign location of the mobile unit. Alternately, the mobile unit can join the multicast group at a local multicast router on the subnet it is visiting. This assumes the local subnet supports multicast. Because many multicast protocols rely on the address of the sender to properly forward messages along a multicast tree, corresponding changes must be made in order for the mobile unit to send to a multicast group address

In logical mobility, many agent systems, notably Telescript, Aglets, and Voyager, provide the capability to multicast messages only within the context of a single runtime support. Mole [11] provides a mechanism for group communication that assumes agents are stationary during a set of information exchanges.

3.4 Concluding Remarks

Focusing on the problem of message delivery incurs no loss of generality because more complex mechanisms such as remote procedure call and method invocation are easily built on top of message passing. Chapters 4 through 6 describe the development of algorithms for reliable message delivery using our strategy of treating a mobile unit as a message.

Chapter 4

Message Delivery to Physically Mobile Hosts

This chapter describes the first application of our design strategy as described in Chapter 3. We show how applying this approach to distributed snapshot algorithms yields an algorithm for reliable message delivery to physically mobile units, an inherently difficult mobility problem.

The remainder of this chapter is organized as follows: Section 4.1 explores the details of the use of snapshot algorithms for message delivery, presenting an overview, algorithm properties, and possible extensions. In the following section, we consider adaptations that make the approach viable in a model similar to the cellular telephone system. Finally, Section 4.4 provides a brief discussion of the results.

4.1 Snapshot Delivery

In this section we present the details of applying the snapshot algorithm to message delivery. In order to avoid confusion in terminology between the control traffic generated by the snapshot algorithms and the data traffic containing the information being communicated to the mobile unit, from this point forward, we will use the term *announcement* to refer specifically to the data message being delivered while a *message* can be either data or control.

4.1.1 From Distributed Snapshot Algorithms to Announcement Delivery

To guarantee delivery in any circumstance, we propose an alternative broadcast algorithm which is based on the classical notion of distributed snapshots. Before addressing announcement delivery, we first note the general properties of snapshot algorithms and those that will be important in announcement delivery. The goal of a snapshot algorithm is to provide

a consistent view of the state of a network of nodes and channels. The state consists of the process variables, and any messages in transit among the nodes. A simple snapshot algorithm would freeze the computation until all messages are out of the channels, record the state of the processors (including outgoing message queues), then restart the computation. Although this is an impractical solution in most distributed settings, it provides the intuition behind a snapshot algorithm, in particular that the consistent global state is constructed by combining the local snapshots from the various processors. In general, a snapshot is started by a single processor and control messages are passed to neighboring nodes informing them that a snapshot is in progress, and initiating local snapshots. The main property of snapshots that we will exploit is that every message will appear exactly once in the recorded snapshot state.

Although snapshot algorithms were developed to detect stable properties such as termination or deadlock by creating and analyzing a consistent view of the distributed state, minor adjustments which we describe in this chapter adapt them to perform announcement delivery in the dynamic, mobile environment. To move from the network of nodes and channels into the mobile computing environment, we return to the cellular structure of mobile support centers. These components and the wires connecting them map directly to the network graph of standard distributed computing. The mobile units are simply represented as persistent messages in the distributed environment, meaning they are always somewhere in the system, either at a node (when in communication with a base station) or on a channel (during a handover). At this point, we have a structure on which to run the snapshot, and note that because the mobile unit is a message, and the snapshot records the location of messages, the global snapshot of the mobile system will show the location of the mobile unit. Therefore, one option is to simply deliver the announcement directly to this location; however, it is possible (and likely in systems with rapidly moving mobile units) that the mobile unit will move between the time its position is recorded and when the announcement arrives at the recorded position. Therefore, we alter the snapshot recording to delivery of the message by augmenting the control messages with the announcement and changing the recording of messages into the delivering of announcements. We further note that the global state of the system is no longer important for delivery, so no system state information needs to be collected.

4.1.2 Snapshot Delivery Algorithm

Throughout this section, we will use the Chandy-Lamport snapshot algorithm [18] and show its adaptation to announcement delivery. In making the transition to the mobile environment, we carry the restrictions of the original distributed algorithm, and clarify certain characteristics of the mobile model moving from the cell structure to the graph setting. First, the Chandy-Lamport algorithm relies on unidirectional, FIFO channels. To

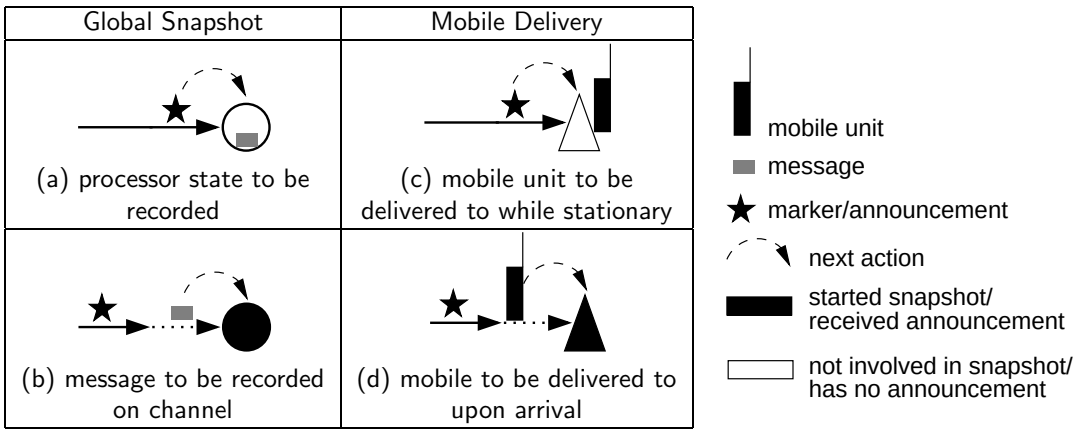


Figure 4.1: Translation of concepts from global snapshots into mobile delivery. The curved arrow shows the processing of an element from a channel while the text describes the action triggered by such movement.

model bidirectional channels, we place two unidirectional channels in opposite directions; however, to handle the FIFO assumption, extensions to the handover protocol are necessary. These will be presented in Section 4.3. In the cellular model, a mobile unit moves directly between cells; however, in the graph representation, the mobile must move onto a channel before arriving at the new cell. This is a natural assumption when the details of the handover are considered.

In the Chandy-Lamport algorithm, it is possible for the snapshot to be initiated at more than one location in the graph, however, we assume that the announcement will be located initially at one point in the network, therefore the snapshot will originate from a single MSC. The Chandy-Lamport algorithm consists of two main localized actions to collect the local snapshot: the processing of the control messages (markers) and the arrival of the messages to be recorded. The *marker arrival rule* states that when a marker arrives at a node not involved in a snapshot, the node begins its local snapshot by recording the processor state, then sends the marker on all outgoing channels (Figure 4.1a). In the mobile environment, this is analogous to the announcement arriving at a node. If the mobile unit is present, it will receive the announcement, otherwise the node will remain in the local snapshot state and will store the copy of the announcement until the local snapshot is complete. The local snapshot is complete when the marker/announcement has arrived from all incoming channels. The message arrival rule states that if the message arrives at a node from channel C before the marker arrives on channel C, and the node is in the middle of the local snapshot, the message is to be recorded as on the channel during the snapshot (Figure 4.1b). In the mobile setting, this condition is the arrival of the mobile unit at an

State

flushed$_{A,B}$ boolean, true if announcement traversed the link from A to B; initially false everywhere

AnnAt$_A$ boolean, true if announcement stored at A; initially true only where announcement starts

MobileAt$_A$ boolean, true if mobile unit at A; initially true only where mobile located

Actions

```
AnnArrives A(B)   ;arrival at A from B
    Effect:
        flushed_{B,A}:=true
        if ¬AnnAt_A
            send ann. on all outgoing channels
            AnnAt_A:=true   ;save ann.
            if MobileAt_A
                deliver announcement
            endif
        endif

MobileArrives A(B)   ;arrival at A from B
    Effect:
        MobileAt_A:=true
        if ¬flushed_{B,A} and AnnAt_A
            deliver announcement
        endif

MobileLeaves A(B)   ;leaves from A to B
    Preconditions:
        MobileAt_A and channel (A,B) exists
    Effect:
        MobileAt_A:=false
        mobile unit moves onto (A,B)

CleanUp A   ;A finishes local snapshot
    Preconditions:
        Forall neighbors X, flushed_{X,A}=true
    Effect:
        AnnAt_A:=false   ;delete ann.
        Forall neighbors X, flushed_{X,A} :=false
```

Figure 4.2: Snapshot Delivery Code

MSC which is storing a copy of the announcement. Therefore, the arrival of the mobile triggers the transmission of the announcement to the mobile.

We capture these actions in an I/O Automata-like pseudo code program shown in Figure 4.2. In addition to the announcement arrival and mobile arrival, we also include statements to terminate the local snapshot (cleaning up the state) and to allow the mobile unit to move from a node to a channel. The channels are assumed to be FIFO, and hold both mobile units and all messages.

We assume the system is initialized with the location of the mobile unit(*MobileAt*, MSC *A* in Figure 4.3) and a single announcement copy at some node (*AnnAt*). Channels are assumed to be empty. We introduce one state variable quantified over the channels (*flushed*) which is used to identify when the local snapshot is complete. Basically a flushed channel has received a marker. As noted previously, when all incoming channels have received a marker, the local snapshot is complete.

These actions describe the local node state transitions which are sufficient for message delivery. No global information must be maintained. A node will be in one of three states: not yet aware of the snapshot (*unnotified*), taking a local snapshot (*notified*), and

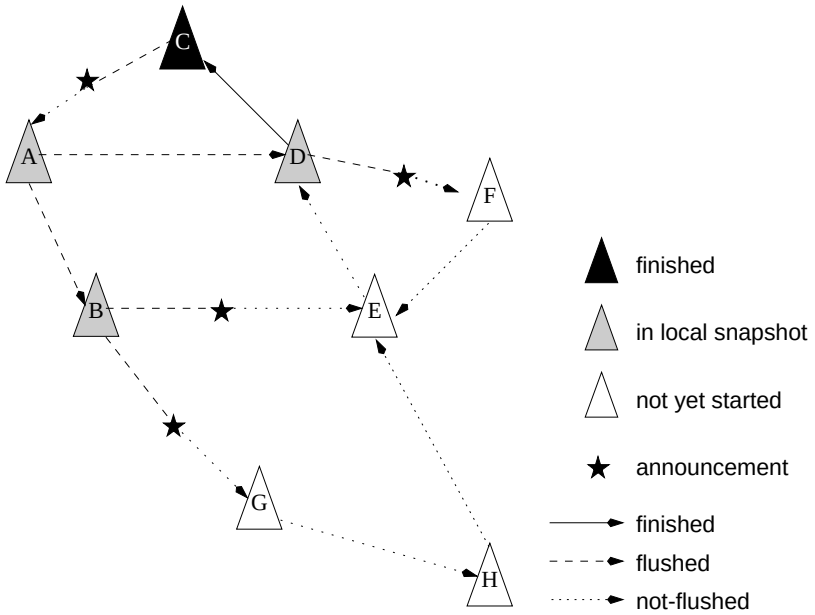


Figure 4.3: Phases of delivery

finished with the local snapshot (*finished*). In Figure 4.3, these states are represented by white, grey, and black respectively. All nodes (except the node where the announcement originates) start unnotified. An unnotified node such as E will eventually receive an announcement along one of its incoming channels (AnnArrives) (such as (B, E)). This action causes it to transition to the notified state, delivering the announcement if possible, storing a copy of the announcement, marking the channel the announcement arrived on as flushed, and sending announcement copies on all outgoing channels. Once a channel is flushed, if the mobile unit arrives on that node, it is guaranteed to have seen the announcement at some other node (to have been recorded in some other local snapshot). Therefore, to avoid multiple delivery, if the mobile arrives on a flushed channel, delivery is not repeated (MobileArrives). If the announcement arrives at a notified node such as A (MobileArrives), the channel it arrives on will be marked as flushed, but since the announcement is already stored, no additional copy is made. When all incoming channels have been flushed (as in B), the node's local snapshot is complete and the local state (including the flushed status of the channels and the stored announcement) can be deleted (CleanUp). The final action, MobileLeaves, models the movement of a mobile away from a node. The mobile is simply placed on the channel and the state variables updated to reflect this change. This effectively achieves random mobile unit movement, however if a particular movement pattern is desired, it can be added to this action.

4.1.3 Properties

In the previous section we presented a global snapshot algorithm modified to perform announcement delivery in a mobile system. Because the approach is based on a well understood algorithm from distributed computing, we can adapt the proven properties from the distributed computing environment into the mobile environment. The three primary properties proven for the Chandy-Lamport distributed snapshot are: (1) there is no residual storage in the system at some point after the algorithm begins execution, (2) every message is recorded once, and (3) no message is recorded more than once. We translate these properties directly into the mobile environment stating that (1) eventually there is no residual storage in the system at some point after the delivery process begins, (2) the announcement is delivered to the intended recipient, and (3) the announcement is delivered only once. In this section, we present first the intuition behind why these properties hold, followed by a reduction style proof outline moving from the distributed snapshot properties to the mobile announcement delivery properties stated above.

To show that eventually all information concerning the announcement is removed from the system, we must show that the program will eventually reach a state where there are no announcements at any nodes, there are no announcements in any channels, and all channels are unflushed. This can be shown by observing that by the connectivity of the graph and the dispersion rules of the algorithm, every node eventually receives a copy of the announcement on each of its channels. In Figure 4.3, this is analogous to each node eventually becoming notified. Once this occurs, each of the channels is flushed, and the cleanup action is enabled, clearing the flushed variable and removing the copies of the announcement at each node. In the figure, the node will transition to finished. There will be no announcements in channels because once an announcement has passed through a channel, it is not possible for another announcement to be sent down that channel. Therefore, we have shown that eventually the system is clear.

Next we must show that the announcement is eventually delivered. First we note (from the previous paragraph) that eventually all nodes receive the announcement. When this occurs, if the mobile unit has been delivered to, then the proof is complete. Otherwise, if the mobile unit has not been found then it must be the case that the mobile unit is located on an unflushed channel and the mobile unit must be in front of the announcement on this channel. In Figure 4.3, an unnotified mobile unit could be located at a white node, or on a dotted channel, but in this case, we are describing a graph in which all nodes are already notified, therefore the mobile unit must be located on an unflushed channel segment in front of an announcement. If the mobile unit was behind the announcement, it would have already received the announcement. Therefore, because the mobile unit arrives on an unflushed channel, and by the assumption that all nodes have received a copy of

the announcement, the node it is heading toward must have a copy of the announcement. Delivery will occur when the mobile unit moves onto the node.

Having shown the announcement will be delivered, it remains to be shown that the announcement is delivered only one time. To do this, it is sufficient to show that after delivery occurs, a mobile unit cannot be in the position to be delivered to. Delivery can occur in two situations: (1) by the mobile being on an unnotified node when the announcement arrives, and (2) by the mobile arriving (along an unflushed channel) to a notified node. To show each of these cases will not arise after delivery, we characterize a region of the graph called *will-be-notified* and define it as the set of all unnotified nodes, and the channels or channel segments which have not had a copy of the announcement pass through them. In Figure 4.3, this region corresponds to the white nodes (unnotified) and dotted channels (unflushed). Having defined this region, and noted that both of the above delivery cases occur when the mobile is in this region, it is sufficient to show that after delivery the mobile unit will not be in the *will-be-notified* region. To show this, we use the intuitive notion of a path as a sequence of nodes and channels between a mobile unit and the *will-be-notified* region. It can be shown that after delivery on such a path, there must always be an announcement. Therefore, in order for the mobile unit to move back into the *will-be-notified* region, it must overtake this announcement, and because of the FIFO assumption of the channels and the rules for dispersion of the announcement, this is not possible. Therefore, the announcement can only be delivered once.

To more explicitly show the relationship between the snapshot algorithm and announcement delivery, we provide a reduction proof outline from the Chandy-Lamport distributed algorithm to the adapted snapshot delivery algorithm, showing the mapping between the actions (such as marker arrival and announcement arrival) and the system variables (such as the marker and announcement). In the Chandy-Lamport algorithm, a process begins its local snapshot when it receives the first marker. When this occurs, the marker is sent on all other outgoing channels and the state of the processor is recorded. If there are any messages at the node, they are recorded as part of the processor state (Figure 4.1a). If the node has already started its local snapshot when a message arrives along a channel (that the node has not seen the marker on), the message is recorded as being on the channel (Figure 4.1b). Recording continues until a marker is received on all incident links.

We translate these actions to the mobile environment. The announcement corresponds to the marker, and the mobile node corresponds to a message in the Chandy-Lamport algorithm. When an MSC receives the announcement for the first time, it sends copies on all outgoing channels and attempts delivery to any mobile unit present. If the mobile unit is at the MSC, it will receive the announcement (Figure 4.1c).

Just as a node continues recording until it has received the marker on all links, the delivery algorithm *will keep a copy of the announcement until it receives a copy of*

the announcement from all neighbors. Intuitively, this prevents the mobile from hopping from node to node eluding the announcement. Thus if the mobile arrives prior to the announcement on a channel, the MSC delivers the data as soon as the handover is complete (Figure 4.1d) because it has a stored local copy.

4.2 Reality Check

When moving from the distributed computing environment to the mobile environment, we made several assumptions about the nature of the network and the behavior of the components in the network. In this section, we reexamine these assumptions, showing why they are reasonable, or how the algorithm can be adapted to make them more reasonable. Specifically, we will look at the issues of non-FIFO channels, base station connectivity, reliable delivery on links, the involvement level of MSCs, and storage requirements.

FIFO Channels. One major objection to using the Chandy-Lamport algorithm is its reliance on FIFO behavior of channels. More specifically, in Section 4.1.1, we modeled both the mobile units and the messages as traveling on the same channel. This seems to be an unreasonable assumption given that mobile units move much more slowly through space than messages move through a fixed network. To further explore this problem, we must look in detail at the movement of messages and mobile units, specifically the handover of mobile units between cells. We will show how the FIFO assumption can be broken, and propose a simple mechanism to restore it.

One of the U.S. standards for analog cellular communication is AMPS [83], in which cellular telephones tune to only one frequency at a time. When the signal between the MSC and a mobile unit begins to degrade, the MSC searches for a neighboring MSC with a stronger communication signal potential indicating the mobile unit is moving into that particular cell, and an available frequency within that cell. When the frequency is requested, a handover begins. Figure 4.4a shows the control messages exchanged as a mobile unit, m, moves from cell A to cell B. First the *frequency request* is exchanged between the MSCs. At this point the mobile unit is made aware of the handover by receiving a new frequency from its current MSC, A. After switching to the new frequency, the mobile sends a *hello* on the new frequency, alerting B that the mobile is now listening on the new frequency. Finally, B sends a *handover complete* to A, which releases the old frequency.

By using the AMPS approach, we know when a mobile unit is moving between cells and which cells it is moving between. We also note that the mobile unit is not involved in the handover until the moment it changes the frequency it is tuned to.

Our primary concern is making the channels FIFO with respect to mobile units and messages (both control messages and announcements). Even if we assume that channels

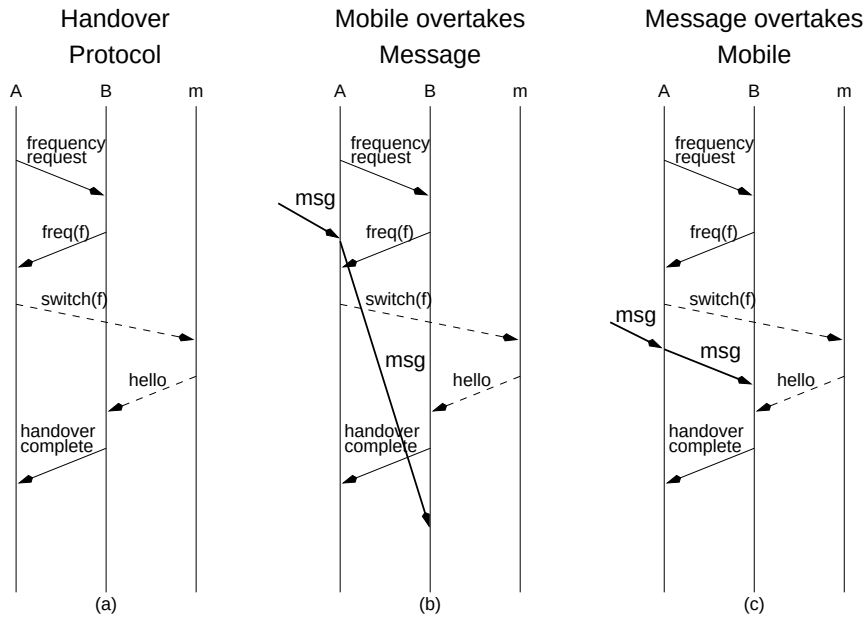


Figure 4.4: AMPS handover protocol (a). If messages are processed (i.e., broadcast to the mobile) immediately upon receipt, it is possible (b) for the mobile to move faster than the message along the channel, or (c) for the message to move faster than the mobile, thus breaking the FIFO channel property.

between MSCs are FIFO, reordering is possible because part of the handover takes place over wireless channels which are not synchronized with the wired channels. Specifically we address the two cases of non-FIFO behavior where (1) the mobile overtakes a message and (2) the messages overtakes the mobile.

It is important to define the point at which the mobile logically moves onto the channel. We define this to be when communication with A is terminated, or when the *switch* message is transmitted. Similarly, the mobile moves off of the channel when the wireless transmission of the *hello* message is accepted at the destination. As can be seen in Figure 4.4b, it is possible for a messages sent on the wired channel before the *switch* message to arrive at the destination after the arrival of the mobile unit, breaking the FIFO ordering. Similarly, a message sent after the *switch* message can move quickly through the channel and arrive at the destination before the mobile (Figure 4.4c).

We propose a minor change in the protocol to involve both the wired and wireless channels in the handover. The only change to the source side (A in this case) is the wire transmission of a message atomically with the wireless *switch* transmission. We call this message the *virtual mobile unit* (VMU) because it identifies the point on the wired channel at which the mobile leaves the source. All messages sent before the VMU were sent before the mobile unit left, and all messages after the VMU were sent after the mobile unit left.

Therefore, we change the behavior of the destination (B in this case) to yield this behavior. The goal is to have the virtual mobile unit and the physical mobile unit arrive at the destination at the same time. Therefore, if the *hello* arrives before the VMU, all incoming messages on the wired channel are treated as if the mobile is not present even though communication is possible. Conversely, if the VMU arrives before the *hello*, all messages sent on the wired channel are buffered until the *hello* arrives. When both messages have arrived and have been processed, B continues processing all messages in the order in which they are received. By forcing the receiver to wait for both messages, the wired and wireless channels are synchronized, effectively yielding a single FIFO channel for both mobile units and messages.

Base station connectivity. Another possible concern with the model we presented is the necessity for physical connections between all MSCs whose cells border one another. Because of the high cost for such connectivity, it is possible that the physical wires may not exist. To allow the snapshot to function in such a setting, we add virtual channels between adjacent cells and include such channels in the snapshot as a channel along which control messages must be sent and received. In the implementation, however, we must be careful to ensure the FIFO nature of this virtual channel. It would also be possible to add a virtual channel between two non-adjacent cells if a mobile unit was likely to move between them in a disconnected manner. This would only work if the same type of handover was used for disconnected movement, which is not likely the case because disconnection is typically a longer-lived situation, but the possibility is worth mentioning.

Reliable delivery on links. The snapshot delivery algorithm assumes that link delivery is reliable. Most of the Internet uses unreliable links like Ethernets, frame relay, and ATM. The probability of error on such links may be small but packets are indeed dropped. A possible solution is to add acks for multicast messages as is done, for example, in the intelligent flooding algorithm used in Links State Routing in OSI [89] and OSPF [57]. Another solution is to only provide best-effort service. Since lost messages can lead to deadlock we need to delete a message after a timeout even if it is still expected along a channel.

Involvement level of MSCs. In every snapshot, every MSC must be involved to guarantee delivery and termination. In a paper on running distributed computations in a mobile setting [9], the authors warn against requiring involvement of all mobile units in a computation, especially due to the voluntary disconnection often associated with mobile computing. Such disconnection is often done to conserve power, or in some cases, to allow disconnected operation. In either case, the mobile unit is not available for participation in the distributed algorithm. These arguments are important for creating distributed algorithms for mobile

computing environments, however our goal is not to create a global snapshot of the mobile units, but instead to employ the snapshot technique to a different end, namely announcement delivery. Additionally, the control messages of the snapshot are not being run over the mobile units, but rather over the fixed mobile support centers. In order to guarantee delivery, the mobile unit must be present in the system, however, this is a reasonable assumption because there are no means to reach a disconnected mobile unit. At this point, it is interesting to note that if the mobile unit is not present in the system during a delivery attempt, the algorithm will terminate normally, removing all trace of the announcement from the system, but without delivery.

Storage requirements. In snapshot delivery, we assume that the MSCs hold a copy of the announcement for delivery to the mobile agents for a bounded period of time limited to the duration of the local snapshot. This is more efficient than another proposal [3] which broadcasts the announcement to all nodes, which then store the announcement until notified that delivery has occurred. This is essentially a *network propagation delay* factor. In a system with bi-directional channels, because the local snapshot terminates when the announcement arrives on all incoming channels, a local snapshot can be as short as a single round trip delay between the MSCs, or a *local propagation delay*. One can argue that it is not the place of the MSCs to be maintaining copies of announcements when their primary purpose is routing. However, in this case, because no routing information is being kept about the mobile units, the system will be required to keep additional state in order to provide delivery guarantees. Therefore, keeping a copy for a short duration is a reasonable assumption.

4.3 Extensions

The snapshot delivery algorithm is extendable to perform delivery to rapidly moving mobile units, route discovery, multicast, and working with the mobile code environment. Each of these will be examined in turn.

Rapidly moving mobile units. One advantage to this algorithm is the ability to operate in rapidly changing environments with the same delivery guarantees. In Mobile IP, mobile units must remain in one place long enough to send a message with their new address to their home agent for forwarding purposes, and remain at that foreign agent long enough for the forwarded messages to arrive. With forwarding enhancements added to the foreign agents in Mobile IP, the issue is minimized because the former location of a mobile unit becomes a kind of packet forwarder. However, even with forwarding, if the agent moves too rapidly and the system is unable to stabilize, forwarded packets will chase the mobile

unit around the system without ever being delivered. Because snapshots do not maintain a notion of home or route, movements are immediately accounted for by the delivery scheme.

Route discovery. In more moderately changing environments, route discovery can increase efficiency and decrease overhead. In these situations, the snapshot delivery algorithm can be used to perform route discovery. When the discovery message from source S located at MSC_S arrives at the destination mobile unit D located at MSC_D, the mobile unit responds with a packet directed toward MSC_S with MSC_D and a particular RBS in its data to identify its location to the source. This assumes that the source is also moving relatively slowly, however a route response message could be sent to S from D in a similar manner as the route discovery. All packets sent to D after the discovery should contain MSC_D and RBS, and routing along the fixed network is now possible.

Multicast. As stated previously, another concern in the mobility community is multicast. While Mobile IP addresses the issues of macromobility, and some work has been done on reliable multicast for micromobility [3] when the set of recipients is known, our algorithm easily extends to perform multicast to all mobile units in the system during the execution of the snapshot without knowing the list of recipients. Without changing the snapshot algorithm, it can be shown that delivery of an announcement is attempted to all mobile units in the system before the algorithm terminates. If the announcement contains a broadcast or multicast destination address, delivery can be carried out whenever a connection with the mobile unit accepting those addresses is established. Interestingly, even in broadcast or multicast, the restriction of single delivery of an announcement holds. Although this description is concise, the importance of it should not be lost in its simplicity.

Our modified snapshot algorithm has worst-case overhead of one announcement per link in each direction to multicast. By contrast, the algorithm used in IP DVMRP [22] effectively computes a tree. Its overhead is the number of links in the tree plus the number of links that have endnodes that participate in this multicast.

Logical Mobility. Thus far we have only considered physical movement of mobile units, but another possible application that is characterized by rapid mobile movement is logical mobility where it is not a physical component that moves, but rather a program moving along the fixed network doing computation at various network nodes. We consider this in much more detail in the next chapter.

4.4 Concluding Remarks

In this chapter we presented an algorithm for reliable message delivery by applying the design strategy presented in Chapter 3. Its mechanics are borrowed directly from the established literature of distributed computing, specifically distributed snapshots. The ease of extending this algorithm to reliable multicast in a mobile environment without knowing the recipients, as well as working in the mobile code arena are added benefits to the approach. More generally, we show that treating mobile units as messages provides an effective means for transferring results from classical distributed algorithm literature to the emerging field of mobile computing.

Chapter 5

Communication among Highly Mobile Agents

The previous chapter showed how the distributed snapshot algorithm by Chandy and Lamport can be applied to reliable message delivery in the physically mobile environment. This chapter starts with the same premise, but from the perspective of logical mobility. By moving to this arena, we uncover many issues that are not addressed by the basic snapshot delivery algorithm. For example, in a physical mobile environment, the support infrastructure consists of large pieces of dedicated hardware creating a known and stable environment. In contrast, the support structure of a logical mobility system consists of multi-purpose computing components executing agent support software. It is likely that this network will not be known in advance, making the original snapshot approach unusable, however, the need for reliable message delivery persists. This chapter shows how the original algorithm can be extended to allow for an expanding network of support nodes, increasing the overall flexibility of these abstractions.

Section 5.1 begins by briefly summarizing the original results from the previous chapter, introducing the notation used for logical mobility and using it as the starting point for the extensions which follow. Section 5.2 discusses the applicability and implementability of a communication mechanism embodying our algorithm in a mobile agent platform.

5.1 Logical Snapshot Delivery

5.1.1 Delivery in a Static Network Graph

We begin the description of our solution with a basic algorithm which assumes a fixed network of connected nodes. For simplicity, we describe first the behavior of the algorithm under the unrealistic assumption of a single message being present in the system, and then show how this result can be extended to allow concurrent delivery of multiple messages.

Distributed Snapshot	Snapshot Delivery
Node	Mobile agent server
Message	Mobile agent
Token	Message
Record message	Deliver message
Local snapshot ended	Message deleted

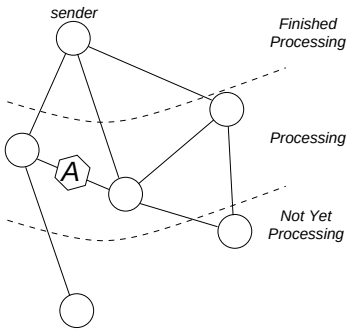


Figure 5.1: Using distributed snapshots for message delivery. Each concept from the traditional snapshot is mapped to a concept in the mobile environment. The result is the ability to trap an agent in a region of the network from which it cannot escape without receiving a copy of the message.

Single message delivery.

Chapter 4 approached reliable message delivery in the physical mobility environment by adapting the notion of distributed snapshots [58]. In snapshot algorithms, the goal is to record the local state of the nodes and the channels in order to construct a consistent global state. Critical features of these algorithms include propagation of the snapshot initiation, the flushing of the channels to record all messages in transit, and the recording of every message exactly once. Our approach to message delivery comes directly from the ideas in the original snapshot paper presented by Chandy and Lamport [18], applying our design strategy to treat the mobile unit as a message. In this move to the mobile environment, the meaning of basic algorithm properties change: instead of spreading knowledge of the snapshot using messages, we spread the actual message to be delivered; instead of flushing messages out of the channels, we flush agents out of the channels; and instead of recording the existence of the messages, we deliver a copy of the message. This correspondence of concepts in the two domains can be seen in Figure 5.1.

The algorithm works by associating a state, OPEN or FLUSHED, with each incoming channel of a node. Initially all channels are OPEN and no node is aware of a snapshot delivery in progress. Delivery is initiated from outside the system, e.g., by an agent requesting its current host to deliver a message. When the message arrives, the state of the channel it came through is changed to FLUSHED, implying that all the agents on that channel ahead of the message have been forced out of the channel (by the FIFO assumption). When the message arrives for the first time at a node, it is stored locally, and delivery is attempted to all agents present at the node. If the agent identifier does not match the message destination, no delivery occurs. In the same atomic step, the message is propagated on all outgoing channels, thus starting the flushing process on those channels. Each agent that arrives

1:	*precondition:*	no incoming channels OPEN
	action:	$curMsg = \perp$
2:	*precondition:*	message j arrives $\wedge$ $(curMsg = \perp \vee curMsg = j)$
	action:	if $curMsg = \perp$ deliver, store, propagate
3:	*precondition:*	message j finished processing
	action:	
4:	*precondition:*	message i arrives $\wedge$ $(curMsg = j \wedge i > j)$
	action:	buffer message i

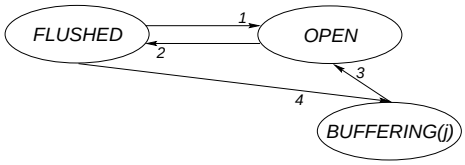


Figure 5.2: State transitions and related diagram for multiple message delivery in a static network graph.

through an OPEN channel on a node storing the message must be delivered a copy of it. When all the incoming channels of a node are FLUSHED, which is guaranteed to occur by the network connectivity assumption, the node is no longer required to deliver the message to any arriving agents, therefore the message copy is deleted and all of the channels are atomically reset to OPEN.

Intuitively, this processing partitions the network into the three regions as shown in Figure 5.1: regions not yet aware of the message, currently processing the message, and where delivery has completed. An agent which has not yet received the message must either be in the first region or on a channel in the currently processing region. In order for the agent to move to the completed region, it must pass through a node in the processing region and receive the message. Because the entire graph will eventually finish processing, it is guaranteed that the agent will receive the message.

Multiple message delivery.

A possible adaptation of the previous algorithm to multiple message delivery is to require a node to wait for the termination of the current message delivery and to coordinate with the other nodes before initiating a new one, in order to ensure that only one message is being delivered at any time. However, this unnecessarily constrains the behavior of the sender and requires knowledge of non-local state. In practical scenarios, it is desirable to allow multiple messages to flow concurrently in the network. Typically, this is needed for two reasons: to allow a source to transmit a burst of of messages without waiting for the delivery of the first one to complete, and to allow multiple sources to transmit at the same time.

We propose here a variant of the algorithm that encompasses both cases without

requiring coordination among hosts. This is accomplished by requiring that each message is tagged with the identifier of the host that initially sent it, as well as an ever-increasing sequence number (or, in practice, a sufficiently large circular window of numbers). The sequence number addresses the case of a message burst coming from a single source, while the host identifier allows multiple sources to transmit at the same time.

To handle a burst of messages from a single source, additional logic must be added to deal with the arrival of a new message while the previous one is still being processed. This case is identified by the arrival of a message on an already FLUSHED channel. To handle the new message, we introduce a new state, BUFFERING, as shown in Figure 5.2 (transition 4). The new message and any additional message arriving on a BUFFERING channel are put into a temporary buffer to be processed at a later time. A buffering channel is considered FLUSHED for the purposes of determining whether the local processing for the delivery of the current message is complete. When this transition to OPEN is finally made for all incoming channels (transitions 1 and 3), the messages in the buffers are treated as if they are new messages on the front of the channel, and are processed again. It is possible that after processing the first buffered message the next message causes a transition to BUFFERING, but the fact that the head of the channel is processed ensures eventual progress through the sequence of messages to be delivered.

While the above addresses multiple messages from a single source host, it does not allow for multiple sources. To do this, we effectively execute concurrent copies of the above algorithm. Instead of keeping a single channel state and buffer for each incoming channel, a vector of states and buffers is maintained. Each entry in the vector corresponds to a single source, and any message arriving from the source is processed only with respect to this entry. Additionally, the transition of channels to OPEN is made on a per source basis by using the corresponding values in the vectors of each incoming channel.

Although messages are buffered, agent arrival is not restricted, allowing the agent to move ahead of any messages it originally followed along the channel. Effectively, the agent may move itself back into the region of the network where the message has not yet been delivered. Therefore, duplicate delivery is possible, although duplicates can be discarded easily by the runtime support or by the agent itself based on the message identifier.

5.1.2 Delivery in a Dynamic Network Graph

Although the solutions proposed so far provide delivery guarantees in the presence of mobility, the necessity of knowing the network of neighbors a priori is sometimes unreasonable in the dynamic environment of mobile agents. Furthermore, the delivery mechanism is insensitive to which nodes have been active, and delivers the messages also to regions of the network that have not been visited by agents. Therefore, our goal is still to flush channels and trap agents in regions of the network where the messages will propagate, but also to

allow the network graph used for the delivery process to grow dynamically as the agents migrate. A channel is only included in the message delivery if an agent traversed it, and therefore, a node is included in the message delivery only if an agent has been hosted there. We refer to a node or channel included in message delivery as *active*.

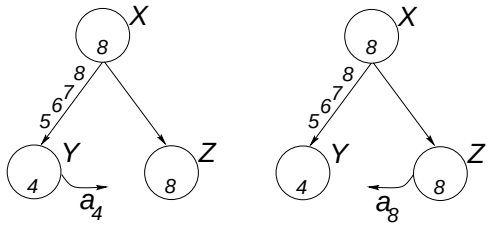
Our presentation is organized in two phases. First, we show a restricted approach where all the messages must originate from a single, fixed source. This is reasonable for monitoring or master-slave scenarios where all communication flows from a fixed initiator to the agents in the system. Then, we extend this initial solution to enable direct inter-agent messaging by allowing any active node in the graph to send messages, without the need for a centralized source.

Single message source.

First, we identify the problems that can arise when nodes and channels are added dynamically, due to the possible disparity between the messages processed at the source and destination nodes of a channel when it becomes active. We initially present these issues by example, then develop a general solution.

Destination ahead of source. Assume a network as shown in Figure 5.3(a). X is the sender of all messages and is initially the only active node in the system. The graph is extended when X sends an agent to Y, causing Y and (X, Y) to become active. Suppose X sends a burst of messages 1..4, which are processed by Y, and later a second sequence of messages 5..8. This second transfer is immediately followed by the migration of a new agent to node Z, which makes Z and (X, Z) active. Before message 5 arrives at Y, an agent is sent from Y to Z, thus causing the channel (Y, Z) to be added to the active graph.

A problem arises if the agent decides to immediately leave Z, because the messages 5..8 have not yet been delivered to it and may never be delivered. Furthermore, what processing should occur when these messages arrive at Z along the new channel (Y, Z)?



(a) Destination ahead of source.

(b) Source ahead of destination.

Figure 5.3: Problems in managing a dynamic graph. Values shown inside the nodes indicate the last message processed by the node. The subscripts on agent a indicate the last message processed by the source of the channel being traversed by a right before a migrated.

If the messages are blindly forwarded on all Z's outgoing channels, message ordering is possibly lost and messages can possibly continue propagating in the network without ever being deleted.

Our solution is to hold the agent at Z until the messages 5..8 are received and, when these messages arrive, to deliver them only to the detained agent, i.e., without broadcasting them to the neighboring nodes. Therefore, no messages are lost and the system wide processing of messages is not affected. Notably, although we do inhibit the movement of the agent until these messages arrive, this takes place only for a time proportional to the diameter of the network, and even more important, only when the topology of the network is changing.

Source ahead of destination. To uncover another potential problem, we use the same scenario just presented for nodes X, Y, and Z. However, instead of assuming an agent moving from Y to Z, we assume it is moving from Z to Y, thus making (Z, Y) active (Figure 5.3(b)). Although the agent will not miss any messages in this move, two potential problems exist.

First, by making (Z, Y) active, Y will wait for Z to be FLUSHED or BUFFERING before proceeding to the next message. However, message 5 will never be sent from Z. Our solution is to delay the activation of channel (Z, Y) until Y catches up with Z. In this example, we delay until 8 is processed at Y. Second, if message 9 is sent from X and propagated along channel (Z, Y), it must be buffered until it can be processed in order.

Solution. Given this, we now present a solution that generalizes the previous one. We describe in detail the channel states and the critical transitions among these states, using the state diagram in Figure 5.4.

- CLOSED: Initially, all channels are CLOSED and not active in message delivery.
- OPEN: The channel is waiting to participate in a message delivery. When an agent arrives through an OPEN channel on a node that is storing a message destined to that agent, the agent should receive a copy of such message.
- FLUSHED: The current message being delivered has already arrived on this channel, and therefore this channel has completed the current message delivery. Agents arriving on FLUSHED channels need no special processing.
- BUFFERING(j): The source is ahead of the destination. Messages arriving on BUFFERING channels are put into a FIFO buffer. They are processed after the node catches up with the source by processing message j. Agents arriving on BUFFERING channels need no special processing.

1:	*precondition:*	no incoming channels OPEN $\wedge$ no incoming channels HOLDING
	action:	$curMsg = \perp$
2:	*precondition:*	message j arrives $\wedge$ ($curMsg = \perp \vee curMsg = j$)
	action:	if $curMsg = \perp$ deliver, store, propagate
3:	*precondition:*	message j finished processing
	action:	
4:	*precondition:*	message i arrives $\wedge$ ($curMsg = j \wedge i > j$)
	action:	buffer message i
5:	*precondition:*	message j arrives $\wedge$ ($curMsg = \perp \vee curMsg > j$)
	action:	deliver to held agents, release held agents
6:	*precondition:*	message j arrives $\wedge$ $curMsg = j$
	action:	deliver to held agents, release held agents
7:	*precondition:*	agent arrives $\wedge$ D ahead of S $\wedge$ ($curMsg = j \ \vee curMsg = \perp$)
	action:	
8:	*precondition:*	agent arrives $\wedge$ $curMsg \neq \perp \wedge$
		S and D processing same message
	action:	
9:	*precondition:*	agent arrives $\wedge$ (D not active $\vee$
		(S and D processing same message $\wedge$ $curMsg = \perp$))
	action:	
10:	*precondition:*	agent a_j arrives $\wedge$ S ahead of D
	action:	

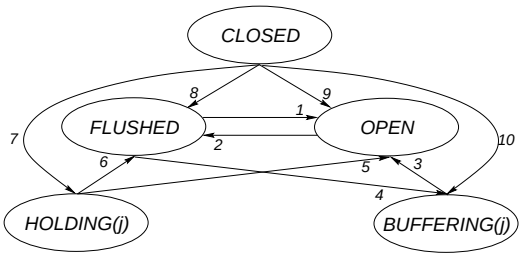


Figure 5.4: State transitions and related diagram for multiple message delivery with a single source in a dynamic network graph. The state transitions refer to a single channel (S, D).

- HOLDING(j): The destination is ahead of the source. Messages with identifiers less than or equal to j which arrive on HOLDING channels are delivered to all held agents. Agents arriving on HOLDING channels, and whose last received message has identifier less than j, are held until j arrives.

The initial transition of a channel from CLOSED to an active state is based on the current state of the destination node and on the state of the source as carried by the agent. The destination node can either still be inactive or it can have finished delivering the same message as the source (9), it can still be still processing such message (8), it can be processing an earlier message (10), or it can be processing a later message (7). Based on this comparison, the new active state is assigned. Once a channel is active, all state transitions occur in response to the arrival of a message. Because we have already taken measures to ensure that all messages will be delivered to all agents, our remaining concerns

are that detained agents are eventually released, and that at every node the next message is eventually processed.

Whether an agent must be detained or not is determined by comparing the identifier of the latest message received by the agent, carried as part of the agent state, and the current state of the destination node. Only agents that are behind the destination are actually detained. If an agent is detained at a channel in state HOLDING(j), it can be released as soon as j is processed along this channel. In this case, the agent was delivered a copy of the message when the agent first arrived, but we assume that this out of order message is ignored by the runtime support, based on the message identifier. Therefore, message j is processed when it arrives on the channel the agent is holding on. By connectivity of the network graph, we are guaranteed that j will eventually arrive[1]. When it does, the destination node will either still be processing j, or will have completed the processing. In both cases the agent is released. In the former case, the channel transitions to FLUSHED (6) to wait for the rest of the channels to catch up, while in the latter case the channel transitions to OPEN (5) to be ready to process the next message.

To argue that eventually all messages are delivered, we must extend the progress argument presented in Section 5.1.1 to include the progress of the HOLDING channels as well as the addition of new channels. As noted in the previous paragraph, message j is guaranteed to eventually arrive along the HOLDING channel, thus ensuring progress of this channel. Next, we assert that there is a maximum number of channels that can be added as incoming channels, bounded by the number of nodes in the system. We are guaranteed that if channels are continuously added, eventually this maximum will be reached. By the other progress properties, eventually all these channels will be either FLUSHED or BUFFERING, in which case processing of the next message (if any) can begin.

Multiple message sources.

Although the previous solution guarantees message delivery and allows the dynamic expansion of the graph, the assumption that all messages originate at the same node is overly restrictive. To extend this algorithm to allow a message to originate at any active node, we effectively superimpose multiple instances of the same algorithm on the network, in a manner similar to the multiple message delivery in a static network. For the purposes of explanation, let n be the number of nodes in the system. Then:

- The state of an incoming channel is represented by a vector of size n where the state of each node is recorded. Before the channel is added to the active graph, the channel

[1]The connectivity assumption we make here is that in the initial system state, before any agents migrate, the node identified as the source of all messages must be connected to all nodes which can spontaneously generate agents. Spontaneous generation of agents means that a node can create an agent without the prior arrival of another agent.

is considered CLOSED. Once the channel is active, if no messages have been received from a particular node, the state of the element in the vector corresponding to that node is set to OPEN.

- Processing of each message is done with respect to the channel state associated with the node where the message originated.
- Nodes can deliver n messages concurrently, at most one for each node. As before, if a second message arrives from the same node, it is buffered until the prior message completes its processing.
- An agent always carries a vector containing, for each message source, the identifier of the last message received. Moreover, when an agent traverses a new outgoing channel, it carries another vector that contains, for each message source, the identifier of the last message processed by the source of the new channel right before the agent departed.
- An incoming agent is held only as long as, for each message source, the identifier of the last message received is greater than the corresponding HOLDING value (if any) of the channel the agent arrived through.
- To enable any node to originate a message, we must guarantee that the graph remains connected. To maintain this property we make all links bidirectional. In the case where an agent arrives and the channel in the opposite direction is not already an outgoing channel, a *fake agent message* is sent to S with the state information of D. This message effectively makes the reverse channel active.

Again, we must argue that detained agents are eventually released and that progress is made with respect to the messages sent from each node. Assume that message i is the smallest message identifier from any node which has not been delivered by all nodes. There must exist a path from a copy of i to every node where i has not arrived, and every node on this path is blocked until i arrives. By connectivity of the network graph, i will propagate to every node along every channel and will complete delivery in the system. No node will buffer i because it is the minimum message identifier which is being waited for. When i has completed delivery, the next message is the new minimum and will make progress in a similar manner. Because the buffering of messages is done with respect to the individual source nodes and not for the channel as a whole, the messages from each node make independent progress.

Holding agents requires coordination among the nodes. The j value with respect to each node for which the channel is being held, e.g., HOLDING(j), is fixed when the first agent arrives. Because the messages are guaranteed to make progress, we are guaranteed that eventually j will be processed and the detained agents will be released.

5.1.3 Multicast Message Delivery

In all the algorithms described so far, we exploited the fact that a distributed snapshot records the state of each node exactly once, and modified the algorithm by substituting message recording with message delivery to an agent. Hence, one could describe our algorithm by saying that it attempts to deliver a message to every agent in the system, and only the agents whose identifier match the message target actually accept the message. With this view in mind, the solution presented can be adapted straightforwardly to support multicast. The only modification that must be introduced is the notion of a multicast address that allows a group of agents to be specified as recipients of the message—no modification to the algorithm is needed.

5.2 Discussion

In this section we analyze the impact of our communication mechanism on the underlying mobile agent platform, argue about its applicability, discuss the current implementation and comment on possible extensions and future work on the topic.

5.2.1 Implementation Issues

A fundamental network property that must be preserved in order for our communication algorithms to function properly is the FIFO behavior of communication channels—a legacy of the fact that the core of our schema is based on the Chandy-Lamport distributed global snapshot. The FIFO property must be maintained for every piece of information traveling through the channel, i.e., messages and agents. This is not necessarily a requirement for mobile agent platforms. Rather, a common design is to map the operations that require message or agent delivery on data transfers taking place on different data streams, typically through sockets or some higher-level mechanism like remote method invocation. In the case where these operations insist on the same destination, the FIFO property may not be preserved, since a data item sent first through one stream can be received later than another data item through another stream, depending on the architecture of the underlying runtime support. Nevertheless, the FIFO property can be implemented straightforwardly in a mobile agent server by associating a queue that contains messages and agents that must be transmitted to a remote server. This way, the FIFO property is structurally enforced by the server architecture, although this may require non-trivial modifications in the case of an already existing platform.

Our mechanism assumes that the runtime support maintains some state about the network graph and the messages being exchanged. In the static single message delivery algorithm we present, this state is constituted only by the last message received. In a

	Guaranteed Delivery	Multicast Capable	Delivery Overhead	Knowledge Maintained
Forwarding	No	No	One indirection	Agent location
Broadcast	No	Yes (no guarantees)	One msg. per spanning tree edge	Spanning tree
Static Snapshot	Yes	Yes	One msg. per edge	Neighbors (known in advance)
Dynamic Snapshot	Yes	Yes	One msg. per traversed edge	Neighbors (discovered)

Figure 5.5: Tradeoffs when choosing a communication mechanism.

system with bidirectional channels, this message must be stored only for a time equal to the maximum round trip delay between the node and its neighbors. At the other extreme, in the dynamic variant of our algorithm with multiple message sources, each server must maintain a vector of identifiers for the active (outgoing and incoming) channels and, for each incoming channel, a vector containing the messages possibly being buffered. The size of the latter is unbounded, but each message must be kept in the vector only for a time proportional to the diameter of the network.

5.2.2 Applicability

It is evident that the algorithm presented in this work generates considerable overall traffic overhead if compared, for instance, to a forwarding scheme. This is a consequence of the fact that our technique involves contacting the nodes in the network that have been visited by at least one agent in order to find the message recipient, and thus generates an amount of traffic that is comparable to a broadcast. Unfortunately, this price must be paid when both guaranteed delivery and frequent, unconstrained agent movement are part of the application requirements, since simpler and more lightweight schemes do not provide these guarantees, as discussed in Chapter 3.3. Hence, the question whether the communication mechanism we propose is a useful addition to mobile agent platforms will be ultimately answered by practical mobile agent applications, which are still largely missing and will determine the requirements for communication.

In any case, we do not expect our mechanism to be the only one provided by the runtime support. To make an analogy, one does not shout when the party is one step away; one resorts to shouting under the exceptional condition that the party is not available, or not where expected to be. Our algorithm provides a clever way to shout (i.e., to broadcast a message) with precise guarantees and minimal constraints, and should be used only when conventional mechanisms are not applicable. The runtime support should leave to the programmer the opportunity to choose different communication mechanisms, and even different variants of our algorithm. For instance, the fully dynamic solution described in Section 5.1.2 is not necessarily the most convenient in all situations. In a network configuration such as

the one depicted in Figure 3.2, where the graph is structured in clusters of nodes, the best tradeoff is probably achieved by using our fully dynamic algorithm only for the "gateway" servers that sit at the border of each cluster, and a static algorithm within each cluster. Such an approach leverages off of the knowledge of the internal network configuration and the inherent network knowledge and broadcast capability of a local area network. Along the same lines, it is possible to exploit hybrid schemes. For instance, in the common case where the receipt of a message triggers a reply, bandwidth consumption can be reduced by encoding the reply destination in the initial message and using a conventional mechanism, as long as the sending agent is willing to remain stationary until the reply is received.

Figure 5.5 highlights some of the tradeoffs among our solutions and those discussed in Chapter 3.3. A fully reliable communication can only be provided by the modified snapshot delivery algorithms, but guaranteed delivery comes at the cost of increased traffic overhead to deliver a single message, and additional network information that must be maintained at each host. At the other extreme, forwarding exhibits a minimal traffic overhead for message delivery, namely the path from the message source to the home node and from the home node to the current location of the mobile unit, but the current location of each mobile unit must be maintained. In the case of frequent movement and infrequent communication, the cost of updating the location information may outweigh the limited overhead of delivery, especially as far as the traffic around the home node is concerned.

We are currently investigating further the tradeoffs of the various communication schemes by exploiting a communication package developed for the μCODE [68] mobile code toolkit. The package contains an implementation of the algorithms presented here, as well as of broadcasting and forwarding schemes. Hence, the application programmer can choose among the most appropriate message delivery schemes, and possibly different choices may coexist in the same application. The implementation enabled us to validate the feasibility of our approach, and will allow further exploration of its interplay with more conventional mechanisms.

5.2.3 Enhancements

In Chapter 3, we argued that the problem of reliable message delivery is inherently complicated by the presence of mobility even in the absence of faults in the links or nodes involved in the communication. In practice, however, these faults do happen and, depending on the execution context, they can be relevant. If this is the case, the techniques traditionally proposed for coping with faults in a distributed snapshot can be applied to our mechanism. For instance, a simple technique consists of periodically checkpointing the state of the system, recording the state of links, keeping track of the last snapshot, and dumping an image of the agents hosted. (Many systems already provide checkpointing mechanisms for mobile

agents.) This information can be used to reconcile the state of the faulty node with the neighbors after a fault has occurred.

A related issue is the ability not only to dynamically add nodes to the graph, but also to remove them. This could be used to model faults or to optimize the network to remove hosts which are not active in hosting agents. Alternately a host may request to be removed because it is no longer willing to host agents, e.g., because the mobile agent support is to be intentionally shut down. A simple solution consists of "short circuiting" the node to be removed, by setting the incoming channels of its outgoing neighbors to point to the node's incoming neighbors. However, this involves running a distributed transaction and thus enforces an undesirable level of complexity. In this work, we disregarded the possibility for a couple of reasons. First of all, while it is evident that the ability to add nodes dynamically enables a better use of the communication resources by limiting communication to the areas visited by agents, it is unclear whether a similar gain is obtained in the case of removing nodes, especially considering the aforementioned implementation complexity. Second, very few mobile agent systems provide the ability to start and stop dynamically the mobile agent runtime support: most of them assume that the runtime is started offline and operates until the mobile agent application terminates.

5.3 Concluding Remarks

This chapter demonstrated first how our design strategy of treating a mobile unit as a message can be applied to snapshot algorithms and mobile agent to derive an algorithm for reliably message delivery. The key contribution, however, lies in the extension of this algorithm to meet the needs of the environment, demonstrating the power of our strategy to guide the design of meaningful abstractions tailored to the needs of the mobile environment.

Chapter 6

Tracking Mobile Units for Dependable Message Delivery

While the previous two chapters address reliable message delivery by developing algorithms which search the entire network for the mobile unit, this chapter focuses on message delivery schemes based on tracking a mobile unit as it moves through the network. In this chapter, we start with the idea of employing diffusing computations proposed by Dijkstra and Scholten [23] and adapt it to message delivery applying the strategy of treating the mobile unit as a message. By equating the root node of the computation to the concept of a home agent from Mobile IP, and by replacing the messages of the computation with mobile units, the result is an algorithm which, instead of tracking a computation as messages are passed through a system of processing nodes, tracks the movement of a mobile unit as it visits various base stations in the system. Essentially, the graph of the Dijkstra-Scholten algorithm defines a region within which the mobile unit is always located. Although this is not directly a message delivery algorithm, by propagating a message throughout this region, we can achieve message delivery. The algorithm can be readily adapted for this purpose and can be optimized for message delivery, e.g., our solution prunes unnecessary portions of the graph reducing the area to which a message must be propagated.

The remainder of this chapter is organized as follows: Section 6.1 explores the details of a message delivery algorithm derived directly from the Dijkstra-Scholten model for diffusing computations. Section 6.2 presents another algorithm inspired by the first, but reduces the message delivery overhead. For this algorithm, we provide a formal verification of its properties. Finally, Section 6.3 contains related work and analysis.

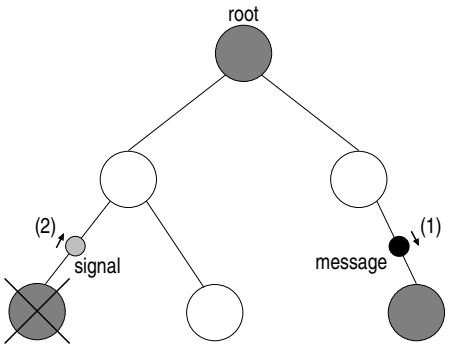


Figure 6.1: Dijkstra-Scholten for detecting termination of a diffusing computation. Shaded nodes are idle, white nodes are active.

6.1 Applying diffusing computations to mobile unit tracking

Diffusing computations have the property that the computation initiates at a single root node while all other nodes are *idle*. The computation spreads as messages are sent from *active* nodes. Dijkstra and Scholten [23] describe an algorithm for detecting termination of such computations. The basic idea is that of maintaining a spanning tree that includes all active nodes, as shown in Figure 6.1. A message sent from an active node to an idle node (message (1) in Figure 6.1) adds the latter to the tree as a child of the former. Messages sent among tree nodes have no effect on the structure but may activate idle nodes still in the tree. An idle leaf node can leave the tree at any time by notifying its parent (signal (2) in Figure 6.1). Termination is detected when an idle root is all that remains in the tree.

We adapt this tree maintenance algorithm to the mobile environment. A node is seen as active when the mobile unit is present. The resulting algorithm maintains a tree identical to Figure 6.1 with the mobile unit at an active node or on a channel leaving a tree node. This enables us to guarantee the continued existence of a path from the root to the mobile unit along tree edges. We use this property to develop a guaranteed message delivery algorithm. The latter is superimposed on top of the graph maintenance algorithm. To maintain the distinction between the data messages being delivered and any control messages used to effect the delivery, we will refer to the data message as an *announcement*. In this section, we first describe the details of the graph maintenance algorithm, then present the guaranteed data message, i.e., announcement, delivery algorithm. A short discussion and possible extensions follow.

State

$MobileAt_A$	boolean, *true*if mobile unit at A, initially *false*except at root
Parent(A)	the parent of node A, initially NULL
Children(A)	multiset of children of node A, initially $\emptyset$

Actions

MobileArrives$_A$(B) *;arrival at A from B*
  Effect:
    MobileAt$_A$:= *true*
    if Parent(A) $\neq$ NULL **then**
      send *signal*(A) to B
    else
      Parent(A) := B

SignalArrives$_A$(B) *;arrival at A from B*
  Effect:
    Children(A) := Children(A) $-$ {B}

SendMobile$_A$(B) *;mobile moves from A to B*
  Preconditions:
    MobileAt$_A$ and channel (A,B) exists
  Effect:
    MobileAt$_A$:= *false*
    Children(A) := Children(A) $\cup$ {B}

CleanUp$_A$(B) *;remove node A from tree*
  Preconditions:
    Children(A) = $\emptyset \wedge \neg$MobileAt$_A$
    Parent(A) = B
  Effect:
    send *signal*(A) to B
    Parent(A) := NULL

Figure 6.2: Diffusing computations adapted for tracking a mobile unit

6.1.1 Mobile tracking

Although the Dijkstra-Scholten algorithm can be easily described and understood, the distributed message passing nature of the algorithm leads to subtle complexities. The details of the algorithm can be found in Figure 6.2. Each action is one atomic step and we assume weak fairness in action selection. For the purposes of discussion, we assume that the mobile unit is initially located at the root and moves nondeterministically throughout the graph (Figure 6.2, operation MobileLeaves).

In the introduction of this section, we described an algorithm which maintains a tree structure with edges from parent to child. By the distributed nature of the environment, the sender of a message cannot know whether or not the destination node is already in the tree, and cannot know whether or not to add the destination as a child. Therefore, the tree structure is maintained with edges from child to parent (recorded in *Parent(A)* in Figure 6.2).

For detecting termination and removing nodes from the tree, a node must be able to detect when it is an idle leaf node. This is done by tracking each message sent by each node. The Dijkstra-Scholten algorithm requires that every message be acknowledged by the destination with a *signal*. If the message arrives and the destination node is already part of the tree, the spanning tree topology does not change and the *signal* is sent immediately. Otherwise the *signal* is delayed and sent when the destination node removes itself from the

tree. The source node tracks all messages by destination in a multiset or bag. Nodes in this bag indicate children nodes of the spanning tree, nodes to which the message has not arrived, or nodes from which the signal has been sent but not yet received. When the bag is empty, the node has no children and can remove itself from the tree by signaling its parent. For detecting termination of a diffusing computation, it is only necessary to keep a count of the number of successors. Because we intend to use this information during announcement delivery, we must maintain the bag of children.

Similar processing must occur in the mobile setting. Each movement of the mobile unit is tracked in a multiset (e.g., *Children(A)*). An element is removed from this multiset when the node receives a signal (Figure 6.2, operation SignalArrives). A signal is sent immediately when the mobile unit arrives and the node is already part of the tree (Figure 6.2, MobileArrives) and is delayed otherwise. A delayed signal is released when the node becomes a leaf to be removed from the tree (Figure 6.2, CleanUp).

6.1.2 Superimposing announcement delivery

Having described the graph maintenance algorithm, we now present an algorithm to guarantee at-least-once delivery of an announcement. The details of this are shown in Figure 6.3 as actions superimposed on the graph maintenance actions of Figure 6.2. Actions with the same label execute in parallel while new actions are fairly interleaved with the existing actions.

For announcement delivery we assume that the announcement originates at the root and we rely on the property that there is always a path from the root to the mobile unit along edges in the tree. We note that the reverse edges of the tree (from parent to child) are a subset of the edges from parent to child maintained as successors of the parent (e.g., *Children(A)*). It is only necessary to send the announcement along edges in the spanning tree. But, because this tree is maintained with pointers from child to parent, the announcement must be propagated along the successor edges, from parent to child. When an announcement arrives from a source other than the parent, the announcement is rejected (Figure 6.3, AnnouncementArrives). In this manner, the announcement is only processed along the tree paths. Effectively, a frontier of announcements sweeps through the spanning tree. When the announcement and the mobile unit are co-located at a node, the announcement is delivered (Figure 6.3, AnnouncementArrives, MobileArrives).

In a stable environment where the mobile unit does not move, this announcement passing is sufficient to guarantee delivery. However if the mobile unit moves from a node in the tree below the frontier to a node above the frontier, delivery may fail. Therefore, each node stores a copy of the announcement until delivery is complete or the node is removed from the tree (Figure 6.3, AnnouncementArrives). Storing the announcement in this manner ensures that the mobile unit cannot move to a region above the frontier without

State
⟨same as before⟩
$AnnouncementAt_A$ boolean, true if announcement stored at A, initially false everywhere
started boolean, true if delivery has started, initially false

Actions

MobileArrives$_A$(B)
;arrival at A from B
Effect:
⟨same as before⟩
if AnnouncementAt$_A$ **then**
deliver announcement
send *ack* to Parent(A) and children(A)

SignalArrives$_A$(B)
;arrival at A from B
⟨same as before⟩

SendMobile$_A$(B)
;moves from A to B
⟨same as before⟩

AnnouncementArrives$_A$(B)
;arrival at A from B
Effect:
if Parent(A) = B **then**
if MobileAt$_A$ **then**
deliver announcement
send *ack* to B
else
AnnouncementAt$_A$:= *true* *;save ann.*
send *announcements* to children(A)

CleanUp$_A$(B)
;remove node A from tree
Preconditions:
⟨same as before⟩
Effect:
⟨same as before⟩
AnnouncementAt$_A$:= *false ;delete ann.*

AckArrives$_A$(B) *;arrival at A from B*
Effect:
if Parent(A)=B ∨ B ∈ Children(A)
if AnnouncementAt$_A$ **then**
AnnouncementAt$_A$:= *false ;delete ann.*
send *acks* to children(A) except B

AnnouncementStart
;root sends announcement
Preconditions:
started = *false*
Effect:
started := *true*
if MoibleAt$_{\text{root}}$ = *true***then**
deliver announcement
else
AnnouncementAt$_{\text{root}}$:= *true*
send *announcement* to children(root)

Figure 6.3: Announcement delivery on top of diffusing computations.

receiving a copy of the announcement. Because there is always a path from the root to the mobile unit, there must be an announcement on the frontier traversing this path and the announcement will eventually reach the mobile unit thus leading to delivery (Figure 6.3, MobileArrives). This path may change as the mobile unit moves from one region of the tree to another, however, the existence of a path is guaranteed by the graph maintenance algorithm presented in the previous section and the existence of the announcement on this path is guaranteed by the delivery algorithm of this section.

In the worst case, it is possible for the mobile unit to continuously travel with the announcement on the channel exactly one step behind. Eventually the mobile unit must either stop moving when the maximum length path is reached (equal to the number of nodes in the system), or the mobile unit will return to a previously visited tree node. When the mobile unit returns to a tree node, which, by the assumptions, must be above the frontier of announcements, it will receive the announcement stored there.

Storing the announcement requires an additional cleanup phase to remove all copies. When the mobile unit receives the announcement, an acknowledgement is generated and sent along the successor and parent edges (Figure 6.3, AnnouncementArrives, MobileArrives). As before, the acknowledgment is rejected along paths which are not part of the tree (Figure 6.3, AckArrives). The connectivity of the tree ensures that the acknowledgement will propagate to all nodes holding copies of the announcement. Leaf nodes being removed from the tree must also delete their copy of the announcement (Figure 6.3, CleanUp).

This algorithm ensures at-least-once delivery of the announcement. Because the announcement copies remain in the graph until an acknowledgment is received, it is possible for the mobile unit to move from a region where the acknowledgments have propagated to a region where they have not. When this occurs, the mobile unit will receive an additional copy of the announcement, which it can reject based on sequence numbers. It is important to note that each time delivery occurs, a new set of acknowledgments will be generated. It can be shown that these acknowledgments do not inhibit the clean up process, but rather lead to a faster clean up. Each set of acknowledgments spreads independently through the tree removing announcement copies, but terminates when a region without announcement copies is reached.

6.1.3 Discussion

By superimposing the delivery actions on top of the graph maintenance, the result is an algorithm which guarantees at least once delivery of an announcement while actively maintaining a graph of the system nodes where the mobile unit has recently traveled.

It is not necessary for the spanning tree be pruned as soon as an idle leaf node exists. Instead this processing can be delayed until a period of low bandwidth utilization.

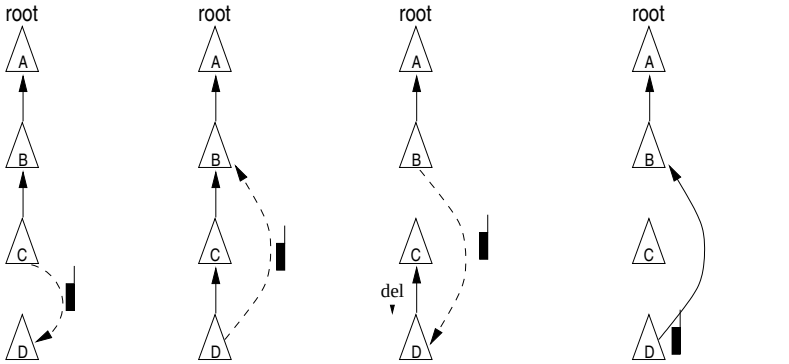
An application may benefit by allowing the construction of a wide spanning tree within which the mobile units travels. Tradeoffs include shorter paths from the root to the mobile unit versus an increase in the number of nodes involved in each announcement delivery.

By constructing the graph based on the movement of the mobile unit, the path from the root to the mobile unit may not be optimal. Therefore, a possible extension is to run an optimization protocol to reduce the length of this path. Such an optimization must take into consideration the continued movement of the mobile unit as well as any announcement deliveries in progress. The tradeoff with this approach is between the benefit of a shorter route from the root to the mobile unit and the additional bandwidth and complexity required to run the optimization.

Although in our algorithm only one mobile unit is present, the graph maintenance algorithm requires no extensions to track a group of mobile units. The resulting spanning tree can be used for unicast announcement delivery without any modifications and for multicast announcement delivery by changing only the announcement clean up mechanism. As presented, the delivery of the announcement triggers the propagation of acknowledgments. In the multicast case, it is possible for the announcement not to reach all mobile units before the cleanup starts. One practical solution is to eliminate the cleanup rules entirely, and assign a timeout to the announcement. This timeout should be proportional to the time it takes for the announcement to traverse the diameter of the network.

6.2 Backbone

We now introduce a new tracking and delivery algorithm inspired by the previous investigation with diffusing computations. Our goal is to reduce the number of nodes to which the announcement propagates, and to accomplish this we note that only the path between the root and mobile unit is necessary for delivery. In the previous approach, although the parts of the graph not on the path from the root to the mobile unit can be eliminated with *remove* messages, announcements still propagate unnecessarily down these subtrees before the node deletion occurs. To avoid this, the algorithm presented in this section maintains a graph with only one path leading away from the root and terminating at the mobile unit. This path is referred to as the *backbone*. The nodes in the remainder of the graph form structures referred to as *tails* and are actively removed from the graph, rather than relying on idle leaf nodes to remove themselves. Maintenance of this new structure requires additional information to be carried by the mobile unit regarding the path from the root, as well as the addition of a *delete* message to remove tail nodes. The announcement delivery mechanism remains essentially the same as before, but the simpler graph reduces the number of announcement copies stored during delivery.



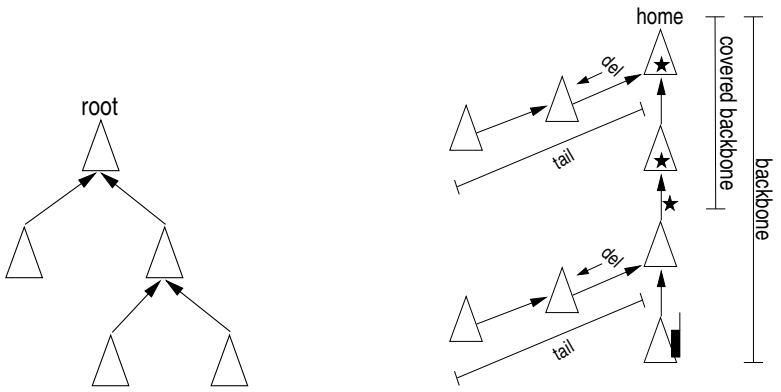
(a) Backbone extended (b) Backbone shortened (c) Tail node added (d) After movement completes

Figure 6.4: The parent pointers of the backbone change as the mobile moves to (a) a node not in the backbone, (b) a node higher in the backbone, and (c) a tail node. (d) shows the state after all channels have been cleared.

To understand how the backbone is kept independent of the tails, we examine how the graph changes as the mobile unit moves. It is important to note that by the definition of the backbone, the mobile unit is always either at the last node of the backbone, or on a channel leading away from it. In Figure 6.4a, the backbone is composed of nodes A, B, and C and the dashed arrow shows the movement of the mobile unit from node C to D where D is not part of the graph. This is the most straightforward case in which the backbone is extended to include D by adding both the child pointer from C to D (not shown) and the parent pointer in the reverse direction (solid arrow in Figure 6.4b).

In Figure 6.4b, the mobile moves to a node B, a node already in the backbone and with a non-null parent pointer. It is clear from the figure that the backbone should be shortened to only include A and B without changing any parent pointers, and that C and D should be deleted. To explicitly remove the tail created by C and D, a delete message is sent to the child of B. When C receives the delete from its parent, it will nullify its parent pointer, propagate the delete to its child, and nullify its child pointer.

If at this point the mobile moves from B onto D before the arrival of the delete (See Figure 6.4c), D still has a parent pointer (C) and we cannot distinguish this case from the previous case (where B also had a non null parent pointer). In the previous case the parent of the node the mobile unit arrived at did not change, but in this case, we wish to have D's parent set to B (the node the mobile unit is arriving from) so that the backbone is maintained. To distinguish these two cases, we require the mobile unit to carry a sequence of the identities of the nodes in the backbone. In the first case where the mobile unit arrives at B, B is in the list of backbone nodes maintained by the mobile unit, therefore B keeps its parent pointer unchanged, but prunes the backbone list to remove C and D. However, when



(a) Sample diffusing computation (b) Modified graph showing new structure.

Figure 6.5: By adapting diffusing computations to mobility, we construct a graph reflecting the movement of the mobile. In order to deliver an announcement, the only part of the graph we need is the path from the root to the mobile, the *backbone*. Therefore we adapt the Dijkstra-Scholten algorithm to maintain only this graph segment and delete all the others.

the mobile arrives at D, only A and B are in the backbone list, therefore the parent pointer of D is changed to point to B. But, what happens to the delete message moving from C to D? Because C is no longer D's parent when the delete arrives, it is simply dropped and the backbone is not affected.

The delivery algorithm is then superimposed on top of the generated graph. It is not sufficient to send the announcement down the spanning tree created by the backbone without keeping copies at all nodes along the path because the mobile is free to move from a region below the announcement to one above it (as in Figure 6.4b, assuming the announcement had propagated to C but not to D). Therefore, to guarantee delivery, as the announcement propagates down the backbone, a copy is stored at each node until delivery is complete. We refer to the portion of the backbone with an announcement as the *covered backbone*, see Figure 6.5b. Delivery can occur by the mobile unit moving to a location in the covered backbone, or the announcement catching up with the mobile unit at a node. In either case, an acknowledgment is generated and sent via the parent pointers toward the root. If the announcement is delivered by the mobile unit moving on to the covered backbone, a delete is generated toward the child and an acknowledgment is generated toward the parent. Therefore any extra copies of the announcement on the newly created tail will be deleted with the nodes.

State

$AnnouncementAt_A$	boolean, true if announcement stored at A, initially false everywhere
$MobileAt_A$	boolean, true if mobile unit at A, initially false except at root
Parent(A)	the parent of node A, initially NULL
Child(A)	the child of node A, initially NULL
started	boolean, true if delivery has started, initially false
MList	list of nodes carried by the mobile, initially contains only the root

Actions

AnnouncementArrives$_A$(B)
;*arrival at A from B*
Effect:
if Parent(A)=B **then**
if MobileAt$_A$ **then**
deliver announcement
send *ack* to B
else
AnnouncementAt$_A$:=*true* ;*save ann.*
send *announcement* to Child(A)

MobileArrives$_A$(B)
;*arrival at A from B*
Effect:
MobileAt$_A$:=*true*
if A ∈ MList **then**
A keeps old parent
MList truncated after A to the end
if AnnouncementAt$_A$ **then**
deliver announcement
Send *ack* to Parent(A)
AnnouncementAt$_A$:=*false*;*delete ann.*
else
Parent(A):=B
MList := MList ∘ A
AnnouncementAt$_A$:=*false* ;*delete ann.*
send *delete* to Child(A)
Child(A):=NULL

AckArrives$_A$(B)
;*arrival at A from B*
Effect:
if Child(A)=B ∧ AnnouncementAt$_A$ **then**
AnnouncementAt$_A$:=*false* ;*delete ann.*
send *ack* to Parent(A)

SendMobile$_A$(B)
;*moves from A to B*
Preconditions:
MobileAt$_A$ and channel (A,B) exists
Effect:
MobileAt$_A$:=*false*
Child(A):=B

DeleteArrives$_A$(B)
;*arrival at A from B*
Effect:
if Parent(A)=B **then**
if AnnouncementAt$_A$ **then**
AnnouncementAt$_A$:=*false* ;*delete ann.*
send *delete* to Child(A)
Parent(A):=NULL
Child(A):=NULL

AnnouncementStart
;*root initiates ann.*
Preconditions:
started = *false*
Effect:
started:=*true*
if MoibleAt$_{\text{root}}$=*true***then**
deliver announcement
else
AnnouncementAt$_{\text{root}}$:=*true*
send *announcement* to Child(root)

Figure 6.6: Tracking and delivery algorithm derived using some initial ideas from termination detection

6.2.1 Details

The details for the tracking algorithm are shown in Figure 6.6. As before, we model arbitrary movement of the mobile by an action, called SendMobile$_A(B)$, that allows a mobile at a node to move non-deterministically onto any outgoing channel.

MobileArrives shows the bulk of the processing and relates closely to the actions described in Figure 6.4. When the mobile unit arrives at a node, the changes to be backbone must be determined. If the mobile is doubling back onto the backbone, the parent pointers remain unchanged and the path carried by the mobile is shortened to reflect the new backbone (as in Figure 6.4b). If the node is not in the backbone (Figure 6.4a) or is part of a tail (Figure 6.4c), then the parent pointers must change to add this node to the backbone, and the node must be appended to the backbone list carried by the mobile. In any case, the children of this node (if any) are no longer necessary for announcement delivery, therefore a delete message is sent to the child, and the child pointer is cleared.

In addition to maintaining the graph, we must also address announcement delivery. As in the previous algorithms, when the mobile unit arrives at a node where the announcement is stored, delivery occurs, yielding *at-least-once* semantics for delivery. In this algorithm, we introduce a sequence number to ensure *exactly-once* delivery semantics. Therefore, when the mobile arrives at a node with the announcement, delivery is attempted if the sequence number of the last announcement received by the mobile is less than the sequence number of the waiting announcement. In all cases (whether or not delivery was just accomplished), at this point the announcement has been delivered to the mobile unit and an acknowledgment is generated along the path toward the root to clean up the announcement copies. No acknowledgment needs to be generated toward the tails because any announcement copies on tails will be removed at the same time the tail node is removed from the graph.

When the propagating announcement arrives at a node, AnnouncementArrives, it is either arriving from a parent or some other node. If the announcement arrives from a node other than the parent, it should be discarded because to guarantee delivery the announcement need only propagate along the backbone. However, when an announcement arrives from the parent it must be processed. If the mobile is present, delivery is attempted with the same restrictions as before with respect to the sequence number and the acknowledgment is started toward the root. If the mobile is not present, the node stores a copy of the announcement in case the mobile arrives at a later time. Additionally, the announcement is propagated to the child link.

AckArrives enables the cleanup of the announcements by propagating acks along the backbone toward the root via the parent pointers. Acks can also be present on tail links, but these are essentially redundant to the delete messages and do not affect the correctness of the algorithm.

The purpose of the delete messages is to remove the tail segments of the graph. Recall that a tail is created by a backbone node sending a delete to its child. Therefore, a delete should only arrive from a parent node. If we were to accept a delete from a non-parent node, as in the delete from C to D in Figure 6.4c, we could destroy the backbone. However if the delete arrives from the parent, we are assured that the node no longer resides on the backbone and should be deleted. Therefore, the arrival of a delete from a parent triggers the deletion of the stored announcement, the propagation of the delete to the child, and the clearing of both child and parent pointers.

6.2.2 Discussion and Generalizations

Keeping the backbone sequence is a similar methodology to routing protocols passing complete paths to the destination as in BGP [73] to avoid loops. It has been argued that keeping such information in the packet greatly increases its size. However, in our case, the information is being kept by the mobile unit and we assume there is sufficient storage on such a device for this additional information.

A simple extension of this algorithm is to allow for multiple concurrent announcement deliveries as in sliding window protocols. The announcements and all associated acknowledgments would have to be marked by sequence numbers so that they do not interfere, but the delivery mechanism uses the same graph. Therefore the rules governing the expansion and shrinking of the graph are not affected but the proofs of garbage collection and acknowledgment delivery are more delicate.

6.2.3 Correctness

Because this algorithm deviates significantly from the original Dijkstra-Scholten model of diffusing computations, essential properties necessary for announcement delivery are proven in this section: 1) announcement delivery is guaranteed, 2) after delivery announcement copies are eventually removed from the system, and 3) any tail node is eventually cleared. Although the third property is not essential to announcement delivery, it is necessary to show announcement cleanup.

Before approaching the proof, we formalize several useful definitions in Figure 6.7[1]. The most important of these are the backbone, covered backbone, and tails. Intuitively, the backbone is the sequence of nodes starting at the root and terminating at either the

[1]The three-part notation used in the equations of the figure $\langle op\ quantified\text{-}variables : range :: expression \rangle$ used throughout the text is defined as follows: The variables from *quantified-variables* take on all possible values permitted by *range*. If *range* is missing, the first colon is omitted and the domain of the variables is restricted by context. Each such instantiation of the variables is substituted in *expression* producing a multiset of values to which *op* is applied, yielding the value of the three-part expression. If no instantiation of the variables satisfies *range*, the value of the three-part expression is the identity element for *op*, e.g., *true* when *op* is $\forall$.

D.1	$\text{reachable}(m,n) \equiv m = n$ $\quad \vee(n = \text{Child}(m) \wedge m = \text{Parent}(n))$ $\quad \vee\langle \exists m' :: \text{reachable}(m,m')$ $\quad\quad \wedge \text{reachable}(m',n)\rangle$	Node n is reachable from node m if there is a path from m to n where every channel on the path has the parent and child pointers of the channel endpoints pointing toward one another.
D.2	$\text{path}(p,n) \equiv n \in p$ $\quad \wedge\langle \forall m : m \in p :: \text{reachable}(m,n)$ $\quad\quad \vee \text{reachable}(n,m)\rangle$ $\quad \wedge\langle \forall i,j : 1 \leq i,j \leq \lvert p\rvert \wedge i \neq j :: p_i \neq p_j\rangle$	Path p includes node n and is an acyclic sequence of reachable nodes.
D.3	$\text{maxpath}(p,n,R) \equiv \text{path}(p,n)$ $\quad \wedge\langle \forall m : \text{path}(m \circ p, n) :: \neg R(m \circ p)\rangle$ $\quad \wedge\langle \forall m : \text{path}(p \circ m, n) :: \neg R(p \circ m)\rangle$ $\quad \wedge R(p)$	Path p is the maximal length path including node n that satisfies the predicate R. Extending p in either direction through concatenation ($\circ$) either violates the path relationship or the condition R.
D.4	$\text{backbone}(p) \equiv \text{maxpath}(p, \text{root},$ $\quad \lambda q.\langle \forall m,n:\ m \in q$ $\quad\quad \wedge n \in q$ $\quad\quad \wedge n = \text{Parent}(m)$ $\quad\quad :: \text{MOB} \notin \text{Chan}(n,m)\rangle)$	Path p is the backbone, i.e. the path of maximal length which includes the root and does not include the mobile unit on any channel. The constant MOB is used to identify the mobile unit.
D.5	$\text{coveredBone}(p) \equiv \text{maxpath}(p, \text{root},$ $\quad \lambda q.\langle \forall m:\ m \in q$ $\quad\quad :: \text{AnnouncementAt}(m)\rangle)$	Path p is the covered backbone, i.e. the maximal length path including the root (the backbone) where all nodes are storing announcement copies.
D.6	$\text{tail}(p,n) \equiv\ \text{maxpath}(p,n,\lambda q.\langle \forall m : m \in q :: m \neq \text{root}\rangle)$	The tail is the maximal length path of any node n where no node on the path is part of the backbone.

Figure 6.7: Useful definitions for proving Dijkstra-Scholten message delivery.

node holding the mobile unit or the node the mobile unit just left if it is on a channel. The covered backbone is the sequence of backbone nodes with announcement copies. Tails are any path segments not on the backbone.

Announcement Delivery Guarantee

Our overall goal is to show at-least-once delivery of an announcement to a mobile unit. Therefore, the first property that we prove is (A) that from the state where no announcement exists in the system (*predelivery*), eventually a state is reached where the mobile unit has a copy of the announcement(*postdelivery*)[2]:

$$\text{predelivery} \mapsto \text{postdelivery} \tag{A}$$

[2]Progress properties are expressed using the UNITY relations $\mapsto$ (read *leads-to*) and **ensures**. Predicate relation $p \mapsto q$ expresses progress by requiring that if, at any point during execution, the predicate p is satisfied, then there is some later state where q is satisfied. Similarly, p **ensures** q states that if the program is in a state satisfying p, it remains in that state unless q is established, and, in addition, it does not remain forever in a state satisfying p but not q.

Although it is possible to make this transition in a single step (by executing AnnouncementStart while the mobile unit is at the root) it is more common for the system to move into an intermediate state where delivery is in progress (A.1). We must show that from this state (*delivery*), either the announcement will be delivered, or, in the worst case, the covered backbone will increase in length to include every node of the system (A.2). Once this occurs, delivery is guaranteed to take place when the mobile unit arrives at any node (A.3).

$$\text{predelivery } \textbf{ensures } \text{delivery} \vee \text{postdelivery} \tag{A.1}$$

$$\text{delivery} \mapsto \text{postdelivery} \vee (\text{delivery} \wedge \langle \exists \alpha :: \text{coveredBone}(\alpha) \wedge \langle \forall n : n \in N :: n \in \alpha \rangle \rangle) \tag{A.2}$$

$$\text{coveredBone}(\alpha) \wedge \text{delivery} \wedge \langle \forall n : n \in N :: n \in \alpha \rangle \textbf{ ensures } \text{postdelivery} \tag{A.3}$$

We approach each of these properties in turn, first showing that from *predelivery*, either *delivery* or *postdelivery* must follow (A.1). Until the action AnnouncementStart fires, the system remains in *predelivery* and AnnouncementStart remains enabled. Trivially, when it fires, either the announcement will be delivered (if the mobile unit is present at the root) or the announcement will begin to propagate through the system.

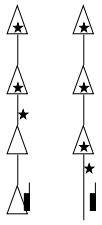
Once the delivery state is reached, we must show that the covered backbone will increase in length to include all nodes or the announcement will be delivered (A.2). To do this, we strengthen the progress property A.2 to state that the covered backbone cannot decrease in length.

$$\text{delivery} \wedge \text{coveredBone}(\alpha) \wedge k = |\alpha| < N \mapsto (\text{delivery} \wedge \text{coveredBone}(\alpha) \wedge |\alpha| > k) \vee \text{postdelivery} \tag{A.2.1}$$

In order to formally make this assertion, we must first show that during delivery the covered backbone exists. Showing the existence of the covered backbone independent from other system attributes is not possible. Therefore we prove a stronger invariant that not only establishes the existence of the covered backbone, but also the existence of the backbone and the relationship between the two. By definition, the covered backbone is a subset of the backbone. We further assert that if the covered backbone is shorter than the backbone, there is an announcement leaving the last node of the covered backbone (where last(α) returns the final element of the path α). Alternately, if the covered backbone and backbone are equivalent, the mobile unit must precede

the announcement (indicated by the constant ANN) in the channel leaving the last node.

$$\begin{aligned}\text{delivery} \Rightarrow \langle \exists \alpha, \beta, f : \ & \text{backbone}(\beta) \wedge \text{coveredBone}(\alpha) \wedge f = \text{last}(\alpha) :: \\ & (\alpha \subset \beta \wedge \text{ANN} \in \text{Chan}(f, \text{Child}(f))) \\ & \vee (\alpha = \beta \wedge \text{mobile.preceeds.ann}(f, \text{Child}(f)))\rangle \quad \text{(I.1)}\end{aligned}$$



This invariant is proven by showing that it holds initially as well as over all statements of the program. Throughout this proof, we use several supporting properties which appear in Appendix A. Specifically: Inv I.1.1 the integrity of the backbone, Inv I.1.2 that the backbone always exists, Inv I.1.3 that there is at most one announcement in a channel, Inv I.1.4 that there are no announcements during predelivery, and Inv I.1.5 that there are no acknowledgments during delivery. We now show the proof of the top level property concerning the existence of the covered backbone during delivery (I.1):

- It is trivial to show the initial conditions of I.1 because initially, *delivery* is false.

- MobileArrives$_A(B)$: We assume the integrity of the backbone (Inv I.1.1). First we consider the case where the system is in delivery and the right hand side of this invariant (I.1) holds. The covered backbone is not affected if the mobile unit arrives at a non-backbone node or a backbone node below the covered backbone. If the mobile unit arrives at a covered backbone node, the announcement is delivered and the invariant is trivially true by falsifying the left hand side.

  Next we consider when the system is not in delivery. If the system is in predelivery and we assume there are no announcements during predelivery (Inv I.1.4), the movement of the mobile unit cannot affect the delivery status. Once the system is in postdelivery, it cannot return to delivery, so the invariant remains true.

- AnnouncementArrives$_A(B)$: We assume there is at most one announcement on a channel in the system (Inv I.1.3). Therefore, if the system is in delivery and we assume this invariant is true before the announcement arrives, the announcement must be leaving the covered backbone. Further, since the announcement is at the head of the channel, it cannot be the case that the mobile unit and announcement are in the same channel, so the covered backbone must be a proper subsequence of the backbone. Therefore, by the definitions of the covered backbone and backbone, the node the announcement arrives at is on the backbone, and either the announcement is delivered or is propagated.

  If delivery occurs, this invariant is trivially satisfied by falsifying the delivery condition.

If the announcement is propagated into the next channel, then the covered backbone is extended by one node which has already been shown to be part of the backbone. The announcement is put onto the child link of this node, which by the backbone definition must be a channel on the backbone or backbone extension. The announcement must follow the mobile unit if the mobile unit is on the same channel.

As before, if the system is not in delivery, then the delivery status of the system cannot change with the execution execution of this statement.

- SendMobile$_A(B)$: Before the statement executes, the mobile unit must be at a node, otherwise the statement is a skip. Since we assume this invariant to be true, it must be the case that the covered backbone is a proper subsequence of the backbone. Therefore, when the mobile unit leaves the node, the backbone is only changed to include the new backbone extension, the covered backbone is not affected, and the invariant remains true.

- AnnouncementStart: We assume that if the system is in predelivery, there are no announcements in the system (Inv I.1.4). Therefore after this statement executes, either delivery occurs and Inv I.1 invariant is trivially true, or the announcement is placed at the root and on the outgoing link, establishing the right hand side of the invariant. If the system is in delivery or postdelivery, this statement is a skip.

- AckArrives$_A(B)$: We assume there are no acknowledgments in the system during delivery (Inv I.1.5), and therefore this statement is essentially a skip during delivery. This statement cannot change the delivery status, therefore, if the system is in predelivery or postdelivery, the invariant is trivially true.

- DeleteArrives$_A(B)$: We assume that delete messages do not affect the backbone (Inv I.1.1), therefore they will not affect the covered backbone, and this invariant will remain true. As before, this statement cannot change the delivery status.

This concludes the proof that during delivery, the covered backbone exists. We now show that the covered backbone must grow, as defined by property A.2.1. We note two specific cases that the system can be in with respect to the mobile unit and the announcement and show how either the covered backbone must increase or delivery will occur. The first case is where *the mobile unit and announcement are not on the same channel*. Since the system is in delivery, there cannot be an acknowledgment on the channel (Inv I.1.5). Since the announcement is on a backbone channel, there cannot be a delete on the channel (Inv I.1.1). The assumption is that the mobile unit is not on the same channel. This covers all possible message types that could precede the announcement on the channel, therefore the announcement must be at the head of the channel. So, in this case, the progress property A.2.1 which concerns the growth of the covered backbone becomes an ensures because

the announcement will remain at the head of the channel until processed, lengthening the covered backbone, or the mobile unit will arrive at a node causing delivery. In either case, the condition on right hand side becomes true.

In the second case, *the mobile unit and announcement are on the same channel.* By Invariant I.1, the mobile unit precedes the announcement in this channel. We state a trivial progress property that if the mobile unit is at the head of a channel, it is ensured to arrive at the destination node:

$$\text{mobile.at.head}(m, n) \textbf{ ensures } \text{MobileAt}(n) \qquad \text{(A.2.1.1)}$$

Because there is only one mobile unit, after the mobile unit is removed from the channel, either the system is taken out of delivery by the mobile unit receiving the announcement, or the system has been reduced to the first case where the mobile unit and announcement are not on the same channel.

The previous discussion effectively shows property A.2.1, namely that the covered backbone must grow until all nodes in the system are part of the covered backbone or delivery has occurred. To complete the proof that delivery is guaranteed, we need to show that when all nodes are part of the covered backbone, delivery *must* occur. By the definitions of the covered backbone and backbone, when all nodes are part of the covered backbone, the two are equivalent. The mobile unit must be on a channel because all nodes have announcement copies and if the mobile unit is at a node, it must have received the announcement copy (either when the mobile unit arrived or when the announcement arrived). The destination of the channel the mobile unit is on must be part of the backbone because all nodes are part of the backbone. If there is a delete in front of the mobile unit , it will not have any effect on the backbone (Inv I.1.1). There cannot be an acknowledgment in the channel (Inv I.1.5). The announcement must be behind the mobile unit (Inv I.1.1). Therefore, after the delete (if any) is processed, the mobile unit is at the head of the channel. The $\text{MobileArrives}_A(B)$ action will cause delivery. Therefore, announcement delivery is guaranteed from the initial state of the system.

Backbone announcements cleaned up

Once the announcement has been delivered, we show that eventually all stored announcement copies are removed. There are two cases to address: the announcements on nodes on the backbone and those not on the backbone. In the next section, we will show how all nodes which are not part of the backbone will be cleaned up, while this section focuses on the cleanup of the backbone nodes. In particular, we wish to show that after the announcement has been delivered, eventually all announcement copies on the backbone will be deleted.

$$\text{postdelivery} \mapsto \langle \forall m, \beta : \text{backbone}(\beta) \wedge m \in \beta :: \neg \text{AnnouncementAt}(m) \rangle \qquad \text{(B)}$$

We introduce a safety property describing the state of the backbone in postdelivery. Namely, (a.) the backbone and covered backbone exist, (b.) there is an acknowledgment in the channel heading toward the last node of the covered backbone, (c.) all nodes in the backbone not in the covered backbone do not have announcement copies, (d.) there are no announcement copies on any backbone channels or the backbone extension, and (e.) there are no acknowledgments on the channels of the covered backbone. Intuitively, this invariant shows that there is only one segment of the backbone with announcement copies and the nodes on this segment are poised to receive an acknowledgment.

$$\begin{aligned}\text{postdelivery} \Rightarrow \langle \exists \alpha, \beta : \ & \text{backbone}(\beta) \wedge \text{coveredBone}(\alpha) \wedge f_\alpha = \text{last}(\alpha) \wedge f_\beta = \text{last}(\beta) :: \\ & \alpha \subset \beta \wedge \textsc{ack} \in \text{Chan}(\text{Child}(f_\alpha), f_\alpha) \\ & \wedge \langle \forall m : m \in (\beta - \alpha) :: \neg \text{AnnouncementAt}(m) \rangle \\ & \wedge \langle \forall m, n : m, n \in \beta \wedge m = \text{Parent}(n) :: \textsc{ann} \notin \text{Chan}(m, n) \rangle \rangle \\ & \wedge \textsc{mob} \in \text{Chan}(f_\beta, \text{Child}(f_\beta)) \\ & \quad \Rightarrow \neg \text{mobile.preceeds.ann}(f_\beta, \text{Child}(f_\beta)) \\ & \wedge \langle \forall m, n : m, n \in \alpha \wedge n = \text{Child}(m) :: \neg \textsc{ack} \in \text{Chan}(n, m) \rangle \end{aligned} \qquad \text{(I.2)}$$

We now show the proof of this statement by showing that if it holds before the execution of each statement, it must hold after the execution of the statement:

- MobileArrives$_A(B)$: When the mobile unit arrives at a non-backbone node, the backbone is extended to include this node. The channel just traversed will become part of the backbone, but will not have an announcement on it by the last part of this invariant. The covered backbone will not change. If there is an announcement at this node, it will be removed so that there are still no announcement copies at nodes other than the covered backbone.

  If the mobile unit arrives at a backbone node that is not part of the covered backbone, the covered backbone does not change. There are still no announcements at backbone nodes other than the covered backbone, and because no new channels are added to the backbone, there are no announcement copies on the channels of the covered backbone.

  If the mobile unit arrives at a backbone node that is part of the covered backbone, the covered backbone is shortened to be all nodes above this new location of the mobile unit. Each of these nodes must have an announcement copy because they were part

of the covered backbone before the mobile unit arrived. Also, an acknowledgment is generated in the channel heading toward the new covered backbone and this invariant is established. As before, there are no new channels in the backbone, so there are still no announcements on any backbone channels.

We must also consider the case where the postdelivery condition is established by the arrival of the mobile unit. The components of this invariant are established because the only announcement in the system was at the end of the covered backbone which must be downstream from where the mobile unit arrived, so there are no announcements in backbone channels. The remainder of the invariant is established in a manner similar to the case where the system is in postdelivery and the mobile unit arrives at a covered backbone node.

- AnnouncementArrives$_A(B)$: If the system is in delivery, the arrival of the announcement could establish postdelivery. In this case, the components of this invariant are established because all nodes above the mobile unit are part of the covered backbone. The acknowledgment is put into the channel above the mobile unit, which is the end of the covered backbone. The announcement copy at the node the mobile unit is at is deleted.

  If the system is in postdelivery and the announcement arrives, the announcement could arrive at a backbone node only from a non-backbone channel and will be dropped. Therefore, the invariant will not be affected because neither the backbone nodes nor links are affected.

- SendMobile$_A(B)$: If the mobile unit leaves a node, the covered backbone does not change. Also, the mobile unit is at the end of the channel, so any announcements in the same channel must be before the mobile unit.

- DeleteArrives$_A(B)$: The arrival of a delete at a backbone node will not affect the backbone or the covered backbone. The arrival of a delete elsewhere in the system will not affect this invariant.

- AckArrives$_A(B)$: If an acknowledgment arrives at a covered backbone node, it must be at the end of the covered backbone. Therefore, the processing of this acknowledgment will shrink the covered backbone by one node and put the acknowledgment farther up in the backbone. Alternately, the root could receive the acknowledgment and there would no longer be a covered backbone.

  If an acknowledgment arrives at a non-backbone node and is accepted, it will not be put onto the backbone. If it is not accepted, nothing changes in the covered backbone or backbone, therefore the invariant is maintained.

- AnnouncementStart: This statement has no effect during postdelivery. During predelivery, this statement could establish this invariant by delivering the announcement to a node at the root. In this case, the covered backbone does not exist, and the invariant is true.

Our goal is to show progress in the cleanup of announcements on the covered backbone. To do this, we use a progress metric that measures the reduction in length of the covered backbone. Because the only nodes on the backbone with announcement copies must be on the covered backbone by Invariant I.2, once the covered backbone has length zero, all announcements on the backbone have been deleted.

$$\text{postdelivery} \wedge \text{coveredBone}(\alpha) \wedge |\alpha| = k \geq 1 \mapsto |\alpha| < k \tag{B.1}$$

To prove this statement, we note that by the previous invariants, it has been established that there is an acknowledgment in the channel heading toward the covered backbone. If the acknowledgment is not at the head of the channel, then there must be something else in front of it. There cannot be a delete on the channel, because that would mean there is a delete on the backbone which is not allowed by invariant I.1.1. An announcement would have no effect because it is not arriving on a parent link. If the mobile unit were on the channel, then the arrival of the mobile unit would cause delivery because the announcement must be at the last node of the covered backbone, and the covered backbone would change.

So, either the mobile unit will arrive from the same channel as the acknowledgment or on another channel and will cause the covered backbone to shrink, or the acknowledgment will be processed and cause the covered backbone to shrink. Since there are only a finite number of messages in the channel in front of the acknowledgment, these will be processed and eventually either the acknowledgment will reach the head of the channel or the covered backbone will shrink in another way (through the arrival of the mobile unit at a covered backbone node).

When the covered backbone shrinks to zero length, there will be no more announcement copies on any backbone nodes, accomplishing backbone cleanup.

Tail Cleanup

In addition to backbone cleanup, we must also ensure that any announcement copies not on the backbone will eventually be deleted. More precisely, any node which is on a tail will eventually be cleared or put on the backbone (C), where clear(n) indicates that n's parent and child pointers are NULL (which will be shown to imply the announcement is no longer stored there). Since a node can only accept an announcement from a parent, this implies that only nodes with non-null parent pointers could have announcement copies. Since a

node can only clear its pointers at the same time as it clears its storage, there is no way for a node (other than the root) to have a copy of the announcement and non-null pointers.

We cannot guarantee that the mobile unit will eventually arrive at the node thereby adding that node to the backbone, we must prove that there is a delete message that will eventually arrive at the node if it remains on the tail. We show that every tail has a delete message on the channel heading toward the first node of the tail (I.3), where the *first* node of the tail is defined to be the node whose parent pointer points toward a node that does not point toward it as the child. This delete message will eventually be processed, shrinking the length of the tail (C.1). When the tail contains only node, the tail is guaranteed to be cleared (C.2).

$$\text{tail}(\tau, n) \mapsto \text{clear}(n) \vee \langle \exists \beta : \text{backbone}(\beta) :: n \in \beta \rangle \tag{C}$$

$$\begin{aligned} &\langle \exists \tau : \text{tail}(\tau, n) \wedge (n \neq \text{first}(\tau)) \wedge (|\tau| = k) \wedge (|\tau| > 1) \rangle \\ &\qquad \mapsto \langle \exists \tau : \text{tail}(\tau, n) \wedge |\tau| < k \rangle \vee \langle \exists \beta : \text{backbone}(\beta) :: n \in \beta \rangle \end{aligned} \tag{C.1}$$

$$\langle \exists \tau : \text{tail}(\tau, n) \wedge |\tau| = 1 \rangle \textbf{ ensures } \text{clear}(n) \vee \langle \exists \beta : \text{backbone}(\beta) :: n \in \beta \rangle \tag{C.2}$$

To show that the tail can shrink, we must guarantee the existence of the delete message at the end of the tail. We do this by assuming the invariant before each statement execution and showing it holds after statement execution.

$$\langle \forall n, \tau : \text{tail}(\tau, n) \wedge n = \text{first}(\tau) :: \text{DEL} \in \text{Chan}(\text{Parent}(n), n) \rangle \tag{I.3}$$

- MobileArrives$_A(B)$: If the mobile unit arrives at a backbone node, one or zero tails are created. If no tails are created, the invariant trivially holds. If a tail is created, it consists of the nodes that are removed from the backbone. These nodes by definition point toward one another as parent and child, making them a tail. No other tails are affected. The new tail by definition has a first node. The first node of the new tail is the node formerly pointed to as the child of the node where the mobile unit is currently at. A delete is put onto this channel, establishing this invariant.

  If the mobile unit arrives at a node that is not on the backbone and not on a tail, no new tails are created, no deletes are sent, and no old tails are affected.

  If the mobile unit arrives at a tail node, the tail is cut into two segments around the mobile unit. The nodes above the mobile unit are not affected because the first node of the tail is still the same and the delete is not affected. The nodes below the mobile unit are similar to the first case, and the delete generated down the old child pointer establishes the invariant.

- DeleteArrives$_A(B)$: If a delete arrives at any node on a link other than from the parent, this delete could not be critical to any tail, and therefore dropping it has no effect on the invariant.

  If a delete arrives at the first node of a tail along the parent link, the delete is propagated to the new first node of the tail and a node is removed from the tail.

- AnnouncementArrives$_A(B)$: SendMobile$_A(B)$: AckArrives$_A(B)$: AnnouncementStart do not affect the invariant

With this invariant, it is clear that when the delete is processed, a node is removed from the tail, and the tail shrinks (property C.1). If the delete is not at the head of the channel, the messages ahead of it must be processed. Neither an acknowledgment nor an announcement will affect progress. If a mobile unit arrives, the node is added to the backbone, satisfying the progress condition.

Finally, we formally define clear (D.7), then show that if a node is clear, it has no announcement copies (I.3.1):

$$\text{clear}(n) \equiv \text{Parent}(n) = \text{Child}(n) = \text{NULL} \tag{D.7}$$

$$\text{clear}(n) \Rightarrow \neg\text{AnnouncementAt}(n) \tag{I.3.1}$$

This invariant is easily shown over every statement. Intuitively, when a node sets both its parent and child pointers to NULL, as in DeleteArrives$_A(B)$, the announcement copies at the node are deleted. Since it is not possible to set the child and parent pointers to NULL any other way, and an announcement is only accepted from a non-NULL parent link, there cannot be an announcement at a node that has both NULL pointers.

Therefore, once a node is either clear or put back on the backbone, it will not have an announcement copy. As the tails shrink, we are guaranteed that the announcements not on the backbone will be removed from the system.

6.3 Discussion

We have described two algorithms to guarantee the delivery of an announcement to a mobile unit with no assumptions regarding the speed of movement. In this section, we compare our approach with other tracking based delivery schemes designed for the mobile environment including Mobile IP, a scheme by Sony, another by Sanders et. al., and finally a multicast scheme by Badrinath et. al.

Each of these algorithms uses the notion of a home node toward which the announcement is initially sent. In Mobile IP [65], the home node tracks as closely as possible the current location of the mobile unit and all data is sent from the home directly to this

location. This information is updated each time the mobile unit moves, introducing a discrepancy between the actual location and the stored location during movement. Any data sent to the mobile unit during this update will be dropped. Mobile IP has no mechanism to recover this data, but rather assumes that higher layer protocols such as TCP will handle buffering and retransmitting lost data. One proposal within Mobile IP is to allow the previous location of the mobile unit to cache the new location and forward data rather than dropping it. While this can reduce the number of dropped announcements, it still does not guarantee delivery as the mobile unit can continue to move, always one step ahead of the forwarded announcement.

One proposal is to increase the amount of correct location information in the system by distributing this information to multiple routers, as in our tree and backbone maintenance algorithms. The Sony [59] approach keeps the home node as up to date as possible, but also makes the other system routers active components, caching mobile unit locations. As the packet is forwarded, each router uses its own information to determine the next hop for the packet. During movement and updating of routing information, the routers closer to the mobile unit will have more up to date information, and fewer packets will be lost than in the Mobile IP approach. This approach still does not provide delivery guarantees, and few details are given concerning updates to router caches. One benefit is that announcements need not be sent all the way to the home node before being forwarded toward the mobile unit. Instead any intermediate router caching a location for the mobile unit can reroute the packet toward the mobile unit.

The Sanders approach [78] has this same advantage, allowing intermediate routers to forward the announcement. Sanders describes precisely how intermediate routers are updated. A hop by hop path is kept from the home node to the last known location of the mobile unit. When the mobile unit moves, the path is shortened one node at a time until the common node between the shortest path to the old location and the shortest path to the new location is reached, then the path is extended one node at a time. Any announcements encountering the hop by hop path are forwarded toward the last node of this path. During the updating of this path, announcements move with the update message which are changing the path and will eventually reach the next known location of the mobile unit. However, the mobile unit may have moved during the update, in which case, the data messages will continue to travel with the next update message. Although no messages are dropped, the slow update time and ability of the mobile unit to keep moving could prevent delivery.

In each of these approaches, a single copy of the announcement is kept in the system, while our approach stores multiple copies throughout the system until delivery is complete. We believe that sacrificing this storage for the limited times that our algorithms require is worthwhile to provide guaranteed delivery of the announcement. If we weaken these

requirements, our approaches can be modified to reduce storage. In the first algorithm, the announcement can be sent down the spanning tree in a wave. When the announcement arrives at a node, if the mobile unit is present, it is delivered, otherwise it is sent on all outgoing spanning tree channels. Although multiple copies are generated, they will not be stored, but simply passed to the next hop. When the announcement reaches a leaf node it will be dropped. If the mobile unit remains in a region of the graph below the announcement propagation, it will receive the message, however if it is in transit or moves to a region above the message propagation, the announcement will not be delivered. In our second approach, a single announcement copy can be sent down the backbone path. Even if the announcement ends up on a tail, it will continue toward the mobile unit. Because the path we define to the mobile unit is based on movement pattern rather than shortest path as in the Sanders algorithm, there is only one pathological movement pattern (a figure eight crossing the backbone) where the mobile unit can continue to avoid delivery.

Another approach which keeps multiple announcement copies is Badrinath's guaranteed multicast algorithm [5] which stores announcement copies at all system nodes until all mobile units in the multicast group have received the announcement. This information is gathered by the announcement initiator from the nodes that actually delivered the announcement. A disadvantage of this algorithm is that all recipients must be known in advance, a property not always known in multicast. Our first algorithm can trivially be extended to track the movement of multiple mobile units, and because it is based on the actual movement of the mobile units can reduce the number of nodes involved in the multicast delivery with respect to the Badrinath approach.

6.4 Concluding Remarks

The primary contribution of this chapter is its application of the algorithm design strategy of Chapter 3 using the Dijkstra-Scholten diffusing computations to develop two algorithms for message delivery to mobile units. The first is a direct derivative of the diffusing computations distributed algorithm, and the second is an optimizing refinement of the first based on careful study of the problem and the solution. Each algorithm is applicable in a variety of settings where mobile computing components are used and reliable communication is essential.

Chapter 7

Global Virtual Data Structures: A design strategy for the development of ad hoc mobility abstractions

The previous chapters show how the development of abstractions in nomadic and logical mobile computing can simplify programming in these environments by taking care of many of the complexities of the environment on behalf of the programmer. The ad hoc computing environment is another complex domain which can benefit from the development of abstractions to simplify the programming task.

Ad hoc mobility distinguishes itself from base station mobility by completely removing the fixed infrastructure, leaving only direct communication among hosts. In a wireless ad hoc network, the distance between components determines connectivity. As components move, the system is continuously reshaped into multiple partitions, with connectivity available within each partition but not across partitions.

Freeing mobile users from a fixed infrastructure makes the ad hoc network model ideal for many scenarios such as systems of small components with limited resources to spend on communication, situations in which the infrastructure has been destroyed such as following a natural disaster, and for settings in which establishing an infrastructure is impossible as in a battlefield environment or economically impractical as in a short duration meeting or conference.

The application needs in these scenarios can be classified broadly by how they interact with their changing environment, or *context*. The context of a mobile unit consists of two primary components: system configuration and data. System configuration context

describes the knowledge about which mobile units are connected and possibly topology information concerning physical location in space or logical connectivity. This knowledge is limited to the current partition of the network in which the mobile unit finds itself. We refer to this as the current transient community. Because communication cannot extend beyond the community, knowledge of configuration beyond the boundaries of the community is not possible. Data context refers to the more passive data elements and resources which are carried by the mobile components.

This view of context fosters two distinct programming styles: *context aware programming* and *context transparent programming*. Context aware applications are those which access both the system configuration context and the data context explicitly. For example, a context aware application may store a piece of new data on a specific mobile host, or retrieve a piece of data from a named mobile host. All operations must be carried out within the current connectivity context, but this style is distinguished by the needs of the application to be aware of the current context. In contrast, context transparent applications can be developed without explicit knowledge of the current context. Data access is performed on the data in the current context without regard to where it is located. Such applications do not need to be aware of the details of the configuration changes, but simply aware that they are occurring and that these changes affect the available resources. Many applications require a combination of both context aware and context transparent programming.

Our goal is to enable the rapid and dependable development of both styles of application programs for the ad hoc mobile environment. Fundamentally our approach is to design abstractions tailored to the ad hoc environment which hide many of the unnecessary details, but give the programmer sufficient power to tailor the abstraction to their specific needs. This involves providing both context aware and context dependent operations within the same abstraction. At the same time, implementations of these abstractions must be responsive to the technical challenges of the environment.

Section 7.1 presents a general model for ad hoc mobility abstractions we call global virtual data structures. Section 7.2 discusses some of the technical challenges of the environment and how global virtual data structures are affected by each. Finally, Section 7.3 poses several possible data structures, including one in particular which serves as a case study for the remainder of this thesis.

7.1 Global Virtual Data Structures

Our approach to abstractions to simplify the programming task comes from a study of coordination models for distributed computing which separate the computation, or the task-specific programming, from the communication, or the interaction among processes.

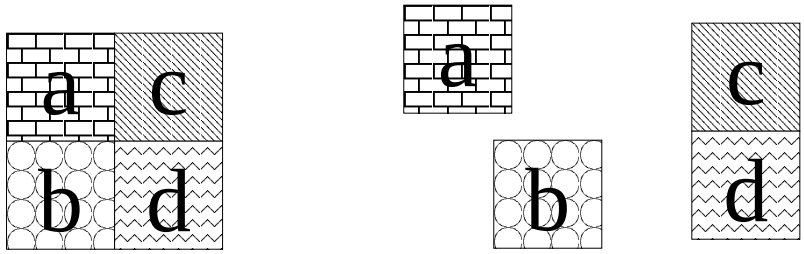


Figure 7.1: Transforming a matrix into a global virtual data structure by distributing it among mobile units.

Distributed coordination models also consider the need to take local decisions while still conceptualizing the effect of these actions on the global scale. Thus, our driving design strategy can be summarized by the desire to coordinate ad hoc mobile applications by thinking globally but acting locally.

One common coordination mechanism in distributed systems is shared memory, or more structured shared data structures. Through this, the complexity of large systems is managed by accessing a single, global data structure. An implementation may be distributed, but the user is not aware of this. The concept of shared memory is appealing in the mobile environment, which is itself a distributed system, however disconnections and the resulting inaccessibility of data make a direct application of shared memory to mobile systems impossible.

By applying our design strategy to shared memory data structures the global data structure emerges as the concept we wish to conceive of globally, but connectivity does not allow this. The first step toward a mobility-viable global data structure is to make explicit the distribution of data across the mobile components, or mobile hosts. For example, Figure 7.1(a) shows how a large matrix can be evenly divided among four hosts. While all mobile components are within communication range, the entire structure remains accessible to all processes.

When mobile components move and connectivity changes, the available portions of the data change to reflect only reachable data. In Figure 7.1(b) mobile components a and b are each isolated from all other components, restricting access only to their local partitions of the matrix. However, components c and d remain connected to one another and have access to the combination of c's and d's data.

The global matrix data structure of Figure 7.1(a) can be visualized at any time from outside the system by ignoring connectivity constraints and combining the data from all mobile components. This is, however, only a *virtual* data structure because it cannot be

created in reality. Despite this it remains a powerful concept to the programmer to view how local changes affect the entire system.

We have discussed how connectivity limits the availability of data, but we must also consider how the operations which access the data structure change in response to this accessibility constraint. Some operations must clearly be restricted if full connectivity is not available. In the matrix example, computations such as matrix inversion require the entire matrix and must be restricted. Many operations, however, require no changes and can simply be evaluated over the current projection of the global virtual data structure. These operations play an important role in implementing context-transparent applications as they do not require the programmer to be aware of the details of the environment, but simply aware that it is changing. Finally some operations can be extended to explicitly address the distribution of data over the hosts. Consider an alternate division of the matrix example that distributes data based on some aspect of the data other than its location in the matrix. In this case, it may be meaningful to query the part of the distributed data located at a specific agent. These operations are likely to play a role in context-aware applications.

For any global virtual data structure to be successful, its development cycle must include not only the model definition, but also formal specification and implementation to inform the model. The informal model presents the underlying data structure, how it changes with respect to connectivity, what the primitives are, and how they are affected and extended. Most importantly, the informal model also describes the abstraction provided to the programmer and a way of thinking to effectively develop applications on top of the model. Next, formal semantics force clear definitions of all model concepts and how they are affected by mobility before beginning an implementation. The formal specification also enables user applications to be formally specified and reasoned about, lending dependability to the resulting system. Finally, the data structures must be implemented and applications built. One mechanism to deliver the data structures is via a middleware which sits between the application and the operating system, providing the abstractions defined by the model and formal specification.

The key to development from these three key perspectives is to allow each step to inform the others in an iterative fashion. By considering the needs of the applications, the primitives of the model can be defined and extended to meet the demands of the application programmer. A formal specification can reveal key parts in the model where restrictions must be made to keep the operations computable in the presence of disconnections. The formal specification also informs the implementation, showing where the complexity is involved in the interactions of concurrent programs. A proper implementation must adhere to the formal specification. The process of implementing may reveal atomicity assumptions of the model which are either impossible or impractical to implement. This can lead to an expansion of the model to include more elements of the environment, or to a weakening of

the model constructs to make them more practical. Complementary changes must also be made to the formal specification.

7.2 Technical Challenges

While designing abstractions for the ad hoc mobile environment is difficult, the technical challenges make implementation of the abstractions similarly challenging. In this section we address the potentially open environment, unintentional disconnections, heterogeneous components, and network level issues, and how each of these affect the implementation of global virtual data structures.

By nature, ad hoc networks consist of transiently connected mobile components. While it is possible that all components can be known in advance, it is more likely that components must discover one another on the fly. This is similar to neighbor discovery at the network level. The primary difference is that in ad hoc networks, propagating this information up to the programmer is essential to allow context aware programming, and thus must be available as part of the data structure abstraction.

Another challenge with respect to disconnection is intentionality. When a user intentionally disconnects from the network for any reason, it is possible to use this advance knowledge to put the system into a consistent state before the actual disconnection occurs. We term this announced disconnection, because a user announces an intent to disconnect, and then waits for acknowledgment before actually disconnecting. However, in a real network, unintentional, or unannounced disconnection must be accounted for. For example, when a user suddenly moves into a region where the radio transmission is lost, not only is the user isolated from the rest of the network, but the status of any ongoing transmissions is unknown. Neither the party that was disconnected, nor those it was disconnected from know the status of the messages that were in transit. The problem is identical to that of distributed consensus. In practice, the impossibility of distributed consensus is addressed by using protocols such as two-phase commit, however these protocols rely on the fault model that a dropped link will eventually be restored and any inconsistent state can be reconciled at that time. In the model of ad hoc networks, this is not the case because a mobile component can disconnect permanently or for a long period, leaving an inconsistent, irreconcilable state. In the presence of unannounced disconnections, the consistency of a global virtual data structure cannot be guaranteed. Therefore we must consider weaker consistency models which can be tailored to application needs.

Another characteristic emphasized in the ad hoc environment is the heterogeneity of components. Wireless networks can consist of small components with extremely limited functionally such as wireless sensors. The same network may also need to support hand held devices and full functionality laptop computers. Not only do processing and storage vary

across devices, but also reliability and range of communication. Thus a key characteristic of any global virtual data structure implementation is to address the integration of this variety of computing elements in a single abstraction. This necessitates a modular design with components tailored to specific platforms. Possible changes may be needed at the model level to provide appropriate restrictions on some components.

Finally, many network-level issues must be addressed, such as expensive and unreliable communication links. One common strategy employed by base station networks to work around these concerns is to push the burden of computation and communication onto the fixed network. This is not possible in an ad hoc network where no such infrastructure exists. Therefore, new, lightweight protocols and models that enable coordination with limited communication must be developed. To aid in this, many cases can exploit the natural broadcast nature of ratio communication to reach multiple hosts while expending less energy than multiple point to point communications.

7.3 Sample Global Virtual Data Structures

Many standard distributed data structures have the potential to be converted into global virtual data structures. For each structure, the fundamental issues to address as part of the evaluation and development processes are similar: Does the data structure match the basic needs of the underlying application? Is there a natural and useful partitioning of the data structure across units in an ad hoc network? How is the data structure perceived by the individual units as changes in connectivity occur?

A tree, as in Figure 7.2, could be partitioned among units with the nodes where a cut occurs being replicated. A global naming convention would allow communicating units to determine the relation between the tree fragments they carry and make content and structural changes (e.g., swapping subtrees) as long as no disconnected units are affected. In principle, certain operations (e.g., adding a leaf node) could be issued at any time with their evaluation being delayed until such time that the affected units are within range. Attempts to access nodes on disconnected units may result in blocking the respective agent. The generalization to a directed graph is straightforward and can overcome the problems caused by the possible loss of one of the units.

Other data structures may be devised to meet the needs of highly specialized applications. For instance, resource-limited units searching a physical space may appear logically as ants crawling on a fixed network of passageways (Figure 7.3). Each unit's knowledge of the surrounding geography is enhanced by the knowledge of all the other units within range. As the density of units decreases, each unit must maintain more and more information. Finally, at a point when the unit's memory is full, information needs to be dropped, e.g., only the main passageways are kept. In an application involving the construction of

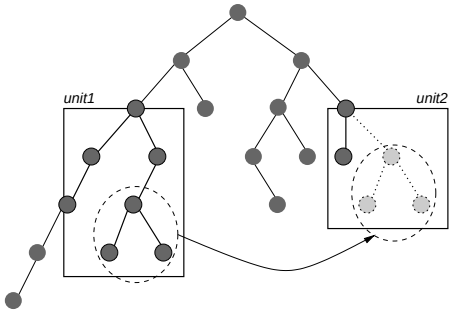


Figure 7.2: A hierarchical data structure where units in range agree to transfer a subtree which is under their jurisdiction even though parts of the global structure remain hidden. Moving a subtree distributes data to a different location to satisfy changing access patterns.

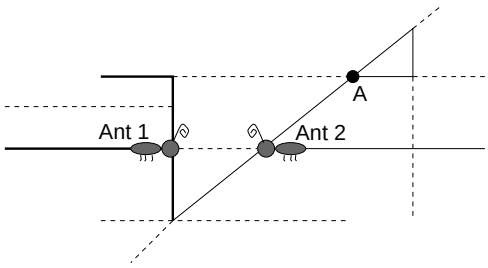


Figure 7.3: Ant 1 learns from Ant 2 about landmark A when, by virtue of being in range, the locally built maps are merged. Solid lines denote paths explored by Ants 1 and 2, and dashed lines denote unexplored regions. After sharing, each ant has the same knowledge of the global structure.

distributed predictive models of the changes taking place in a physical environment it is conceivable to have the units tied together by a complex structure that combines information about space and time. Each unit may be exploring and collecting data in the present while simulating the future in order to build a predictive model. As units meet they may exchange information about the present but also about various points in the future since some units may be further ahead than others in their simulation.

In the field of parallel programming, tuple space communication à la Linda provides a good example of how coordination can simplify the programming task. Tuple space coordination facilitates temporal and spatial decoupling among parallel programs. By limiting the power of the tuple space access primitives, efficient implementation is achieved as well. The programmer is presented with the appearance of a persistent global data structure which can be readily understood and operated on: a set of tuples accessed by content. Applying the concept of global virtual data structures to Linda yields a model which distributes the

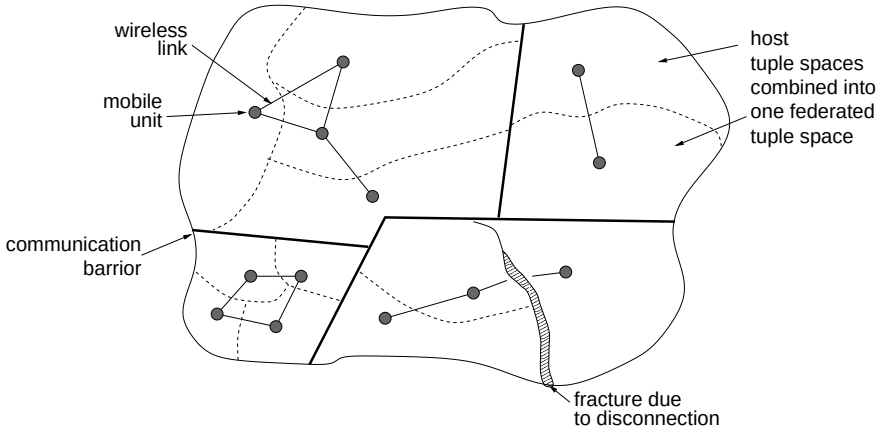


Figure 7.4: Creating the illusion of a globally shared tuple space.

global tuple space among the mobile units and limits access to the confines of each ad hoc network. For a programmer, mobility is perceived simply as an independently evolving host tuple space, i.e., a continuously changing context. When the mobile components are co-located, the tuple spaces are transiently shared and all tuple space accesses, including pattern matching for reading and removing data, are done on the now shared data space (see Figure 7.4). Additional primitives with extensions for location are straightforward to access specific tuple spaces, however the presence of the specified tuple space is dependent on connectivity. This data structure has been explored in detail and has resulted in the LIME model, Linda in a Mobile Environment. In Chapters 8 to 11, the details of LIME are presented. LIME serves as a proof of concept for the construction of global virtual data structures.

7.4 Concluding Remarks

Ad hoc networks represent a complex technology which pose many challenges to application development. By completely removing the traditional wired network, many assumptions about the community of distributed components are fundamentally changed. In this chapter, we have set forth the goal of developing a programming environment for the rapid development of dependable applications in the ad hoc mobile environment. We present the general concept of *global virtual data structures* which can be used to guide the development of a variety of abstractions for mobility based on the shared memory concept from standard distributed computing. The core idea is to move standard distributed data structures to mobility by redefining the accessibility of and operations over data with respect to connectivity. The development path of global virtual data structure should integrate model

development, formal specification, and implementation. In the remainder of this thesis, we present the LIME system, showing how the Linda tuple space model has been extended into the mobile environment. This serves as a positive proof of concept of the global virtual data structure approach to abstractions for ad hoc mobile computing.

Chapter 8

LIME: Linda Meets Mobility

In this chapter we present the informal model for a system called LIME (Linda in a Mobile environment) which is designed by with the concept of global virtual data structures. The underlying assumption is that both physically mobile hosts and logically mobile agents can be regarded as instances of a generic concept of mobile component, and *coordination* takes place through the use of transiently shared *tuple spaces* accessed via the basic set of Linda primitives [30].

In Linda, coordination is achieved through a tuple space globally shared among components which, independent of their actual location, can access the tuple space by inserting, reading, or withdrawing tuples containing information. The model provides both spatial and temporal decoupling. The components do not need to co-exist in time for in order to communicate and can reside anywhere in the distributed system. Since decoupling is intrinsic to mobility, the Linda model is a natural choice for our system.

LIME retains the basic philosophy and goals of Linda while adapting them to mobility. Simple and rapid application development is facilitated by the same mechanisms which made parallel programming in Linda attractive to implementors. Programs (written in a variety of languages) view the world as a sea of tuples accessible by contents. Movement, logical or physical, results in implicit changes of the tuple space accessible to the individual components. The system, not the application program, is responsible for managing movement and the tuple space restructuring associated with connectivity changes.

The remainder of this chapter is organized as follows: Section 8.1 provides a brief review of Linda. Section 8.2 motivates LIME and its design philosophy. Sections 8.3, 8.4, and 8.5 present a set of coordination primitives supporting transiently shared tuple spaces, location-aware computing, and reactive programming—the fundamental concepts underlying LIME.

8.1 Linda

Linda has been proposed at the beginning of the past decade [30] as a new model of communication among concurrent processes. The fundamental abstraction provided to each process is a shared *tuple space* that acts as a repository of elementary data structures—the *tuples*. Each tuple is a list of typed parameters, such as ⟨"foo", 9, 27.5⟩, that contain the actual information being communicated. A tuple space is a multiset of tuples that can be accessed concurrently by several processes.

Tuples are added to a tuple space by performing an **out**(t) operation on it. After its execution, the tuple t is available to any subsequent operation on the tuple space. The update of the tuple space is performed atomically. Tuples can be removed from a tuple space by executing **in**(p). Tuples are anonymous, thus their removal takes place through pattern matching on the tuple contents. The argument p is often called a *template*, and its fields contain either *actuals* or *formals*. Actuals are values; the parameters of the previous tuple are all actuals, while the last two parameters of ⟨"foo", ?integer, ?long⟩ are formals. Formals are like "wild cards", and are matched against actuals when selecting a tuple from the tuple space. For instance, the template above matches the tuple defined earlier. (Tuples can contain formals as well.) The matched tuple and the template must have the same arity. If multiple tuples match a template, the one returned by **in** is selected non-deterministically and without being subject to any fairness constraint. The **in** operation is blocking, i.e., if no matching tuple is available in the tuple space the process performing the **in** is suspended until a matching tuple becomes available. Tuples can also be read from the tuple space using the **rd** operation. The execution of a **rd**(p) proceeds identically to **in**, except for the fact that the tuple matched and delivered to the process that executes the operation is copied rather than withdrawn from the tuple space. Similarly to the **in** operation, **rd** is blocking. Linda implementations typically include also an **eval** operation which provides dynamic process creation and enables deferred evaluation of tuple fields. For our purposes, we use hereafter only **out**, **in**, and **rd**.

Communication in Linda is decoupled in *time* and *space*. Decoupling in time refers to the fact that senders and receivers do not need to be in communication in order to exchange information. Tuples are stored in the tuple space and can be retrieved later, even if the process that produced the tuple terminated its execution already. Decoupling in space refers to the fact that a tuple in the tuple space is available to processes dispersed on the nodes of a distributed system—the actual location of a tuple is hidden from the tuple producer and consumer. Decoupling is appealing, as it separates clearly the behavior of the individual processes from the communication needed to coordinate their actions. The idea enjoys wide acceptance in many scientific communities ranging from economics to artificial intelligence, and is at the core of a new interdisciplinary research area that investigates technologies and methodologies for the *coordination* of components in complex systems [43, 29]. Notably,

the decoupling between components and their coordination fostered by Linda has several points of contact with the distinction between components and their interconnection that constitutes the core of recent research on software architecture [81]. We chose the Linda model as the basis for this work due to its minimality and decoupling in time and space.

8.2 Linda Extensions for a Mobile Environment

Linda provides coordination among concurrently executing components accessing a shared tuple space that is persistent, globally accessible, and statically created. Maintaining these properties in the presence of physical mobility is complicated, because connectivity can no longer be taken for granted. Early research on fault-tolerant distributed implementations of Linda [88] tackled the problem under the assumption that disconnection was just an unfortunate accident, and employed replication schemes to increase tuple availability. However, in physical mobility disconnection is often forced explicitly by the user, e.g., to save battery power during movement or to reduce communication costs over expensive cellular phone lines. To make matters worse, mobile hosts (and mobile agents) are completely independent and controlled by users occasionally forming transient communities; thus, they may come in contact once and never again. This makes unreasonable any assumptions about eventual delivery of data. Finally, logical mobility complicates the implementation of a distributed tuple space even in the absence of any physical mobility. For instance, replication schemes that rely on the locality of processes may no longer be applicable, because processes can move freely around the network. The idea of a static, persistent, and globally visible tuple space is then unreasonable. Mobility demands weaker constraints on the tuple space and dynamic reconfiguration of its contents.

In the model underlying LIME, mobile agents are programs that can travel among mobile hosts. They are "active" components of the system. Mobile hosts are roaming containers for agents, to which they provide connectivity. The Linda model is adapted by LIME through the notion of a *transiently shared tuple space* that ties together physical and logical mobility. Tuple spaces are permanently bound to mobile agents and mobile hosts. Transient sharing enables dynamic reconfiguration of their contents according to agent migration or connectivity variations. At a high level of abstraction, mobile agents interacting in the logical space provided by a single host are akin to mobile hosts interacting in the physical space spanned by communication links. Hence, transiently shared tuple spaces and the associated access primitives provide a single coordination toolkit for both scenarios. In practice, however, distribution and mobility do complicate implementability. As a result a more constrained use of the LIME primitives is allowed when physical mobility is present.

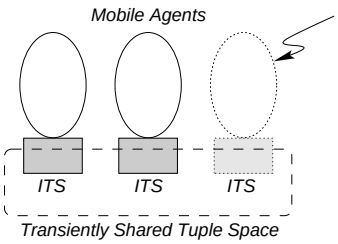


Figure 8.1: Transiently shared tuple spaces in LIME.

8.3 Transiently Shared Tuple Spaces

The fundamental abstraction provided by LIME is the notion of a *transiently shared tuple space*. As summarized in Figure 8.1, in LIME each mobile agent has access to an *interface tuple space* (ITS) that is permanently associated with that agent and transferred along with it when movement occurs. Each ITS contains information that the mobile agent is willing to share with others, and is accessed using the conventional Linda operations **in**, **rd**, and **out** described in Section 8.1, whose semantics are unaffected. On the other hand, the actual content of the ITS is determined differently from Linda. The set of tuples that can be accessed through the ITS is dynamically recomputed in such a way that, for each mobile agent, the content of its ITS gives the appearance of having been merged with those of the other mobile agents which are currently co-located. This way, each mobile agent "sees" through its own ITS the same transiently shared tuple space presented to the others. Operations performed on the ITS are effectively performed on the contents of the transiently shared tuple space; e.g., if agents A and B are co-located and A performs an **out**(t) on its ITS, after the tuple t is inserted in A's ITS it is available for retrieval with an **in**(t) by agent B.

The tuple space that can be accessed through the ITS of an agent is *shared* by construction and is *transient* because its content changes according to the migration of agents. The action of making the contents of an ITS accessible to other agents through the transiently shared tuple space takes place in reaction to changes in the set of co-located agents. Upon arrival of a new mobile agent, the transiently shared tuple space is recomputed by taking into account the ITS of the new agent. The result is made accessible to all the agents currently co-located. This sequence of operations is called *engagement* of the tuple spaces, and is performed as a single atomic transaction. Similar considerations hold for the departure of a mobile agent, resulting in the *disengagement* of the corresponding ITS. Its content is removed atomically from the transiently shared tuple space according to rules that are discussed later.

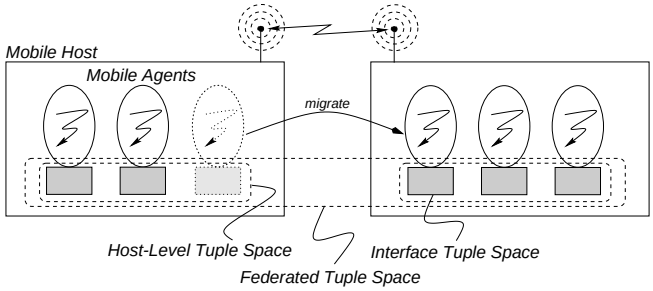


Figure 8.2: Transiently shared tuple spaces to handle physical and logical mobility.

In LIME, agents may have multiple ITSs, and also *private* tuple spaces, i.e., not subject to sharing. Tuple spaces are *named*; the name effectively defines a notion of typing for the tuple space and, in the case of ITSs, determines the sharing rule. If an agent has multiple ITSs, these are shared independently with the corresponding ITSs of other co-located agents, if any. In other words, if agent A owns the ITSs named S, T, and U, while agent B owns the ITSs named T, U, and V, only T and U will become transiently shared between A and B. For instance, this enables an agent to exchange information with a service broker about the available CD resellers by transiently sharing the corresponding ITS, and then subsequently share information about a given title and the payment options with the reseller selected through a different ITS, thus keeping separate the information concerned with different tasks and different roles. By construction, all agents are bound to a LimeSystem ITS whose tuples can be read but not withdrawn. This ITS contains system information concerning the agent, e.g., its identifier, as well as information concerned with the host, e.g., quality of service information. Transient sharing of LimeSystem enables co-located agents to access global system information. We will detail the role of this tuple space later on. To identify the tuple space on which a given operation is performed, we use the dot notation, e.g., $T.\mathbf{out}(t)$. However, in this chapter we will focus on agents with a single ITS, unless otherwise specified, and drop the name of the tuple space.

So far, the discussion has been focused on mobile agents. However, LIME applies the notion of transiently shared tuple space to a generic mobile component regardless of its nature—physical or logical. This relies on an extended notion of connectivity that encompasses both kinds of components. Mobile hosts are connected if a communication link connecting them is *available*. Availability may depend on a variety of factors, including quality of service, security considerations, or connection cost; in principle, all of these can be represented in LIME. However, in this chapter we limit ourselves to a simple notion of availability based on the presence of a functioning link. Because we assume bidirectional

links and the presence of routing capabilities in the physical network, our notion of connectivity is commutative and transitive. Mobile agents are connected if they are co-located on the same host or they reside on hosts that are connected. Changes in connectivity among hosts depend only on changes in the physical communication link. Connectivity among mobile agents may depend also on arrival and departure of agents with creation and termination of mobile agents being regarded as a special case of connection and disconnection, respectively. Finally, we assume that both mobile agents and mobile hosts are given globally unique identifiers. Figure 8.2 depicts the model adopted by LIME.

The content of the ITS of each mobile agent is determined by the presence of connectivity, in the aforementioned extended meaning. By definition, agents co-located on the same host are connected, and this creates a *host-level tuple space* that is transiently shared among all such agents and accessible through each agent's ITS. As evident in Figure 8.2, the host-level tuple space can be regarded as the ITS of a mobile host, as it is permanently associated with it; if no mobile agents are currently hosted, the host-level tuple space is empty. Next, the mechanism of transient sharing is applied to the host-level tuple spaces. Hosts that are connected merge their host-level tuple spaces into a *federated tuple space* whose content is transiently shared across the network. A tuple in the ITS of an agent can be either local and thus belonging to the local host-level tuple space, or remote and thus belonging to the host-level tuple space of a mobile host that is currently accessible.

The notion of transiently shared tuple space is a natural adaptation of the Linda tuple space to a mobile environment. When physical mobility is involved, there is no place to store a persistent tuple space. Connection among machines comes and goes and the tuple space must be partitioned in some way. In the scenario of logical mobility, maintaining locality of tuples with respect to the agent they belong to may be complicated. LIME enforces an a priori partitioning of the tuple space in subspaces that get transiently shared according to precise rules, providing a tuple space abstraction that depends on connectivity. In a sense, LIME takes the notion of decoupling proposed by Linda further, by effectively decoupling the mobile components from the global tuple space used for coordination.

In this model physical and logical mobility are separated in two different tiers of abstraction. It is worth noting, however, that many applications do not need both forms of mobility, and straightforward adaptations of the model are possible. For instance, applications that do not exploit mobile agents but run on a mobile host can employ one or more stationary agents, i.e., programs that do not contain migration operations. In this case, the design of the application can be modeled in terms of mobile hosts whose ITS is a fixed host-level tuple space. Applications that do not exploit physical mobility—and do not need a federated tuple space spanning different hosts—can exploit a host-level tuple space as a local communication mechanism among co-located agents.

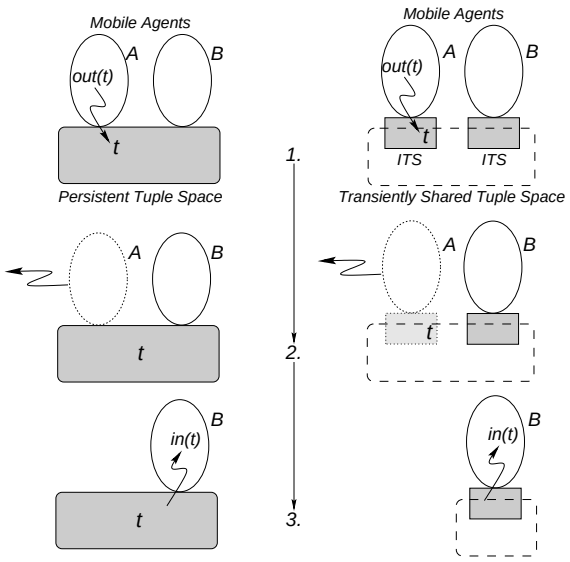


Figure 8.3: Persistent vs. transiently shared tuple spaces.

8.4 Location-Aware Computing

The transient nature of the LIME shared tuple space poses some additional challenges, as illustrated in Figure 8.3. Let us consider a simple system composed of two mobile agents, A and B. A is initially co-located with B; before departing, A leaves some information that is intended to be eventually processed by B. This simple scenario, depicted in the left hand side of Figure 8.3, is represented straightforwardly in Linda. A performs **out**(t) and deposits in the tuple space the tuple t containing the information to be communicated to B (step 1). Thanks to the persistence and global availability of the tuple space, the tuple t remains in the tuple space even after A departs (step 2), thus it is still available when B eventually tries to withdraw it by performing **in**(t) (step 3). The situation with transiently shared tuple spaces is depicted on the right hand side of the same figure. Since A and B are initially co-located, when A performs **out**(t) the tuple t is inserted in the ITS of A and becomes available to B thanks to the transient sharing of the agents' ITSs (step 1). However, when A departs, it takes along its own ITS (step 2). If B tries to pick up the tuple t after A has already departed, the corresponding **in**(t) will be performed on a transiently shared tuple space consisting of B's ITS only, and thus may block since t is no longer present there and there might not be any other tuple satisfying the match (step 3).

This problem is a consequence of the fact that in LIME there is no well-known persistent tuple space that can be used as a global repository of tuples. The configuration of the tuple space, i.e., its content, varies dynamically according to the location of components. Still, it is desirable to retain the advantage provided by the decoupling in time and space

characterizing the Linda model of communication. In our example, from B's perspective it would be desirable for the withdrawal of t to be possible any time after A becomes co-located with B. To accomplish this, A should be allowed to specify that the effect of an **out**(t) on the transiently shared tuple is to place t in B's ITS rather than keeping it in A's ITS as we assumed so far.

In LIME this is accomplished by exploiting the notion of *location*—central to mobility in general. The location of a tuple is a tuple space. In the case of an ITS, the location of a tuple is identified uniquely by the name of the tuple space and by the identifier of the mobile agent owning the ITS, since the agent and its ITS are permanently bound. Given this notion, in LIME a tuple can be placed into the ITS T of a mobile agent λ by simply using T.**out**$[\lambda](t)$, a version of **out** annotated with the tuple's intended location. The semantics of the **out**$[\lambda]$ operation take into account the location of agents, and involve conceptually two steps. The first step is equivalent to a conventional **out**(t), the tuple t is inserted in the ITS of the agent calling the operation, say ω. At this point the tuple t has a *current location* ω, and a *destination location* λ. If the agent λ is currently connected, i.e., either co-located or located on a connected mobile host, the LIME system reacts to the **out** operation by moving the tuple t to the destination location. The combination of the two actions—the insertion of the tuple in the ITS of ω and its instantaneous migration to the ITS of λ—are performed as a single atomic operation. On the other hand, if λ is not currently connected, the tuple remains at the current location, the ITS of ω.

Thus, in our example, A could circumvent the problem described earlier by performing **out**$[B](t)$ on its ITS. Note that performing **out**$[\lambda](t)$ does not necessarily imply guaranteed delivery of t to λ; the rules for non-deterministic selection of tuples as defined by Linda are still in place and the tuple t is always available in the transiently shared tuple space, even when it is not yet within the intended ITS. Thus, it might be the case that while waiting for λ to connect, or after it becomes connected and t is transferred to λ's ITS, some other agent may withdraw t from the tuple space before λ.

User-specified tuples are automatically augmented by the run-time support with fields recording their current and destination locations. This information enables LIME to detect the presence of "misplaced" tuples (i.e., tuples whose current location is different from their intended destination) and facilitates state reconciliation upon engagement and disengagement of tuple spaces. For instance, if t still belongs to the ITS of ω, and λ becomes connected, the system detects the presence of the misplaced tuple t and migrates it to the ITS of λ, also changing the current location of t to the value of its destination, λ. Since this action is part of the engagement of tuple spaces, the actions of becoming connected, merging of the ITS, and migrating misplaced tuples take place in a strictly sequential order and are executed as if they were a single atomic operation.

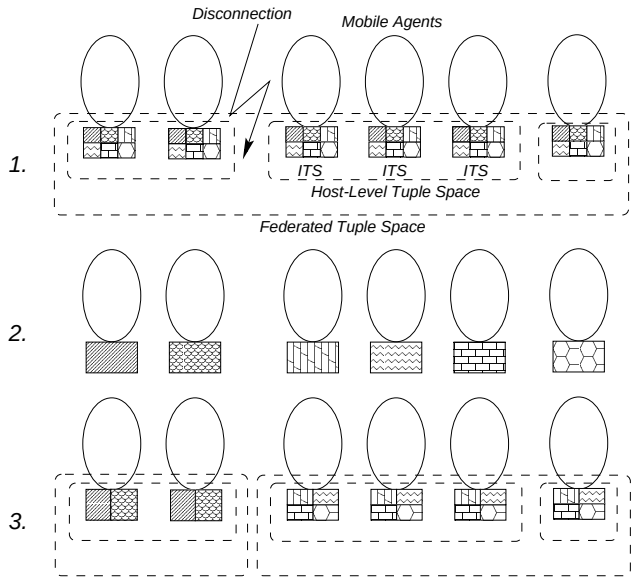


Figure 8.4: Recomputing transiently shared tuple spaces on disengagement.

Disengagement relies on the notion of location, as well. When an agent departs, the transiently shared tuple space that is presented to mobile agents must be recomputed accordingly. Conceptually, this process can be described as depicted in Figure 8.4. Initially (step 1), the ITS of each mobile agent has access to the (bag) union of the tuples contained in the physical ITS actually associated with the agent. This is represented by using different fill patterns for the boxes representing the ITSs. Upon occurrence of a disconnection, be it the departure of an agent or a physical disconnection like in Figure 8.4, the transiently shared tuple spaces are partitioned into their constituents (step 2) such that the ITS of each agent, say ω, contains only the portion of the shared tuple space it is responsible for, i.e., all the tuples whose current location is ω. This includes also misplaced tuples, whose final destination is different from ω; if the destination is not present, placing these tuples with the agent that generated them is the only meaningful solution. Thus, this step basically partitions the contents of the transient tuple spaces back into the "concrete" ITSs that are actually physically carried by each mobile agent. Finally, these ITS are merged again according to the new configuration of the system, thus completing the tuple space update process caused by disconnection (step 3).

The ability to abstract away the location of mobile agents and hosts as well as to access their data seamlessly simplifies certain programming tasks. However, there are situations where it may be desirable to limit access only to a portion of a transiently shared tuple space, either for programming convenience or for performance reasons. For instance, the programmer may be temporarily concerned only with the ITS of a given agent

or a particular host-level tuple space. These situations demand new forms of **in** and **rd** annotated with locations, similar to **out**[λ]. In LIME, annotations for these operations come in the form **in**[ω, λ] and **rd**[ω, λ], where the current and destination locations defined earlier are used. Either of the two locations can be left unspecified, in which case the symbol "_" will be used. Different combinations of location parameters identify different projections of the transiently shared tuple space. In the following, we review first the combinations allowed in LIME when dealing with logical mobility on a single host. Later on, we show what operations are permitted in the presence of physical mobility. The discussion focuses on the **in** operation.

The operation **in**[ω, _] performs an **in** on the projection of the transient tuple space whose tuples have current location equal to ω. This operation provides a means to restrict the scope of the **in** operation to the ITS of a single agent, including misplaced tuples that have been created by this agent. LIME allows one also to refer to tuples that are in a given ITS but whose destination is actually another agent λ, in the form **in**[ω, λ]. The ability to perform these operations can be useful, for instance, in a kind of tuple "garbage collection." If a garbage collector agent has some application knowledge that it will never meet the agent λ again, then it can purge useless tuples from the ITS of ω. As a special case, **in**[ω, ω] performs the **in** on the tuples in ω's ITS that are not misplaced, i.e., whose destination is ω and whose current location is ω as well. **in**[_, λ] restricts the scope of the **in** to the tuples in the transiently shared tuple space whose final destination is λ. If λ is currently not present, such tuples are currently misplaced in the ITS of some other agent. If λ is present, then the operation becomes equivalent to **in**[λ, λ] because, according to semantics of transient sharing, if λ is present all misplaced tuples directed to it migrate atomically to the correct ITS at the time of engagement.

When physical mobility is present, LIME provides also the capability to restrict the query performed by **in** to a specific host-level tuple space. This is achieved by specifying the identifier Ω of a mobile host as the first parameter, e.g., **in**[Ω, _]. In general, the current location can be specified either as the identifier of a mobile agent or of a mobile host. On the other hand, the destination location must always be the identifier of a mobile agent. This restriction holds also for the **out**[λ] operation whose λ parameter, representing the destination location for t, cannot be a mobile host. If this were allowed, the tuple would eventually belong to a host-level tuple space. However, we already discussed that this tuple space is actually made of the concrete ITSs associated to mobile agents, they are the ones physically holding tuples. According to our model, if no agent is specified as a destination location for t, the tuple vanishes upon disconnection. Nevertheless, if in some applications the host must own a tuple space, this can be realized in LIME by using a stationary and persistent agent that is responsible for holding incoming tuples destined to that host and for making them available to the agents located there.

8.5 Reactive Programming

In the rapidly changing environment that characterizes mobility, the ability to react to events, and to do it as soon as possible, is of great importance. Events can be concerned with the physical environment, like disconnection or changes in the quality of service, or with the application, like the availability of data carried by other parties. A typical example might be the arrival or departure of a mobile agent.

Events are naturally represented in the Linda model as tuples. This is actually the solution exploited by LIME as well. It leverages off the LimeSystem ITS described earlier. The run-time support, which can be modeled as a stationary agent, continuously monitors the underlying layers of the virtual machine for system events. These events, that include departure or arrival of mobile agents, changes in connectivity, and changes in the quality of service, are transformed into event tuples and are inserted in the LimeSystem ITS of the stationary agent. There, this information becomes available to all the agents connected through the federated tuple space generated by transient sharing of all the LimeSystem ITSs.

In Linda, the **in** operation already provides a basic mechanism for coping with events. Processes can suspend on the occurrence of an event, modeled by the appearance of the corresponding tuple in the tuple space, and will take appropriate actions after the tuple is retrieved by the **in**. However, this solution has some drawbacks. From a semantic point of view, there is no guarantee that the event is actually caught by a process λ interested in it. Another process μ could be interested in the same event and perform the **in** operation before λ. To guarantee event delivery, more complex schemes must be used. They must involve a priori knowledge about the number of processes interested in an event. From a performance point of view, since **in** is blocking, listening to multiple events requires one thread of control per event, which is often impractical. The fundamental problem rests with the fact that Linda forces applications to "pull" out tuples, while reactive programming demands to have tuples "pushed" to applications. Finally, it is desirable to encapsulate the reaction to an event into its own definition, thus providing a higher level of abstraction to the programmer who should be freed from the burden of dealing explicitly with synchronization issues.

LIME introduces the notion of a *reactive statement* having the form $T.\textbf{reactsTo}(s, p)$ where s is a code fragment containing non-reactive statements that is to be executed when a tuple matching the pattern p is found in the tuple space T. T can be any tuple space. However, in the next section, we will see that some additional constraints are necessary to deal with remote tuple spaces. The execution of **reactsTo** "registers" the reaction with T; a complementary operation to "de-register" the reaction is also provided by LIME. The semantics of the **reactsTo** statement are the same as those defined for Mobile UNITY reactive statements, described in [50]. After each non-reactive statement, a reaction is selected non-deterministically among those registered, and its guard is evaluated. If the guard is true, the corresponding action is executed, otherwise the reaction is equivalent to

a skip. This selection and execution proceeds until there are no enabled reactions and the normal execution of the non-reactive statements can resume. Thus, reactive statements are executed as if they belong to a separate reactive program that is run to fixed point after each non-reactive statement. These semantics offer an adequate level of reactivity, because all the reactions registered are executed before any other statement of the co-located agents, including the migration statements. Thus, the programmer's effort in dealing with events is minimized.

Although in principle s and p could have arbitrary forms, in practice their structure is constrained by implementation considerations. For example, if the definition of p contained arbitrary expressions in the host language—Java in the current implementation of LIME—the truth of p would need to be evaluated after each statement. This would require either access to the innards of the Java virtual machine or the development of a higher level language built on top of Java whose run-time support manages arbitrary reactive statements. For this reason, in the current implementation of LIME the specification of p is reduced to a pattern that is matched against tuples in the tuple space on which the reactive statement is executed. This way, the only statement that can trigger a reaction is the insertion of a tuple in the tuple space, which is under the control of LIME. Similar issues relate to s. Conceptually, s could be any code fragment containing non-reactive statements, thus including tuple space operations. In practice, reactions are executed in a separate thread of control belonging to the LIME system. Thus, blocking operations are forbidden in s, as they would actually block the processing of all the reactions. In general, the presence of tuple space operations in s complicates matters. From a performance point of view, s should be kept as small as possible to allow fast processing of reactions; as execution of tuple space operations is usually more demanding than conventional statements and they should be used with much care. From a semantic point of view, if an **out**(t) operation is allowed in s, t may match the pattern p specified by some other reaction and thus generate a potentially infinite reaction loop. LIME presently allows tuple space operations in s, although subject to the constraints described below. Our rationale stems from the notion that preventing the programmer from modifying reactively the tuple space is worse than leaving up to the programmer the evaluation of the semantic and performance tradeoffs. Application development with the LIME API will hopefully validate this hypothesis.

The actual form of the **reactsTo** operation is annotated with locations—this has been omitted so far to keep the discussion simpler. A reaction assumes the form T.**reactsTo**$[\omega, \lambda]$ (s, p), where the location parameters have the same meaning as discussed for **in** and **rd**. However, reactive statements are not allowed on federated tuple spaces; in other words, the current location field must always be specified, although it can be the identifier either of a mobile agent or of a mobile host. The reason for this lies in the constraints introduced by the presence of physical mobility. If a federated tuple space is present, its content, accessed

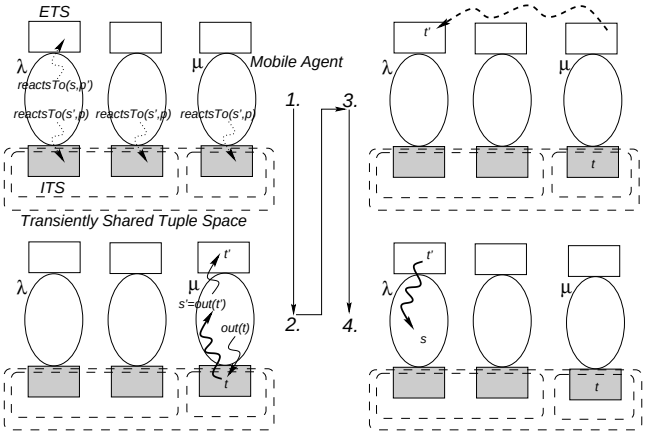


Figure 8.5: Reacting to remote events in LIME. Thick solid lines represent reactions, while the thick dotted line represents an asynchronous action.

through the ITS of a mobile agent, actually depends on the content of ITSs belonging to remote agents. Thus, to maintain the requirements of atomicity and serialization imposed by reactive statements would require a distributed transaction encompassing several hosts for each tuple space operation on any ITS—very often, an impractical solution.

For these reasons, LIME offers the capability to react asynchronously to the availability of tuples in a remote ITS by providing an operation T.**weakReactsTo**(s, p) that has weaker semantics than the **reactsTo**. Conceptually, it works as depicted in Figure 8.5, where an additional, private tuple space associated with each mobile agent is shown. The purpose of this *event tuple space* (ETS) is to collect event tuples and is introduced here only to simplify the presentation—it is not actually provided among LIME constructs. The semantics of **weakReactsTo** can then be described as follows:

1. When **weakReactsTo**(s, p) is invoked on the ITS T_λ of a given agent λ, a strong reaction **reactsTo**(s', p) is registered on each ITS currently shared with T_λ. A strong reaction **reactsTo**(s, p') is also registered on λ's ETS, E_λ.

2. When an **out**(t) operation with t matching the pattern p is performed on the ITS T_μ of a given agent μ, the reaction T_μ.**reactsTo**(s', p) fires and s' is executed. s' performs an E_μ.**out**(t') where t' is a copy of t augmented with information that enables the system to bind an event tuple to the agent that is performing the **weakReactsTo**.

3. An asynchronous action moves the tuple from the E_μ to E_λ. There is no guarantee that this action happens as soon as t' shows up in E_μ, because this would require starting a distributed transaction and suspending the execution of all the connected

agents. Instead, LIME guarantees eventual delivery of t' to E_λ, if connectivity is available.

4. When t' reaches E_λ, the reaction E_λ.**reactsTo**(s, t') fires, where s is actually the statement specified originally in the **weakReactsTo**.

The operation **weakReactsTo** in LIME is similar to the `notify` operation provided by Sun's JavaSpace [53] and to the event registration mechanism provided by IBM's T-Spaces [33]. The development of a richer event model, allowing reactions to arbitrary events other than insertion of a new tuple, is the subject of on-going work.

8.6 Concluding Remarks

In this chapter we presented the design of LIME, a system that follows the design strategy of global virtual data structures by adapting the Linda model of coordination to mobility, introducing the notions of transiently shared tuple spaces, tuple location, and reactive statement. The hypothesis behind our research is that the minimalist set of operations and concepts provided by LIME, in particular the transient sharing of tuple spaces, enable rapid and dependable development of applications which involve mobility.

Chapter 9

Formalizing LIME

The ultimate goal of our research is to develop dependable mobile applications. One way to address dependability is the use of formal specifications to define precise semantics of a system. In this chapter we present a formal specification of LIME for two main reasons. First, the specification of the data structures and the constructs used to manipulate them provide a foundation for implementation of the LIME middleware. Second, the specification allows applications built on LIME to be formally specified and reasoned about.

Section 9.1 starts this chapter with a formal specification of the Linda model which underlies LIME, introducing the constructs of the formal modeling language UNITY as necessary. Following this, Section 9.2 defines the LIME semantics using Mobile UNITY, an extension of UNITY tailored to the specification of mobile systems. For simplicity of presentation, the informal definitions of both the Linda and LIME semantics are repeated from the previous chapter and interleaved with the formal definitions. Section 9.3 shows how parallelism can be exploited when implementing the model, and exposes other aspects of the model which require system-wide coordination.

9.1 Formal Foundations: UNITY and Linda

Linda fosters a programming style with a clear separation between communication and computation and provides a clean, easy to comprehend model with a minimal set of primitives. As shown in Figure 9.1, the model is defined as a shared memory *tuple space*, which is accessible to multiple processes. Data is stored in the space as elementary data structures called *tuples*. Processes coordinate through the tuple space by writing (**out**), removing (**in**) and copying (**rd**) tuples. Typically these primitives are presented as a programming interface from a specific host language such as Fortran or Java. However, in this chapter we are interested in the Linda model and not the technicalities of a particular implementation. Further we are interested in providing a precise semantic definition for the Linda operations upon which we will eventually build the formal semantics of our mobile middleware.

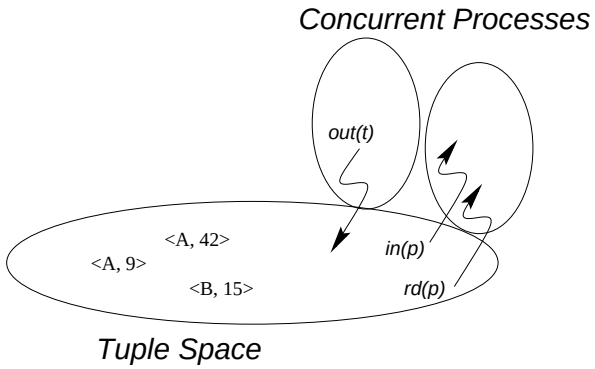


Figure 9.1: A simple Linda tuple space.

For these reasons, we present a formal model for Linda using UNITY, a model put forth by Chandy and Misra [19] to study the essential characteristics of distributed computing and to aid in the development of reliable concurrent programs. The programming language independent notation provides for the formal specification and verification of distributed systems, and the minimal nature of the constructs allows one to focus on the essence of the problem being specified. The fundamental elements of the model are variables and conditional, multiple assignment statements. A program specification consists of a set of assignment statements which execute atomically and are selected for execution in a nondeterministic, weakly fair manner. Concurrent systems can be specified as a set of programs, composed through program *union* ($\cup$). Figure 9.2 shows two simple UNITY programs which together describe a producer/consumer system using the Linda abstraction of a shared tuple space to share tasks to be completed. The producer randomly generates jobs of different types, putting each job and its description into a tuple space called *jobs*. The consumer removes jobs from the tuple space and performs the defined task. The details of the Linda operations will be described in the second part of this section, for now we focus on the main UNITY concepts.

Every UNITY program contains a **declare** section for naming variables and their types, an **initially** section for defining the allowed initial conditions of the variables, and an **assign** section for specifying the guarded assignments which define the state transitions of the system. Statements are separated by the "[]" operator. The "||" operator is used to construct multiple assignments to be executed in a single atomic step. For example, the consumer program contains two statements; the second simulates the performance of a task by setting the value of *curTask* to ε and the first removes a task from the tuple space and keeps a running count of the number of tasks to be performed.

Variables with the same name from different programs are the same variable after composition through union. In the example, the tuple space *jobs* is shared between the

Program *Producer*
  declare
    $jobs$: TupleSpace
    $jobType$: {A, B}
  initially
    $jobs = \emptyset$
  assign
    $jobType$:= A
    [] $jobType$:= B
    [] **out**($jobs$,
      createTuple($jobType, jobInfo()$))
end

Program *Consumer*
  declare
    $jobs$: TupleSpace
    $curTask, myJobPattern$: Tuple
    $count$: Integer
  initially
    $myJobPattern$ = createTuple(A, String)
    [] $curTask = \varepsilon$
    [] $count = 0$
  assign
    ($curTask$:= **in**($jobs, myJobPattern$)
      || $count := count + 1$)
      if $curTask = \varepsilon$
        $\wedge$matchExists($jobs, myJobPattern$)
    [] $curTask := \varepsilon$ || $doJob(curtask)$
end

Figure 9.2: A standard UNITY specification with two component programs. *Producer* $\cup$ *Consumer*.

composed programs so that when the producer writes a tuple to its tuple space it is immediately available in the consumer's tuple space, they are actually the same data structure. The assignment statements from the programs are combined by union to form a single set of statements and the concurrent execution is modeled by the interleaved execution of these statements. At each computation step, one statement is nondeterministically selected for execution and the system state is atomically modified. Because UNITY does not provide any sequential constructs, guards are used in the consumer to prevent it from taking a new task before the previous task has been completed (i.e., before *curTask* is reset to ε).

In this example, the Linda operations and data structures appear as elementary UNITY primitives, however their semantics have not been defined. They are more properly regarded as *macros* defined for notational convenience, whose meaning is actually represented in terms of the basic UNITY statements which as defined formally in the remainder of this section.

Linda Tuple. The first concept to formalize is the Tuple. A tuple is the data structure used to communicate information among processes, and is simply an ordered sequence of typed fields generated with the createTuple() function. Parameters to this function can include actual values, as in the producer, or any combination of actuals and formals (types), as in the consumer, e.g., (String). A tuple with formal values is also known as a template or pattern tuple.

Data is accessed from the tuple space based on content by specifying a pattern to match against the data. Therefore it is important to define what it means for two tuples to *match*. Following the standard Linda semantics, matching tuples must have the same arity, and the corresponding fields of each tuple must match. Within a field, an actual matches

an actual of the same value, a formal matches an actual of the same type, but a formal does not match a formal. In this way, a formal acts as a "wild card", matching an actual of any value. Thus the consumer in the example identifies a task of a specific type (A) but with any description (String).

To formally express the fact that a tuple θ matches a template or pattern tuple p, we use the notation $\mathcal{M}(\theta, p)$. To add power to pattern matching beyond that of standard Linda, we extend the formal field definition to allow the specification of subtypes. For example, $\langle$"foo", Integer $i : 1 \leq i \leq 10\rangle$ requires a matching tuple to have, in its second field, an integer value between 1 and 10. While we do not use this function directly in our example program, it is used in the definitions of several LIME constructs.

Linda Tuple space. The TupleSpace data structure can be conceptualized as a multiset or bag of tuples. For the purpose of our formalism, we specify the tuple space as a set, but we must still allow for multiple tuples with identical data to exist in the set. Therefore, we associate a unique tuple identifier to each tuple in the tuple space. This is accomplished by adding an identifier field to the beginning of every tuple. Because this field is introduced only for the convenience of the formal specification, we wish to make it invisible to the programmer who simply wants to use the Linda constructs. This is the primary motivation for the createTuple() function which, as part of the tuple creation process, prepends a formal field of type TupleID to every tuple to eventually contain the tuple identifier. The management of this new field must be accounted for throughout the Linda specification, but remains transparent at the user level.

For convenience we define a general notation for setting and accessing *labeled* tuple fields. For example, we define the tuple identifier field to have the label ID, and [ID : TupleID] $\oplus\, t$ sets the value of tuple t's labeled field ID to the formal TupleID. Alternately, a value can replace the type in the previous formula, establishing the labeled field as an actual. The notation t.ID retrieves the value of the field with label ID or undefined if the field is formal.

In typical Linda applications, only a single, unique tuple space is available for process coordination. By defining the tuple space as a regular UNITY variable, it is possible for a single process to define and access multiple, disjoint tuple spaces, each of which is identified by the name of the tuple space variable. This gives applications another degree of freedom to separate the different coordination concerns into multiple tuple spaces.

Basic Linda constructs. The operations over the tuple space as they appear in the example programs are not UNITY primitives, but rather represent the abstract Linda constructs. The precise semantics of these constructs are given below in the form of macro definitions. In other words, when one of the Linda constructs appears in a UNITY program,

Standard Linda operations	$\textbf{out}(T, t)$	
	$t := \textbf{in}(T, p)$	$t := \textbf{rd}(T, p)$
Probing operations	$t := \textbf{inp}(T, p)$	$t := \textbf{rdp}(T, p)$
Group operations	$s := \textbf{ing}(T, p)$	$s := \textbf{rdg}(T, p)$
	$s := \textbf{ingp}(T, p)$	$s := \textbf{rdgp}(T, p)$
Miscellaneous	$\mathsf{matchExists}(T, p)$	
	$\mathsf{createTuple}(\ldots)$	

Figure 9.3: Summary of the Linda macros. T is a TupleSpace, t is a Tuple, p is a pattern Tuple, and s is a set of tuples. The parameter of createTuple is a sequence of any number of values (actuals) and types (formals).

its complete semantics can be extracted by doing a textual substitution of the appropriate definition. A summary of the Linda constructs available to programmers appears in Figure 9.3.

All process coordination in Linda-based applications takes place through access to the shared tuple space. In our example, the producer process generates tasks and inserts them as tuples into the tuple space, from which they are retrieved and processed by the consumer. The required functionality is insertion into (**out**), removal from (**in**), and copying (reading) from (**rd**) the tuple space. Typical Linda implementations also provide an **eval** operation which provides dynamic process creation and enables delayed the evaluation of tuple fields, but for our purposes we use only **out**, **in**, and **rd**.

The final assignment statement of the producer demonstrates the use of $\textbf{out}(T, t)$ to insert a tuple in to the tuple space. Because a process can access multiple tuple spaces, the tuple space variable appears as a parameter of the operation. The formal semantics defining the macro definition describe the change to the tuple space as a result of the **out**, namely the addition of the tuple t.

$$\textbf{out}(T, t) \triangleq T := T \cup \{[\textsc{id} : \mathsf{newID}()] \oplus t\}$$

In addition to inserting the tuple into the tuple space, the formal definition establishes the uniqueness of the tuple within the tuple space by setting the identifier field. The function newID() returns a new, system-wide unique tuple identifier. It should also be noted that by the assignment semantics of UNITY, the value in braces is actually the value of the tuple, and thus a *copy* of the tuple is effectively made and inserted into the tuple space.

Information from the tuple space is retrieved based on the content of the data by specifying a tuple pattern to look for inside the tuple space. Removal of a tuple matching a specific pattern is achieved with the **in** construct which identifies a tuple matching the pattern p in the tuple space T and assigns it to the variable t. If more than one tuple

matches, one is chosen non-deterministically. Tuples can also be copied from the tuple space using **rd**.

To simplify the formal definitions of these concepts, we introduce several helper functions. We have already defined the matches function $\mathcal{M}$ which compares two tuples. Here we define a predicate which identifies whether a tuple matching a pattern exists in a given tuple space, and two macros which identify a matching tuple in a given space and either remove or copy it[1].

$$\mathsf{matchExists}(T, p) \equiv \langle \exists\, \theta : \theta \in T :: \mathcal{M}(\theta, [\textsc{id} : \mathsf{TupleID}] \oplus p) \rangle$$

$$t := \mathsf{remove}(T, p) \triangleq$$
$$\langle \| \ \theta : \theta = \theta'.(\mathcal{M}(\theta', [\textsc{id} : \mathsf{TupleID}] \oplus p) \wedge \theta' \in T) :: T, t := T - \{\theta\}, \theta \rangle$$

$$t := \mathsf{copy}(T, p) \triangleq \ \langle \| \ \theta : \theta = \theta'.(\mathcal{M}(\theta', [\textsc{id} : \mathsf{TupleID}] \oplus p) \wedge \theta' \in T) :: t := \theta \rangle$$

We choose to model the non-deterministic tuple selection $\theta = \theta'.(\mathcal{M}(\theta', [\textsc{id} : \mathsf{TupleID}] \oplus p) \wedge \theta' \in T)$ in the remove and copy macros by means of *non-deterministic assignment* [7]. In a non-deterministic assignment of the form $x := x'.Q$, the variable x is assigned a value x', selected non-deterministically among those satisfying condition Q. In our case, we use a similar notation to select non-deterministically a single matching tuple θ', bind it to the tuple θ, and use it to quantify the three-part notation. Because a single tuple is selected and bound, the parallel operator in the three-part notation serves the purpose of creating a quantified statement, and does not express parallel execution of multiple statements.

With these helper functions, the definitions of the **rd** and **in** constructs follow naturally.

$$t := \mathbf{in}(T, p) \triangleq \ t := \mathsf{remove}(T, p) \ \textbf{if} \ \mathsf{matchExists}(T, p)$$
$$t := \mathbf{rd}(T, p) \triangleq \ t := \mathsf{copy}(T, p) \ \textbf{if} \ \mathsf{matchExists}(T, p)$$

It is worth noting that our semantics differ from the original Linda definitions in one minor way. The value of the returned tuple is *not* bound to the pattern. Instead, the constructs defined here bind the value to the explicitly named tuple type variable on the left hand side of the assignment statement.

[1] These statements use a three part notation of the form ⟨ **op** quantifiedVariables : range :: expression ⟩. The variables from *quantifiedVariables* take on all possible values permitted by *range*. If *range* is missing, the first colon is omitted and the domain of the variables is restricted by context. Each such instantiation of the variables is substituted in *expression* producing a multiset of values to which **op** is applied, yielding the value of the three-part expression. If no instantiation of the variables satisfies *range*, the value of the three-part expression is the identity element for ***op***, e.g., *true* when ***op*** is ∀. If the operator is + we generate a summation $\sum$, if the operator is ∧ we express universal quantification ∀.

The previous definitions describe the *blocking* form of the **in** and **rd** operations. In standard Linda processing, a process which encounters a blocking operation will suspend until a matching tuple is found. In UNITY, however, there is no notion of process blocking, but instead the statements are selected nondeterministically. Therefore, if these statements are selected when no matching tuple exists in the tuple space, they are equivalent to a skip. In other words, they have no effect and it is as if the statement to remove the tuple was *blocked* waiting for a matching tuple. In the example, in the case where no tasks exist in the tuple space, the consumer will be effectively blocked waiting for the **in**.

With this in mind, it is possible and meaningful to put these blocking operations in parallel with other statements. For example:

$$t := \mathbf{in}(T, p) \parallel count := count + 1$$

is expected to count the number of tuples removed by this statement. However, after the macro expansion of the **in**, the semantics of this statement are such that each time it is selected for execution, the counter increments even if no matching tuple is taken from the tuple space. This is because only the assignment to the left of the parallel bar is inhibited until a match is found.

In order to allow the more meaningful style of parallel assignment in which both assignments are inhibited until a match is found, we expose the matchExists predicate to the programmer, enabling the following assignment statement with the correct counting semantics, similar to that shown in the consumer example:

$$t := \mathbf{in}(T, p) \parallel count := count + 1 \textbf{ if } \mathsf{matchExists}(T, p)$$

It should be noted that the same parameters, namely (T, p), must be used for the **in** and the matchExists in order to have the desired effect.

Extended constructs. While the above constructs define the basic Linda operations, there are a number of extensions which have been shown to be useful when programming with Linda. We define them here.

In contrast to the blocking version of the **in** and **rd** which have no effect until a matching tuple is found, it is sometimes desirable to take action when no matching tuple exists. Therefore, non-blocking or probing versions of both the **in** and **rd** have been introduced to assign the value ε to the tuple if no tuple matching the specified pattern exists in the tuple space at the moment when the operation is performed. The formal definitions

build upon the blocking operations.

$$t := \textbf{inp}(T, p) \triangleq t := \mathsf{remove}(T, p) \textbf{ if } \mathsf{matchExists}(T, p)$$
$$\| \; t := \varepsilon \textbf{ if } \neg\mathsf{matchExists}(T, p)$$

The definition of the probing read, **rdp**, is identical to **inp** except the copy macro is used in place of remove, leaving the tuple space unchanged after the execution of the operation involving a successful match.

Another useful operation is to remove or copy *all* tuples matching a pattern from the tuple space in either a blocking or non-blocking manner. For simplicity we introduce another macro which creates a set containing all tuples that match a given pattern, p, in a named tuple space, T:

$$\mathsf{matchingSet}(T, p) \equiv \langle \textbf{set}\, \theta : \mathcal{M}(\theta, [\textsc{id} : \mathsf{TupleID}] \oplus p) \wedge \theta \in T :: \theta \rangle$$

Using this definition, we can easily define the blocking version of the removal construct which simultaneously finds and removes all matching tuples in T. The result is bound to a variable of the type set of tuples. Similarly, the non-blocking construct is extended to return the emptyset if no matches are present in the tuple space.

$$s := \textbf{ing}(T, p) \triangleq s, T := \mathsf{matchingSet}(T, p), T - \mathsf{matchingSet}(T, p) \textbf{ if } \mathsf{matchExists}(T, p)$$
$$s := \textbf{ingp}(T, p) \triangleq s, T := \mathsf{matchingSet}(T, p), T - \mathsf{matchingSet}(T, p)$$

By removing the condition matchExists for the probing operation, the macro for **ingp** unconditionally returns the value returned from matchingSet. If no matches are found, matchingSet returns the the empty set, $\emptyset$, which is exactly the desired functionality for the probing operation. The corresponding **rdg** and **rdgp** can be constructed by removing the assignment to the tuple space T in the above definitions.

9.2 LIME

The features provided by the core Linda model as presented in Section 9.1 already provide for coordination among components that execute concurrently. However, the concept of context as captured by the Linda tuple space is a static one. Even in a distributed setting, Linda relies on the accessibility of a persistent, globally available tuple space that resides permanently on a pool of constantly interconnected machines. The approach we describe in this section makes different assumptions that are motivated by the unique characteristics of mobility.

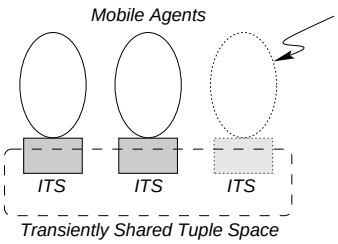


Figure 9.4: Transiently shared tuple spaces associated with each agent.

The ability of Linda to decouple processes from one another continues to hold a great deal of appeal in an open, mobile environment where components that have no a priori knowledge of one another may come into contact and need to work together. Further, the simplicity of the Linda model is attractive in the complex mobile environment. These advantages have motivated the LIME model (Linda in a Mobile Environment) which associates a tuple space, called the *interface tuple space* (ITS), with each mobile component. Initially we will consider only one ITS, however this will be expanded in a later discussion. This ITS is accessed by the component using the conventional Linda **in**, **rd**, and **out** operations as previously described. The semantics of these operations is conceptually unaltered; simply, the tuple space they are evaluated over is the component's ITS rather than a globally known one. The mobile component and its corresponding ITS migrate as a unit.

To enable coordination through the interface tuple space when connectivity is available, LIME defines a dynamic composition of ITSs conditional on connectivity. Effectively the accessible content of the ITS expands as more components are connected, and contracts as components disconnect, as shown in Figure 9.4.

To this point we have been intentionally vague about the definition of the mobile component. Conceptually either a physically mobile or logically mobile component can be associated with an interface tuple space, but the process centered model of coordination in Linda leads us to a two tiered model as shown in Figure 9.5 where logical processes, or mobile agents, are the active components in the system. As such, each mobile agent is associated with an ITS while the hosts on which these processes reside, the mobile hosts, are simply containers. A host does not have its own tuple space. We assume that communication exists among all agents on the same host. Therefore the composition of the tuple spaces of the mobile agents executing on a host constitutes the first tier or *host-level* tuple space. Host tuple spaces can be further composed among the mobile hosts that are directly or transitively connected, providing the second tier of abstraction or the *federated* tuple space. Each mobile agent accesses the current federated tuple space by issuing operations on its own ITS. Thus an agent's interaction with the data space remains consistent despite the dynamic nature of the data.

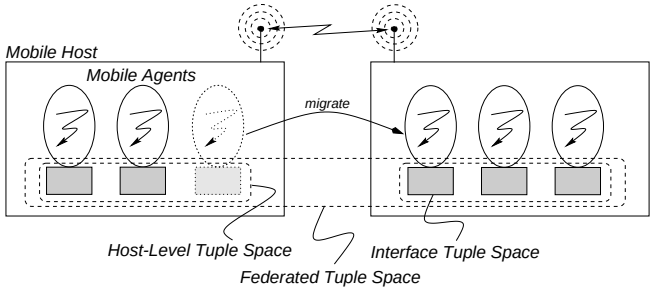


Figure 9.5: Physical and logical mobility, a two tier structure.

Standard LIME operations	**out**$[d](T,t)$	$t :=$ **in**$[x,y](T,p)$ $t :=$ **rd**$[x,y](T,p)$
Probing operations	$t :=$ **inp**$[x,y](T,p)$	$t :=$ **rdp**$[x,y](T,p)$
Group operations	$s :=$ **ing**$[x,y](T,p)$ $s :=$ **ingp**$[x,y](T,p)$	$s :=$ **rdg**$[x,y](T,p)$ $s :=$ **rdgp**$[x,y](T,p)$
Reactions	$\rho ::$ $s(\tau)$ **reacts-to**$[x,y](T,p,mode)$ **enableReaction**(ρ) **disableReaction**(ρ)	
Miscellaneous	matchExists$[x,y](T,p)$ isConnected(d)	createTuple$(\ldots)$

Figure 9.6: Summary of the LIME macros. T is a TupleSpace, t is a Tuple, p is a pattern Tuple, s is a set of tuples, x and y are formals of AgentID or AgentID subtypes, and ρ is a unique statement label. The parameter of createTuple is a sequence of any number of values (actuals) and types (formals).

Formalizing agent location and connectivity. As in the previous section, we are concerned with providing formal semantics for our model. The independence of mobile components and the transient connectivity based on location play central roles in the mobile environment, and are the first concepts formalized in this section. A summary of all formal LIME concepts appears in Figure 9.6

While regular UNITY programs can be specified independently as in the producer/consumer system of Figure 9.2, composition through union is static and all variables with the same name are implicitly shared for the duration of the system execution. For mobility we need a more dynamic composition to account for transient sharing. For this reason we move to the Mobile UNITY specification model in which the variables of each program are considered unique, and no variables are implicitly shared. Instead, the rules for sharing variables are made explicit in a new section of the specification called the **Interactions** section. Figure 9.7 shows a mobile version of the producer/consumer example specified in

Mobile UNITY and using LIME constructs defined in the remainder of this section. The programs in the Mobile UNITY specification are the active components of the system, and therefore represent the mobile agents.

In Mobile UNITY, each program component has a special location variable λ, which we use to define the host upon which the mobile agent is executing. This can be considered the IP address of a host. Agents migrate by assigning a new value to this location variable. Connectivity among agents is formally expressed as a symmetric and transitive relation κ. Agents are *connected* when they are on the same host, when the hosts they reside on are directly connected, or when the hosts they reside on are transitively connected. When an agent migrates, the κ relation changes to reflect the new configuration. Changes in connectivity among hosts are intentionally left outside of the specification, but nevertheless contribute to the κ relation. In a scenario where agents do not migrate, connectivity among the agents defines the connectivity among hosts. This is merely a consequence of choosing agents as the only active components in the system.

In many cases, both in physical and logical mobility, it is desirable for an application to access the current connectivity of mobile components. Therefore, exposing the κ relation to the programmer seems a natural course of action. However, directly exposing κ actually gives the programmer more power than can be realized in the mobile environment.

Consider the case where agent programs can use κ directly, for example to query whether agent b is connected to agent c with the predicate $b\,\kappa\,c$. If b and c are out of range of a, there is no reasonable way for a to resolve the query about components it cannot communicate with. However, it is both useful and reasonable, to allow queries about a's current connectivity state. Therefore, we allow restricted access to the κ relation through a function $\mathsf{isConnected}(b)$ defined to return the result of $a\,\kappa\,b$ when the query is issued by agent a. This limits the range of the query to a's current connectivity context.

In the remainder of this section we continue to describe the informal semantics of the LIME model, interleaved with the formal semantics as defined in Mobile UNITY.

9.2.1 Dynamic tuple space configuration

In LIME, no static, globally accessible tuple space exists a priori. Rather, the contents of each component's ITS is reconfigured dynamically and automatically by the system according to the current composition of the community of mobile components. In a sense, LIME takes the notion of decoupling proposed by Linda further, by effectively decoupling the mobile components from the global tuple space coordinating them.

A question arises at this point about the criteria needed to rearrange the contents of a mobile agent's tuple space as connections come and go. Clearly when a new connection is made, the tuples of all connected components should be visible to all connected components. Disconnection is slightly more complex as one must decide which agents keep which tuples.

System *Producer/Consumer*
  Program *Producer* **at** λ
    declare
      $jobs$: TupleSpace
      $jobType$: $\{\mathrm{A}, \mathrm{B}\}$
    initially
      $jobs = \emptyset$
    assign
      $jobType := \mathrm{A}$
      $[\!]$ $jobType := \mathrm{B}$
      $[\!]$ **out**$[Consumer](jobs,$
        createTuple$(jobType, jobInfo()))$
  end
  Program *Consumer* **at** λ
    declare
      $jobs$: TupleSpace
      $curTask, myJobPattern$: Tuple
      $count$: Integer
    initially
      $myJobPattern =$ createTuple$(\mathrm{A}, \mathrm{String})$
      $[\!]$ $curTask = \varepsilon$
      $[\!]$ $count = 0$
    assign
      $(curTask :=$ **in**$(jobs, myJobPattern)$
        $\|$ $count := count + 1)$
        if $curTask = \varepsilon$
          $\wedge$ matchExists$(jobs, myJobPattern)$
      $[\!]$ $curTask := \varepsilon \,\|\, doJob(curTask)$
  end
  Components
    Producer $[\!]$*Consumer*
  Interactions

end

Figure 9.7: A mobile producer/consumer.

This tuple selection criteria is provided by introducing the notion of tuple location. The location of a tuple is defined as the identifier of a mobile agent. In other words, when an agent migrates (changes location), all tuples associated with that agent migrate with the agent. By default, the location of a tuple is the agent which inserts it into the shared tuple space.

An agent's ITS always contains all the tuples which are the responsibility of that agent. The ITS also contains the tuples which are the responsibility of other connected agents, but no tuple can be in the ITS if its responsible agent is not connected. Because tuples and agents migrate together, the agent's ITS changes each time its connectivity changes.

Formalizing tuple location. Similar to the tuple identifier field, tuple location is introduced to aid in the specification but it is desirable for it to remain transparent to the programmer. Therefore, we expand the createTuple() function to insert a new labeled field, CUR, of type AgentID as a formal. This field will be managed by the macro definitions defined later in this section.

Formalizing tuple space reconfiguration. The definition of the changing content of the shared tuple space is handled through the shared variables of Mobile UNITY. Specifically, the identically-named tuple space variables in multiple agents are transiently shared when connectivity is available. The Mobile UNITY model of shared variables is dynamic. By default, no variables are shared across programs; the conditions for sharing are explicitly defined by the programmer. To this end, Mobile UNITY provides a construct which names the variables to be shared, the value the shared variable assumes when the condition is established (**engage**), and the value each variable takes when the condition is falsified (**disengage**). Intuitively, the condition for sharing is the existence of connectivity, the engagement value is the union of the tuple spaces and the disengagement values partition the contents of the shared tuple space according to tuple location and the new connectivity.

For convenience we define an operator to generate a set containing all tuples from a tuple space T with a specific responsible agent a. Intuitively, $T \downarrow a$ identifies all tuples which a is responsible for.

$$T \downarrow a \triangleq \langle \mathbf{set}\, \theta : \theta \in T \wedge \theta.\mathrm{CUR} = a :: \theta \rangle$$

With this definition, the engagement and disengagement of tuple spaces is defined as:

$$
\begin{array}{ll}
a.T \approx b.T & \textbf{when } a \,\kappa\, b \\
& \textbf{engage } a.T \cup b.T \\
& \textbf{disengage } \langle \cup c : c \,\kappa\, a :: c.T \downarrow c \rangle,\ \langle \cup c : c \,\kappa\, b :: c.T \downarrow c \rangle
\end{array}
$$

This expression describes the fact that the two variables corresponding to the tuple spaces of agents: a and b are shared when connectivity exists between the agents, i.e., $a \,\kappa\, b$. The now shared variable takes on a value representing the contents of the tuple spaces of both agents calculated with set union. Correctness of the engagement clause relies on the commutativity of the union operator to generate the same value whether $a.T \approx b.T$ or $b.T \approx a.T$, and on the fact that the tuple space is a set rather than a multiset, ensuring that when $a.T \approx a.T$, no tuple duplication takes place. On disengagement, or falsification of the sharing condition $a \,\kappa\, b$, the shared tuple space variable is partitioned such that the contents of the tuples spaces of each agent reflect the tuples available in the new connectivity. In other words, after a disconnection, the visible tuple are only those whose responsible agents are still connected to one another.

By specifying the interface tuple space as a shared variable, whenever a tuple in inserted or removed by one agent, the change in the tuple space is immediately visible to all connected agents. This naturally reflects the standard Linda concept of a global tuple space, while the reconfiguration due to mobility is transparently handled by the above expression.

9.2.2 Location Extended Constructs

The modifications to Linda mentioned thus far enable a style of mobile application development we refer to as *location transparent*. Processes need not know the details of connectivity, the identity of the other processes, or the distribution of data in order to successfully interact with one another. Instead, applications use the standard content-based Linda operations over a tuple space that changes by some mechanism which the programmer need not be concerned with.

An alternate style of application development, *location aware programming*, takes into account application knowledge of the changing connectivity, and allows a programmer to have control both over where data resides and from where it is accessed. These extensions are motivated by a fundamental difference between transiently shared tuple spaces and persistent, global tuple spaces.

For example, consider a producer/consumer system where the producer only generates tasks for a specific, known consumer, and this consumer is the only process that services these tasks. In the Linda model, as shown in Figure 9.8, this is straightforward, even in the case where the producer terminates after producing the task. In the mobile

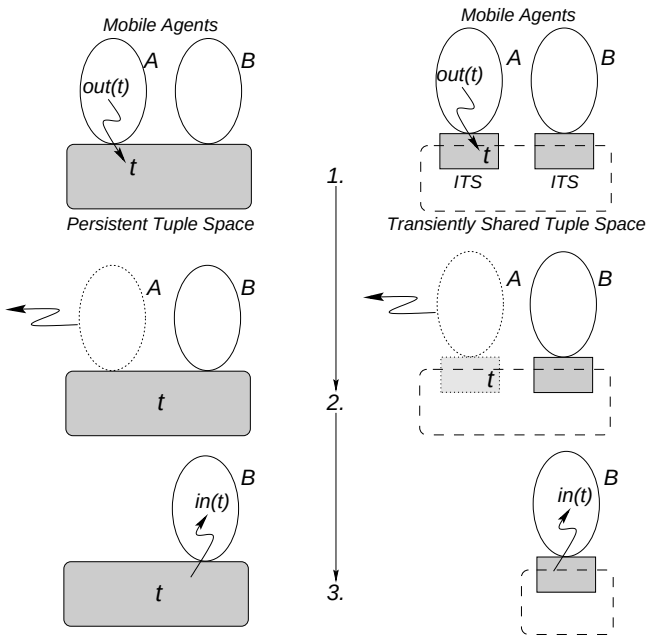


Figure 9.8: Persistent Linda tuple space vs. Lime transiently shared tuple Space.

setting, the produced data is visible to the consumer only while connected. If the consumer requests the data at this time, it will be successful. However, if the consumer fails to issue the request during connectivity, a later query will not succeed, but it will block instead.

This scenario suggests the need to allow an agent to specify an alternate location (agent) for the tuple. This is straightforwardly accomplished by augmenting the output operation with an agent identified as the intended destination location for the tuple, as in **out**$[d](T, t)$. In other words, whenever connectivity allows, the destination agent d should be made the agent responsible for the tuple, thus migrating the tuple with the destination rather than the source. In the producer/consumer example, the producer can output tasks with the consumer as the destination, and after a disconnection, the tuples remain accessible to the consumer. Thus, if the consumer requests a task after disconnection, as in the case described in Figure 9.8, the task is accessible because it has migrated with the consumer.

It should be noted that if the destination agent is not connected when the tuple is written, the writing agent must assume temporary responsibility for the tuple. If not, then when the tuple space is partitioned due to disengagement, the tuple would simply disappear because no agent was responsible for keeping it. A tuple whose destination is different from its currently responsible agent is referred to as *misplaced*. When connectivity is established between a and d, the responsibility for the tuple is transferred to d by a process we refer to as *tuple migration*.

CUR : x	DEST : y	Defined projection
AgentID	AgentID	entire currently shared tuple space
AgentID	$\langle$AgentID $i : i \in \{c, d\}\rangle$	current shared tuple space, destination of c or d
$\langle$AgentID $i : i \in \{a, b\}\rangle$	AgentID	a or b's responsibility, any destination
$\langle$AgentID $i : i \in \{a, b\}\rangle$	$\langle$AgentID $i : i \in \{c, d\}\rangle$	a or b's responsibility, destination of c or d

Figure 9.9: Summary of the meaning of several possible query projections.

It is important to note that specifying a destination location does not enforce a notion of ownership. Misplaced tuples still belong conceptually to a shared tuple space, and can be withdrawn or read by any connected process. Thus, it may be the case that another agent can remove the tuple from the space before the intended destination agent even has a chance to see it. This is consistent with the Linda model, where no notion of ownership is associated with the tuples. Our notions of responsibility and destination are only clues given to the system to relocate tuples to the best position according to some application specific notion.

Just as it can be useful to write tuples to a specific location, it can be equally useful to read or remove data from a specific location. Consider a consumer process only willing to accept tasks from a specific producer. In this case, we allow the consumer to query a projection of the tuple space, where the projection is defined through the responsibility attributes of the tuple, or more specifically the current and destination agents. The operations take the form **in**$[x, y](T, p)$ and **rd**$[x, y](T, p)$, where the values in square brackets define the projection of the tuple space over which to resolve the query. Specifically x restricts the value of the currently responsible agent and y restricts the intended destination agent. When these queries are used, both x and y must be formal AgentID fields. To enable a specific subset of agents to be matched for either x or y, we use the subtype definitions that extend the power of the matches function as described in Section 9.1. For example, to match either agent a or b, the subtype $\langle$AgentID $i : i \in \{a, b\}\rangle$ replaces x or y. To specify all of the agents on the same host as agent a, the subtype definition is $\langle$AgentID $i : i.\lambda = a.\lambda\rangle$. The formal AgentID matches any agent. A description of the projections defined by different combinations of x and y values appears in Figure 9.9.

Mathematically, arbitrary subtypes can be defined, but not all combinations of current and destination AgentID subtypes yield computable operations. Consider a system as shown in Figure 9.10 of two disconnected hosts A and D. Agents a and b are on host A while agents c and d are on host D. Agent d can write the query **in**[$\langle$AgentID $i : i.\lambda =$

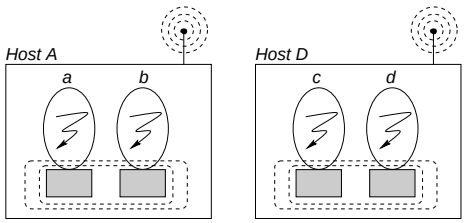


Figure 9.10: A sample LIME system with two disconnected mobile hosts, A and D, each with two mobile agents. The inner dashed line indicates the host-level tuple space while the outer dashed line indicates the federated tuple space.

$a.\lambda\rangle, \mathsf{AgentID}](T, p)$ to restrict the query to the projection of the tuple space to tuples whose currently responsible agent are on the same host as agent a. Although the subtype cannot be computed because d cannot access this information through the isConnected() function, the query can still be correctly computed based on available information. Specifically, agent d can discover that it is not connected to agent a because $\mathsf{isConnected}(a)$ is false. Therefore, by the transitivity of the κ relation, d can deduce that it is not connected to any agents a is connected to. Therefore, the projection defined by the operation is empty and the operation blocks. If this had been a probing operation, the result should have been to return ε.

If d uses the same subtype to restrict the destination field of the tuples, the result of the query can no longer be computed. Consider the same scenario where d issues the query $\mathbf{in}[\mathsf{AgentID}, \langle \mathsf{AgentID}\ i : i.\lambda = a.\lambda\rangle](T, t)$ in which d asks for all tuples in the current tuple space which are destined for agents on the same host as a. Unlike the previous case, there is no relationship between the currently connected agents and the value of the destination fields of the tuples, and the projection cannot be defined.

To prohibit such impossible queries, we restrict the subtype of the y projection parameter to only contain explicit sets of agents as in $\langle \mathsf{AgentID}\ i : i \in \{a, b\}\rangle$. Because evaluation of this subtype does not involve the calculation connected agents, the query can always be successfully performed.

In general, if the agents specified by the d parameter are not connected to the agent issuing the query, a blocking operation will block and a probing operation will return ε. It is important to note that the programmer cannot distinguish between a probe returning ε for connectivity reasons versus the unavailability of a matching tuple.

To allow operations in parallel with the blocking query operations, the matchExists() function is extended to include location parameters, as in $\mathsf{matchExists}[x, y](T, p)$.

Formalizing location specific output. As described earlier, the labeled tuple field CUR is used to specify the agent responsible for the tuple. To track the intended destination, we introduced already a similar labeled field DEST which is added by createTuple. When

an agent a performs $\mathbf{out}[d](T,t)$, the tuple t is written with a as the currently responsible agent and d as the destination, even if d is connected. The atomic migration of the tuple to d when connectivity exists is discussed shortly. As with the standard Linda constructs of Section 9.1, the augmented constructs of LIME are specified as macro definitions, and in fact, are built upon the Linda macros. The formal semantics for agent a inserting a tuple t are:

$$\mathbf{out}[d](T,t) \triangleq \mathbf{out}(T, [\text{CUR}:a, \text{DEST}:d] \oplus t)$$

For notational convenience, programmers that wish to specify operations in a location transparent mode are not forced to use the location augmented construct, but instead may continue to use the standard Linda **out** notation. In this case, when agent a issues $\mathbf{out}(T,t)$, the operation is equivalent to $\mathbf{out}[a](T,t)$, as the default destination for tuples is to remain with the agent that created them.

Formalizing tuple migration. It is the intent of the destination field to suggest an agent to assume the responsibility for a tuple as soon as possible, but the formalism of tuple output always places the tuple initially as the responsibility of the writing agent. Therefore, we must construct an operation which is able to take advantage of a connection as soon as it exists to migrate tuples to their destination. In the case where the writing and the destination agents are connected, this tuple migration should be immediate. In the case where connectivity does not immediately exist, the migration should occur as soon as connectivity is available. This style of operation, namely taking an action opportunistically when a condition is true, is described naturally using the reactions concept of Mobile UNITY.

The reactive statement extends UNITY with the capability to specify actions that must be executed immediately after a given condition is established rather than eventually as dictated by the fair interleaving semantics. For example, reactions can be used to model interrupts or event driven computations. Reactive statements can appear with an individual program specification or within the **Interactions** section. The reactive statements of the entire system are logically combined to form a single reactive program that is executed to fixed point after the execution of every conventional (non-reactive) statement. It is the responsibility of the programmer to ensure fixed point is actually reached for the reactive program. The syntax of a reactive statement, s **reacts-to** p, augments a conventional statement, s, with a reactive clause that strengthens its guard, p. The introduction of reactive statements affects the UNITY logic, and proper inference rules are needed to prove the correctness of reactive programs. The interested reader can find details about this in [50].

For our purposes, the reactive statement is used to ensure that whenever connectivity is available, misplaced tuples are migrated to their destinations. This is formally expressed

in the following where a tuple migrates from agent c to agent d:

$$\langle \| \ \theta, c, d : \theta = \theta'.(\theta' \in d.T \wedge \theta'.\text{CUR} = c \wedge \theta'.\text{DEST} = d) :: d.T = d.T - \{\theta\} + \{[\text{CUR} : d] \oplus \theta\} \rangle$$
$$\textbf{reacts-to } \textit{true}$$

Although connectivity is not explicitly mentioned in this formula, it is implicit that tuples can migrate only from c to d when connectivity is available. This can most easily be seen by interpreting the above formula from the perspective of agent d. Agent d selects a misplaced tuple from its tuple space, $d.T$. The tuple is migrated to d by removing the tuple and reinserting it with the CUR field properly reset. Because $d.T$ is the shared tuple space, the presence of a tuple belonging to c guarantees that c is connected because a tuple cannot exist in a tuple space without the agent of its current field being connected.

The semantics of reactions, i.e., immediate execution after each non-reactive statement, guarantee that tuples migrate atomically when connectivity is available. If a tuple is written while the current and destination agents are connected, the reaction is fired immediately after the **out** operation, creating the appearance to the programmer of instantaneous tuple migration to the destination. Alternately, when components come into range and have misplaced tuples which can now be migrated to their destinations, the reaction executes in the same atomic step that engages the previously disconnected tuple space variables, thus migrating all misplaced tuples from the programmer's perspective in one atomic step.

Formalizing location dependent queries. Although the existence and management of the CUR and DEST fields are hidden from the programmer, the location extended LIME construct provide limited access to tuple location information. When agent a issues a query, the specifications are as follows:

$$t := \textbf{in}[x, y](T, p) \triangleq t := \textbf{in}(T, [\text{CUR} : x, \text{DEST} : y] \oplus p)$$
$$t := \textbf{inp}[x, y](T, p) \triangleq t := \textbf{inp}(T, [\text{CUR} : x, \text{DEST} : y] \oplus p)$$

As previously described, x and y can be either the formal AgentID or a subtype of the same type, properly restricted for the destination field. These formal values become part of the pattern tuple and the $\mathcal{M}$ function effectively accesses the defined projection.

Because the query is specified by agent a, the tuple space T is the shared tuple space as perceived by agent a. Thus, if none of the agents specified by x are connected to a, there will not be any tuples to match the current field. In this case, the blocking operations block and the probing operations return ε.

As with the augmented output operation, we still allow the non-augmented versions in a program with the meaning that x and y are both the formal AgentID, thereby evaluating the operation over the currently shared tuple space. The augmented read and group operations can be built in a similar manner.

9.2.3 Reactions

Linda presents a computational model in which data is pulled from the data space on demand by a process by issuing one of the query operations. While this is a minimal model to enable process coordination, the event model in which a process registers for specific information and the environment pushes that information to the process when it becomes available is a useful extension. This is especially true in the dynamic mobile environment where taking advantage of a transient connection can be critical to enabling application progress.

Both JavaSpaces [35] and T Spaces [33] provide event notification within the Linda environment. In these systems, a programmer registers interest in the operations performed over the tuple space. In other words, an application can listen for the writing or reading of a tuple matching a specific pattern. When these operations are performed, the tuple space implementation runs the programmer's registered method.

LIME modifies this notion of event programming by focusing on the state of the system rather than the performing of an action. Specifically, the programmer specifies an action to be taken when a tuple matching a pattern is present in the currently shared tuple space. With this model, any system property which can be represented as data can be reacted to by adding it to the tuple space.

As mentioned in the tuple migration description of Section 9.2.2, Mobile UNITY provides the **reacts-to** primitive to specify an action to be taken when the system is in a specified state. LIME reactions are similar to Mobile UNITY reactions in both form and function. The most fundamental difference between the two is that LIME reactions are limited to reacting only to the presence of tuples in the tuple space and not to arbitrary predicates. Second, for simplicity of discussion, we initially limit LIME reactions to firing only one time. In other words, after the action is executed the first time, the LIME reaction becomes a skip. We will later extend our model to allow a reaction to fire multiple times.

As with Mobile UNITY reactions, a LIME programmer can specify multiple LIME reactions, all of which are combined to form a single reactive program. This reactive program is executed to fixed point after each change in the system state. Intuitively, the LIME reactive program reaches fixed point when there are no more matching tuples in the tuple space or all reactions are disabled.

The atomic execution of the reactive program is in contrast to the asynchronous notion of event notification of JavaSpaces and T Spaces. In these systems, after an event

occurs the registered processes are *eventually* notified. In contrast, the LIME model *atomically* executes the statement of the reaction in the same state that the matching tuple is found in the tuple space. Therefore, the statements of a LIME reaction can make assumptions about the state of the system when they are executing. Specifically, when the statement is run, the tuple is guaranteed to be in the tuple space. With this execution model, it is possible for one reaction to remove a tuple before all reactions have a chance to see it. In other words, there are no guarantees that if a tuple matching a pattern p is written that all reactive statements registered on that pattern will fire their actions.

Another important thing to note about reactions is that they fire only on the tuples currently accessible to the interested agent. In other words, if agents a and b are connected when b writes a matching tuple, a's reaction fires. After a and b disconnect, if b writes and immediately removes a tuple matching pattern p, a will never be notified of the existence of this tuple because it never existed in a's ITS.

To increase the flexibility of reactions beyond a single execution, we introduce the *mode* parameter. The model described up to this point has only reactions with mode being ONCE, i.e., each reaction fires exactly one time. We also provide the ONCEPERTUPLE mode in which a reaction fires one time for each matching tuple in the tuple space.

Further, LIME allows the programmer to explicitly control the enabling and disabling of reactions. When a reaction is disabled, it will not fire. When a reaction is explicitly enabled, it is treated as a new reaction, with no state. In other words, an enabled ONCEPERTUPLE reaction which is disabled and then reenabled will react to any matching tuples in the tuple space, whether or not it reacted on them before being disabled.

The specific form of a LIME reaction is $\rho :: s(\tau)$ **reacts-to**$[x, y](T, p, mode)$ where s is the statement to be performed when a tuple matching pattern p is found in the tuple space T. As in the query operations, $[x, y]$ indicates a projection of the tuple space to evaluate the reaction over. The variable τ is a free variable which can appear within the statement s and will be bound to the matched tuple when the action fires. The mode is either ONCE or ONCEPERTUPLE. The label ρ uniquely identifies this reaction and can be used as a parameter to the enabling and disabling functions as in **enableReaction**(ρ) and **disableReaction**(ρ).

To make the use of reactions more concrete, we return to the producer/consumer example and extend it to include *high priority* tasks generated by the producer. The idea is that a high priority task should be performed as soon as possible by the consumer, where as soon as possible is determined by connectivity. If the producer and consumer are connected when the task is generated, then the consumer should immediately take and perform the task. Alternately, if the components are not connected when the task is generated, as soon as connectivity is available, the consumer should take and complete the task. The generation of the high priority task is no different from the point of view of the consumer, simply the

task tuple is written with a different pattern indicating the increased priority, as shown in the first statement below. The consumer program contains the corresponding reactive statement with the pattern matching high priority tuples and the action (statement) that performs the action, as shown in the second statement below.

$$\mathbf{out}(jobs, \mathsf{createTuple}(\textsc{priority}, jobInfo()))$$

$$doJob(\tau)\ \mathbf{reacts\text{-}to}[\mathsf{AgentID}, \mathsf{AgentID}]$$
$$(jobs, \mathsf{createTuple}(\textsc{priority}, \mathsf{String}), \textsc{oncepertuple})$$

Formalizing reactions. Because the semantics of a LIME reactions closely match those of a Mobile UNITY reaction, the formalization if fairly straightforward, defining a LIME reaction as a Mobile UNITY reaction. Much of the complexity of the reaction macro deals with the bookkeeping necessary to ensure the proper execution pattern with respect to the mode. The details of the formalism are discussed below.

$$\begin{aligned}\rho :: s(\tau)\ \mathbf{reacts\text{-}to}[x, y](T, p, mode) \triangleq\\ \langle \| \ \theta : \theta = \theta'.(\mathcal{M}(\theta', [\textsc{id} : \mathsf{TupleID}, \textsc{cur} : x, \textsc{dest} : y] \oplus p)\\ \wedge\ \theta' \in T \wedge \theta'.\textsc{id} \notin reactedSet_\rho\\ \wedge (mode = \textsc{once} \Rightarrow reactedSet_\rho = \{\}))\\ :: s\{\tau/\theta\} \parallel reactedSet_\rho := reactedSet_\rho \cup \{\theta.\textsc{id}\}\rangle\end{aligned}$$

$$\mathbf{reacts\text{-}to}$$

$$\mathsf{matchExists}(T, [\textsc{id} : \mathsf{TupleID}, \textsc{cur} : x, \textsc{dest} : y] \oplus p) \wedge \mathsf{enabled}_\rho$$

To keep track of the tuples which have been reacted to and whether the user has enabled or disabled the reactions, two auxiliary variables indexed by ρ, namely $reactedSet_\rho$ and $\mathsf{enabled}_\rho$, are introduced. By subscripting these variables with the unique reaction identifier ρ, we are guaranteed that reactions will not interfere with one another.

The selection of a matching tuple uses the same nondeterministic selection notation as described in the Linda formalism for the copy function. The necessity to bind the matched tuple to the free variable τ prohibits the use of the copy function here, but the semantics are the same. A matching tuple is selected from the shared tuple space.

The condition for actually firing the reaction (executing the user action s) depends on the contents of the *reactedSet*. This variable tracks the identity of the tuples that have caused the user action to execute in the past. When s is executed, in parallel the unique tuple identifier is written to the set. A ONCEPERTUPLE reaction will only fire if the selected tuple has not been recorded in this set. A ONCE reaction will only fire if no tuples have been reacted to.

The user action $s(\tau)$ can be any regular UNITY statement except for a Mobile UNITY transaction. The reason for this exception comes directly from the Mobile UNITY formalism. For more details, see [50]. The LIME **in** operation can be used to remove the tuple from the tuple space, or a reaction can be enabled using the macros below.

The matchExists function is necessary as a condition to the reaction because we must guarantee that at least one tuple can be selected from the tuple space. Otherwise, the θ' variable cannot be bound and the formalism will be incorrect. The second condition for the reaction, namely $\mathsf{enabled}_\rho$ tracks whether the user has explicitly enabled or disabled the reaction with the following macros:

$$\begin{aligned} \textbf{enableReaction}(\rho) &\triangleq \mathsf{enabled}_\rho, \mathit{reactedSet}_\rho := \mathit{true}, \{\} \\ \textbf{disableReaction}(\rho) &\triangleq \mathsf{enabled}_\rho := \mathit{false} \end{aligned}$$

By clearing the *reactedSet* when a reaction is reenabled, it is treated as a new reaction which has not fired on any tuples in the tuple space. It is important to remember that the reactive program is executed after every regular statement, including a statement that enables a reaction. This means that a reaction may fire in the same atomic step as the statement which enables it, assuming a matching tuple exists in the tuple space.

Initially, each reaction must either be disabled or enabled with an empty reacted set. To easily allow the programmer to specify which state a reaction starts in, the **always** section of a program can contain one of the following predicates for each reaction:

$$\mathsf{initEnable}(\rho) \triangleq \mathsf{enabled}_\rho = \mathit{true} \wedge \mathit{reactedSet}_\rho = \{\}$$
$$\mathsf{initDisable}(\rho) \triangleq \mathsf{enabled}_\rho = \mathit{false}$$

9.3 Practical Considerations in Implementing the Model

The model in the previous sections provides a clean abstraction for describing the transient interactions of mobile components using tuple spaces. The formalism allows us to specify and reason about the mobile systems and serves as a first step toward the development of reliable mobile applications.

The next logical step is to actually implement these mobile systems in such a way that their characteristics match those defined by the formal model. Rather than begin the development process from scratch for each project, we choose to provide an implementation of the LIME model as a middleware. Intuitively, the LIME middleware sits between the operating system and the applications, completely hiding the details of the network communication. LIME provides an interface for tuple space coordination which is very similar to the abstractions described by the model.

The implementation is not identical to the model for several reasons. First, we choose for the implementation Java, an object oriented, sequential language. This cannot match exactly the properties of Mobile UNITY, a programming notation with no sequential constructs and transiently shared memory. Further, some of the constructs as previously defined are often inefficient to implement in a distributed mobile environment and must be weakened to define similar functionality at a lower cost.

Next, the mobile environment is by definition also a distributed environment where parallel processing is possible. An efficient implementation should exploit parallelism whenever possible while maintaining the semantics of the model.

The remainder of this section describes how some of the constructs can be implemented to exploit parallelism and how some constructs must be weakened to increase parallelism.

9.3.1 Distribution and Parallelism

The implementation of the LIME model describes a distributed system with multiple active processes (mobile agents), multiple processors (on each mobile host), and data stored in multiple physical locations (in the memory of the hosts). The LIME model leads to a natural distribution of the contents of the tuple space among the hosts according to the location of the mobile agents and the agent currently responsible for the tuple. Recall that we refer to the current tuple field as the location of the tuple. This relates directly to an implementation in which a tuple resides in the memory of the host where the responsible agent is executing. When a tuple is migrated in the model by changing the value of the current field, the implementation physically moves the tuple to the host of the new current agent. Therefore, if a host disconnects, no data needs to be moved because the tuples to be migrated with the agents on that host have already been moved into the memory of the host. The illusion of a transiently shared tuple space shared among connected components is implemented using the message passing primitives of the runtime system to perform remote queries and return results.

The implementation of the distributed tuple space model of LIME has many of the characteristics of a distributed database. Parallel operations can be performed on the data as long as they do not conflict. For example, multiple tuple space read operations can be performed simultaneously because the contents of the tuple space are not affected and the operations can be serialized in any order without affecting the correctness of the system. Further, multiple tuples can be simultaneously removed as long as they are removed from disjoint partitions of the tuple space. The physical distribution of tuples across hosts defines a natural partitioning of the shared tuple space. Non-conflicting operations can be performed independently on each of these partitions. Within a partition, the operations

themselves must be serialized. In other words, the operations of all local agents must be coordinated. Fortunately, this kind of local control is inexpensive to implement.

9.3.2 Probing Operations

Other operations cannot be as easily parallelized. For example, consider the probing operation **inp**[AgentID, AgentID](T, p) which queries the entire tuple space for a tuple matching the pattern p. The precise semantics of this operation as defined by our model requires that ε be returned only if no tuple exists in the state when the operation is executed. To phrase this another way, in a consistent snapshot of the distributed tuple space, no matching tuple exists and ε should be returned. If a matching tuple does exist, the semantics do not require a system-wide snapshot. The probing read and group operations have similar requirements.

While such a transaction can be implemented, the cost of this operation on the overall system can be significant, both in terms of message overhead as well as with respect to the time to perform the query in which all other processing is halted.

One way to eliminate the expense of system-wide transactions caused by probe operations is to limit them to a partition of the tuple space over which a transaction is more easily implementable. For example, within a host, transactions to coordinate agents require little overhead. It should be noted that a probe query can be done on either the local host or a remote host without a multi-host transaction. Only the host where the query is actually performed needs to guarantee the atomicity of the operation.

While restricting a probing operation to a single host clearly limits its power, the tradeoffs associated with system complexity, message overhead, and loss of parallelism must be considered. In fact, it is reasonable to provide both restricted and unrestricted versions of the operations, however the programmer must be aware of the impact on system performance when strong forms are used.

9.3.3 Reactions

Just as the probing operations which specify a projection which includes multiple hosts require a transaction, LIME reactions that respond to the presence of tuples located at multiple hosts have similar requirements. A first step toward limiting the scope of a reaction is to restrict the projection to a single agent or single host as was done for the probes, however this is not sufficient to remove the need for a transaction.

As described in the model, the user action s can be any program statement, including LIME operations which modify the tuple space. The implementation must guarantee that these operations are performed atomically with the firing of the reaction. If any of these operations involve queries of tuples on remote hosts, then a transaction is necessary to guarantee atomicity. It should be noted that even the **out** operation which specifies a

remote destination introduces the need for a transaction. The migration of the tuple occurs in the same atomic step as the writing of the tuple, which is in the same atomic step as the firing of the reaction. For this reason, we restrict the strong reaction to fire only over tuples on the same host as the agent that installed the reaction, and for the user action to only access and affect the local projection of the tuple space.

Another natural line for parallelism is to separate the operations of the differently named and therefore disjoint tuple spaces. Again, the user action is not restricted from accessing other tuple spaces, and may require a transaction involving multiple tuple spaces. The natural restriction in this case is to prohibit user actions from accessing any tuple space except the one specified on the right hand side of the reactive statement.

Weakening the model. While these restrictions allow us to build reactive statements that do not require unreasonable transactions involving multiple hosts, we have lost the ability to proactively push information to an agent from a remote projection of the tuple space. Clearly the above arguments show that the strong atomicity guarantees of the reaction make the implementation costly.

We therefore propose to extend the model itself with a new construct that has similar capabilities, but eliminates the tight coupling between the detection of the tuple and the firing of the user action. To this end, we define a *weak reaction* which can be installed on projections of the tuple space which span multiple hosts. By separating the execution from the detection, transactions are no longer required, and the system performance can be greatly enhanced.

Because this notion of weak reaction is radically different from that of strong reactions, we take the extra step of providing the formal definition of the new construct below. Syntactically, the specification of a weak reaction by the programmer includes the same elements as the specification of a strong reaction, namely a unique reaction label, the user action, the tuple space projection, the name of the tuple space, a pattern, and a mode.

$$\rho :: s(\tau)\textsf{weak-reacts-to}[x, y](T, p, mode)$$

To specify the semantics of this operation, we take advantage to its similar functionality to strong reactions, but force an asynchronous step between the identification of the tuple and the execution of the user reaction. To identify a tuple, we employ the LIME strong reaction, but define our own *system* action which places the identified tuple into a special variable called the *events* set, indexed by the reaction identifier. The events set serves as a temporary holding place for all tuples which should be reacted to, but for whom the user reaction has not fired. A second statement which executes in a different atomic step from the reaction removes tuples from the event set and fires the user action.

$$\rho :: s(\tau)\ \textbf{weak-reacts-to}[x,y](T,p,mode) \triangleq$$
$$\rho :: events_\rho := events_\rho \cup \{\tau\}\ \textbf{reacts-to}[x,y](T,p,mode)$$
$$[\!] \langle \| \theta : \theta = \theta'.(\theta' \in events_\rho) :: events_\rho := events_\rho - \{\theta\} \| s\{\tau/\theta\} \rangle$$
$$\textbf{if}\ events_\rho \neq \emptyset$$

One point to notice about this formula is the use of nondeterministic selection to remove the tuple from the *events* set. Similar to the definition of remove definition of Section 9.1, exactly one tuple will be selected, ensuring that only one user action s will fire at a time.

As with strong reactions, only tuples in the currently shared tuple space will be reacted to, meaning that after the hosts disconnect, any tuples written and removed from the remote tuple space during the disconnection will not be reacted to, even in the weak model.

9.3.4 Engagement and Disengagement

Two other points in the LIME model define atomic access to the distributed contents of the shared tuple space and therefore require transactions in order to be properly implemented: engagement and disengagement. On the engagement of tuple spaces, the sharing operation defines that the tuple space now contains the union of all connected tuple spaces. When combined with the reactive statement to migrates all misplaced tuples, a transaction is needed to ensure all misplaced tuples are atomically transferred and that all connected hosts have the same perception of which agents are accessible in the connectivity graph.

On disconnection, the tuple space is partitioned based on the current location field of the tuples. Because the implementation guarantees that all tuples are physically transferred to their destination as soon as connectivity is available, no tuples need to be migrated during a disconnection. However, a transaction is still necessary to ensure that all agents have the same view of the connectivity graph.

Although it is possible to weaken the engagement and disengagement semantics such that a transaction is not necessary, for the initial implementation of the model, we keep the strong semantics. By weakening the semantics of these actions, the consistency model of the tuple space significantly changes because each agent can potentially have a different perception of the contents of the tuple space at any given moment. To avoid these confusions, the implementation employs a system-wide transaction each time the connectivity graph changes.

9.4 Concluding Remarks

This chapter presented the formal model of LIME using constructs from Mobile UNITY. We then presented several implications of the model in a real implementation, showing where the model can and cannot be parallelized and suggesting the weakening of some operations to increase parallelism while remaining useful. The result is a complete specification of LIME, including the data structures and the operations available to manipulate these data structures.

The model serves two main purposes. First, an application programmer can build a formal model of their system using these abstractions. Because the constructs of LIME have been fully specified, the user specification can be formally reasoned about, and critical properties can be proven. The second objective of this specification is to serve as the complete definition of the concepts to be implemented in the middleware presented in the next chapter. Because the middleware is implemented in Java, a language fundamentally different from Mobile UNITY, we must be careful to maintain the same semantics put forth by this specification in the implementation.

Chapter 10

Implementing and Applying LIME

Middleware has emerged as a new development tool which can provide programmers with the benefits of a powerful virtual machine specialized and optimized for tasks common in a particular application setting without the major investments associated with the development of application-specific languages and systems. The approach is intellectually attractive and economical at the same time. For the programmer, middleware offers a clean model that can be easily understood and readily adopted without the need to acquire a new set of programming skills or to delve into the intricacies of a sophisticated formal model. For the software engineer, middleware provides a vehicle by which new concepts and design strategies may be packaged and disseminated without the high cost associated with complex tool sets and compilers. For these reasons, middleware is enjoying growing popularity in the distributed computing arena. Given the complexities associated with software involving mobile hosts and agents, middleware is expected to establish itself as an important new technology in mobility as well. This chapter presents a middleware which implements the abstractions of LIME.

This chapter begins with a brief outline of the interface presented to the programmer, followed by a description of the design and implementation of the middleware. Section 10.3 presents two specific applications built using LIME, and Section 10.4 provides reflections on the development process of LIME.

10.1 Programming with LIME

We begin this presentation of the LIME model by briefly commenting upon the programming interface that is currently provided in the implementation of LIME.

The class `LimeTupleSpace`, shown in Figure 10.1[1], embodies the concept of a transiently shared tuple space. Objects of this class are created by specifying an instance of the

[1]Exceptions are not shown for the sake of readability.

```
public class LimeTupleSpace {
  public LimeTupleSpace(Agent agent, String name);
  public String getName();
  public boolean isOwner();
  public boolean setShared(boolean isShared);
  public static boolean setShared(LimeTupleSpace[] lts,
                                  boolean isShared);
  public boolean isShared();
  public void out(ITuple tuple);
  public void out(AgentLocation destination, ITuple tuple);
  public ITuple in(ITuple template);
  public ITuple in(Location current, AgentLocation destination,
                   ITuple template);
  public ITuple inp(Location current, AgentLocation destination,
                    ITuple template);
  public ITuple rd(ITuple template);
  public ITuple rd(Location current, AgentLocation destination,
                   ITuple template);
  public ITuple rdp(Location current, AgentLocation destination,
                    ITuple template);
  public RegisteredReaction[]
         addStrongReaction(LocalizedReaction[] reactions);
  public RegisteredReaction[] addWeakReaction(Reaction[] reactions);
  public void removeReaction(RegisteredReaction[] reactions);
  public RegisteredReaction[] getRegisteredReactions();
  public boolean isRegisteredReaction(RegisteredReaction reaction);
}
```

Figure 10.1: The class `LimeTupleSpace`, representing a transiently shared tuple space.

`Agent` class, which essentially provides a means to uniquely identify a mobile agent. The thread associated to such agent object will be the only one allowed to perform operations on the `LimeTupleSpace` object; accesses by other threads will fail by returning an exception. This represents the constraint that the ITS must be permanently and exclusively attached to the corresponding mobile agent.

In LIME, agents may have multiple ITSs distinguished by a name, which is the second parameter for the constructor of `LimeTupleSpace`. The name determines the sharing rule; only tuple spaces with the same name are transiently shared. For instance, this enables an agent to exchange information with a service broker about the available CD resellers by transiently sharing the corresponding ITS, and then subsequently share information about a given title and the payment options with the reseller selected through a different ITS, thus keeping separate the information concerned with different tasks and different roles.

Agents may have also *private* tuple spaces, i.e., not subject to sharing. A private `LimeTupleSpace` can be used as a stepping stone to a shared data space, allowing the agent to populate it with data prior to making it publicly accessible, or it can turn out to be

useful just as a primitive data structure for local data storage. As a matter of fact, all tuple spaces are initially created as private, and sharing must be explicitly enabled by calling the instance method `setShared`. The method accepts a boolean parameter specifying whether the transition is from private to shared or vice versa. Calling this method effectively triggers engagement or disengagement of the corresponding tuple space. Sharing properties can also be enabled in a single atomic step for multiple tuple spaces owned by the same agent by using the class method `setShared`.

`LimeTupleSpace` contains also the Linda operations needed to access the tuple space, as well as their variants annotated with location parameters. Tuple objects must implement the interface `ITuple`, defined in a separate package that provides a definition for a Linda tuple space that is independent on the actual runtime support used. As for location parameters, LIME provides two classes, `AgentLocation` and `HostLocation`, which extend the common superclass `Location` by enabling the definition of globally unique location identifiers for hosts and agents. Objects of these classes are used to specify different scopes for LIME operations. Thus, for instance, a probe `inp(cur,dest,t)` may be restricted to the tuple space of a single agent if `cur` is of type `AgentLocation`, or it may refer the whole host-level tuple space, if `cur` is of type `HostLocation`. The constant `Location.UNSPECIFIED` is used to allow an unspecified location parameter. Thus, for instance, `in(cur,Location.UNSPECIFIED,t)` returns a tuple contained in the tuple space of `cur`, regardless of its final destination, thus including also misplaced tuples. Note how typing rules allow to constrain properly the nature of the current and destination location according to LIME rules. Thus, for instance, the `destination` parameter is always an `AgentLocation` object, as agents are the only carriers of a "concrete" tuple space in LIME. Specifying a `HostLocation` as a destination for a tuple would result in the impossibility to assign a responsible for the tuple when the host-level tuple space becomes partitioned due to disengagement. Note also how, in the current implementation of LIME, probe are always restricted to a subset of the federated tuple space, as defined by the location parameters. An unconstrained definition, like the one provided for `in` and `rd`, would involve a distributed transaction in order to preserve the semantics of the probe across the whole transiently shared tuple space.

All the operations retain the same semantics on a private tuple space as on a shared tuple space, except for blocking operations. Since the private tuple space is nonetheless permanently and exclusively associated to an agent, the execution of a blocking operation would immediately suspend the agent forever, waiting for tuples that no other agent is allowed to insert. In this case, a run-time exception is thrown instead.

The remainder of the interface of `LimeTupleSpace` is devoted to managing reactions; other relevant classes for this task are shown in Figure 10.2. Reactions can either be of type `LocalizedReaction`, where the current and destination location restrict the scope of the

```
public abstract class Reaction {
  public final static short ONCE;
  public final static short ONCEPERTUPLE;
  public ITuple getTemplate();
  public ReactionListener getListener();
  public short getMode();
  public Location getCurrentLocation();
  public AgentLocation getDestinationLocation();
}
public class UbiquitousReaction extends Reaction {
  public UbiquitousReaction(ITuple template,
                            ReactionListener listener,
                            short mode);
}
public class LocalizedReaction extends Reaction {
  public LocalizedReaction(Location current,
                           AgentLocation destination,
                           ITuple template,
                           ReactionListener listener,
                           short mode);
}
public class RegisteredReaction extends Reaction {
  public String getTupleSpaceName();
  public AgentID getSubscriber();
  public boolean isWeakReaction();
}
public class ReactionEvent extends java.util.EventObject {
  public ITuple getEventTuple();
  public RegisteredReaction getReaction();
  public AgentID getSourceAgent();
}
public interface ReactionListener extends java.util.EventListener {
  public void reactsTo(ReactionEvent e);
}
```

Figure 10.2: The classes `Reaction`, `RegisteredReaction`, `ReactionEvent`, and the interface `ReactionListener`, required for the definition of reactions on the tuple space.

tuple space scanned for matching, or `UbiquitousReaction`, that specify the whole federated tuple space as a target for matching. The type of reactions is used to enforce the proper constraints on the registration of reactions through type checking. These classes have the abstract superclass `Reaction` in common, which defines a number of accessors for the properties set on the reaction at creation time. Creation of a reaction is performed by specifying the template that needs to be matched in the tuple space, a `ReactionListener` object that specifies the actions taken when the reaction fires, and a mode. The `ReactionListener` interface requires the implementation of a single method `reactsTo` that is invoked by the runtime support when the reaction actually fires. This method has access to the information about the reaction carried by the `ReactionEvent` object passed as a parameter to the method. The reaction mode can be either of the constants ONCE and ONCEPERTUPLE, defined in `Reaction`. ONCE specifies that the reaction is executed only once and then deregistered automatically in the same atomic step. When ONCEPERTUPLE is specified instead, the reaction remains registered but it never executes twice for the same tuple.

Reactions are added to the ITS by calling either `addStrongReaction` or `addWeakReaction`. Only `LocalizedReaction` can be passed to the former, as prescribed by the LIME model. Due to the different semantics, this operation has different atomicity guarantees. The former guarantees that all the reactions passed as a parameter are registered in a single atomic step, i.e., processing of reactions takes place only after all reactions have been inserted in the `LimeTupleSpace`, and yet before any other operation takes place on it. The latter does not provide such guarantee, as weak reactions could be spread on multiple hosts and thus enforcing the property above would entail a distributed transaction among all the nodes involved.

Registration of a reaction in any case returns an object `RegisteredReaction`, that can be used to deregister a reaction with the method `removeReaction`. `RegisteredReaction` basically acts as a "ticket stub" for the registration of the reaction, and provides additional information about the registration process. The decoupling between the reaction used for the registration and the `RegisteredReaction` object returned allows for registration of the same reaction on different ITSs, or to register the same reaction with a strong and then subsequently with a weak semantics.

10.2 Design and Implementation of LIME

In this section we look behind the scenes of the LIME programmer interface, by providing some insights about the internal structure of the `lime` package and of the associated runtime support. The presentation will proceed through increasing levels of complexity. We first describe how the simple notion of a private, non-shared tuple space is made available through the `LimeTupleSpace` class. Then, we move on to describe the components that

enable the local transient sharing that determines a host-level tuple space. Finally, we show how the illusion of a federated tuple space enabling transient sharing across remote nodes is provided.

Private Tuple Space A private tuple space essentially provides a Linda tuple space that is permanently attached to an agent. It enjoys exclusive access to the tuple space and can leverage off the strong reaction feature of LIME. Furthermore, since the private tuple space can later be engaged through transient sharing, support for operations annotated with tuple location parameters must be provided as well.

The core functionality above is supported by two objects that belong to every `LimeTupleSpace` object. The first object has type `ITupleSpace` and provides exactly the functionality of a plain Linda tuple space, including blocking operations. The second object is of type `Reactor` and is in charge of running the reactive program constituted by the reactions registered on the `LimeTupleSpace` object after each operation.

When designing LIME, we had to face the decision about how to implement the core tuple space support. Analysis of available systems revealed that they provide a very rich set of features, with big variations in terms of expressiveness, performance, and often also semantics. The need for a simple, lightweight implementation, combined with the desire to provide support and interoperability with industry-strength products, led us to the development of an adaptation layer that hides from the rest of the LIME implementation the nature of the underlying tuple space engine. This layer is provided by a separate package called LIGHTS, developed by one of the authors. `ITupleSpace`, together with the already mentioned `ITuple`, and `IField`, are the interfaces that provide access to the core tuple space functionality. Adapter classes implementing this interfaces can be loaded at startup time to translate these operations into those of the tuple space engines supported. Currently, adapters are in place for our lightweight tuple space implementation and for IBM's TSpaces [33]. Short term activities include the development of an adapter for Sun's JavaSpaces [35].

Support for operations annotated with tuple locations relies on the `ITupleSpace` object, although a change in the format of tuples is performed along the way. In fact, the design choice we made was to represent tuple location parameters as tuple fields, in order to simplify the implementation of the corresponding extended operations and, as we will see later, to simplify the retrieval of misplaced tuples during engagement. Nevertheless, this representation is hidden from the programmer, who is prevented from tampering directly with the location fields (which would possibly lead to changes in the semantics) and can refer to location fields only through the corresponding parameters in the operations provided by `LimeTupleSpace`. Thus, upon insertion, a tuple specified by the programmer is augmented with two location fields representing the current and destination location. These fields are

then stripped down when operations accessing the tuple space, like **rd**, are performed. As it will be discussed in the remainder of this section, a third field containing a globally unique tuple identifier is also added, and it is used exclusively to support reactions with a ONCEPERTUPLE mode.

The `Reactor` object is the other key component of the `LimeTupleSpace`. It contains the list of registered reactions forming the reactive program, which gets changed through the methods of the `LimeTupleSpace` that add and remove reactions. The current implementation supports only reactions to changes in state and not to the mere occurrence of an operation. This means that execution of the reactive program must be triggered only when the contents of the tuple space changes, i.e., as part of the execution of the `out` method of the `LimeTupleSpace`. The requirement for the reactive program to run to fixed point after every such change is achieved by cycling through the whole list of reactions in a round robin fashion until no reaction is enabled to fire. A straightforward implementation of this processing would probe the whole tuple space for a tuple matching the reaction's template every time a reaction is evaluated. Clearly, this would be quite highly inefficient even for a small number of reactions and tuples, especially in the case of reaction listeners that insert tuples of their own. For this reason, our `Reactor` adopts an optimized strategy that mirrors and separates, during execution of the reactive program, the tuples written to the tuple space as a consequence of the firing of a reaction from those that have already been checked, thus avoiding looking at the same tuple more than once per evaluation of a reaction. This complicates the management of the tuple space during the evaluation of reactions because it must be kept consistent with the `Reactor`'s view. Nevertheless, in our experience this added complexity is far outweighed by the advantages gained, especially during the processing of ONCEPERTUPLE reactions which, as we will discuss, represent a major asset during development.

Host-Level Tuple Space Transient sharing of `LimeTupleSpace` objects is under the explicit control of the respective agent. Once sharing is turned on, a host-level tuple space is created. Implementation of this abstraction requires a host-wide, centralized management of the access to the individual tuple space objects, in order to properly enforce the semantics of transient sharing and to take into account engagement and disengagement of local tuple spaces. This management is provided by instances of the class `LimeTSMgr`.

At run-time, there exist one `LimeTSMgr` per each named transiently shared tuple space currently active. A `LimeTSMgr` object is created as soon as the first `LimeTupleSpace` instance with a given name is engaged. Subsequent engagements of `LimeTupleSpace` objects with the same name will refer to the same `LimeTSMgr`. Engagements of objects with a different name will refer to a different `LimeTSMgr`.

Upon local engagement of a given tuple space, the `LimeTupleSpace` object surrenders the control of its own `ITupleSpace` object. Thus, the implementation of the methods providing access to the tuple space no longer operate directly on the `ITupleSpace`. Instead, operation requests are forwarded to the corresponding `LimeTSMgr`, and the calling agent is suspended, waiting for the result. Operation requests are enqueued by the `LimeTSMgr`, which runs in a separate thread of control, and thus their execution is serialized. This way, synchronization among concurrent accesses performed through different `LimeTupleSpace` instances is obtained structurally, by confining concurrent accesses in a synchronized queue.

In our current implementation, not only the `LimeTupleSpace` surrenders control of its tuple space, but the contents of the `ITupleSpace` object are physically merged upon engagement in another `ITupleSpace` object associated with the `LimeTSMgr`. This latter object becomes then a concrete representation of the host-level tuple space. Similarly, the reactive statements of each `LimeTupleSpace` instance are all moved, upon engagement, into a `Reactor` object associated to the `LimeTSMgr`. The rationale for this design decision lied in the fact that this solution optimizes for tuple queries, especially if the underlying tuple space engine adopts indexing mechanisms to provide faster access to tuples, like in the case of T Spaces. Thus, this solution is appropriate in the case where changes in the configuration are not very frequent. However, experience with applications and the development of our own lightweight tuple space made us consider more carefully the alternate solution of keeping tuples and reactions in the objects associated with the `LimeTupleSpace`, and simply allow the `LimeTSMgr` to reference them. This latter solution, by eliminating the transfer of tuples and reactions during engagement and disengagement, is likely to provide better performance in the case of frequent mobility. In the short term, we will extend our run-time support to let the choice of the more appropriate strategy to the designer, who will evaluate it against application needs.

In contrast with `LimeTupleSpace` objects that are still private, when sharing is enabled blocking operations are allowed as well because multiple agents can write tuples to the host level tuple space. In the case where a matching tuple is found, no special processing is necessary and the `LimeTSMgr` releases the agent with the appropriate result. However, if no matching tuple exists, a mechanism must be established to detect when the tuple shows up, and immediately to notify and release the waiting agent. The realization that this kind of processing is somehow reactive led us to a design solution that exploits the notion of reaction not only as part of the programming interface, but also as a core element of system design.

For each blocking operation that does not find immediately a matching tuple, a strong reaction with the specified template is created, together with a system-defined `ReactionListener`. This listener will be called as any other LIME reaction listener, that is, with a `ReactionEvent` parameter containing the matching tuple triggering the reaction. In

the case of a **rd**, the listener will simply return a copy of the tuple in the `ReactionEvent` object to the suspended agent; in the case of an **in**, the listener will also first remove the matching tuple from the host. Note that, in this latter case, the listener is guaranteed that the tuple is still in the tuple space, because the reactive program runs as a single atomic step.

Federated Tuple Space Creating the illusion of a transiently shared federation of tuple spaces is the ultimate goal of the abstractions provided by LIME. This is accomplished by building upon the choices and mechanisms discussed thus far. While, the ultimate target environment for LIME is an ad hoc network where mobile hosts may move unconstrained and mobile agents can roam among them, we recognize that such a task of monumental proportions is likely to fail if not backed up by an initial evaluation of the primitives chosen. For this reason, our first version of LIME is based on a more constrained scenario that allows us to quickly develop a first implementation and gather feedback from applications, as discussed in the next two sections. We assume that mobile units announce explicitly their intentions to join and leave the LIME community, which determines the ability to control programmatically in LIME the engagement and disengagement process. Also, the scenario we assume involves a single community of mobile hosts, all in communication range. This latter constraint will change as we integrate an ad hoc routing protocol within our testbed network. Thus far, disengagement of a host always leaves the rest of the community connected, and hosts are able to join the community only one at a time, i.e., we do not yet support the engagement of two distinct LIME communities.

The management of changes in the configuration of the hosts is one of the key additions needed to move from the host-level to the federated tuple space. The engagement and disengagement protocols are implemented as community-wide transactions in order to maintain a consistent view among all hosts. To coordinate change requests in the configuration of the community and to ensure a total ordering of transactions among all hosts, the current version of our engagement protocol determines the presence of a leader in the community, with an election mechanism in place to deal with leader departure. The details of the protocols can be seen in Figure 10.3.

The first step of the protocol involves the engaging host, and presumes the availability of multicast support, which is exploited by the host to send a first message requesting the engagement. Upon receipt of the message, all hosts in the community prepare locally for engagement and inform the leader that they are ready. When the leader knows that all hosts are ready, the distributed transaction begins and the hosts in the community begin to exchange any misplaced tuples and new weak reactions with the new host.

One critical aspect of engagement is the update of the LimeSystem tuple space. This is accomplished in two steps. When a host first joins the LIME community, it sends a

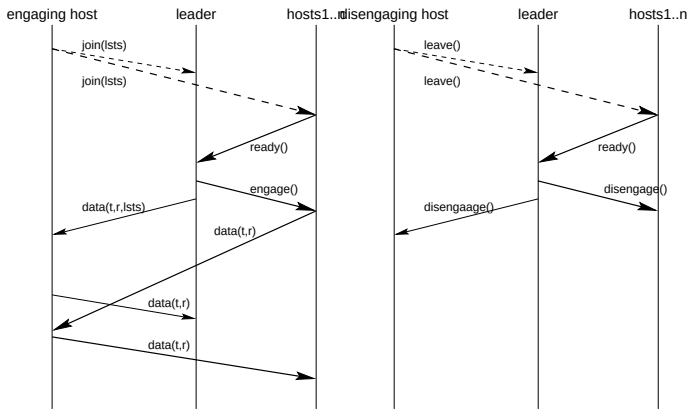


Figure 10.3: The engagement and disengagement protocols. Dashed lines indicate multicast messages, heavy solid lines represent multiple unicast, and regular lines are unicast messages. The data message from the leader may contain tuples (t), weak reactions (r), and all the tuples in the LimeSystem tuple space (lsts).

copy of the content of its own LimeSystem in the multicast message sent to all the hosts. This information is bound to contain only agents and tuple spaces present on that host, since the engaging host is not part of any other community. The information distributed to the members of the community serves to update their own LimeSystem in a way that is consistent with the configuration the system will assume after engagement is completed. In addition, when the leader sends its own tuple space information it will also send a copy of its LimeSystem tuple space to the new host. This way, the engaging host will be able to obtain a consistent view of the new configuration being built. Incidentally, this will also allow the engaging host to determine when it has exchanged data with all the members of the community, and thus it can resume regular processing. The disengagement protocol is similar albeit notably shorter than engagement, as there is no need to exchange data. With minor modification, these protocols are also used for the engagement and disengagement of agents and tuple spaces, both when the host is and is not part of a LIME community, although for simplicity, this was not discussed earlier.

Besides changes in the configuration, the very task of enforcing the semantics of the operations we described in the previous section for the whole federated tuple space is complicated by distribution. In particular, much of the complexity actually lies in the mechanisms supporting weak reactions. These are based on the same idea exploited to handle blocking operations on the host-level tuple space. When a weak reaction is registered, the `ReactionListener` object specified by the programmer is inserted in a separate `weakReactionMgr` object, while a system-defined strong reaction is registered with the reactor associated with the `LimeTSMgr` of the hosts involved in the weak reaction. These

strong reactions guard the host-level tuple space (or a single agent tuple space, depending on the value of its current location parameter). If the strong reaction is fired locally to the subscribing agent, the listener simply looks into the local `weakReactionMgr` and the user `ReactionListener` is executed. Alternately, if the reactor is remote, the listener sends a message to the subscriber's host with a `ReactionEvent`. When this message arrives, the user's `ReactionListener` is executed. In the case of a ONCE reaction, we must be careful to only execute the `ReactionListener` one time even though multiple matching tuples may be returned from different hosts in the system.

Federation also has an impact on remote processing of basic tuple space operations. Just as we were able to exploit the local reactor for the remote operations at the host level, here we utilize the weak reaction structure. A remote blocking **rd** is identical to a ONCE, weak reaction with a system defined `ReactionListener` which releases the blocked agent. A remote blocking **in** is slightly more complex as we may get responses back from multiple hosts, must return to that host to actually retrieve the tuple (using an **inp**) before releasing the agent.

Although the same `Reactor` serves both ONCE and ONCEPERTUPLE reactions, the processing of matching tuples are tailored based on this mode. After a ONCE reaction fires, the reaction is removed from the reactive program because the user's request has been satisfied. Alternately, a ONCEPERTUPLE reaction remains registered and must ensure that no single tuple causes the reaction to fire more than once. This is accomplished by keeping a list of the tuple identifiers which have already been reacted to within the `RegisteredReaction` itself. Each time a matching tuple is found, this data structure is queried and updated to determine if the `ReactionListener` should be executed. The implementation of the `Reactor` which separates newly written tuples from those which were in the tuple space prior to this round of the reactive program greatly improves the performance of ONCEPERTUPLE by not selecting a tuple more than once from a single local tuple space. However, because tuples can migrate and weak reactions can be uninstalled and reinstalled as connectivity changes, it is possible for a tuple to be selected more than one for a reaction, making the list of tuple identifiers necessary. By passing a relevant subset of the list of tuples already reacted to when an upon is reinstalled, additional duplication can be eliminated.

The uninstalling and reinstalling of weak reactions during disengagement and engagement provides another opportunity for optimization. Consider the situation where two hosts, A and B, connect while an agent on A has a ubiquitous reaction registered. All of the matching tuples in B's host level tuple space will be sent to A's agent and the reaction fired. If A and B disconnect and reconnect at a later time, all of the tuples previously reacted to, need not be retransmitted. However, because the reaction is uninstalled during disconnection, on reconnection B has no recollection of the tuples which it has already sent. Therefore, as part of the reinstallation process, the list of tuples at B which have already

been reacted to are sent. Although this increases the amount of information exchanged during engagement, this data can be greatly compressed and unnecessary traffic may be eliminated.

Details about the Current Implementation LIME is currently implemented completely in Java, with support for version 1.1 and higher. Communication is completely handled at the socket level—no support for RMI or other additional communication mechanisms is needed or exploited in LIME. The `lime` package is about 5,000 non-commented source statements, for about 100 Kbyte of `jar` file. The `lighTS` package providing a lightweight implementation of a tuple space and the adapter layer integrating multiple tuple space engine adds an additional 20 Kbyte of `jar` file. Thus far, it has been tested successfully on PCs running various versions of Windows and using Lucent WaveLAN wireless technology.

10.3 Developing Mobile Applications with LIME

Application development is the last phase of our research strategy, and the one where the abstractions inspired by formal modeling and embodied in the middleware are evaluated against the real needs of practitioners.

In this section we present two applications that exploit the current implementation of LIME in a setting where physical mobility of hosts is enabled. The two applications are typical of the physical mobility domain. The first one involves the ability to perform collaborative tasks in the presence of disconnection, while the second one revolves around the ability to detect changes in the system configuration. In each case, we present the corresponding application scenarios and a report about the way LIME has been exploited during development. The lessons learned from these experiences and the results of our empirical evaluation of LIME are presented in the next section.

10.3.1 ROAMINGJIGSAW: Accessing Shared Data

Scenario Our first application, ROAMINGJIGSAW as shown in Figure 10.4, is a multiplayer jigsaw assembly game. A group of players cooperate on the solution of the jigsaw puzzle in a disconnected fashion. They construct assemblies independently, share intermediate results, and acquire pieces from each other when connected. Play begins with one player loading the puzzle pieces to a shared tuple space. Any connected player sees the puzzle pieces of the other connected players and can select pieces they wish to work with. When a piece is selected, all connected players observe this as a change in the colored border of the piece, and within the system, the piece itself is moved to be co-located with the selecting player. When a player disconnects, the workspace does not change, but the pieces that have been selected by the departing player can no longer be selected and manipulated.

Figure 10.4: RoamingJigsaw. The left image shows the view of a disconnected player which is able to assemble only pieces it selected. The right image shows the view after the player re-engages with the other players, seeing assembly that occurred during disconnection.

From the perspective of the disconnected player, pieces whose border is tagged with the player's color can be assembled into clusters. Additionally, the player can connect to other players to further redistribute the pieces, and to view the progress made by the other players with respect to any clusters formed since last connected.

Play begins with one player loading the puzzle pieces to a shared tuple space. All users connected see the puzzle pieces in the shared workspace displayed in the user interface. Players can select pieces, resulting in a different color of the piece outline, and bring them to their own workspace. When a piece is selected, moved, or otherwise manipulated, all connected players see the change. When a player disconnects the workspace does not change, but the pieces that have been selected by the player cannot be manipulated by other players. While disconnected, the player may build assemblies by manipulating only the pieces displayed with the player's color. These changes will become available to the other group of players only when they become in contact again. Of course, connectivity may be restored only with some of the players initially belonging to the group, who can share they intermediate results and manipulate each other's pieces.

This application is based on a pattern of interaction where the shared workspace provides an accurate image of the global state of connected players but only weakly consistent with the global state of the system as a whole. The user workspace contains the last known information about each puzzle piece. It is interesting to observe that the globally set goal of the distributed application, i.e., the solution of the puzzle, is built incrementally through successive updates to the local state, distributed to all other players either immediately if connected or in a "lazy" fashion if connectivity is not available at that time.

RoamingJigsaw is a simple game that exhibits the characteristics of a general class of applications in which data sharing is the key element. The RoamingJigsaw design strategy may be adapted easily to any applications found in which the data being shared may change, e.g., sections of a document in a collaborative editing application, paper submissions to be evaluated by a program committee, etc.

Design and Implementation The basic data element of RoamingJigsaw is the individual puzzle piece. For efficiency purposes, each piece is stored as a pair of tuples. The first contains the image of the piece which remains unchanged throughout the game. The second contains a descriptor that includes information about a piece or the cluster that includes it and the current owner of the cluster. When a player selects a piece or joins together several pieces, a new tuple with the updated information is inserted, and the old descriptor is removed.

The critical operations in the game are the detection of piece selection and clustering actions, the reconciliation on reconnection, and the engagement of a new player. All are handled by exploiting a single mechanism in Lime: a weak reaction with mode oncepertuple and type `UbiquitousReaction`, its scope is the whole federated tuple space. The reaction is registered for tuples that match any cluster descriptor. The corresponding reaction listener updates the user workspace with the information in the matched descriptor and correctly maintains the weakly consistent view of the workspace. The amount of data transmitted for each update is minimized, because the reaction looks for descriptors and not for the individual piece images. However, in the case where a puzzle descriptor is received for a piece which the player never encountered before, as is the case during the engagement of a new player, the puzzle image is explicitly requested directly from the tuple space before the workspace is updated. Since the reaction is registered on the federated tuple space, the program receives updates about new descriptors without any need to be explicitly aware of the arrival and departure of players. Thus, the programming effort can focus just on handling data changes without worrying about the actual system configuration.

Although all processing described so far has operated on the federated tuple space, fine-grained control over the location of tuples is critical in dealing with disconnections caused by mobility. When a player selects a piece to work with, the piece must remain part of the transiently shared tuple space, but its location is changed to that of the selecting player in order to enable it to disconnect without losing access to the descriptor. In addition, since we deal with a weakly consistent workspace, a player must be prevented from selecting a piece that is currently not present in the federated tuple space. For these reasons, our implementation of RoamingJigsaw responds to an attempt to select a piece by first performing an **inp** operation on the tuple space of the player last known to have the piece. If the piece is returned, it is properly rewritten to the local tuple space of the new owner, and the selection is successful. If no tuple is returned, it means that the piece is unavailable for selection because the corresponding player is currently disconnected. This fact is communicated to the user by an audible beep.

We are presently developing a version of RoamingJigsaw that presents the user with a workspace that represents the current state of the system in a fully consistent way, i.e., only pieces belonging to users that are currently connected are seen through the workspace.

Figure 10.5: RedRover. The main console of RedRover, and the most recent camera image of a connected player.

All other pieces are removed from the workspace upon disconnection of the player that owns them and are redisplayed as soon as the player becomes connected again. In other words, the player sees exactly the pieces which are currently in the federated tuple space. This version may be easily coded by utilizing the `LimeSystemTupleSpace` to react to player arrivals and departures and by monitoring which puzzle pieces have been handed off to other players.

10.3.2 RedRover: Detecting Changes in Context

Scenario Our second target application is a spatial game we refer to as RedRover in which individuals equipped with small mobile devices form teams and interact in a physical environment augmented with virtual elements. This forces the participants to rely to a great extent on information provided by the mobile units and not solely on what is visible to the naked eye.

RedRover is the initial step in the development of a suite of virtually augmented games to be carried out in the real physical world. *BodyWare* will provide each player with global positioning system access, audio and video communication, range finding capabilities, and much more. For now, the game is limited to seeking and discovering the physical flag of the other team and clustering around the player who finds the flag. Each player is equipped with a digital camera which can be used to share a snapshot of the current environment with team members who may be separated physically by walls or other barriers, but remain within radio communication range. Finally, players know and share their precise location in space so that all connected players can maintain an image of the playing field displaying the relative location of all participants.

The user has several options available from the main display. The most dominant display element is a view of the playing filed indicating the current position of all players within range. Each player is equipped with a camera to share images of the surroundings with team members. Also, players are expected to identify key elements of the physical environment and share them with others. For example, a user may find the flag of another team or some relevant clue about the environment. Another player can register interest in a particular kind of information, and be notified when it has been found. For example, when the flag has been captured, any connected, interested team members can be notified.

As with RoamingJigsaw, RedRover is a simple game but it has great potential to be extended to real world scenarios such as the exploration of an unknown area by a group of people or robots. Our current efforts include the incorporation of a mapping mechanism which will allow users to define the elements of a region and share these results as they meet other users. Finally, the current implementation employs an artificial notion of player location, however, this can be trivially replaced with a global positioning system.

Design and Implementation in Lime The dominant feature of the user display is the current location of each connected player within the playing field. This is maintained in a strongly consistent manner, i.e., by displaying precisely the players which are connected and their most recent location update. Each time a player moves, a tuple representing its location is written to the federated tuple space. All players register a weak oncepertuple ubiquitous reaction for these tuples and the screen is updated with each reaction.

To detect when a player disconnects, we make use of the LimeSystem tuple space and register a reaction for the departure of a player (represented by a host). The listener of this reaction changes the connected status of the player and the user display is updated to replace the standard image of the player with a "ghost image" indicating that the player was once present, but is no longer connected. Upon reconnection, once the player moves, the oncepertuple reaction gets the new location for the player. However, if the player stays put, the reaction on the location tuple will not fire. Nevertheless we can still update the player's status to connected by registering a reaction on the LimeSystem tuple space for the arrival of a player (host).

To handle the notification of the flag capture, each player can register a once or oncepertuple upon on the federated tuple space. When a player finds the flag it writes a tuple to the tuple space indicating this fact, and all registered players receive notification in the form of a dialog box indicating which player has the flag. To facilitate clustering around this player, it is useful to request the camera image of the player in order to identify obstacles not visible on the screen which must be maneuvered around. Because the image is requested from a specific player, we simply use the **rdp** operation rather than incurring of the overhead of the reaction.

Another feature of the implementation is the separation of data to be shared with teammates versus information available to all game players. For example, it is desirable to inform only team members of the flag capture. Therefore, this information is written to a team-only tuple space, while general information. such as player location is written to a game tuple space. The ability to have selective sharing of tuple spaces is an important feature and the first step towards introducing security considerations in LIME.

10.4 Discussion

In this section we discuss our research contributions with an emphasis on lessons learned from exploiting LIME in the mobile applications presented in the previous section. We also compare LIME to similar projects found in literature and report about future work and enhancements we plan for our middleware.

10.4.1 Reflections and Lessons Learned

The development of LIME is the result of a continuous interplay among the definition of the underlying formal model, the design and implementation of the middleware, and its evaluation on mobile applications. The development of a model for LIME, and its formalization, favored a better understanding of the abstractions provided by the middleware. In particular, by keeping the programming interface as close as possible to the operations defined in the formal model, we made it easy to communicate and reason about the functionality of the system and its use in applications. In an incidental way, this task also provided an evaluation of the applicability of Mobile UNITY to the specification of a middleware for mobility. The ability to think about abstractions in a setting unconstrained by implementation details favored a style of investigation characterized by a more radical perspective, where the decisions driving the modeling and the definition of the main abstractions where mostly determined by the need for expressiveness and completeness.

This view was greatly refined when we started the design and implementation of the middleware. An example of refinements that took place is provided by the notion of reaction. Reactions were motivated by an intuition of the importance of reacting to events in a mobile environment and were inspired by the notion of reactive statements in Mobile UNITY. Nevertheless, reactions as defined in Mobile UNITY imposed atomicity requirements that are in general too strong to be practical in a distributed setting. This consideration led to the notion of a weak reaction, which represents a seemingly reasonable compromise between the loose guarantees provided by common event mechanisms like those found in T Spaces and JavaSpaces and the full atomicity guarantees of strong reactions.

Other refinements were the result of unforeseen needs on the part of the application programmer. This was the case with the reaction mode. In the reactive model of

Mobile UNITY, reactions are permanently enabled and it is up to the designer to specify the conditions under which they become disabled. Nevertheless, programming practice with the LIME model showed early on that some sort of automatic disabling of reactions is needed. In particular, the ONCEPERTUPLE mode turned out to be an important mechanism in developing both applications discussed in this chapter.

The feedback coming from applications was not limited to the discovery of new primitives. The use of LIME made it possible for us to evaluate the usefulness of its programming abstractions and constructs. Experience with ROAMINGJIGSAW and REDROVER corroborated our hypothesis that the ability to register weak reactions on the whole transiently shared tuple space provides the programmer with highly effective constructs that simplify the programming task. The execution of a single operation is sufficient to guarantee future notification of every event occurring over the whole federated tuple space, independently of changes in the configuration. Interestingly, this power has a cost; the implementation of weak reactions is probably the most complicated portion of the current LIME software—this should be expected, since we are shifting a great deal of complexity away from the programmer and into the run-time support.

Another interesting byproduct of these empirical evaluations is an understanding of the programming and architectural styles fostered by LIME and recurring in mobile applications. A possible distinction can be made between applications whose main requirement is to enable sharing of data despite mobility and those where most of the computation is driven by reactions to changes in context, as is the case with the two applications we presented here. Interestingly, in one case the functionality of the application must be provided *despite* mobility, while in the second case, the functionality *exists because of* mobility.

In this and other application typologies, a recurring dilemma is between an application style that provides a weakly consistent view of the system in the presence of mobility, and one that provides a fully consistent view that takes into account departure and arrival of mobile units. Choosing one representation style or the other has non-trivial implications on the complexity of the overall design and development task, and on the primitives that must be used. If weak consistency is enough, the view can be built incrementally by exploiting the notification mechanism provided by weak reactions, usually in the ONCEPERTUPLE mode. If, instead, a fully consistent view is required, additional, application specific machinery must be added in addition to using the LimeSystem tuple space to react (immediately) to changes in the system configuration. In our experience both styles are naturally accommodated by the abstraction of a transiently shared tuple space. Our "developers", mostly graduate and undergraduate students, found it easy not only to *program* applications with LIME but, most importantly, to *think* about the application in terms of the metaphors characteristic of the underlying LIME model.

Actually, the particular programming style induced by LIME, albeit biased by the limited range of applications considered thus far, is quite different from what we initially expected. This is especially true in the case of weak reactions and the LimeSystem tuple space. Reactive programming was not part of the initial core of LIME was envisioned to be a coordination framework founded on the idea of transiently shared tuple spaces accessible exclusively through Linda operations. Similar circumstances surrounded the LimeSystem. It was initially thought of as an add-on to support very specific needs. Instead, these abstractions turned out to play a key role in the design of both ROAMINGJIGSAW and REDROVER. We already reported about the use of weak reactions and ONCEPERTUPLE and we noted that the LimeSystem tuple space provides full context awareness by exposing changes in the configuration of the system. Although we initially thought this explicit knowledge could be bypassed by the observation of changes in the data context, experience with our applications (especially with REDROVER), showed that this hypothesis does not hold in general. The developer must resort to the LimeSystem tuple space. This causes no difficulties since the LimeSystem tuple space is perceived by the user as just another transiently shared tuple space with a different name and restricted access.

Finally, an issue that deserves careful evaluation is the extent to which the programmer is induced to duplicate the data present in the tuple space into some other run-time data structure, for performance reasons. For instance, in ROAMINGJIGSAW the re-paint of the workspace would involve retrieving from the federated tuple space all the pieces present at that moment. Clearly that is impractical, and the content of the tuple space is mirrored in a data structure that is kept consistent as changes are notified through reactions. In ROAMINGJIGSAW, such a mirroring is necessary in order to preserve weak consistency of the workspace, and to keep track of pieces that are no longer available. Nevertheless, this issue has more profound implications that have to do with the way the tuple space is actually used (i.e., as a coordination means or as a data repository). A comparative evaluation of the LIME programming style relative to the programming style induced by other middleware based on tuple spaces, like T Spaces [33] or JavaSpaces [35] remains to be carried out in the future.

10.4.2 Related Projects

LIME is not alone in its exploitation of the decoupled nature of tuple spaces for the coordination of mobile components. The Limbo platform [13] builds the notion of a quality of service aware tuple space which resides on mobile hosts. The quality of service information itself is stored in the tuple spaces and can be made accessible to agents on remote hosts. There is no notion of sharing data among the tuple spaces, however a *bridge agent* can be built which has references to multiple tuple spaces and can monitor and copy information among the spaces. Agents must also know explicitly which tuple space they wish to connect

to. A *universal tuple space* exists which registers all tuple spaces and can be used to locate a space. This notion is similar to the LimeSystem tuple space. Limbo does not provide any mechanisms beyond the regular Linda operations to react to changes in the tuple space.

In contrast, the main focus in the TuCSoN coordination model [62] is a reactive mechanism which is used to create *programmable tuple spaces* which respond to the queries of mobile agents. When an agent poses a query to the tuple space, the registered event which matches the operation and template fires, and an action is atomically performed. Another feature of TuCSoN is the ability to either fully qualify a tuple space name, identifying the specific host where the tuple space relies, or providing a partial name and gaining access to a local version of the tuple space. There is no coordination between tuple spaces, and mobile agents only have access to the tuple spaces fixed at the hosts.

It is interesting to note how the notion of reaction put forth in LIME is profoundly different from similar extensions that allow notification of events in the tuple space, such as those provided by TuCSoN, T Spaces [33], and Javaspaces [35]. In these systems, the events that are detected are the actual operations performed by the accessing processes, while in LIME, reactions fire based on the state of the tuple space itself. One common application task we discovered early is the need to look for a tuple, and, if it is not present, and then wait for its appearance. Without transactions, this is complicated by the possibility for the tuple to be written in between the initial query and the installation of the event listener. However, transactions are complex and expensive. In LIME, a single reaction accomplishes the desired task. Furthermore, the atomicity guarantees of the local reactions are relatively powerful. For example, with a localized reaction, the execution of the listener is guaranteed to fire in the same state in which the matching tuple was found. No such guarantee can be given with an event model where the events are asynchronously delivered.

10.5 Concluding Remarks

LIME is our first attempt at designing middleware for mobile systems based on the strategy of global virtual data structures. The ease of implementation of the two applications presented in this chapter demonstrates both the flexibility of the LIME abstraction to accommodate different application needs and the viability of the middleware approach to make the abstractions available to application programmers.

Chapter 11

Extending LIME

The current implementation of LIME serves as a proof of concept that the global virtual data structure style of abstraction is useful in aiding in the development of applications in the ad hoc mobile environment. However, the current assumptions of LIME are not reasonable in all mobile environments. In this chapter, we explore three specific enhancements to LIME, namely removing the assumption of announced disconnection (Section 11.1), weakening the engagement and disengagement transactions (Section 11.2), and limiting the scope of engagement to include more than the presence of connectivity (Section 11.3).

11.1 Unannounced Disconnection

In an ad hoc mobile environment, the ability of hosts to communicate is related to the distance between them. If a host suddenly moves out of range, any ongoing transmissions are terminated before they can be gracefully closed. We refer to this as unannounced disconnection because a host terminated communication without first announcing its intention to disconnect. Some work has been done in this area to try to predict when disconnection is likely, and intentionally disconnect before the actual disconnection is necessary [74]. While this has potential to reduce the number of unnannounced disconnections, it cannot eliminate them entirely. For example, a mobile host may suddenly lose power when a battery dies, or move into an anomalous dead done where interference prohibits communication.

Within LIME, unannounced disconnection has two main effects. First, if a component moves out of range without going through the disengagement protocol, the LimeSystem tuple space, which is intended to reflect current connectivity, is inconsistent with respect to reality. Second, if a communication was in progress between two LIME servers to transfer a piece of data from one host to another (e.g., during a tuple migration or in response to a remote **in** or **inp** query) the data being transferred may be corrupted or lost entirely, leaving the tuple space in an inconsistent state. This section presents several mechanisms for deciding what action to take in response to an unannounced disconnection. The options place varying

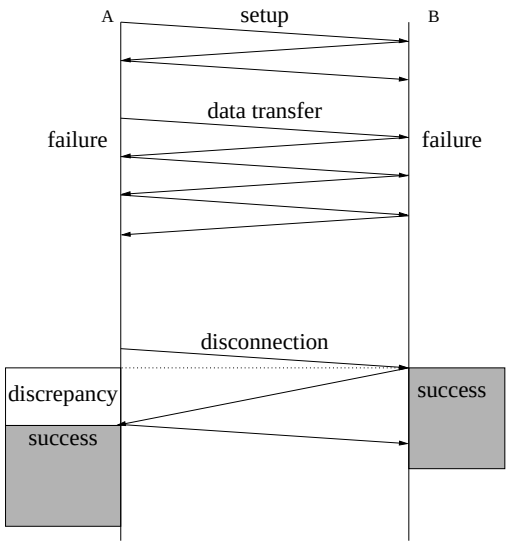


Figure 11.1: TCP data transmission.

demands on the system and thus can be selected and tuned by the programmer according to the demands of the application. We independently address the loss of communication during a data transfer and the loss of communication during an idle period in communication (e.g., when no remote tuple space operations are being executed by the LIME servers).

11.1.1 Disconnection during Data Transfer.

To provide solutions to unannounced disconnection during tuple transfers, we must first understand the underlying protocol and what error states are possible. For communication between two hosts, LIME employs TCP to ensure reliable transfer. In normal operation, as shown in Figure 11.1, a transfer goes through a setup phase to establish the connection, a data transfer phase, and finally a cleanup phase to terminate the connection. A temporary loss of communication will readily be handled by retransmission of lost packets, but unannounced disconnection arbitrarily terminates all communication from the instant disconnection occurs onward. In other words, if a host moves out of range in the middle of the data transfer, any messages sent by either side will be lost.

For simplicity, we treat the notification of the end of data with the beginning of the disconnection phase. At the moment when the receiver is notified that no more data will be sent, the receiver reports a successful data transmission, passing the data to the application and returning and acknowledgment to the sender. On the sender side, success is reported when this acknowledgment is received. If an unannounced disconnection happens at any time above these points, failed transmission is reported by TCP.

Sender Perceives	Receiver Perceives	Conservative	Liberal Loss	Liberal Duplication
success	success			
failure	success	none	none	duplication
failure	failure	protected	loss	none

Figure 11.2: Possible states for a transmission to terminate and the consequences of the unannounced disconnection policies.

In any data transfer, only the states shown in the first two columns of Figure 11.2 are possible. If the sender reports success, it is guaranteed that the receiver also reports success. However, if the receiver reports failure, it is provably impossible for the sender to know whether the receiver successfully received the data (and the disconnection acknowledgement was lost, abbreviated failure-success) or reported failure (because the disconnection message was lost, failure-failure). Success-failure is not possible because of the TCP model that we choose.

Whenever failure is reported, we must take a decision about what action to take and the possible consequences of those actions. Consider the failure-failure state. If no additional processing is taken at the sender, the data being transferred would be lost. One possible action is for the sender to *reverse* the action that was in progress (e.g., write a misplaced tuple locally or put the tuple which was in response to the remote **in** or **inp** back into the local tuple space). This guarantees that data will not be lost. However, consider the failure-success case. If we follow the previous course of action, the data will be duplicated at both the sender and receiver. Rather than make an arbitrary decision mandated to all applications built on LIME, we introduce several policies and allow the programmer to decide which applies best to their application.

Conservative policy. In some cases, the possibility of duplication is not acceptable, nor is the possibility of permanent loss of data. Therefore we define a *conservative* policy which under no circumstances duplicates data but may make it temporarily unavailable to regular tuple space operations. When the sender reports failure, a copy of the data being transferred is held in a special, *inconsistent* state. Data in this state is not accessible to regular queries over the tuple space, but may be accessed by new *administrative* operations. If a connection is reestablished between the sender and receiver, a reconciliation protocol must be executed to determine whether the receiver did or did not complete the transfer (reported success or failure). The receiver always remembers the identity of the last piece of data successfully received with respect to each sender. In the reconciliation protocol, if this value matches

the value held at the sender, the transfer was successful and the sender's copy is removed. If it does not match, the sender must retransmit the data.

This conservative policy guarantees that data is never duplicated, but in the failure-failure case, data is inaccessible between the failed transmission and the reconciliation. If a connection is never reestablished, the data will forever remain in the inconsistent state. Operations on the inconsistent data allow application administration over this information.

Liberal policy. One of the disadvantages of the conservative policy is the overhead, including storage of inconsistent data at the sender, storage of the identity of the last received data at the receiver, time to execute the reconciliation protocol, and the possible administration of the inconsistent data. In some cases, this strong policy is not required by the application and loss or duplication can be tolerated. For this reason, we introduce two more liberal policies in which the programmer can choose to accept either loss or duplication without the expense of the bookkeeping necessary for reconciliation.

If the programmer selects the LOSS mode, when a transfer fails, the receiver does nothing. If the user chooses DUPLICATION, the sender reverses the operation in progress, writing the data back to its local tuple space. At the receiver, no special processing occurs and there is no reconciliation if the connection is reestablished.

In the failure-success case, LOSS has no effect on the data consistency, while DUPLICATION creates multiple copies. Oppositely, in failure-failure, LOSS mode creates an inconsistency while DUPLICATION does not.

11.1.2 Duplication during Idle Communication

The previous discussion addressed inconsistency in the data as a result of loss of communication. This subsection deals with inconsistencies in the LimeSystem tuple space that arise when a disconnection occurs and is not immediately detected (i.e., there is no communication in progress between the LIME servers).

The best way to detect this type of disconnection is by introducing a passive mechanism, namely a periodic *hello* and a timeout. The *hello* is a short message containing the host's identify which is sent on a well-known broadcast address. Upon receipt of a *hello*, the local entry in the LimeSystem tuple space is updated with the time the message arrived. Periodically the LimeSystem tuple space is checked for the presence of hosts that it has not recently received a *hello* from, where *recently* is defined by the timeout period. When one is identified, we assume that an unannounced disconnection occurred and the LimeSystem tuple space is updated. We need not be concerned with the consistency of the tuple space.

It is possible that an inconsistent LimeSystem tuple space could be detected in another way. Consider an agent on host A that writes a tuple destined for an agent on host B. If B has moved out of range, but its timeout has not expired in A's LimeSystem tuple

space, when A attempts to establish a connection to send the misplaced tuple, it will fail. In this case, A detects the unannounced disconnection and original operation is executed as if B was not present. This is different from a failure during transmission because A knows definitively that this is equivalent failure-failure because B was never contacted.

11.2 Weakening Engagement and Disengagement

The current model of LIME assumes a system-wide transaction every time the system configuration changes. As was shown in the previous section, in the presence of unannounced disconnection, a host may leave the network without running the disengagement protocol. Alternately, the cost of running the transaction may be higher than an application is willing to tolerate simply to achieve the level of consistency that it provides. For these reasons, this section presents alternatives to the system-wide transactions which focuses only on the pairwise interactions between hosts.

Before launching into the details of the engagement protocol, it is important to remember what processing is done on disconnection. The most critical operation to consistency is the updating of the LimeSystem tuple space to remove the hosts that are no longer reachable. In addition, any weak reactions installed by these hosts are uninstalled. This processing occurs both during announced disconnection and when unannounced disconnection is detected.

The protocol we propose for engagement makes no consistency guarantees with respect to the LimeSystem tuple spaces of all connected components, but instead considers only pairwise interactions among hosts. We assume that the periodic broadcast messages described in Section 11.1 exist and can be used to trigger an engagement. Due to the possibility of unannounced disconnections, we must be careful to consider the various states that the hosts may be in and what operations can actually trigger the engagement.

The usual case is that neither host will know of the other. Therefore when B's *hello* arrives at A, A sends an *engagementRequest* to B. It is possible that B will simultaneously send an *engagementRequest* to A, otherwise upon receipt of A's *engagementRequest*, B will send its own *engagementRequest*, ensuring that the engagement always takes place both from A to B and from B to A. To prevent an infinite chain of *engagementRequest* messages, a source host only allows one outstanding *engagementRequest* per destination host. To process an *engagementRequest*, a host sends any misplaced tuples or remote weak reactions. The arrival of this data message updates both the LimeSystem tuple space and the data tuple space, concluding the engagement.

Although it is unlikely, it is possible that A thinks it is engaged with B, but B determined an unannounced disconnection took place. In this case the arrival of any regular messages from A's LIME server or A's hello will trigger an *engagementRequest* from B.

As previously mentioned the arrival of an *engagementRequest* always triggers a reciprocal *engagementRequest* (unless one is already underway), so A will send an *engagementRequest* to B. At first this seems unnecessary because A already thought that it was engaged with B. However, when B detected the unannounced disconnection, it removed all weak reactions from A, requiring engagement data to be sent from A to B, and any misplaced tuples destined for agents on A were written locally to B, requiring engagement data to be sent from B to A.

By assuming the timeouts can be used to detect disconnection, there is no need to specify a separate pairwise disengagement protocol. If desired, a simple announcement can be made to gracefully clean up the LimeSystem information.

With this pairwise model for engagement and disengagement, there is no longer any guaranteed consistency between the perception that each host has concerning the connectivity of the network. In the previous model, if A, B, and C were connected, they all shared the identical view of the tuple space, which was modeled in Mobile UNITY as a shared variable. With pairwise engagement and disengagement, it is possible for A's view to be $\{A, B\}$, B's view to be $\{A, B, C\}$ and C's view to be $\{B, C\}$. This changes the semantics of the operations because connectivity is no longer transitive. One way to model this is to keep the same model of a single transitively shared tuple space, but restrict the operations of each agent to a projection that matches the set of hosts it has completed the engagement process with. This includes both regular operations as well as the reaction for migrating misplaced tuples.

11.3 Limiting the Scope of Engagement

Under the current LIME model, the scope of a LIME network is limited only by the ability to communicate, which is related to the distance between hosts. However, this model does not scale as the density of hosts increases and the stress on the LIME system increases. In extreme conditions, the increased stress on LIME can lead to a point where the network becomes unreasonably inefficient, motivating a mechanism to restrict the scope of LIME operations.

This section proposes one solution to the scoping problem in LIME. The general idea is to use an additional, host specific parameter in addition to connectivity to determine optimal groupings in an ad hoc network. Various algorithms are presented that demonstrate how this parameter can be used to create groups of a manageable size.

11.3.1 Problem Statement

It is apparent that some method is desired to manage growth in a LIME-based network. On the surface, a variety of simple solutions exist to this problem, however a naive solution could

cause more harm than good. For example, consider an algorithm that creates arbitrary groupings that are restricted to a maximum size. While this solution prevents growth problems, it may fail to create groups that allow desired host coordination. Any reasonable method for managing network growth must fulfill the criteria outlined below.

Best Mutual Arrangement The solution should achieve a grouping that maximizes desired coordination between hosts. This means that if there are two hosts that desire to communicate with each other, there should be a high probability that they will be placed in the same group.

Minimize Complexity Ad hoc networks are constantly changing, which means that the groupings will need to be reevaluated frequently to ensure that the best mutual arrangement is achieved. Therefore, each reevaluation needs to occur quickly. In general, this means that the overall complexity of the solution should be minimized. The complexity has several components including the computation time on a single host, the overall computation time on all hosts and the network utilization. It is important to minimize the complexity on both a per host basis and a per network basis because an algorithm should not have to leverage a large network to complete its computations efficiently.

Adapt Dynamically Over time the hosts present in an ad hoc network will change, thus changing the connectivity of the network. The solution should accommodate for this change by ensuring that the current grouping is always optimal with respect to the hosts that are currently connected. Failure to do this could lead to a network with suboptimal groups that are based on an outdated optimality model.

An ideal solution will balance each of these properties to create a grouping that is equally and maximally beneficial to each host on the network. Specifically, the solution described below will sacrifice optimality in host groupings to minimize the algorithm's complexity.

11.3.2 Creating Groups Dynamically

As stated above, each grouping should create the best mutual arrangement of hosts on the network. This implies that the groupings will be deliberately chosen, which is beneficial to both LIME and the application-level program. In the case of LIME, properly chosen groups will allow maximal scalability along with a minimization of network traffic. At the application-level, groups provide a greater abstraction of the network, which increases LIME's utility to a programmer.

The term group has been used casually up to this point. However, for the rest of the section it is important to clarify the distinction between a *group* and a LIME *network*. A

group is a set of one or more hosts that transiently share their tuple spaces. On the other hand, a LIME network is a set of groups that are connected by the underlying network (e.g., a wireless network). On a standard LIME system, the group and the LIME network are equivalent. The algorithms described in the following sections break the LIME network into multiple groups.

The process of creating groups is best understood by breaking it down into several logical components. First, we define the concept of *happiness* between hosts, which allows us to distinguish the optimality of a given grouping of hosts. Next, we discuss the general protocol used by arriving hosts to request permission to join a group. Finally, we explain the decision-making algorithms. The first algorithm determines which hosts are allowed to join a group and the second determines when groups should be split in order to create smaller, more optimal groups.

Defining Host Preferences and Happiness

In order to create a meaningful grouping of hosts, each host must have some mechanism to describe the characteristics of groups it would prefer join. For example, an ad hoc network at a university could have a group of professors and a group of students. A host arriving at this ad hoc network would need to indicate which group it would prefer to join. This is accomplished with a vector of preferences for each host. The preference vector, p for a given host, i, is of the form

$$p_i = \{\langle l_0, v_0\rangle, \ldots, \langle l_k, v_k\rangle, \ldots, \langle l_n, v_n\rangle\}$$

where each l_k is a unique label for a component of happiness with a corresponding $v_k \in \Re$, which is the relative importance of that component to host i. Practically speaking, each label will be a unique string used to identify the meaning of its associated value.

Each preference vector can be viewed as a vector in $\Re^n$. Therefore, a convenient method to determine the similarity between two hosts' preferences is to determine the angle between their preference vectors. This can be done using the inner product, resulting in the angle, q, between p_i and p_j being defined as

$$\theta_{ij} = cos^{-1}\left(\frac{p_i \cdot p_j}{\|p_i\|\|p_j\|}\right) \tag{11.1}$$

Notice that as $\theta \to 0$ the vectors p_i and p_j are increasingly similar. Therefore, θ determines the happiness between two hosts such that happiness is inversely proportional to θ. The happiness between two hosts is a measure of how strongly the hosts desire to be grouped together.

Using this concept of happiness, the overall happiness of a host, i, within its group, G, is defined as

$$h_i = \frac{i}{|G| - 1} \sum_{(\forall j \neq i) \in G} \left| \frac{1}{\theta_{ij}} \right| \tag{11.2}$$

We have already discussed the reason for using the inverse of θ in this equation. The other portion of the equation divides the summation by $|G| - 1$, which is done to allow the happiness of hosts from different size groups to be easily compared. This definition of happiness is integral to a grouping algorithm because it provides a simple metric to determine the relative optimality of different groupings. Based on the definition, greater happiness indicates a more desirable grouping and $0 \leq h_i < \infty$.

Group Decision and Engagement Protocols

When a host first enters a LIME network, it will need to join a group in order to coordinate with the group through transiently shared tuple spaces. Typically, multiple LIME groups will already exist. Therefore, the entering host will desire to join a group that maximizes the overall happiness of the network. To achieve this goal, before joining a group (i.e., before initiating a LIME engagement) a host must decide which group to join. This is achieved through a group decision protocol which is described by the following basic steps:

1. The arriving host *applies* to several groups by passing each group leader its preference vector. The application requests permission to join the group.

2. Each group replies to the arriving host with a binding *decision* represented as a single bit indicating that the arriving host is or is not allowed to enter the group. If the decision is affirmative, then an *offer* estimating how the addition of the arriving host will increase the overall happiness of the network is also sent along with the decision.

3. After receiving each decision, the arriving host must *choose* which group to join. Typically, the host chooses the group offering maximal happiness.

4. The arriving host sends a message to each group that made it an offer to *accept* or *decline* the offer.

In the application stage (step 1), the arriving host sends a *group join request* to the multicast address. The request is a single message with the arriving host's preference vector. This request is received by the leader of the LIME network and the leader of each group. Upon receiving this request, the leader of the LIME network will reply to the arriving host to give the number of group leaders on the network. Using this number, the arriving host ensures that it receives replies from each of the group leaders.

The group decision stage (step 2) is a computational step that occurs at the leader of each group. Each leader will run an algorithm to determine if it would be advantageous

to allow the arriving host to enter the group. Practically speaking, the arriving host will be allowed to enter the group if it would increase the happiness of that group. If the results of the algorithm indicate that the arriving host may join the group, then the leader replies with an offer including the estimated amount of happiness that the arriving host would add to the group. After receiving replies back from each leader, the host should join the group that it would add the most happiness to. This ensures that the overall happiness of the network is maximized instead of attempting to just make the arriving host happy. If the results of the algorithm indicate that the arriving host may not join the group, then the leader replies by declining the join request. The details of the decision-making algorithm are explained next under the heading *Building a Single Group*.

The host choice stage (step 3) begins after the arriving host receives a reply from every group that was sent a join request. At this stage, the arriving host simply decides which group to join. Since the goal is to maximize happiness, the decision will be made such that the arriving host is happiest.

Finally, in the host decision stage (step 4) the arriving host sends its decision back to each group. From this point forward, the arriving host will only need to communicate with the group that it decided to join. The next step is for the arriving host to engage with its new group.

The engagement protocol only requires slight modifications from its current state in LIME. Specifically, every host in the group needs to know the preference vectors of every other host in the group. This ensures that each host can become a leader without requiring the old leader to pass the group's preference vectors when changing leaders. To achieve this functionality, the preference vectors are exchanged during the engagement protocol at the same time as the misplaced tuples and the remote weak reactions.

After engagement, the leader reevaluates its group by deciding if it would be more optimal by splitting into two or more groups. This decision is made by the algorithm described under the heading *Deciding When to Split a Group*.

Building a Single Group

Before building a group, we choose to establish a maximum group size, τ. This value is chosen such that each group is never larger than the maximum manageable size of a group in a LIME-based network. Until the current number of hosts in a group, η, equals τ the group will simply accept hosts into the group as group join requests are made because you can never decrease a group's happiness by adding another host. However, a decision must be made if $\eta = \tau$ and another host requests to join.

The decision making process attempts to choose group members that create the greatest happiness for the group, where the group happiness is defined as

$$H_G = \sum_{\forall i \in G} h_i \tag{11.3}$$

Achieving the greatest group happiness may require that a current host be removed to allow the arriving host to join the group. Unfortunately, the complexity for calculating the optimal H_G for the addition of one host is $O(\tau^2)$ when $\eta = \tau$. As τ grows the amount of computation required would be unreasonable, however, it is possible to achieve a close approximation to the quantity of happiness that will be added by a single host with less computation.

The approximation algorithm follows the steps below:

1. Calculate the happiness, $h_{\eta+1}$, for the arriving host saving each of the intermediate values from the summation.

2. Find the smallest intermediate value from step 1 and subtract its value from $h_{\eta+1}$.

3. Find the host with the smallest intermediate value from step 1. Compare the happiness of that host in the current group with the value for $h_{\eta+1}$. If $h_{\eta+1}$ is greater, then add the arriving host to the group and remove the other host. Otherwise, do not allow the arriving host to enter the group.

To illustrate the algorithm, consider the following example. The preference vectors given below are for four participants at a conference. Each participant has preferences regarding whom he or she talks to at the conference. For example, person A prefers to talk with people involved in security research, who have PhDs and who are affiliated with Washington University. The numbers indicate the relative importance of this person's preferences for each category. It is important to notice that magnitude of each number is only relevant within the same preference vector. This is because all of the vectors are normalized during the computation.

$\text{Person}_A = \{\langle\text{Security, 7}\rangle, \langle\text{PhD, 2}\rangle, \langle\text{Washington University, 1}\rangle\}$
$\text{Person}_B = \{\langle\text{Security, 20}\rangle, \langle\text{Mobility, 5}\rangle, \langle\text{Masters, 1}\rangle, \langle\text{MIT, 4}\rangle\}$
$\text{Person}_C = \{\langle\text{PhD, 1}\rangle\}$
$\text{Person}_D = \{\langle\text{Mobility, 9}\rangle, \langle\text{Security, 2}\rangle\}$

In this example we let $\tau = 3$, so the addition of the first three hosts proceeds without any computation. However, the addition of the fourth host uses the algorithm given above to determine which host, if any, should leave the group to make room for the new host. In

this case, the decision is to admit host D and remove host C. To understand this decision we need to look at the algorithm one step at a time. All of the calculations in this algorithm are performed by the leader of the LIME group.

The first step is to calculate the happiness for the new host with all of the hosts currently in the group. Using equation 11.1 we get the following θ values:

$$\theta_{ad} = 78.07^\circ$$
$$\theta_{bd} = 63.99^\circ$$
$$\theta_{cd} = 90.00^\circ$$

Next, we convert these values to the corresponding happiness values by taking the inverse of each number. (The notation below indicates the host's happiness values along with which hosts were considered in the calculation. For example, $h_{a,cd}$ would represent a's happiness when placed in a group with c and d.)

$$h_{a,d} = 1.28 \times 10^{-2}$$
$$h_{b,d} = 1.56 \times 10^{-2}$$
$$h_{c,d} = 1.11 \times 10^{-2}$$

To complete the first step we find the estimated happiness of host D by using equation 11.2. To simplify calculations we drop the 10^{-2} on each term from this point forward.

$$h_{d,abc} = (1.28 + 1.56 + 1.11)/3 = 1.32$$

For the second step we need to remove the smallest intermediate value from $h_{d,abc}$. Looking at the values for $h_{a,d}$, $h_{b,d}$ and $h_{c,d}$ we see that the smallest value is $h_{c,d} = 1.11$. Therefore, we subtract this value from $h_{d,abc}$.

$$h_{d,ab} = (1.28 + 1.56)/2 = 1.42.$$

To complete the third step we must have the happiness values of each host in the current group. Normally these values would already be known and stored with the group leader. The happiness values for the current group, $G = \{a, b, c\}$, are given below.

$$h_{a,bc} = 2.68$$
$$h_{b,ac} = 2.56$$
$$h_{c,ab} = 1.23$$

The host with the smallest happiness value from our calculations in the first step was host C. Therefore, we need to compare the happiness of host C in the current group with our newly calculated value for $h_{d,ab}$. The comparison shows that $h_{d,ab}$ is bigger, so we allow host D to enter the group and remove host C.

Looking at this example, we can see that the decision to remove host C was a good decision by looking at the group's happiness from before and after the decision. It is important to remember that the goal is to maximize the overall happiness of all hosts on the network. Therefore, even though host C will lose happiness from this decision, the overall happiness on the network has still increased and thus we view this as a good decision. The happiness values for before and after the decision are given in the table below.

	Grouping	H_G
Before adding D	A, B, C	6.46
After adding D	A, B, C	7.56

Examining all possible groups with these four hosts shows that the group $G = \{a, b, d\}$ is optimal. However, it is important to understand that this algorithm is not guaranteed to find the optimal grouping.

To evaluate the expected accuracy of this algorithm, a test program was written to create random preference vectors and use them as input to the approximation algorithm described above. Each preference vector contained five random values. In addition, τ ranged from three to five hosts and there were nine hosts on the network. Each approximation was compared to the exact optimal result and on average was within five percent.

The complexity of the approximation algorithm is $O(\tau)$ when $\eta = \tau$, which allows it to make fast decisions even when τ becomes large. However, this algorithm is only effective for building a single group. The next step is to determine when it is advantageous to split a group into two or more smaller groups.

Deciding when to Split a Group

When hosts join or leave a group, the group should reevaluate itself to verify that it is still in an optimal grouping. This reevaluation involves judging if the group is most optimal in its current configuration or if the group would be more optimal after splitting into two or more smaller groups. Unfortunately, the complexity of finding an optimal split for the grouping is very large because one would need to calculate the happiness for every possible grouping of τ hosts. This works out to

$$\sum_{r=1}^{\tau-1} \frac{\tau!}{r!(\tau - r)!}$$

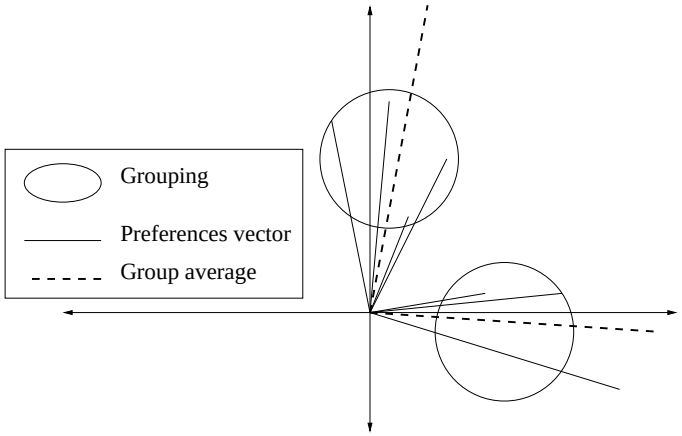


Figure 11.3: A visual interpretation of the grouping algorithm with seven hosts divided into two groups.

possible groups, which grows very rapidly with τ. Therefore, it is desirable to have an approximation algorithm to obtain near-optimal groupings with considerably less computation.

The approximation algorithm leverages the geometric interpretation of the preference vectors to find vectors that are close to one another (i.e., have a relatively small angle between them). Before examining the algorithm in detail, it is important to get a visual understanding of what the algorithm is doing. Figure 11.3 gives an example of seven preference vectors that each have only two values, which means that the vectors are in $\Re^2$. The algorithm uses these preference vectors as input and returns the group average and the groupings. The group average is a vector for each group that is located at the center of the group (shown as a dashed line in Figure11.3). The groupings are simply the algorithm's approximation of an optimal set of groups for the given preference vectors.

The approximation algorithm is surprisingly simple. It is a recursive algorithm with the following steps:

1. Choose two arbitrary vectors, q_1 and q_2, from the input preference vectors.

2. Create two groups, Q_1 and Q_2, such that Q_1 contains all of the preference vectors closer to q_1 than q_2 and Q_2 contains all of the other preference vectors.

3. Let q_1' be a vector located in the middle of Q_1 and let q_2' be a vector located in the middle of Q_2. Note that a vector is located in the middle of a group when it satisfies

$$q = \frac{1}{|Q|} \sum_{\forall p \in Q} \frac{p}{\|p\|}$$

for each component i in q. This means that q represents that average all the vectors in its group.

4. If the new groups, Q_1 and Q_2, contain different vectors than they did in the previous iteration, then let $q_1 = q_1'$ and $q_2 = q_2'$ and go to step 2. Otherwise, continue to step 5.

5. Return the average vectors (q_1 and q_2) and the groupings (Q_1 and Q_2).

If desired, this algorithm can be run successive times adding additional arbitrary vectors to the first step and making the appropriate changes throughout the rest of the algorithm to accommodate the extra vectors. Note that the algorithm always creates a number of groups equal to the number of arbitrary vectors in the first step.

After running the algorithm, one must determine if the suggested grouping returned from the algorithm is more optimal than the existing single group. This is quickly and easily determined using equation 11.3. However, to make the happiness values consistent among groupings of different sizes, H_G should be normalized by dividing by the number of hosts in G. This normalization allows a comparison of the happiness per host in each grouping. The grouping with the highest happiness per host is the desired grouping because it maximizes the overall happiness of the network.

As a demonstration of this algorithm, we use the preference vectors given under the title *Building a Single Group*, however, for this example we will assume that $\tau = 4$, which enables the current group to contain all four hosts. The question is: "Would this set of hosts be more optimal if they were split into multiple groups?"

First, notice that the only type of split that we should consider is to split the hosts into two groups. Anything beyond that would lead to a degenerate problem that is obviously not optimal (i.e., a single host always has a happiness of zero). Therefore, for this example we find an approximation for the best groupings when dividing this set of hosts into two groups. We then compare the happiness per host for the two groups with the happiness per host for the original single group. Whichever grouping offers the maximal happiness is the desired grouping.

The algorithm takes the four preference vectors as input and arbitrarily chooses to let $q_1 = \text{Person}_A$ and $q_2 = \text{Person}_B$. The next step is to find the hosts that are closer to q_1 than q_2. For this step, we list the relative angles in the table below. Note that it is not necessary to calculate the angles between q_1 and q_2 since they will not be grouped together for this iteration.

	$\theta_{q_1,host}$	$\theta_{q_2,host}$
Person_A	0.0	—
Person_B	—	0.0
Person_C	74.21	90.0
Person_D	78.07	63.99

Inspection of the data indicates that for this iteration we should let $Q_1 = \{q_1, \text{Person}_C\}$ and $Q_2 = \{q_2, \text{Person}_D\}$.

The next step is to find q_1' and q_2' such that they each represent the middle of their respective groups. Geometrically speaking, this means that q_1' and q_2' are the average of all vectors in Q_1 and Q_2, respectively. Therefore, we let

$q_1' = \{\langle\text{Security, 0.476}\rangle, \langle\text{PhD, 0.636}\rangle, \langle\text{Washington University, 0.068}\rangle\}$
$q_2' = \{\langle\text{Security, 0.584}\rangle, \langle\text{Mobility, 0.607}\rangle, \langle\text{Masters, 0.024}\rangle, \langle\text{MIT, 0.095}\rangle\}$.

We have now completed one iteration of the algorithm. Next we let $q_1 = q_1'$ and $q_2 = q_2'$ and repeat the above steps with these new values.

Given the new values for q_1 and q_2, we recalculate the angle table to get the values below.

	$\theta_{q_1,host}$	$\theta_{q_2,host}$
Person_A	37.103	48.998
Person_B	55.381	31.995
Person_C	37.103	90.000
Person_D	82.556	31.995

Looking at the data we see that Person_A and Person_C are closer to q_1 than q_2 so they will be placed in Q_1. Likewise, Person_B and Person_D are closer to q_2 than q_1 so they will be placed in Q_2. Now, comparing the vectors in each group for this iteration with the vectors in each group for the previous iteration, we see that the groups have not changed. Therefore, the algorithm has converged on a particular grouping and terminates.

The output from the algorithm is the final groupings, Q_1 and Q_2, along with the average vectors, q_1 and q_2. Using this information we calculate the average happiness per host below (as before, the 10^{-2} is dropped from all numbers in the calculation).

	Q_1 Happiness	Q_2 Happiness
Person_A	1.35	—
Person_B	—	1.56
Person_C	1.35	—
Person_D	—	1.56
H_G	2.70	3.12
Happiness per host	1.35	1.56
Average Happiness	1.455	

Next, we need to calculate the average happiness per host for the original grouping, which had all hosts in the same group.

	Happiness
Person_A	2.21
Person_B	2.22
Person_C	1.19
Person_D	1.32
H_G	6.94
Average Happiness	1.735

Now we simply compare the average happiness values for the two potential groupings. The comparison shows that it is more optimal to keep all of the hosts in a single group because $1.735 > 1.455$.

Notice that if the group in this example were larger, then we could run the approximation algorithm additional times to find the average happiness per host with more than two groups. After collecting all of the average happiness values for each grouping, the final decision would still be made by choosing the grouping with the highest average happiness per host.

11.3.3 Putting it all Together

Having seen the various approximation algorithms for building groups, we now shown an example of the entire process. We start with the arrival of the first host, $host_1$. Since there are no other hosts present, no computation is necessary and $host_1$ becomes the leader and only member of a group.

Next, the second host, $host_2$, arrives. At this point, $host_2$ determines that there is only one group on the network and requests to join it by sending a join request to $host_1$. Assuming that $\tau > 1$, $host_2$ is automatically allowed to join the group. Since the group changed with the addition of $host_2$, the group leader should run the approximation algorithm to determine if splitting the group is a good idea.

As additional hosts arrive, they will continue to be automatically accepted as described in paragraph above. After each host is accepted into the group, the group leader will once again run the approximation algorithm to determine if splitting the group is a good idea.

Once the group reaches its maximal size, $\eta = \tau$, a decision must be made with the arrival of each new host. This decision is determined by the results of the approximation algorithm described in *Building a Single Group*. If this decision is positive and the arriving host decides to join the group, then the group leader will remove another host from the group to make room for the arriving host. After the arriving host engages with the group,

the group leader will once again run the approximation algorithm to determine if splitting the group is a good idea.

This process will continue as new hosts arrive on the network. Furthermore, if multiple groups develop on the network, then each arriving host will submit join requests to each group and choose the group that maximizes the overall happiness of the LIME network.

11.3.4 Discussion

Using the approximation algorithms as described in the previous subsection achieves a close approximation to the optimal groupings on a network. In addition, the methods used to build groups provide a high degree of flexibility in how groups are created. For example, the elements in the preference vectors can be changed to suit many different applications. The example used throughout this section showed how preference vectors could be used to connect hosts at a conference. Some additional ideas are given in the table below.

Setting	Elements in Preference Vectors
University	students, professors, classes, departments, clubs
Business	projects, management,
Highway	destination, next planned stop, average traveling speed
Military	rank, mission, classification level

Building LIME Groups in a "Real World" Setting

A few important details should be addressed when implementing the protocol and algorithms discussed in Section 11.3.2. These details help to bridge the gap between the theoretical algorithms and the potential "real world" problems that may be encountered from using these algorithms.

First, we will look at step 1 from the group decision protocol. In this step, the arriving host is supposed to communicate with the leader of the LIME network to find out how many group leaders are on the LIME network. This solution is very good in that it guarantees that the arriving host will wait to receive a response from all of the groups. However, by having a leader for the LIME network there is a potential for scalability problems. To solve the scalability problem we need to eliminate the LIME network leader. This means that the arriving host will no longer be able to know the number of group leaders on the LIME network. Instead, the arriving host can use a timeout value to estimate whether it has received replies back from every group leader. For example, after sending the group join request the arriving host would wait for a specified amount of time before assuming that it has heard back from all group leaders. The assumption is that if the timeout value is chosen properly, then the arriving host will most likely receive a response

from every group leader before moving on. Naturally, there is the potential for the arriving host to inadvertently exclude a group, however for this price we have made the algorithm more scalable.

Next, we consider the final steps of the group decision protocol. On a busy network it is possible for a group leader to receive join requests from multiple hosts at the same time. Unfortunately, it is difficult for the group leader to give accurate estimates of how each host will affect the happiness of the group because the leader does not know which, if any, of the hosts will ultimately join the group. The only way to solve this problem would be to allow the group leaders to handle requests from only one arriving host at a time. The problem with this idea is that it would require a leader for the LIME network to coordinate the group join requests between the group leaders. For reasons discussed above, this solution suffers from scalability problems. Therefore, we suggest a different, yet conservative approach. This approach is for the group leader to calculate the estimated happiness of the arriving host only with the hosts that have already joined the group. This seems reasonable because as the number of groups on a LIME network increases, the chances that an arriving host will join any particular group decreases.

Finally, we consider what happens to hosts after they are removed from a group. When a host is removed from a group, the group leader should ignore all join requests received from the host until another host joins or leaves the group. This will ensure that the protocol is not stuck in a cycle where it continuously allows a host to join and then removes that host from the group immediately. However, a removed host should be free to send join requests to any other group leader in the LIME network. This allows a removed host to quickly find a new group to join.

11.4 Concluding Remarks

This chapter presented several extensions to LIME to increase its applicability to a variety of different mobility scenario. First we considered the effects of sudden loss of connection on the consistency of the data tuple space and on the connectivity information stored on each host in the LimeSystem tuple space. Next we proposed an alternative engagement protocol that does not require a system-wide transaction. The final section described an approach for limiting the scope of the engaging hosts to keep the operations on the shared tuple space more reasonable in systems with a large number of hosts.

Chapter 12

Conclusions

Mobility is emerging as an important area for computing research, posing many challenges which must be overcome in a society which is increasingly placing demands on computing technology. Research in mobility is being approached from a variety of angles which can broadly be categorized by applications, algorithms, models, and middleware.

This thesis describes contributions in each of these areas. We have developed several testbed applications which embody many important characteristics of the mobile application domain. We have designed several algorithms for reliable message delivery in the base station mobility environment by relying on algorithms from standard distributed computing. We introduced the notion of global virtual data structures as a novel abstract model for coordination among ad hoc mobile components. Finally, we have developed the LIME middleware as proof of concept of the global virtual data structure design strategy. Together, these research contributions accomplished our stated goal of exploring ways to enable rapid development of dependable applications in the mobile environment.

In the area of algorithm development, we have presented a new design strategy to apply established algorithms from traditional distributed computing to the mobile environment. Future work in this area will be two-fold. First new algorithms implementing different abstractions can be built using the same approach. Second, new strategies can be developed for algorithm development based on other techniques from distributed computing. For example, randomized algorithms may provide a convenient mechanism for describing probabilistic guarantees. Also, as non-determinism proved to be a useful model in defining parallelism, randomness may prove to be a useful tool for modeling arbitrary movement of mobile units. Self-stabilization algorithms in distributed computing provide guarantees conditional on the relative stability of the network. Such guarantees are directly translatable to the mobile environment when considering automatic reconfiguration after mobile unit movement. Finally, epidemic algorithms may provide insight into mechanisms to spread information from one mobile unit to another as connections change.

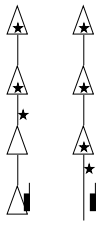
At the higher level of abstraction, other global virtual data structures can be developed following the same strategy as that used in the development of LIME. Our experiences with LIME have shown that the idea of middleware has great potential in mobility as it can be tailored to access low level details of the environment while still providing high level abstractions to the programmer. LIME showed how a single abstraction, namely transiently shared tuple spaces, can be implemented as a middleware. We believe that the notion of middleware can be broadened to define a layer of abstraction which itself is a layered composition of abstractions. This composition should be dynamic and configurable by the application programmer. For example, at the lowest level, middleware can be developed to utilize different protocols based on the characteristics of the communication media. For example, in radio communication, broadcast protocols have no more overhead than unicast and when possible should be exploited in protocols for this environment. Above this, algorithms can be developed to report, with varying degrees of guarantees, current connectivity information. In a logical mobility only setting or a physically mobile setting with restricted mobility where disconnections are always announced, tight guarantees can be made such that at all times, all nodes in the network are guaranteed to have the same view of the network. This of course comes at the cost of system-wide synchronization protocols, therefore such strong guarantees may not be desired and options for this layer of the middleware should include weaker notions of system configuration. At a yet higher level, different devices may have varying resource configurations. Middleware should accommodate this in two ways, first by allowing only the essential modules of the middleware to be installed on the device, and second by implementing the same high level abstractions with different functionality depending on the capabilities of the device on which it is executing. The driving theme is adaptability realized through modular middleware which can be pieced together for the specific needs of the application and the environment the application is running in.

Appendix A

Supporting Invariants of Chapter 6

This appendix proves several supporting invariants needed to prove the existence of the covered backbone (Inv I.1):

$$\begin{aligned}\text{delivery} \Rightarrow \langle \exists \alpha, \beta, f : \ & \text{backbone}(\beta) \wedge \text{coveredBone}(\alpha) \wedge f = \text{last}(\alpha) :: \\ & (\alpha \subset \beta \wedge \text{ANN} \in \text{Chan}(f, \text{Child}(f))) \\ & \vee (\alpha = \beta \wedge \text{mobile.preceeds.ann}(f, \text{Child}(f)))\rangle \quad \text{(I.1)}\end{aligned}$$



A.1 Integrity of the Backbone

The first supporting invariant addresses the **integrity of the backbone**, stating that if the backbone exists, it must have the following properties: (a.) the sequence of nodes in the backbone must be the same as the list of nodes carried by the mobile unit (program variable MList), (b.) if the mobile unit is on a channel, that channel must be the backbone extension (defined as the channel segment between the last node of the backbone and the mobile unit), (c.) if the mobile unit is at a node (program variable MobileAt), that node must be the last node of the backbone sequence, (d.) there are no delete messages (indicated with the constant DEL) in any backbone channels, and (e.) there are no delete messages on the backbone extension.

$$\begin{aligned}\text{backbone}(\beta) \wedge \text{last}(\beta) = f \Rightarrow & \text{MList} = \beta \\ & \wedge \textsc{mob} \in \text{Chan}(m, n) \Rightarrow m = f \wedge \text{Child}(m) = n \\ & \wedge \text{MobileAt}(m) \Rightarrow m = f \\ & \wedge \langle \forall m, n : m, n \in \beta \wedge n = \text{Child}(m) :: \textsc{del} \notin \text{Chan}(m, n) \rangle \\ & \wedge \neg \text{mobile.preceeds.delete}(f, \text{Child}(f)) \end{aligned} \tag{I.1.1}$$

Again, this invariant is proven over each of the program statements.

- MobileArrives$_A(B)$: If A is on the backbone, the MList is truncated after this node, none of the pointers change, and the backbone is maintained and matches the MList.

  If A is not on the backbone, A is appended to MList. The last node of the backbone is pointing toward A as its child by this invariant, therefore when A sets its parent pointer, the backbone is extended because the maximal length path now includes the link just traversed by the mobile unit.

  Since there were no delete messages on the backbone and the mobile unit was processed from the head of the channel, after this statement executes, there are still no delete messages on any channels. However, a delete is generated leaving A. Since the child of A is NULL, this channel cannot be part of the backbone because there is no path that includes it. Therefore the generated delete does not violate the invariant.

- DeleteArrives$_A(B)$: If the delete is accepted, A is not on the backbone because by this invariant, there are no delete messages that will affect backbone nodes. Therefore, changing the pointers of A does not affect the backbone or backbone extension. After the statement executes, A is still not part of the backbone, therefore the outgoing channel of A that now has a delete message is not part of the backbone.

  If the delete is not accepted, the statement is essentially a skip, which trivially maintains the invariant.

- MobileLeaves: The nodes of the backbone do not change, however the mobile unit is no longer on the node. Therefore the backbone extension is created and the last node of the backbone does point toward the mobile unit. Also, it is not possible for the mobile unit to precede a delete because the mobile unit is put onto the end of the channel, and by the FIFO assumption with the channels, the mobile unit follows everything that was in the channel.

- AnnouncementArrives$_A(B)$, AckArrives$_A(B)$, and AnnouncementStart do not affect any of the variables of this invariant.

A.2 Backbone always exists

To be able to assert the previous invariant at any point, we must show that the **backbone always exists**.

$$\langle \exists \beta :: \text{backbone}(\beta) \rangle \tag{I.1.2}$$

Because the root is a constant, the backbone can always be constructed. Note, this invariant does not say anything about the structure of the backbone, only that one can always be constructed. The structure invariant is found in I.1.1.

A.3 A most one announcement

At any time during program execution, there is **at most one announcement** in the system which is on a channel. There might be multiple copies of the announcement in the system, but these copies are at nodes.

$$\langle \sum m, n : \text{ANN} \in \text{Chan}(m, n) :: 1 \rangle \leq 1 \tag{I.1.3}$$

The intuition behind proof of this property is that during predelivery, there are no announcements in the system (on either nodes or channels) (Inv I.1.4). Once AnnouncementStart fires, either one announcement is put on a channel, or none (in the case where delivery occurs immediately). Since AnnouncementStart acts as a skip from this point forward, and no other statement can generate an announcement without consuming an announcement from a channel, there will never be more than one announcement on any channel.

A.4 No announcements during predelivery

To support the previous argument, we need the fact that there are **no announcements in the system during predelivery**.

$$\text{predelivery} \Rightarrow \langle \forall m, n :: \text{ANN} \notin \text{Chan}(m, n) \wedge \neg \text{AnnouncementAt}(m) \rangle \tag{I.1.4}$$

By the initial conditions, there are no announcements in the system. The only statement which can generate an announcement is AnnouncementStart and after this statement fires, the system is no longer in predelivery. The system cannot enter predelivery again because there is no statement to set predelivery to true.

A.5 No acknowledgements during delivery

We must also show that there are **no acknowledgments during delivery** (where the constant ACK indicates an acknowledgment).

$$\text{delivery} \Rightarrow \langle \forall m, n :: \neg \text{ACK} \in \text{Chan}(m, n) \rangle \tag{I.1.5}$$

It is trivial to show that there are no acknowledgments initially, and none are generated during predelivery. the first acknowledgment is generated when the mobile unit receives the announcement, which by definition takes the system out of into postdelivery, making this invariant trivially true.

References

[1] 3Com. Palm VII Connected Organizer web page. `http://www.palm.com/products/palmvii/index.html/`, 2000.

[2] 3G.IP. 3G.IP Web Page. `http://www.3gip.org/`, 2000.

[3] A. Acharya and B.R. Badrinath. A framework for delivering multicast messages in networks with mobile hosts. *Journal of Special Topics in Mobile Networks and Applications (MONET)*, 1(2):199–219, October 1996.

[4] A. Acharya, B.R. Badrinath, and T. Imielinski. Checkpointing distributed applications on mobile computers. In *Proceedings of the Third International Conference on Parallel and Distributed Information Systems*, October 1994.

[5] A. Acharya, A. Bakre, and B.R. Badrinath. IP multicast extensions for mobile internetworking. Technical Report LCSR-TR-243, Rutgers University, 1995.

[6] R.M. Amadio. An asynchronous model of locality, failure, and process mobility. In *Proceedings of COORDINATION 97. Second International Conference on Coordination Models and Languages, LNCS 1282*, pages 374–391, Berlin, Germany, September 1997. Springer-Verlag.

[7] R.J.R. Back and K. Sere. Stepwise Refinement of Parallel Algorithms. *Science of Computer Programming*, 13(2-3):133–180, May 1990.

[8] B.R. Badrinath, A. Acharya, and T. Imielinski. Structuring distributed algorithms for mobile hosts. In *Proceedings of the Fourteenth International Conference on Distributed Computing Systems*, pages 21–28, Poznan, Poland, 1994.

[9] B.R. Badrinath, A. Acharya, and T. Imielinski. Designing distributed algorithms for mobile computing networks. *Computer Communications*, 19(4):309–320, April 1996.

[10] M. Baldi and G.P. Picco. Evaluating the Tradeoffs of Mobile Code Design Paradigms in Network Management Applications. In *Proc. of the 20^{th} Int. Conf. on Software Engineering*, 1998.

[11] J. Baumann et al. Communication Concepts for Mobile Agent Systems. In K. Rothermel and R. Zeletin, editors, *Mobile Agents: 1st Int. Workshop MA'97*, LNCS 1219, pages 123–135. Springer, April 1997.

[12] J. Baumann and K. Rothermel. The shadow approach: An orphan detection protocol for mobile agents. In *Mobile Agents: Second International Workshop MA '98, LNCS 1477*, pages 2–13, Stuttgart, Germany, September 1998. Springer-Verlag.

[13] 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 (IWQoS '97)—Building QoS into Distributed Systems*, pages 37–48, May 1997.

[14] J. Bradshaw, editor. *Software Agents*. AAAI Press/MIT Press, 1996.

[15] J. Broch, D.B. Johnson, and D.A. Maltz. The Dynamic Source Routing Protocol for Mobile Ad Hoc Networks . Internet Draft, October 1999. IETF Mobile Ad Hoc Networking Working Group.

[16] G. Cabri, L. Leonardi, and F. Zambonelli. Reactive Tuple Spaces for Mobile Agent Coordination. In [77], pages 237–248.

[17] L. Cardelli and A. Gordon. Mobile Ambients. *Theoretical Computer Science*, 240(1), 2000. To appear.

[18] K.M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computing Systems*, 3(1):63–75, 1985.

[19] K.M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, NY, USA, 1988.

[20] D. Coore, R. Nagpal, and R. Weiss. Paradigms for structure in an amorphous computer. A.I. Memo No. 1614, Massachusetts Institute of Technology Artificial Intelligence Laboratory, October 1997.

[21] G. Cugola, E. Di Nitto, and A. Fuggetta. The JEDI event-based infrastructure and its application to the development of the OPSS WFMS. *IEEE Trans. on Software Engineering*. To appear.

[22] S.E. Deering and D.R. Cheriton. Multicast routing in datagram internetworks and extended lans. *ACM Transactions on Computer Systems*, 8(2):85–110, 1990.

[23] E.W. Dijkstra and C. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(1), 1980.

[24] M. Dunham, A. Helal, and S. Balakrsihnan. A Mobile Transaction Model that Captures both the Data and Movement Behavior. *ACM-Baltzer Journal on Mobile Networks and Applications (MONET)*, 2(2):149–162, October 1997.

[25] Ericsson, IBM, Intel, Nokia, and Toshiba. Bluetooth. http://www.bluetooth.com, 2000.

[26] H. Eriksson. Mbone: The multicast backbone. *Communications of the ACM*, 37(8):54–60, 1994.

[27] C. Fournet, G. Gonthier, J.-J. Lévy, and L. Maranget. A calculus of mobile agents. In *Proceedings of the International Conference on Concurrency Theory, LNCS 1119*, pages 406–421, Berlin, Germany, 1996. Springer-Verlag.

[28] A. Fuggetta, G.P. Picco, and G. Vigna. Understanding code mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, 1998.

[29] D. Garlan and D. Le Métayer, editors. *Proc. of the 2nd Int. Conf. on Coordination Models and Languages (COORDINATION '97)*, volume 1282 of *LNCS*. Springer, Sept. 1997.

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

[31] R. Gray, D. Kotz, S. Nog, D. Rus, and G. George. Mobile agents for mobile computing. Technical Report PCS0TR96-285, Dartmouth College, May 1996.

[32] R.S. Gray, G. Cybenko, D. Kotz, and D. Rus. Agent Tcl. In *Itinerant Agents: Explanations and Examples with CDROM*. Manning, 1996.

[33] IBM. T Spaces. `http://www.almaden.ibm.com/cs/TSpaces/`, 2000.

[34] V. Jacobson and S. McCanne. Visual audio tool.

[35] JavaSpaces. The JavaSpaces Specification web page. `http://www.sun.com/jini/specs/js-spec.html`, 2000.

[36] D.B. Johnson. Scalable support for transparent mobile host internetworking. In H. Korth and T. Imielinski, editors, *Mobile Computing*, pages 103–128. Kluwer Academic Publishers, 1996.

[37] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained Mobility in the Emerald System. *ACM Trans. on Computer Systems*, 6(2):109–133, February 1988.

[38] J.M. Kahn, R.H. Katz, and K.S.J. Pister. Mobile Networking for Smart Dust. In *Proc. of the 5th Annual ACM/IEEE Int. Conf. on Mobile Computing and Networking*, Seattle, WA, USA, August 1999. ACM.

[39] J. Kiniry and D. Zimmerman. A Hands-On Look at Java Mobile Agents. *IEEE Internet Computing*, 1(4), 1997.

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

[41] F.C. Knabe. *Language Support for Mobile Agents*. PhD thesis, Carnegie Mellon Univ., Pittsburgh, PA, USA, December 1995.

[42] D. Lange and M. Oshima. *Programming and Deploying Mobile Agents with Aglets*. Addison-Wesley, 1998.

[43] T.M. Malone and K. Crowston. The interdisciplinary study of coordination. *ACM Computing Surveys*, 26(1):87–119, Mar. 1994.

[44] manet. IETF Mobile Ad-hoc Networks Working Group. `http://www.ietf.org/html.charters/manet-charter.html`, 2000.

[45] C. Mascolo. MobiS: A Specification Language for Mobile Systems. In P. Ciancarini and A. Wolf, editors, *Proceedings of the 3rd Int. Conf. on Coordination Languages and Models (COORDINATION)*, volume 1594 of *LNCS*, pages 37–52. Springer, April 1999.

[46] C. Mascolo, G.P. Picco, and G.-C. Roman. A Fine-Grained Model for Code Mobility. In *Proc. of the 7th European Software Engineering Conf. held jointly with the 7th ACM SIGSOFT Symp. on the Foundations of Software Engineering (ESEC/FSE '99)*, LNCS, Toulouse (France), September 1999. Springer.

[47] J. Matocha. Distributed termination detection in a mobile wireless network. In *36th Annual ACM Southeast Conference*, Marietta, GA, April 1998.

[48] Lucent Technologies. Orinoco. `http://www.wavelan.com`, 2000.

[49] Sun Microsystems. Jini connection technology. `http://www.sun.com/jini`, 2000.

[50] P.J. McCann and G.-C. Roman. Compositional Programming Abstractions for Mobile Computing. *IEEE Trans. on Software Engineering*, 24(2), 1998.

[51] S. McCanne and V. Jacobson. vic: A flexible framework for packet video. In *ACM Multimedia '95*, pages 511–522, San Francisco, CA, USA, 1995.

[52] J. McLurkin. Using cooperative robots for explosive ordnance disposal. Massachusetts Institute of Technology Artificial Intelligence Laboratory.

[53] Sun Microsystems. *JavaSpace Specification*, March 1998. `http://java.sun.com/products/jini/specs`.

[54] R. Milner. *Communicating and Mobile Systems: The π-Calculus*. Cambridge University Press, 1999.

[55] D. Milojicic, M. Breugst, I. Busse, J. Campbell, S. Covaci, B. Friedman, K. Kosaka, D. Lange, K. Ono, M. Oshima, C. Tham, S. Virdhagriswaran, and J. White. MASIF—The OMG Mobile Agent System Interoperability Facility. *Journal of Personal Technologies*, 2(2), June 1998.

[56] N. Minar, M. Gray, O. Roup, R. Krikorian, and P. Maes. Hive: Distributed Agents for Networking Things. In *Proc. of the 1st Int. Symp. on Agent Systems and Applications and 3rd Int. Symp. on Mobile Agents (ASA/MA '99)*, pages 118–129, Palm Springs, CA, USA, October 1999. IEEE Computer Society.

[57] J. Moy. OSPF version 2. Internet draft, Internet Engineering Task Force, March 1994 1994.

[58] A.L. Murphy, G.-C. Roman, and G. Varghese. An exercise in formal reasoning about mobile computations. In *Proceedings Ninth International Workshop on Software Specification and Design*, pages 25–33, Ise-Shima, Japan, 1998. IEEE Computer Society Press.

[59] A. Myles and D. Skellern. Comparing four IP based mobile host protocols. *Computer Networks and ISDN Systems*, 26(3):349–355, 1993.

[60] R. De Nicola, G.L. Ferrari, and R. Pugliese. KLAIM: A kernel language for agents interaction and mobility. *IEEE Transactions on Software Engineering*, 24(5):315–330, May 1998.

[61] ObjectSpace Inc. *Voyager ORB 3.0—Developer Guide*, 1999. `www.objectspace.com`.

[62] A. Omicini and F. Zambonelli. Tuple Centres for the Coordination of Internet Agents. In *Proc. of the 1999 ACM Symp. on Applied Computing (SAC'00)*, February 1999.

[63] Oracle. Oracle 8*i* Lite web page. `http://www.oracle.com/`, 2000.

[64] V. Park and S. Corson. Temporally-Ordered Routing Algorithm (TORA) Version 1 Functional Specification. Internet Draft, October 1999. IETF Mobile Ad Hoc Networking Working Group.

[65] C.E. Perkins. IP mobility support. Technical Report RFC 2002, IETF Network Working Group, October 1996.

[66] C.E. Perkins and D.B. Johnson. Mobility support in IPv6. In *Proceedings of the Second Annual International Conference on Mobile Computing and Networking (MobiCom'96)*, Rye, NY, USA, November 1996. ACM Press.

[67] C.E. Perkins, E.M. Royer, and S.R. Das. Ad Hoc On Demand Distance Vector (AODV) Routing. Internet Draft, October 1999. IETF Mobile Ad Hoc Networking Working Group.

[68] G.P. Picco. μCODE: A lightweight and flexible mobile code toolkit. In *Mobile Agents: Second International Workshop MA '98, LNCS 1477*, pages 160–171, Stuttgart, Germany, September 1998. Springer-Verlag.

[69] G.P. Picco, A.L. Murphy, and G.-C. Roman. LIME: Linda meets mobility. In *accepted for publication in Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*, Los Angeles (USA), May 1999.

[70] R. Prakash, M. Raynal, and M. Singhal. An Adaptive Causal Ordering Algorithm Suited to Mobile Computing Environments. *Journal of Parallel and Distributed Computing*, pages 190–204, March 1997.

[71] R. Prakash and M. Singhal. A Dynamic Approach to Location Management in Mobile Computing Systems. In *Proc. of the 8th Int. Conf. on Software Engineering and Knowledge Engineering (SEKE'96)*, pages 488–495, June 1996.

[72] M. Ranganathan, A. Acharya, S. Sharma, and J. Saltz. Network-aware mobile programs. Technical Report CS-TR-3659, University of Maryland, College Park, 1997.

[73] Y. Rekhter and T. Li. A Border Gateway Protocol 4 (BGP-4). *RFC 1771*, March 1995.

[74] G.-C. Roman, Q. Huang, and A. Hazemi. On Maintaining Group Membership Data in Ad Hoc Networks. Wucs-00-09, Washington University in St. Louis, Department of Computer Science, April 2000.

[75] G.-C. Roman, P.J. McCann, and J.Y. Plun. Mobile UNITY: Reasoning and specification in mobile computing. *ACM Transactions on Software Engineering and Methodology*, 6(3):250–282, 1997.

[76] 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 ACM SIGSOFT Symp. on the Foundations of Software Engineering (ESEC/FSE97), number 1301 in LNCS, Zurich (Switzerland), September 1997. Springer.

[77] K. Rothermel and F. Hohl, editors. *Mobile Agents: 2nd Int. Workshop MA'98*, LNCS 1477. Springer, September 1998.

[78] B. Sanders, B. Massingill, and S. Kryukova. Derivation of an algorithm for location management for mobile communication devices. *Parallel Processing Letters*, 8(4):473–488, December 1998.

[79] D. Sangiorgi. *Expressing Mobility in Process Algebras: First Order and Higher Order Paradigms*. PhD thesis, Computer Science Dept., Univ. of Edinburgh, 1993.

[80] M. Satyanarayanan. Mobile information access. *IEEE Personal Communications*, 3(1), February 1996.

[81] M. Shaw and D. Garlan. *Software Architecture: Perspective on an Emerging Discipline*. Prentice Hall, 1996.

[82] J.W. Stamos and D.K. Gifford. Remote Evaluation. *ACM Trans. on Programming Languages and Systems*, 12(4):537–565, October 1990.

[83] M. Steenstrup. *Routing in Communication Networks*, chapter 10. Prentice-Hall, 1995.

[84] D. Terry, M. Theimer, K. Petersen, A. Demers, M. Spreitzer, and C. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. *Operating Systems Review*, 29(5):172–183, 1995.

[85] R. Want, A. Hopper, V. Falcao, and J. Gibbons. The Active Badge Location System. *ACM Trans. on Information Systems*, 10(1):91–102, January 1992.

[86] M. Weiser. The computer for the 21st century. *Scientific American*, 265(3):94–104, 1991.

[87] J.E. White. Telescript Technology: Mobile Agents. In [14].

[88] A. Xu and B. Liskov. A design for a fault-tolerant, distributed implementation of Linda. In *Digest of Papers of the 19th Int. Symp. on Fault-Tolerant Computing*, pages 199–206, June 1989.

[89] H. Zimmerman. OSI reference model – The ISO model of architecture for open systems interconnection. *IEEE Transactions on Communication*, 28:425–432, 1980.

Vita

Amy L. Murphy

PERSONAL DATA

Date of Birth:	August 6, 1973
Place of Birth:	Rochester, Minnesota, USA
Citizenship:	USA

EDUCATION

Degree granted:	D.Sc. in Computer Science.
August 2000	Washington University, St. Louis, MO
	Enabling the Rapid Development of Dependable Applications in the Mobile Environment.
	Advisor: Dr. Gruia-Catalin Roman
Degree granted:	M.S. in Computer Science.
May 1997	Washington University, St. Louis, MO.
Degree granted:	B.S. in Computer Science.
May 1995	University of Tulsa, OK. magna cum laude.
	A Formally Specified Negotiation Strategy for the Feature Interaction Problem.
	Advisor: Dr. R.F. Gamble

PROFESSIONAL SOCIETIES

Association for Computing Machinery
IEEE Computer Society

PUBLICATIONS

G.P. Picco, A.L. Murphy, and G.-C. Roman. Developing Mobile Computing Applications with LIME. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE'00)*, Limerick, Ireland, June 2000. (Demonstration).

A.L. Murphy and G.P. Picco. Reliable communication for highly mobile agents. In *Proceedings of the 1st International Symposium on Agent Systems and Applications and 3rd International Symposium on Mobile Agents (ASA/MA '99)*, pages 141–150, Palm Springs, CA, USA, October 1999.

G.P. Picco, A.L. Murphy, and G.-C. Roman. LIME: Linda meets mobility. In *Proceedings of the 21st International Conference on*

Software Engineering (ICSE'99), pages 368–377, Los Angeles, CA, USA, May 1999.

A.L. Murphy. Algorithm development in the mobile environment. In *Proceedings of the 21st International Conference on Software Engineering (ICSE'99), Doctoral Workshop*, pages 728–729, Los Angeles, CA, USA, May 1999. (Abstract).

A.L. Murphy, G.-C. Roman, and G. Varghese. An exercise in formal reasoning about mobile computations. In *Proceedings of the 9^{th} International Workshop on Software Specification and Design*, pages 25–33, Ise-Shima, Japan, 1998.

A.L. Murphy, G.-C. Roman, and G. Varghese. An algorithm for message delivery to mobile units. In *Proceedings of the 16^{th} Annual ACM Symposium on Principles of Distributed Computing (PODC '97)*, Santa Barbara, CA, USA, 1997. (Abstract).

D. Baughman, R.F. Gamble, and A.L. Murphy. Using formal specification to design verifiable hybrid knowledge based dystems. In *Proceedings of the 7^{th} AAAI Workshop of Verification and Validation of Knowledge Based Systems*, Seattle, WA, USA, August 1994.

INVITED PUBLICATIONS

G.-C. Roman, A.L. Murphy, and G.P. Picco. Software engineering for mobility: A roadmap. In A. Finkelstein, editor, *Future of Software Engineering*. June 2000.

A.L. Murphy and G.P. Picco. Reliable communication for highly mobile agents. *Autonomous Agents and Multi-Agent Systems Journal*, special issue on Mobile Agent Technology. In Press (2000).

G.-C. Roman, A.L. Murphy and G.P. Picco. Coordination and mobility. In A. Omicini, F. Zambonelli, M. Klusch, and R. Tolksdorf, editors, *Coordination of Internet Agents*. Springer. In press (2000).