Report Number: WUCSE-2005-58

2005-11-01

Context Aware Service Oriented Computing in Mobile Ad Hoc Networks

Radu Handorean, Gruia-Catalin Roman, and Christopher Gill

These days we witness a major shift towards small, mobile devices, capable of wireless communication. Their communication capabilities enable them to form mobile ad hoc networks and share resources and capabilities. Service Oriented Computing (SOC) is a new emerging paradigm for distributed computing that has evolved from object-oriented and component-oriented computing to enable applications distributed within and across organizational boundaries. Services are autonomous computational elements that can be described, published, discovered, and orchestrated for the purpose of developing applications. The application of the SOC model to mobile devices provides a loosely coupled model for distributed processing in a resource-poor... **Read complete abstract on page 2.**

Context Aware Service Oriented Computing in Mobile Ad Hoc Networks

Radu Handorean, Gruia-Catalin Roman, and Christopher Gill

Complete Abstract:

These days we witness a major shift towards small, mobile devices, capable of wireless communication. Their communication capabilities enable them to form mobile ad hoc networks and share resources and capabilities. Service Oriented Computing (SOC) is a new emerging paradigm for distributed computing that has evolved from object-oriented and component-oriented computing to enable applications distributed within and across organizational boundaries. Services are autonomous computational elements that can be described, published, discovered, and orchestrated for the purpose of developing applications. The application of the SOC model to mobile devices provides a loosely coupled model for distributed processing in a resource-poor and highly dynamic environment. Cooperation in a mobile ad hoc environment depends on the fundamental capability of hosts to communicate with each other. Peer-to-peer interactions among hosts within communication range allow such interactions but limit the scope of interactions to a local region. Routing algorithms for mobile ad hoc networks extend the scope of interactions to cover all hosts transitively connected over multi-hop routes. Additional contextual information, e.g., knowledge about the movement of hosts in physical space, can help extend the boundaries of interactions beyond the limits of an island of connectivity. To help separate concerns specific to different layers, a coordination model between the routing layer and the SOC layer provides abstractions that mask the details characteristic to the network layer from the distributed computing semantics above. This thesis explores some of the opportunities and challenges raised by applying the SOC paradigm to mobile computing in ad hoc networks. It investigates the implications of disconnections on service advertising and discovery mechanisms. It addresses issues related to code migration in addition to physical host movement. It also investigates some of the security concerns in ad hoc networking service provision. It presents a novel routing algorithm for mobile ad hoc networks and a novel coordination model that addresses space and time explicitly.

2005-58

Context Aware Service Oriented Computing in Mobile Ad Hoc Networks

Authors: Radu Handorean

December 7, 2005

Corresponding Author: radu.handorean@wustl.edu

Abstract: These days we witness a major shift towards small, mobile devices, capable of wireless communication. Their communication capabilities enable them to form mobile ad hoc networks and share resources and capabilities.

Service Oriented Computing (SOC) is a new emerging paradigm for distributed computing that has evolved from object-oriented and component-oriented computing to enable applications distributed within and across organizational boundaries. Services are autonomous computational elements that can be described, published, discovered, and orchestrated for the purpose of developing applications.

The application of the SOC model to mobile devices provides a loosely coupled model for distributed processing in a resource-poor and highly dynamic environment. Cooperation in a mobile ad hoc environment depends on the fundamental capability of hosts to

WASHINGTON UNIVERSITY

THE HENRY EDWIN SEVER GRADUATE SCHOOL

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CONTEXT AWARE SERVICE ORIENTED COMPUTING IN MOBILE AD HOC NETWORKS

by

Radu Handorean

Prepared under the direction of

Professors Gruia-Catalin Roman and Christopher Gill

A dissertation presented to the Henry Edwin Sever Graduate School of
Washington University in partial fulfillment of the
requirements for the degree of

DOCTOR OF SCIENCE

December 2005

Saint Louis, Missouri

WASHINGTON UNIVERSITY

THE HENRY EDWIN SEVER GRADUATE SCHOOL

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

ABSTRACT

CONTEXT AWARE SERVICE ORIENTED COMPUTING IN MOBILE AD HOC NETWORKS

by

Radu Handorean

ADVISOR:

Professors Gruia-Catalin Roman and Christopher Gill

December 2005

Saint Louis, Missouri

These days we witness a major shift towards small, mobile devices, capable of wireless communication. Their communication capabilities enable them to form mobile ad hoc networks and share resources and capabilities.

Service Oriented Computing (SOC) is a new emerging paradigm for distributed computing that has evolved from object-oriented and component-oriented computing to enable applications distributed within and across organizational boundaries. Services are autonomous computational elements that can be described, published, discovered, and orchestrated for the purpose of developing applications.

The application of the SOC model to mobile devices provides a loosely coupled model for distributed processing in a resource-poor and highly dynamic environment. Cooperation in a mobile ad hoc environment depends on the fundamental capability of hosts to communicate with each other. Peer-to-peer interactions among hosts within communication range allow such interactions but limit the scope of interactions to a local region. Routing algorithms for mobile ad hoc networks extend the scope of interactions to cover all hosts transitively connected over multi-hop routes. Additional contextual information, e.g., knowledge about the movement of hosts in physical space, can help extend the boundaries of interactions beyond the limits of an island of connectivity.

To help separate concerns specific to different layers, a coordination model between the routing layer and the SOC layer provides abstractions that mask the details characteristic to the network layer from the distributed computing semantics above.

This thesis explores some of the opportunities and challenges raised by applying the SOC paradigm to mobile computing in ad hoc networks. It investigates the implications of disconnections on service advertising and discovery mechanisms. It addresses issues related to code migration in addition to physical host movement. It also investigates some of the security concerns in ad hoc networking service provision. It presents a novel routing algorithm for mobile ad hoc networks and a novel coordination model that addresses space and time explicitly.

To my family. They all paid **too much** for my endeavour.

Contents

List of Figures

Chapter 1

Introduction

1.1 Background

Small, mobile computing devices steadily are becoming more and more part of our everyday lives. More and more of them are wireless communication–enabled and thus can engage in network communication with either an infrastructure deployed by a service provider (e.g., a cell phone connecting to a base station) or directly with each other (e.g., a laptop and a PDA can connect directly, in an ad hoc mode).

There is a myriad of applications that are needed only at some moments or some places and therefore it makes no sense for a user to carry all of them around all the time. These applications can be delivered only *when and where* needed, can be executed by mobile computing devices anyone can carry, and can then be discarded as soon as the goal is accomplished. For example, finding a parking spot in a garage may be an application needed every morning when going to work, but it is useful only for a few minutes and only around a parking garage. It makes no sense to carry such an application on a PDA all day long only to use it the next day again, for a few minutes. This class of applications is well served by dynamic discovery, utilization, and discarding. This type of software exploitation offers a mobile device with limited storage space and/or computational power, a potentially infinite set of applications it can run, by using a natural time-sharing approach, consisting of discover-use-discard cycles. The approach is known as the *Service Oriented Computing* (SOC) paradigm.

The networks formed by such devices (known to the research community as Mobile Ad Hoc Networks, or MANETS), have a dynamic character, as the only infrastructure supporting them is entirely comprised of the mobile devices themselves that communicate on-the-move, without base stations. Hosts within proximity of each other opportunistically form a network which changes due to host mobility. An ad hoc wireless network is a dynamic environment by necessity, which exhibits transient interactions, decoupled computing, physical mobility of hosts, and logical mobility of code. Such mobile devices both generate user and application traffic and carry out network controls and routing protocols. Rapidly changing connectivity, network partitions, higher error rates, transmission collision interference, bandwidth and power constraints together pose new problems which require special attention, as solutions from the wired environment do not transfer transparently to these mobile settings.

1.2 Technological Trends

Technology has followed a path towards miniaturization and nowadays we can carry in a shirt pocket more computational power than we had only years ago on an entire desk. Calculator watches from the 80's became data carriers for file transfer (watches that connect to a computer via USB and can store as much as 512MB of data), data banks (e.g., Casio dual display watches that can store and display and entire addressbook on the glass on top of the regular dial, Swatch watches that can be used for automatic payment at transportation facilities), and even computational platforms running operating systems and Java Virtual Machines (IBM has developed a watch that runs Linux, and Fossil deploys PalmOS on one of their watches).

PDAs and laptops are delivered these days with built-in adaptors for multiple wireless communication protocols including infrared, bluetooth, and ethernet. Even cell phones that always connect to a base station to send and receive voice traffic, have infrared and bluetooth communication capabilities for peer-to-peer interactions with devices for wireless hands-free operation, or for addressbook synchronization with another device. More and more, such devices also have built-in (or can accept) adaptors for GPS localization, which makes them location aware.

All these devices, including the latest wireless sensors, can form ad hoc networks and can take advantage of each other's functionality. The PDA in a vehicle can connect to a network of sensors in a garage to find an empty parking spot, a temperature sensor can connect to a phone to announce fire danger, two children in different cars driving along on a highway can play games over a wireless car-to-car link, and so forth.

1.3 Research Trends

These advances in hardware are both triggered and closely followed by advances in software development. New types of applications are both needed and now possible. Some existing applications can simply be augmented to work in an ad hoc networking environment while others have to be completely redesigned as implicit assumptions from the wired network (which are no longer valid in MANETS) may have implications for the semantics of the application itself (such as a transaction mechanism which is no longer guaranteed connectivity for the entire length of a transaction) or may just be too difficult to eliminate in the new setting. Entirely new applications are now possible as well.

The research community has investigated challenges associated with infrastructure discovery and maintenance (e.g., neighbor host discovery, routing algorithms for ad hoc networks, etc.) [56], [69], [66], [107], [89], [46], [57], [61], [111], [92], etc., new applications (e.g., ad hoc schedule synchronization, etc.), new types of applications (e.g., mobile applications that can migrate from host to host, context-aware applications that adapt to the user's environment, etc.) [16], [78], [94], [3], [90], [34] middleware systems to facilitate the development of these new types of applications (e.g., service provision in ad hoc networks, coordination models deployed for ad hoc networks, sensor networks programming middleware, etc.) The advance of hardware platforms both rises new questions and demands new answers to problems already solved for previously investigated environments. For example, the easy, centralized design of many architectures in conventional, wired networks evolved to decentralized designs for reliability reasons (to eliminate single points of failure), efficiency and convenience (to have each part of a large system closer to where it is really used), security (to have sensitive code and data protected more carefully), etc. In mobile ad hoc networks, these challenges need new answers, even if, answers are already known in distributed designs for reliable, wired networks.

The most basic element is the possibility for two hosts to simply communicate, i.e., to exchange a packet of information. This bare necessity is seriously challenged in mobile ad hoc networks. The transient interactions between hosts and continuous network reconfiguration require dynamic algorithms that discover and maintain communication routes between hosts in an ad hoc community. The research in this area has produced a plethora of algorithms each improving on the previous or addressing a particular type of ad hoc network, or addressing a particular pattern of message delivery for which it can yield a better performance than a general purpose routing algorithm.

As routing algorithms execute at a low level, upper levels in a layered system architecture are usually shielded from the issues specific to the layers below. This is achieved via encapsulation of the solutions specific to certain problems into packages specific to each layer. A coordination layer, usually delivered as middleware, away details related to communication by exposing a high level interface to be used by upper layers which should not be concerned about anything else than the semantics of the operations they need to perform themselves. The research community has addressed the need for such coordination models in mobile add hoc networks to a significant degree, though more needs to be done. Applications for mobile ad hoc networks are concerned with how to deliver the desired functionality to their users and a coordination layer just below helps to encapsulate other low-level issues, like how to send a message to a partner, how to synchronize execution with other applications' progress (e.g., to coordinate access to a resource), and so on. The various coordination models investigated by the research community so far vary from tuple space coordination semantics delivered in mobile ad hoc networks, to event-based coordination in ad hoc networks, to software migration for purpose of local coordination (a popular approach in the mobile agents research community, where such agents migrate and coordinate directly when on the same physical host).

At the application layer, even in commonplace wired network environments, we are used having access to applications we do not store on our machines, but rather execute only when needed, e.g., over the internet via a web browser-friendly interface.

Listening to internet live radio feeds via a applet opened by a radio station's web site, registering for classes at school, booking a flight or hotel, etc., are all applications that few store locally. We instead *find* them as we need them and then discard them by simply closing a window. There is no reason for us to carry them around. This approach is known as service oriented computing and it has seen its own evolutions of models, culminating with what is known as *web services*. Service oriented computing offers a complete solution to the problems of advertising, searching for, and utilizing, previously undiscovered services (as opposed to the way we might register for classes when *we know* which web page to visit for this particular purpose). While service oriented computing seems promising for delivering software in mobile ad hoc networks, especially where hosts have limited capabilities and therefore could use each other's functionality, the research community has only recently started to investigate this area. Preliminary results are available in a number of areas but the field it not nearly as mature as the others previously described.

1.4 Technical Contributions

My contributions are grouped into three layers: the top layer provides a service-oriented computing middleware for mobile ad hoc networks; the middle layer is a coordination model that exposes time and space explicitly to the user, and is also targeted for mobile ad hoc networks; the bottom layer is a routing algorithm for multi-hop communication in disconnected mobile ad hoc networks, which exploits knowledge about hosts motions.

Service Oriented Computing in MANETS is the solution I identified as a novel approach to provide software for mobile ad hoc networks. Applications deployed as services can be discovered when and where needed, and easily discarded afterwards. The SOC paradigm comes from traditional wired networks, and its transfer to mobile ad hoc networks was a natural, useful, and yet non-trivial thing to do. In ad hoc networks one often cannot connect to the Internet or other commonplace features of the wired networking environment. On the other hand, a mobile device can allow its user to discover applications when needed and where needed, such as finding a payment service at a parking meter or highway tollbooth, a parking service that helps a driver find a free spot in a garage. Issues related to service advertisement, deployment, discovery, utilization, maintenance, migration, have their own specific challenges and solutions in ad hoc networks. The description of my solutions to these challenges follows in Chapter 4.

Coordination Across Time and Space is a novel coordination model, inspired by the Linda coordination model, adapted to mobile ad hoc networks. It exposes space and time explicitly to the user, allowing the user to schedule coordination operations in the future, define the lifetime of operations, send them to specific hosts or even specific geographic areas (assuming they are populated). This coordination model takes advantage of contextual information made available by mobile hosts in forms of motion profiles to identify collaboration partners and plan interactions. It is the first coordination model that allows primitives to be scheduled to execute at a particular moment, to be active between certain moments (e.g., allowing a search for some data only between two given times). It is also the first coordination model that allows primitives to execute in specific places (i.e., geographic locations). These new features of the coordination model I developed are useful in mobile ad hoc networks, where nodes aware of their location and future spatio-temporal evolution can schedule interactions. The coordination layer serves to separate the upper SOC layer from the lower routing layer so that their local do should not cross into each other's layer. A veneer to secure the interactions in the coordination model is presented in Chapter 3. The details of this coordination model are available in Chapter 6.

Disconnected Routing is a new routing algorithm for mobile hosts that uses knowledge about host mobility to discover routes that can deliver messages from a source to a destination. What sets this approach apart from the plethora of algorithms already available is the use of contextual information to build potentially disconnected routes from source to destination. A host can hop a message to the next host, where the message can stay for a certain period of time and travel a certain distance in space, and then hop to the recipient long after the source host has disconnected from the intermediary carrier. This is fundamentally different from current approaches which almost all search for end-to-end fully connected paths from source to destination such that the message can be streamed from start to end rather than hopped on link at a time. These algorithms work only in isolated islands of connectivity where all hosts are connected with each other either directly, in a peer-to-peer relationship, or transitively, over multi-hop fully connected routes. My algorithm goes beyond these islands of transitive connectivity by allowing hosts in different such islands to be part of the same message delivery route between two hosts. The algorithm is presented in Chapter 5.

1.5 Significance

My research provides a solution for SOC in mobile ad hoc networks. My contributions to each of the layers previously described not only take each one of them a step further, but also provide building blocks for the complete solution I offer for context aware service provision in mobile ad hoc networks. While the mobility of hosts is the main source of problems in ad hoc networks, I use it to my advantage. I see in host mobility not only disconnections and route breaks, but also an opportunity for hosts to meet other hosts. Well exploited, host mobility can help increase inter-host collaboration. Throughout my research, I have searched for those elements from the surrounding environment that help turn host mobility to my advantage.

The use of motion profiles, assumed to be known for various durations in the future by each host and exchanged with other hosts upon each encounter, is the key feature of my research. The contributions of my work are the development of known mechanisms from the wired environment into the wireless ad hoc networking environment, with the main thrust being the use of the motion profiles. This has opened research paths beyond just the resulting technology transfer. Analysis of known motion profiles has helped my research and progress at all three layers: routing, coordination and SOC.

The disconnected routing algorithm was possible only because of the use of motion profiles. After the initial work on peer-to-peer interactions, where hosts could only collaborate with neighbors within their direct communication range (and therefore the only important question was *"am I in touch with the other host?"*), new routing algorithms extended this scope to include entire islands of connectivity by allowing hosts to relay packets on behalf of others (making the important question become *"am I in transitive contact with the other host?"*). Some algorithms do consider physical location of hosts, such as geographic routing [63], [39], [107], [82], [59] (e.g., send a message to the node the closest to some location). My routing algorithm takes this trend a step further by enabling communication beyond a sender's island of immediate connectivity by taking advantage of (and paying a price for) additional contextual information available in the form of motion profiles. My algorithm answers, based on motion profile analysis, questions like *"**will** I be able to talk to the other host"?*, and *"**will** I be able to deliver this message at some **location** at some **moment**?".*

While the routing algorithm is concerned about delivering a message between two points, the coordination layer above it searches for certain patterns in the disconnected routes the algorithm can discover. These patterns are characteristic to the coordination primitives the model exposes to its users. I can thus provide coordination across space and time with the help of motion profiles. It is the first coordination model that makes time and space explicitly available for manipulation by the user. The user can explicitly request that a certain operation be executed at a specific time at a specific place, as opposed to the limitation to *"now"* and *"here"* exhibited by the currently known coordination models. *"Now"* means there is no explicit notion of time. The coordination primitives execute as soon as they are invoked. Some may exhibit a synchronous behavior and the "now" moment can extend until something else happens and helps the primitive in question complete (e.g., a primitive may block until a certain block of data is available). *"Here"* means the entity issuing the coordination primitive is in contact (peer-to-peer (P2P) or transitive via fully connected routes) with the place where the action takes place (e.g., a tuple space managed by a centralized server, another agent on the same machine, a process at the other end of a communication channel, etc.) Also, there is no sense of a lifetime of a primitive. The primitives are active from invocation until completion.

My exhibits the notion of a lifetime for a primitive as well as for data exchanged as part of the coordination that takes place (time-sensitive data is not a new idea, especially in database research, but it was never before used in coordination models research). It is also the first coordination model that accounts for (present and future) physical location of entities being coordinated, such that data can be sent to or queried from specific locations, no matter who is there to serve the reader. The coordination primitives and data exchanged have their own profiles. Such a profile is built when the operation is issued and defines the entire lifetime of the primitive. The programmer delimits the active period and declares the target of interest (e.g., a particular host, a particular zone), while the coordination layer translates this in a profile that takes the primitive/data from the creator and carries it over disconnected routes to the designated target, where is activated and destroyed as requested by the programmer. This is the first coordination model that uses contextual information about motion profiles to plan ahead of time *what* happens *when* and *where*, and *who* is involved. The *who* part is enforced by a security veneer that provides means for secure coordination in mobile ad hoc networks.

Once this easy interaction is available, my contribution to the SOC layer represents a fundamentally new approach to delivering services in ad hoc networks. The new approach I took to service advertisement, discovery, and utilization has resulted in a completely distributed model, with no centralized single points of failure and with strong consistency guarantees (e.g., if I discover a service, then I can really use it and I didn't discover a zombie advertisement not yet collected by some garbage collector). This makes traditional service oriented computing, as we know it from wired networks, work in ad hoc networks too. In addition to this, I provide a solution for runtime upgrades to services, completely transparent to the client(s) using the service. It's the first such effort in service oriented computing. In addition, I allow the services to move freely in an ad hoc network to better service their clients. This unprecedented level of flexibility increases the degree of assistance clients get from their service providers. The services can actually follow the client migrating from host to host to follow the client's machine as this moves in space and changes connectivity partners.

My overall contribution thus combines research at several levels of abstraction from the networking layer to the application layer to provide context aware secure service oriented computing in mobile ad hoc networks.

Chapter 2

Overview of Tuple Space Coordination in Ad Hoc Networks

Coordination is a paradigm that promises to address some of the issues related to the development of complex parallel and distributed systems. The development of such a complex system can be imagined as made of two different parts: (1) *computing*: the processes that manipulate the data and (2) *coordination*: the abstractions responsible for enabling cooperation and communication among these processes. Coordination thus distinguishes the computational concerns of such distributed systems from the communication concerns, allowing the separate development and occasional intersection of these independent components of such systems. While there are multiple definitions for the technical term "coordination model", a short intuitive statement can be found in [88]: "*A coordination model is the glue that binds separate activities into an ensemble.*"

From the larger domain of all coordination models, I will describe a particular family of models that use a *shared dataspace* to carry out the coordination, as these models are related to and have influenced my research. A data space is a central, content-addressable data structure [98]. All processes involved in coordination use this data structure as a communication buffer between them. These processes can post information in the shared dataspace, can read information or can remove information from the dataspace. The information in the dataspace has a life span mainly independent of the lifespan of the processes that published that data. The consumer of some data doesn't need to know the identity of the producer of that data as they do not need to meet in order to exchange this data. A consumer retrieves data from the shared dataspace by providing a description of the item the consumer is looking for. If available, a matching item is returned to the consumer process.

The exact and detailed semantics of the operations that write, read, or remove items from the dataspace vary among different specific models and various implementations of coordination models based on shared dataspaces. The first coordination model in this family, which inspired all all its successors is the Linda coordination model [37]. In Linda, the data published in the shared dataspace is in form of tuples and therefore the shared dataspace is called a tuple space. A tuple is a sequence of fields, each field having a type and a value. The coordination primitive out(tuple) puts the tuple in the tuple space. To read a tuple from the tuple space a consumer needs to provide a template and the rd(template) operation returns a tuple chosen nondeterministically from all tuples available in the tuple space and that matches the template. A match between a tuple and a template is declared if each field in the tuple matches the corresponding field in the template. The field in the template can describe only the type of data expected in the tuple field or may specify the requisite value as well.

<Integer(25), String(“Boat”), Location(Pacific)>………..tuple
<Integer(25), String.(“Boat”), Location.class>……...template

Figure 2.1: Tuple and template.

Figure 2.1 shows a tuple and a template matched by the tuple. The tuple has three fields: the first is of type Integer and has the value 25; the second field if of type String and has the value Boat and the third is of type Location and has the value Pacific. The first field in the template requires the matching tuples to have a first field of type Integer and with value 25. The second field is required to be of type String and have the value Boat, while the third field is required to be of type Location but no particular value is specified.

Two implementations of the Linda coordination model targeted to MANETs inspired the middleware I will present in this document, and consisted the inspiration for the coordination model I will describe in Chapter 6: Lime [80] and Limone [32]. The active entities are called agents. An agent can be a thread of execution for an entire application. Each agent has it’s own tuple spaces, identified by name. For simplicity, I will assume there is only one tuple space per agent and each such local tuple space has its own name.

The common and essential characteristic of these models is the transient sharing of tuple spaces. In Lime, for example, when two agents are in touch (i.e., run on the same host or on hosts within communication range), they can share their local tuple spaces in a federated tuple space, if the local tuple spaces have the same name. Once the tuple spaces are merged, each accesses the content of the federated tuple space as if it was its own local tuple space. This means the scope of the coordination operations performed by an agent on its local tuple space extends transparently to the entire content of the federated tuple space.

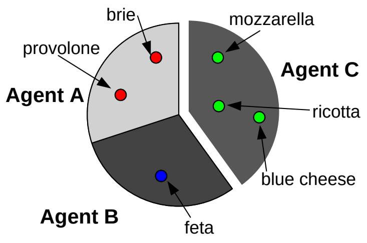


Figure 2.2: Transient sharing of tuple spaces. Each of the three slices are the A's, B's, and C's local "Cheese" tuple spaces and form the federated "Cheese" tuple space.

For example, in Figure 2.2, Agents A, B, and C have each a local tuple space called "Cheese". Agent A has two tuples: "brie" and "provolone". Similarly, Agent B has a "feta" tuple, and Agent C has the "mozzarella", "ricotta", and "blue cheese" tuples. Agents A and B are within communication range and they merged their local tuple spaces in a federated "Cheese" tuple space. Both A and B can operate on this federated "Cheese" tuple space and see its entire content (i.e., "brie", "provolone", and "feta") as if this was the content of their own local tuple space (the figure shows local tuple spaces in different shades of grey so they can be seen as separate contributors to the federated tuple space). Agent C is on a machine too far from the hosts where the other two agents are running. Therefore C has access only to its local inventory of cheese. A similar discussion could be carried assuming C is within communication range and therefore could share it's tuple space with A and B, but C's local tuple space has a different name (e.g., "Dairy") and will not be shared/merged with the other two.

The primitives supported by Lime (originally introduced with the Linda coordination model) for publishing, reading, and removing a tuple are out, rd, and, in, respectively. If no matching tuple is found, rd and in block waiting for a matching tuple to appear. There are non-blocking versions called probes (i.e., rdp and inp) who return a null response without blocking if no tuples matched their template. Group operations outg, rdg, and ing handle groups of tuples.

Lime supports a special type of operations: reactions. A reaction is a piece of code associated with a template. When a tuple matching that template is found in the tuple space, the code of the reaction is executed. Using reactions, agents can, for example, search for tuples without staying blocked in a synchronous rd or continuously polling using rdp.

The use of a coordination model was justified in Chapter 1. There are multiple reasons I chose a tuple space–based coordination model to support service oriented computing in ad hoc networks (initially Lime, then Limone, and then, as we will see in Chapter 6, I developed a new tuple space–based coordination model). Tuple space-based coordination has the advantage of an extremely small yet very powerful interface. A few coordination primitives are sufficient for developing sophisticated scenarios and these fully addressed my needs. In addition, the model applies very conveniently to ad hoc networks where the transient sharing of tuple space is tightly bound with the direct connectivity between peer hosts. The transient sharing of tuple spaces offers transactional guarantees with respect to updates to tuple space contents as connectivity changes. Tuple spaces are also a very convenient communication medium. Tuples can be exchanged over virtual communication channels created inside tuple spaces (e.g., all tuples belonging to a communication session are tagged with a session ID in their first field, so tuples belonging to different communication sessions can share the same tuple space yet remain distinguishable). In addition, the content-based type of communication confers the communication between two entities a location-agnostic character, an extremely useful feature as will be shown later in this document. This eliminates point-to-point types of communication channels, which are very inconvenient in ad hoc networks. As hosts change configurations and applications migrate from host to host, point-to-point communication protocols such as those carried out over direct socket communication are difficult to redirect and entail a significant overhead (sockets have to be closed and (re)opened). Additionally, a tuple space can carry out middleware level or application level communication, and can be a repository for data or code which can migrate from host to host packaged in tuples.

Chapter 3

Security Issues

3.1 Introduction

Ease of coordination in an open environment is a great asset but it must be tempered by security concerns. The open environment of an ad hoc network and the ease of tuple space-based interactions make corruption of information through malicious tampering or simply by accident an important concern. As the coordination model is the foundation on top of which the rest of my architecture is built, addressing security issues at this level is both easier and both more effective and more efficient than at any other level. The layers above the coordination layer can take advantage of these secure coordination primitives and use them directly, while still mainly concentrating on the semantics of the problems specific to their respective levels.

While these problems are not new, the novelty of the challenges arises from the new settings in which these problems have to be solved. Many strategies commonly used in wired networks become problematic in the ad hoc setting. There is no protection against eavesdropping, there are no trusted authentication servers, there are no centralized databases of secure information, etc. Full transparency may be desirable but the ability to hide security concerns from the application developer and user may not always be feasible. In this chapter, I answer the question whether the coordination strategy made available in Lime can be made secure with minimal impact on the Lime middleware and on its fundamental coordination model.

My solution was to extend Lime in two important ways. First, I offer password-protected access to tuple spaces. The sharing policy within a group is extended to require not just the same name but protection with the same password. This is complemented by the ability to password-protect individual tuples regardless of whether they are part of a protected or unprotected tuple space. Interestingly enough, the implementation of these two capabilities employs distinct features of the underlying Lime system itself. Moreover, by exploiting the fact that Lime restricts tuple space access to its creator agent and other agents that create a tuple space with the same name and share it with other agent's local tuple space (i.e., if an agent creates a local tuple space, it cannot hand it to another agent), password usage is limited solely to tuple space creation, thus minimizing the scope of API modifications and also affording some level of robustness in regard to possible programming errors involving incorrect password utilization. The tuples inside a tuple space need to be protected too, against malicious or accidental tampering. For this reason, I developed tuple-level protection mechanisms that allow three levels of access to a tuple: *free access*: any agent with access to the tuple space has access to these tuples; *read-protected* - agent with access to the tuple space need a password to read the tuple; *remove-protected* (read-only tuples) - free to read by all agents with access to the tuple space but still require a password for removal. This can logically subdivide a tuple space into secure subspaces with different levels of protection. The third and last security mechanism is a transparent encryption of all communication related to protected tuple spaces. Any operation and parameter of operation (e.g., tuple or template) addressed to a secure tuple space that needs to travel from a host to another is transparently encrypted before being sent out in the wireless network and is immediately decrypted upon receipt by the remote host. While the mechanism does not protect against physical level attacks (e.g., denial of service attacks by jamming the radio waves), it does protect against eavesdropping by rendering the snooped information useless to the intruder.

By making effective use of the existing Lime design, the secure version of the system is realized by wrapping the existing middleware with a security veneer above and an interceptor below. The interceptor provides the proper encryption of messages associated with secure tuple spaces using a protected table shared with the former. The price for achieving this level of simplicity is the need to accomplish an initial password distribution possibly outside of the application itself and the requirement for the application to manage required password changes in response to possible security compromises.

This research was targeted towards securing an existing coordination model and not towards designing a secure coordination model from beginning. While there is work in the area of secure coordination, the main interest in securing this particular model was the safety of communication rather than exploring *new* coordination models built around the idea of secure interaction domains or any other type of security. This work was done mostly to meet the requirement of having a secure platform rather than to claim fundamental advances in the area of secure interactions.

3.2 Secure Tuple Spaces

The name of the tuple space is the key to gaining access to the information in that tuple space. With the name of the tuple space one can create a local tuple space which can be shared with other local tuple spaces bearing the same name, thus gaining access to the contents of the latter tuple spaces. To make matters worse, all tuple space names are freely available in a public tuple space, maintained by the Lime system, and identified by the known name LimeSystemTupleSpace. To protect the information means to protect the name of the tuple space, even it it is visible in the LimeSystemTupleSpace.

The first step is to make the information obtained from the LimeSystemTupleSpace unusable in its raw form. Changes are required to ensure that extracting the name of a protected tuple space from the LimeSystemTupleSpace will no longer provide enough information for an agent to create a tuple space with the same name and share it with other agents thus gaining unauthorized access to its content.

To achieve this, some processing of the tuple space name is done on the way from the constructor call, when the tuple space is created, to the internal storage of the name inside the system. The information available in LimeSystemTupleSpace will be the processed name of the tuple space. I ensure this information cannot be used in its processed form obtained directly from the LimeSystemTupleSpace and also that it cannot be generated incorrectly (the internal processing of this information entails some processing beyond the control of the external user). Public (unprotected) tuple spaces are not part of this concern.

For this reason tuple spaces are split in two categories: protected and unprotected. If the user creates a tuple space that is intended to be secure, the user will have to provide a password. If no password is provided, the tuple space is assumed to be unprotected. For secure tuple spaces, the password is used to encrypt the name before marking it as a secure tuple space name and forwarding it to the previous implementation of Lime which will use it as if it were a regular string representing a name of a tuple space that will be used for sharing.

The constructor call is the only place where the agent explicitly uses the password. Once the agent has the handle to the tuple space, it does not need the password anymore. The tuple space handle enables the agent to access the tuple space for as long as the agent has it without having to provide the password. All methods will be invoked as before and will use the tuple space protection password transparently to the agent, if needed. As I mentioned before, these tuple space handles are not transferrable. When an operation is called on a tuple space, Lime verifies that it was called by the thread representing the agent that created it. This is why it is not necessary to ask for the password when a tuple space operation is called.

The tuple space name (encrypted name when a password is provided or the plain, clear name if the tuple space is not meant to be protected) will be prefixed by a differentiator: letter "U" for unencrypted, or "S" for secure tuple space. The unprotected tuple space called "blue" is different from the tuple space called "blue" and protected with password "pwd" (the latter will actually have the internal name K_{pwd}(blue)). They can coexist but no sharing takes place. The prefixes ensure that a tuple space cannot be created incorrectly. Since they are internally added, they cannot be manipulated by agents. Reading the name of a (secure) tuple space from LimeSystemTupleSpace will not be enough to create an insecure tuple space with the same name. A prefix will be attached in front of whatever the programmer provides as a tuple space name. If an attacker reads the name of a protected tuple space from LimeSystemTupleSpace and tries to create a tuple space with the same name, there are two ways she/he could follow. One is to create the tuple space as an unprotected tuple space. In this case the system will add the "U" prefix and will not be shared with the original tuple space. The second attempt would be to trick the system to add the "S" prefix. To do so it will be necessary to create a secure tuple space. In this case the information retrieved by the attacker from LimeSystemTupleSpace is useless since he will need to provide the clear name and the correct password. There are no "blank" passwords that can be used to encrypt a text and to yield the same text as result. The prefixes also address the case when the result of encrypting the clear name of a tuple space coincides with the name of an unencrypted tuple space (before adding prefixes).

The encrypted name of a protected tuple space and the password that protects it are important not only when the tuple space is created and shared but also later in inter-host communication. This is why the Lime server was augmented with a SecurityTable that stores entries of the form [encrypted name, password]. An entry is added to this table every time a new secure tuple space is created. When an operation is executed on the tuple space, if it runs on the local host of the issuer no further verification is needed. For executions of tuple space operations that span beyond the limits of issuer's host, the table will be used for more verifications. See Section 3.3 for details.

This **SecurityTable** is a very important target that has to be protected. Currently, only the default Java object protection mechanisms protect this table. I could encrypt it and provide a somewhat more difficult and less efficient form of access to it but this would only shift the problem to protecting the password used for that additional encryption. Since my work does not address the Java security model, I assume this model is secure enough for my research.

3.3 Secure Communication

If a tuple space operation involves a remote execution on some other host whose agent contributes to the federated tuple space, the request is sent across the wireless link and the result is sent back over insecure wired or wireless lines. Eavesdropping is easy because the information traveling across the network consists of clear serialized Java objects. Secure communication between hosts is achieved by encrypting the messages associated with a given tuple space using the password supplied when the tuple space was created first (if any). The remote party is supposed to have access to the same password since sharing of the tuple space is taking place. For tuple spaces which are not protected, the messages will not be encrypted and the other party will need to know only the communication protocol in order to be able to deserialize the objects received in the request.

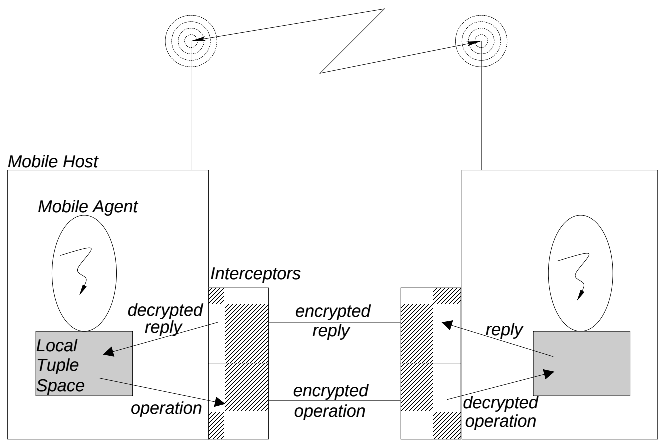


Figure 3.1: Interceptors secure wireless communications.

When an agent executes an operation that spans beyond the limits of the current host, an interceptor catches it, analyzes the tuple space that the message refers to (the name of the tuple space is always present in the message that travels across hosts) and takes the appropriate action (the use of the interceptor pattern [102] is natural for this case, to add security to a system that in its initial design did not address this issue). It also offers a great deal of flexibility with respect to the choice of encryption protocol. Figure 3.1 shows how interceptors secure the communication between two hosts.

The interceptor checks whether the tuple space name appearing in the outgoing message is present in the SecurityTable. If the message refers to an unprotected tuple space (it is not in the table), the interceptor lets it pass through unchanged. If the tuple space is a secure one, the interceptor extracts from the table the password that corresponds to that tuple space and uses it to encrypt the message. The interceptor creates a packet that contains the encrypted message and the encrypted name of the tuple space the message refers to and forwards this packet to the other involved host. On the recipient's side, actions happen symmetrically. Another interceptor catches the incoming message, looks up the encrypted name of the tuple space in the local SecurityTable and if found, uses the corresponding password to decrypt the message. The message is then forwarded to the LimeServer. If the target tuple space is not a secure one, the name is not found in the SecurityTable and the message is forwarded unchanged to the LimeServer. The results are handled in the same way on their way back.

Special attention has to be paid when using password-protected tuples in unprotected tuple spaces. The traffic between two hosts is unprotected if it refers to an unprotected tuple space. If such a tuple space contains a password protected tuple (let's say the tuple has only a read-password) then a rd or in operation will need to provide the password along with the template. Let's assume a rd operation provides the correct password. Since the tuple space is not protected, the communication is not encrypted so the password travels in clear between the two hosts (the content of a tuple can NOT be encrypted because it would destroy the entire matching mechanism and therefore the passwords can only be used as additional fields to filter the access). A hacker could steal the password and use it then to remove the tuple (the tuple does not have a remove password so the read-password will be the only protection against removal as well). Password-protected tuples are safe to use in unprotected tuple spaces as long as the owner does not disclose the password (no message carrying a password should travel over insecure communication channels).

3.4 Secure Tuples Inside Tuple Spaces

Even if an entire tuple space can now be protected, restrictions at the tuple level are still desirable in many applications. The reasons are two fold. In case of a secure tuple space shared among cooperating agents, tuple level protection can protect inadvertent tuple removal or access. Similarly, in an unprotected tuple space this feature affords some level of protection against malicious agents.

A tuple may have a password to protect the tuple from removal (hereafter called remove-password) and a different password that protects the tuple from reading (hereafter called read-password). If these passwords are different and non null, a

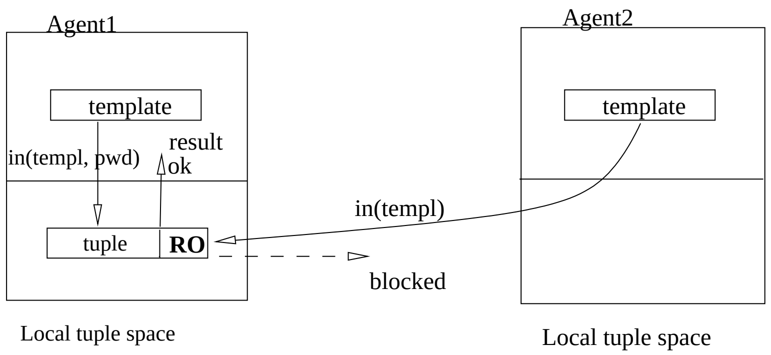


Figure 3.2: The execution of an *in* operation matching a read-only tuple. Agent1 is able to retrieve the tuple because it provides the (correct) remove-password, while Agent2 blocks because its template, even when it matches the data part of the tuple, does not satisfy the security requirements.

rd operation can read the tuple if it provides either one of the passwords, assuming that the fields match. This is because an agent that has the password to remove a tuple is also entitled to read the tuple. If a tuple has a remove-password, an in operation will have to provide the same remove-password to match this tuple. If the tuple has no remove-password but has a read-password, an in operation will need to provide the latter password to remove the tuple (Figure 3.2).

To implement read-only tuples every tuple space operation adds to the end of the user specified fields (if any) three fields. They are in order: the read-password, the remove-password and the name of the operation that uses that tuple or template (e.g., "rd" for any type of read operation, "in" for any type of remove operation and "out" for any type of write operation). This is because both tuples and templates are instances of the Tuple class in Lime and the matching mechanism needs to differentiate them, as well as the operation that requested the matching. If either password is absent the field contains an instance of a NULL class (serializable object replacing Java's null). When a tuple is written to the tuple space, the out method can specify either or both the read-password and the remove-password to protect the tuple.

To read a tuple, the rd method takes a read-password beside the usual template. This method constructs a template that contains the NULL in remove-password's position and the read-password in the right place. For removing a tuple, the situation is similar. The in operation takes an extra parameter, the remove-password. The read-password is filled in with the same value since I consider that a template that is allowed to remove a tuple should also be allowed to read the tuple. In some cases one of the two passwords expands in the other's field from a semantic point of view. For example, if a tuple has a read password but no remove-password, a template trying to remove the tuple will need to have the read-password. Likewise, if a tuple has a read-password and a remove-password, and the template provides the remove-password for a read operation, access will be granted. Group operations are implemented similarly. An outg protects each tuple written to the tuple space with the password(s) provided (if any). The ing and rdg operations return only the tuples that satisfy the matching criteria for both the data and security parts. Figure 3.3 shows examples of tuple space access methods, involving passwords.

lts.out(ITuple tuple, char[] readPwd, char[] removePwd)
— writes a tuple to the tuple space and protects it against reading and/or removing. Any combination of the two passwords is permitted.

lts.rd(ITuple template, char[] readPwd)
— reads a tuple from the tuple space if the tuple and the template match (and the correct password is provided).

lts.in(ITuple template, char[] removePwd)
— removes a tuple from the tuple space if the tuple and the template match (and the correct password is provided).

Figure 3.3: The tuple space interaction operations.

Even though the matching of the fields is carried out internally by Lime, the password fields in particular showed that it is useful to have the possibility to chose the matching policy specific to a particular field. This required an extension to Lime for the ability to select among different matching rules on a field by field basis. First, a field in a tuple may require the template to provide the exact value of the field for a match to be declared (hereafter called exact_value match). Second, the tuple may restrict only the type of the template field to be the exact type of it's own field. (hereafter called exact_type). Finally, the least constrained type of matching is when a tuple's field allows a wildcard in the template's corresponding field. For example, the Java Object object is a wildcard that will always match under these circumstances. This type of matching takes advantage of Java's OO polymorphism and this is why this policy is called poly_type.

When fields are added to a tuple, the type of matching can be specified for each of them. Figure 3.4 shows how fields are added to tuples and how to specify the matching policy for each of them. Field.EXACT_VALUE, Field.EXACT_TYPE and Field.POLY_TYPE are predefined integers in Field class that identify the matching policies. The tuple passwords are transformed into fields subject to the exact_value policy and added at the end of the tuple when written to the tuple space.

```
Tuple t = new Tuple();
t.addActual(new Integer(1), Field.EXACT_VALUE)
 .addActual(new String("WU"));
```

— creates a tuple and adds fields it. To match this tuple, a template will need to have an **exact_value** on its first field (that is an actual of type Integer and value 1). Since the second field doesn't have any matching policy specified, the **poly_type** is assumed.

Figure 3.4: Adding fields and the matching policy to a tuple.

3.5 Summary

This chapter presented a way to add security features to the Lime coordination model. I used Lime because it is the first coordination model designed for ad hoc networks. Later I used Limone and the mechanisms presented above transferred almost transparently to the new coordination model. The overall architecture of secure Lime can be observed in Figure 3.5. The application from the top layer has access to the API that creates and manipulates (secure) tuples and tuple spaces. The secure tuples and tuple spaces layer interfaces with Lime and is linked to the interceptors at the bottom of the architecture via the security table (which contains the passwords for each protected tuple space). The interceptors are the last filter before a message carrying a tuple or a command leaves a host. They encrypt the communication related to secure tuple spaces. On the recipient side, the message works it way up to the application level by clearing its way through the same layers traversed in reverse order, beginning with being decrypted by the receiving interceptor (if applicable).

The security veneer provides mechanisms needed to control *who* can do *what* and *how* with *which* tuples. I have showed that simple changes can transform a coordination

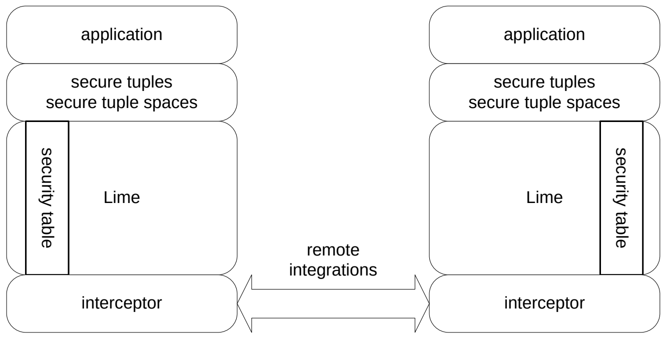


Figure 3.5: Secure Lime architecture.

model into a platform suitable for the development of secure applications. The mechanisms are general and can solve real issues in terms of secure coordination in ad hoc networks.

While not solving all issues that can arise related to security, this research provides a basic level of security for interactions in mobile ad hoc networks. It especially provides a solution to the problem of public key advertisement within the domain of coordination in ad hoc networks. The major problem with the public key distribution is associating a public key with the real owner of that key. By having agents advertise their public keys in read-only tuples which cannot migrate from their own local tuple space, the problem is solved. Once there is a solution for public key advertisement, more sophisticated algorithms (almost any security algorithm) can be developed to exchange private keys, etc.

The initial use of this security mechanisms was to secure coordination in Lime [80]. The mechanism transferred later to secure Limone [32] and can be used in a similar way to secure the coordination model I developed, which is presented in Chapter 6.

The specific contribution of the work described in this chapter is the secure coordination veneer which provides protected tuple space access, transparent encrypted communication, and three levels of tuple access: free access, read-only (remove password protected), read protected (read password protected).

Research dissemination. This material is based on the research published in the paper *"Secure Sharing of Tuple Spaces in Ad Hoc Networks"* published at the 2004 International Workshop on Security Issues in Coordination Models, Languages, and Systems (28-29 June 2003, Eindhoven, The Netherlands) and *"Coordination Middleware Supporting Rapid Deployment of Ad Hoc Mobile Systems"*, published at the ICDCS International Workshop on Mobile Computing Middleware (19-22 May 2003, Providence, RI, USA). In combination with the work described in Section 4.3, a secure service oriented computing middleware was described in *"Secure Service Provision in Ad Hoc Networks"* published at the 2003 International Conference on Service Oriented Computing (15-18 December 2003, Trento, Italy).

Chapter 4

Service Oriented Computing in Ad Hoc Networks

4.1 Introduction

Service Oriented Computing (SOC) is the latest step in a progression of widely used programming paradigms containing, among others, the object-oriented computing and component-oriented computing paradigms. The service oriented computing paradigm is characterized by a minimalist philosophy, in that a user needs to carry only a small amount of code in its local storage, and exploits other services by discovering and using their capabilities to complete its assigned task.

Migrating service oriented computing to ad hoc networks is non-trivial and requires a systematic rethinking of core concepts. Many lessons have been learned from the work done on SOC in the wired setting, especially regarding description and matching of services. However, the more demanding environment of an ad hoc wireless network requires novel approaches to advertising, discovering, exploiting, and maintaining services. I envision such ad hoc networks being used in a range of application domains, such as response coordination by firemen and police at disaster sites or command and control by the military in a battlefield. Such scenarios demand reliability despite the dynamic nature of the underlying network.

The transition to mobile ad hoc networks is facilitated by the use of the previously mentioned tuple space-based coordination models (Chapter 2, which provide support for interactions in the new environment. The work described in this section constitutes a layer on top of the secure coordination layer described in Chapter 3. The coordination model ensures consistent interactions in the mobile ad hoc networking environment and the security extensions provide for safe interactions among service providers and clients.

A service oriented computing framework is a conglomerate of elements, with each element fulfilling a very specific role in the overall framework. The salient elements required for a viable service oriented computing framework are:

- The *service description* element is responsible for describing a service in a comprehensive, unambiguous manner that is machine interpretable to facilitate automation and human readable to facilitate rapid formulation by users. The aim is to specify the functions and capabilities of a service declaratively using a known syntax. A good service description mechanism must have a clear syntax and well defined semantics which facilitate matching of services on a semantic level.

- The *service advertisement* element is responsible for advertising a given service description on a directory service or directly to other hosts in the network. The effectiveness of an advertisement is measured as a combination of the extent of its outreach and the specificity of the information it provides up front about a service, which can help a user determine whether he or she would like to exploit that service.

- The *service discovery* element is the keystone of a service oriented computing framework and carries out three main functions. It formulates a request, which is a description of the needs of a user. This request is formatted in a similar manner to the service description. This element also provides a matching function that pairs requests to services with similar descriptions. Finally, it provides a mechanism for the user to communicate with the service provider. A good discovery mechanism provides a flexible matching algorithm that matches advertisements and requests based on their semantics and maximizes the number of positive matches giving the user a more eclectic choice of services for his or her needs.

- The *service invocation* element is responsible for facilitating the use of a service. Its functions include transmitting commands from the user to the service provider and receiving results. It is also responsible for maintaining the connection between the user and the provider for the duration of their interaction. A good invocation mechanism abstracts communication details from the user and, in the case of network failure, redirects requests to another provider or gracefully terminates.

- The *service maintenance* element is responsible for maintaining the service throughout its entire life. Its functions include upgrading the functionality offered to the clients, upgrading the quality of assistance already offered (e.g., encrypting the communication protocol used between the client and the server while offering the same functionality). A good service maintenance mechanism performs its tasks with no or little impact on the interaction with the clients it is already serving.

- The *service composition* element provides mechanisms to merge two or more services into a single composite service which combines and enhances the functions of the services being composed. A good service composition mechanism leverages off each of the elements listed above to provide automatic composition of services without human intervention.

In the next section I describe how different earlier research efforts have addressed the items from the list above, in an overview of the related work. Then, I present how some of the components from above need to be reconsidered when developing a SOC architecture for mobile ad hoc networks. After the related work overview and motivation are presented, the remainder of the chapter is composed of two main parts: how I have designed basic SOC components for ad hoc networks, and a fundamentally new idea of mobile services realized in a "follow-me session" which provides (within limits) continuous assistance to a client despite the changes in connectivity.

4.2 Related Work and Motivation

4.2.1 Related Work Overview

The SOC elements described previously in this chapter form the building blocks for most service oriented computing models. Various models entail the use of some or all of the elements described, depending on their complexity. I review some of the more popular models as background to my work and highlight features unique to each model.

The **Service Location Protocol** (SLP) [45] was developed by the SRVLOC working group of the Internet Engineering Task Force (IETF) with the idea of creating a service oriented computing standard that was platform independent for the Internet community. In SLP, every entity is represented as an agent. There are three kinds of agents - User Agents, which perform service discovery on behalf of clients, Service Agents which advertise locations and capabilities on the behalf of services and register them with a central directory, and Directory Agents which accumulate service information received from numerous Service Agents in their repository and handle service requests from User Agents. An interesting feature of SLP is that it can operate without the presence of a Directory Agent, i.e., it does not require a central service registry to function. User Agents search for a Directory Agent by default but if they do not receive any replies, they continue to operate directly with peer agents. If a Directory Agent starts operating at some point in the future, it advertises its presence and the User and Service Agents that receive the advertisement automatically start using the Directory Agent as a central service repository. SLP registers services using templates which follow a specific pattern. Requests are made using a similar template though the parameters differ slightly. SLP's ability to operate in the absence of a Directory Agent makes it especially useful in ad hoc environments where it is not feasible to have a central service registry. Overall, SLP offers a clean model that is easy to conceptualize and, due to its design, can be implemented readily in modern languages like C++ and Java.

Universal Description, Discovery and Integration (UDDI) [112] was formulated jointly by IBM Corp., Ariba Inc., and Microsoft Corp. UDDI technology is aimed at promoting interoperability among web services. It specifies two frameworks, one to describe services and another to discover them. UDDI uses existing standards such as Extensible Markup Language (XML) [121], Hypertext Transfer Protocol (HTTP), and Simple Object Access Protocol (SOAP) [122]. The UDDI model envisions a central repository which is called the UDDI Business Registry (UBR). A simple XML document is used to specify the properties and capabilities of a service. Registration with the UBR is done using SOAP. Information present in a service description can include things like the name of the business that provides the service, contact information for people in charge of administering the service, and identifiers for the service and descriptions. The UBR acts as a mediator and assigns a unique identifier to each business and each service. Marketplaces, search engines and enterprize level applications query the UBR for services, which they use to integrate their software better with other business entities like suppliers, dealers, etc. The UDDI model is distinctive in that its approach looks at service oriented computing from a business process perspective. It envisions electronic interactions between businesses resulting in trading of services and goods much like the business-to-business (B2B) model today but with enhanced capabilities and features. With the backing of industry giants like Microsoft and IBM, this technology evolved to become today's web services.

Universal Plug and Play (UPnP) [75] was developed by Microsoft Corp. to facilitate seamless communication between networked devices in close proximity to each other. UPnP leverages TCP/IP and the Internet for its communication needs. It assumes a network that is dynamic with devices frequently joining or leaving. A device, on joining a network, conveys its own capabilities upon request and learns about the capabilities of other devices. When it leaves the network, a device does so without leaving behind any explicit evidence that it was there. Entities in the network may be controllable devices or controllers. Controllers actively search for proximate devices. The network is "unmanaged" in that there is no DNS or DHCP capability. Instead, a mechanism called AutoIP [77] is used to allocate IP addresses. In UPnP a client, also called a control point, can undertake five kinds of actions. The discovery action is based on the Simple Service Discovery Protocol (SSDP) [41]. It exchanges information about the device type, an identifier and a URL for more detailed information. During the description action, the control point uses the URL obtained during the discovery action to get a detailed XML description of the device.

Then the control point sends a control message encoded in XML to a Control URL using SOAP to discover the actions and parameters to which the discovered device responds. The event action consists of a series of events formatted in XML using the General Event Notification Architecture (GENA) [20] which reports changes in the state of the service. The presentation action consists of presenting a URL that can be retrieved by a control point, presumably in a format that allows an end user to control the discovered device. At this point UPnP assumes some external architecture and protocol that handles subsequent interactions with the device. UPnP uses multicast for discovery and unicast for service utilization and event notification. Services and controllers are written using an asynchronous, multithreaded application model. UPnP requires a web server to transmit device and control descriptions, though this server is not required to be on the device itself. This requirement means that UPnP works best on the web and porting it to other environments, especially those with low-power hosts, is non-trivial.

Salutation [100] is an open standard service discovery and utilization model formulated by the Salutation Consortium. The vision for Salutation is not on the scale of the World Wide Web. Instead, Salutation hopes to promote interoperability among heterogeneous devices in settings such as corporate LANs where there is permanent connectivity from either wired networks or wireless gateways and disconnection is not an issue. In addition, Salutation strives to be platform, processor, and protocol agnostic. Salutation has two major components: (1) the Salutation Manager (SLM) which presents a transport independent API called the SLM-API and (2) The Transport Manager (TM) which is dependent on network transport and presents a SLM-TI (transport interface) between the Salutation Manager and the Transport Manager. Services are registered via a local SLM or by a nearby SLM connected to a client. Service discovery occurs when SLMs find other SLMs and the services registered with them. Capability exchange is done by type and attribute matching. The SLM protocol uses Sun's ONC RPC [106]. Periodic checks by the client ensure that it has the most current list of available services. Salutation is interesting in that it solves a highly focused but widely relevant problem. An ideal environment for a Salutation deployment is a busy office space where there are a lot of computers, printers, fax machines and other electronic devices. Using Salutation to automatically discover and use devices within range eliminates the effort of individually configuring every single device. This makes Salutation a useful productivity enhancing tool.

Jini [118] is a distributed service-oriented model developed by Sun Microsystems. Services in a Jini system can be hardware devices, software components or a combination of the two. A Jini system has three types of entities; service providers, lookup services and users. A service provider multicasts a request for a lookup service or directly requests a remote lookup service known to it a priori. Once the handle to a lookup service has been obtained, the service provider registers a proxy object and its attributes with the lookup service. The proxy object serves as a client side handle to the actual service. The user makes a request for some service by specifying a Java type (or interface that the desired service must implement) and desired attributes. Matching is done based on type and values. Once a candidate service is found, the proxy object registered by the service provider is copied to the client using RMI [108]. Clients use the proxy object to interact with the service. The Jini model is designed for ubiquitous computing and is intrinsically scalable. It is language and protocol independent (though Java and TCP/IP seem to be natural choices for an implementation) and is resilient because multiple lookup services ensure that there is no single point of failure. Jini holds much promise for middleware developers that use the Java programming language, and has been formulated with Java-like languages in mind. Also, Jini is unique in that it introduces the idea of shipping a proxy object to the client where it is used as a handle to the actual service. The proxy approach allows complex services to reside on a server at a well known location. The service then simply has to encode its well known location within its proxy, and can then be called from any client without that client having any knowledge of the actual location of the service. It also mitigates the issue of establishing the protocol between the service and the user since the proxy object hides all such communication from the end user.

As the SOC field has matured, most of the activity now gravitates around an area known as **web services** which seems to be the convergence of lessons learned from the research into the preliminary models described above. Web services represents a branch of the service oriented computing paradigm, characterized by the standardization of the elements enumerated above. Service description is realized using standards like RDF [116] and WSDL [117], the advertisement and discovery are realized with the help of UDDI [112] service registries. Invocation is realized via the SOAP [122] protocol which carries XML[121] formatted messages encoding method invocations and returned results. To ensure the semantic compatibility of message contents (i.e., to ensure that both parties understand the same thing when they read/write the string “printer”), ontologies [55] are used and assumed to be known and understood by both parties. To be part of web services, an application has to adhere to these standards.

4.2.2 Challenges in Ad Hoc Networks.

The models above exhibit a common characteristic that makes them all fail in ad hoc networks. The implicit assumption that the Internet is always there to help remote parts find each other and stay in contact is a fundamental limitation when considering the dynamic environment of ad hoc networks.

In an ad hoc network, the convenience of centralized architecture is lost. Centralized service directories where providers advertise their services and clients connect to look them up are not practical anymore. As in Figure 4.1, service advertisements can remain in service registries long after their providers went offline or moved away and disconnected. A lease mechanism (where a provider has to periodically renew the lease of its advertisement or the registry will erase its ad upon expiration) can help collect zombie ads but still does not solve the problem completely.

Another possible scenario is that a client and a service provider that could help the client cannot begin their interaction because they need a service registry to introduce them one to the other (Figure 4.2).

Architectures based on centralized lookup directories are no longer suitable. Mobile ad hoc computing cannot rely on any fixed infrastructure. The interactions among devices are peer-to-peer and entail no external infrastructure support. Since nodes are mobile, the network topology may change rapidly and unpredictably over time. The network is decentralized and all activities, including topology discovery and message delivery must be carried out by the nodes themselves. The network structure at any moment depends on the available direct connectivity between mobile devices and

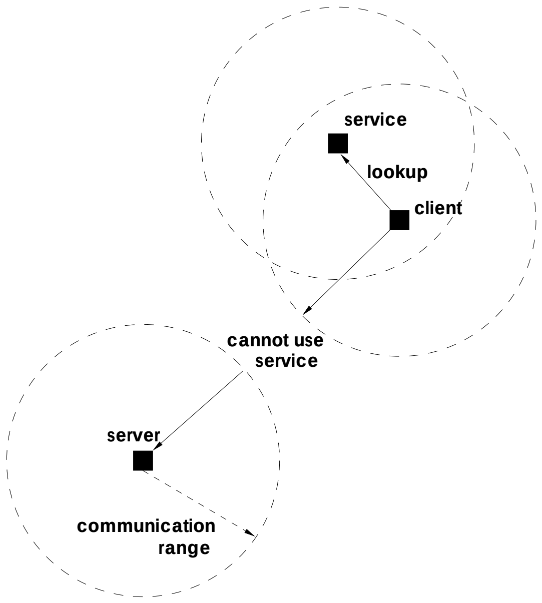


Figure 4.1: A service advertisement is discovered but the service responsible is no longer available.

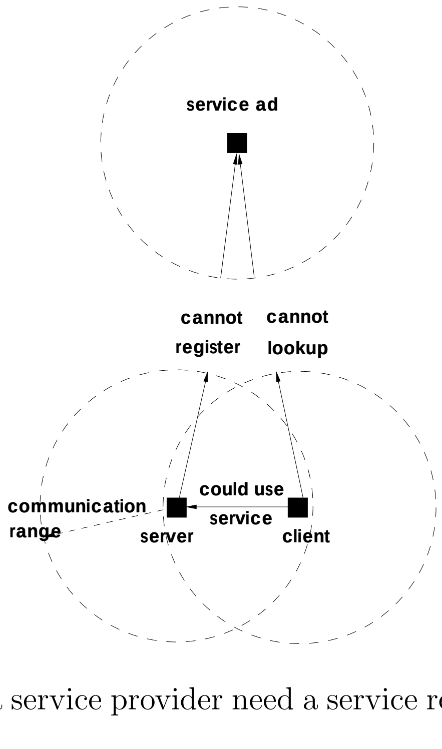


Figure 4.2: A client and a service provider need a service registry to establish contact.

thus all advertisement, discovery, and utilization scenarios have to be reconsidered to account for the characteristics of this dynamic environment.

4.3 Runtime Environment

My model for SOC in ad hoc networks is inspired by Jini. In my architecture, a service consists of some application running on a provider host and a service proxy that the provider advertises. Interested clients may retrieve the service proxy and use it locally. Next, I present a description of the software infrastructure required for a SOC platform in ad hoc networks.

4.3.1 Service Repositories

Centralized service directories are suitable for wired networks where reliable, permanent connectivity ensures prompt access to dedicated directory hosts. However, the dynamism of ad hoc networks makes centralized structures ineffective. Hence, my design entails distributed service directories. Each host maintains a local service directory. Hosts within communication range automatically share these directories to form a federated service directory. A client query for services spans the entire federated service directory. The content of the federated service directory is updated atomically with any change in the configuration of the ad hoc network. The content of the federated directory thus reflects any change in connectivity and real service availability (i.e., there are no orphan advertisements).

Such a directory is implemented as a tuple space. Each host has its local service directory in the form of a local tuple space. When hosts come within communication range, they share their local directories and clients running on these hosts gain access to the federated service directory. This transient sharing of tuple spaces assures a consistent view and access to tuple space contents such that scenarios like those depicted in figures 4.1 and 4.2 are no longer possible.

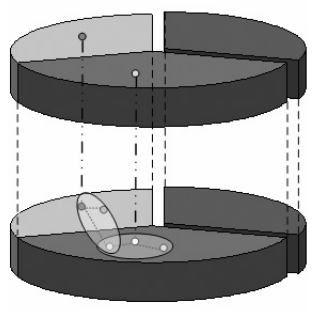


Figure 4.3: Transient sharing of tuple spaces with service advertisements and binary code.

The architecture is depicted in Figure 4.3. The symmetric structure represents two layers at which the sharing takes place. The top level represents the tuple space that contains the service advertisements. As detailed in Subsection 4.3.2, these advertisements are tuples. The top level tuple space (service directory) is visible to users, as it represents the marketplace where clients and providers meet. The lower level is hidden from the application programmers. The lower layer contains the binary code associated with the serialized objects published in the top layer tuple spaces. The bottom layer structurally mirrors the top layer. For each local tuple space at the top layer, where tuples can be published, there's a hidden local tuple space at the bottom layer, which handles the code management for the tuples in the top layer tuple spaces. Each object published in serialized form has to be accompanied by its binary code, in case this code is not available on the machine where the object will be downloaded and used. Because the binary code management is decoupled from the application semantics, the management takes place in the lower level federated tuple space, completely hidden from the user.

4.3.2 Service Advertisement

An advertisement has three parts: the proxy object, the descriptive profile of the service being offered, and the binary for the proxy object (including the class closure). The proxy is the serialized form of the object that becomes the local handle to the service on the client host (similar to a remote control handled by the user to manipulate the TV set). The descriptive profile is a set of attribute-value pairs that describe how well a service can perform a task. For example, for a service that offers access to a printer, a possible attribute-value pair could be resolution:300dpi. The binary is the machine code required to execute the proxy class.

To advertise a service, the service provider passes the proxy object and the attributes to the system. The system automatically scans the proxy object and assembles a list of all the classes that are in the closure of the proxy's class (classes the proxy instantiates as local object variables that are therefore dependencies of the proxy object). It then packages the serialized form of the proxy and the descriptive profile in a service advertisement tuple and places it in the local service directory. The dependencies are placed directly into the code repository (which is also modeled as a tuple space, shown as the lower level pie chart in Figure 4.3) by the system. An argument can be made for packaging the proxy code as well as its dependencies in a single executable archive, e.g., using JAR files in Java. However, in the interest of flexibility, I did not want to restrict a client to obtaining the dependencies of a proxy solely from the advertiser of the proxy. Further, not using JAR files allows the client to obtain dependencies on demand. If a certain branch of execution in the proxy does not require a dependency, then the client is not required to download it, thus saving valuable resources.

4.3.3 Service Discovery

When hosts are connected, the service directory is federated across all connected hosts. Hence, any other agent resident on a host that is connected can view the service advertisement. However, simply viewing the advertisement is not enough. A client must be able to request services that match its criteria. A client requests a service by first formulating a template that describes the kind of service desired. A service request template has two fields: the interface of the service desired, and a set of minimum attributes that the service is required to meet. The tuple matching mechanism compares service advertisement tuples in the service directory with the provided service request template. If a match is found, a copy of the service advertisement tuple that generated the match is sent back to the requesting client.

It should be noted that if more than one match is generated, then the system non-deterministically selects one and returns it to the client. Optionally, the client can request to have all the matches returned and do the filtering itself.

Upon receipt of the service advertisement, the client extracts the name of the service's proxy class from the advertisement and tries to instantiate the proxy locally. If the binary code for the proxy is not present on the client the instantiation step raises an exception. The system catches the exception and launches a discovery protocol to find the required binary code. This process is identical to the process used to discover the service advertisement except that the search is performed on the code directory. Once the binary code has been discovered and retrieved by the client, it is installed on the client.

The installation is light-weight, i.e., the proxy is not written to permanent storage but is simply kept in memory. The reason I made the decision to not write the object to persistent storage is because I regard services as software resources that are used when and where needed and then discarded once their usefulness is at an end. Hence, saving them on a hard disk or flash memory is not necessary, i.e., it would be the same as always carrying all the needed applications on the portable device. I rely on the operating system (Java Virtual Machine, actually) to page out the binaries if available memory is low.

Discovering Dependencies. Once the proxy has been installed on the client, it is loaded into the runtime environment. At this time, the binary code for the proxy is parsed to determine if any additional code (dependencies) are required. If any dependencies are required, the system launches the discovery protocol in the code repository to discover the binary code for the dependencies and install them on the client machine. This process is identical to the one used for discovering the binary code for the proxy as described before.

By design, the architecture for discovering dependencies also supports service composition, since one proxy can have among its dependencies another proxy, representing another self sufficient service. Figure 4.4 illustrates this feature. Each slice in the pie chart represents the code repository local to each host, and the entire pie represents the federated code repository. P_A^1 is the proxy of a service advertised by a service provider on host A depending on D_A^1 and P_B^2. D_A^1 is a dependency that the P_A^1 proxy needs and is advertised by the service provider on host A. P_B^2 is a stand-alone service which can be discovered and used independently by a client but, from P_A^1's perspective, it is just another dependency which is treated in a similar manner to D_A^1.

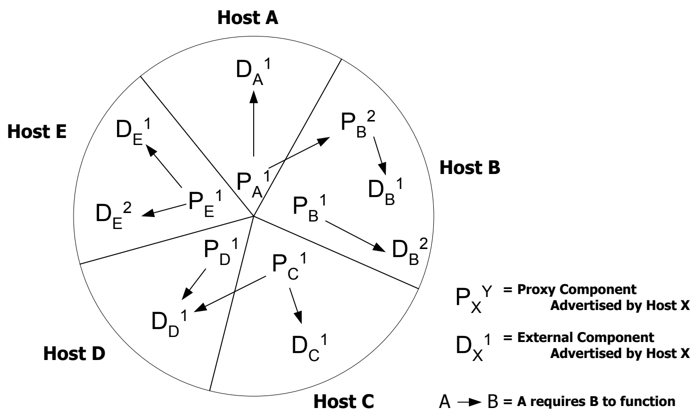

Host A
Host B
Host C
Host D
Host E
D_A^1
P_A^1
P_B^2
D_B^1
P_B^1
D_B^2
P_C^1
D_C^1
P_D^1
D_D^1
P_E^1
D_E^1
D_E^2
P_X^Y = Proxy Component Advertised by Host X
D_X^1 = External Component Advertised by Host X
A → B = A requires B to function

Figure 4.4: Federated code repository - proxies and their dependencies.

Once the installation has been completed, the client can interact with the proxy as if it were a local object (recall that I assume that the client knows the interface of the proxy). The proxy accepts calls from the client and services them locally or delegates them to its parent service on a remote host.

4.3.4 Service Utilization

Once the proxy and its dependencies have been fetched and installed on the client, the proxy can begin executing. At this stage, the client interacts with the proxy as if it were the service itself. The client can call methods on the proxy which either resolve the request locally or tunnel the request to the server on which its parent service resides. While most of the time the proxy will connect back to the instance of the server which published it, it is possible for one proxy to connect to another instance of the same server, running on a different host. This is particularly useful in ad hoc networks, where the client can be within proximity of different servers offering similar functionality.

The communication protocol between the proxy and a server is entirely encapsulated from the client, which is useful since ad hoc networks lack standardized application level protocols such as SOAP [122] and in the absence of encapsulation, the client would be required to be aware of multiple protocols. In my approach, the client is only required to know the interface offered by the proxy (i.e., know how to manipulate the proxy). This is reasonable since the client specifies the interface the proxy must implement in the request.

Though fully encapsulated, the proxy-server protocol needs to address the issue of temporary disconnections which can be caused by the client and server hosts moving beyond communication range or by having the proxy reconnect to a different server. In such cases, the client will only need to wait slightly longer for the result of a method call to be returned, which is equivalent to a simple method call that takes a long time to complete. The proxy object also needs to use a timer to avoid infinite blocking during a method call on the proxy. When the time expires, the proxy searches for another, similar, server. If this is not found within a specified period of time, the system raises an exception. Figure 4.5 illustrates the phases of the proxy's life cycle described thus far.

4.3.5 Service Upgrade

Software upgrades due to bug fixes or to provide enhanced functionality are commonplace. Since a service is at its core a piece of software, it is possible that upgrades may be required. One problem with the upgrade of a service software is that it could make the proxy incompatible. Hence, the proxies may need to be upgraded at the same time as the service itself. Performing such upgrades introduces several technical challenges, including (1) ensuring that the upgrade takes place in a manner transparent to the client while minimizing the downtime of the proxy and (2) ensuring that when the server side software is upgraded, in-progress requests from proxies that have yet to be upgraded can still be handled (since it is unreasonable to expect that the server and all its proxies can be upgraded in a single atomic step).

In this section I present a lightweight service update mechanism that can dynamically upgrade the server as well as its proxies on client hosts. Transparency is achieved by employing a dynamically generated facade to hold calls from the client application temporarily while the old version of the proxy object is swapped for the new one. Orphan calls are avoided due to tuple space–based communication which can hold communications for short periods of time without losing them (similar to when there is a temporary disconnection between client and service provider hosts). These calls are picked up by the new version of the server and, since I require newer versions of the server to be backwards compatible, it can service the calls and return the result to the client. My approach can be divided into two distinctive parts: updating the proxy used by a client and updating the server that the proxy interacts with. When updating the proxy object, problems arise due to the fact that the client is actively using it when the server decides the proxy needs to be upgraded. I aim to swap the proxies in a manner transparent to the client. On the server side, the upgrade may also trigger the upgrade of the proxies or may not affect the proxies currently in use.

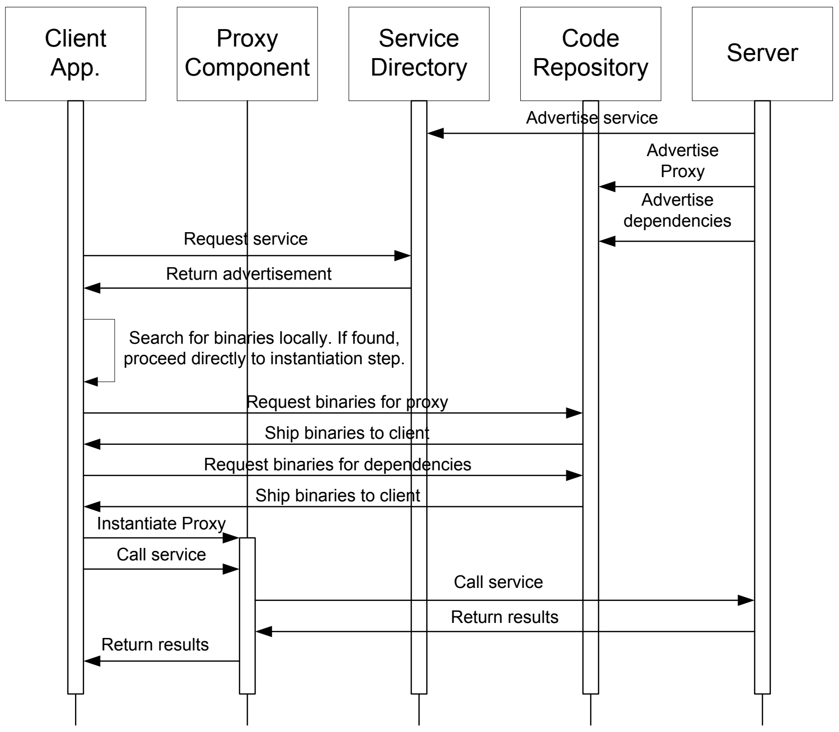


Figure 4.5: Proxy advertisement, discovery, installation, and utilization.

In the second case, the infrastructure aims to replace the server with its newer version transparently even to its own proxies.

Updating the Proxy Object

Also in the interest of transparency, I impose the constraint that the external API advertised by the proxy cannot change from one version to the next unless the new interface extends the old one and hence is backward compatible. Recall that the interface is specified by the client during the lookup process; since I upgrade the proxy without notifying the client, the new proxy is required to provide the same interface (or a subclass) to ensure that the client remains oblivious to the change. Also, if the client is using the proxy directly, the upgrade cannot be transparent since during the swap the reference to the proxy will not be valid. I solved the problem by adding a layer of indirection between the client and the proxy. Using a combination of the facade [103] and interceptor [102] design patterns, I developed an intermediary wrapper layer that isolates the client from the proxy and handles the proxy upgrade in a manner that is transparent to the client. This layer is generated automatically when the service publishes its proxy object. When the client searches for the proxy object, it receives and installs the proxy as well as the wrapper. The functionality this wrapper provides is essentially to decouple the client from the proxy, to forward the client's calls to the proxy, to monitor the server's decision to upgrade the proxy, and to manage the proxy upgrade process. An overview of the architecture is shown in Figure 4.6.

During the normal mode of operation, the wrapper is a simple pass through for calls from client to proxy and for results from proxy to client. In parallel, the wrapper monitors the service advertisement in the directory service. If an upgrade is initiated on the server side, the old advertisement is removed and replaced with one containing the new version of the proxy. The wrapper reacts to the replacement of the advertisements and retrieves the new proxy. During the retrieval of the new proxy the wrapper continues to function as a pass through. Once the wrapper has retrieved the new proxy, it requests lock on the proxy from the synchronization logic module (which ensures consistency during the updating process). The synchronization logic ensures that the proxies are swapped when there is no activity from the client and no remote

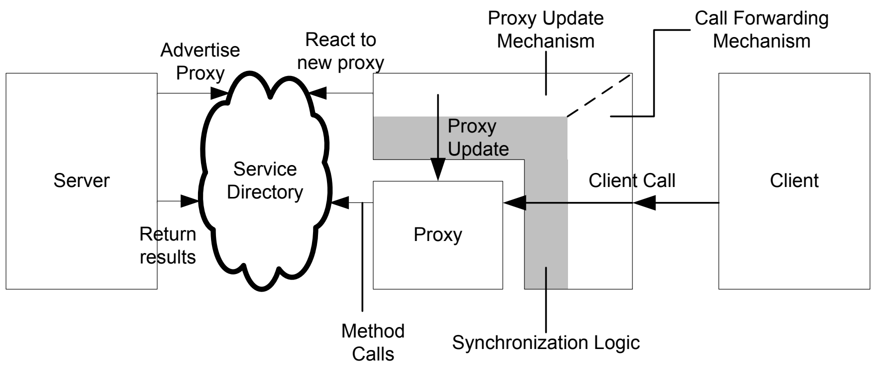


Figure 4.6: Architecture supporting run-time service upgrades.

execution of some method in progress. A method call acquires a lock which guards the exclusive access to the proxy and will not release it until the result is returned from the proxy. During this time, even if the proxy has already been retrieved and is available on the client host, the swap cannot proceed. Note that there can be at most one call on hold per proxy. This is because there is only one client trying to access any given instance of the proxy. A client cannot initiate a second call before the previous one returns. I ensure that the wrapper does not return the flow of control back to the client, and keeps the client blocked until the call returns by forcing a synchronous behavior on the client side, even though the wrapper forwards the client's call to the proxy. This synchronization mechanism also ensures that it is not possible for a client to send out a call using the old proxy and receive the answer from the new proxy. Symmetrically, if a proxy upgrade is in progress, a method call cannot complete and will be blocked by the same lock when it reaches the proxy. Once the swap is finished, the lock is released and the method call is forwarded to the newly installed proxy.

Updating the Server

Upgrades on the server side assume that the server needs to go off line temporarily and be restarted. Before the server goes off line, it removes the service advertisement it registered from the service directory. Clients interested in the functionality offered will not be able to discover the service during this stage, even though the server may be running. At this stage of the process, the server ignores all incoming calls from clients. At the same time, it continues to process the calls in progress, which were generated by the old version of the proxy. Not performing this step can delay the completion of the in-progress calls indefinitely (as the set of in-progress calls can evolve over time, e.g., if new calls come in from clients before the server finishes the calls it is currently working on) and thus defer the upgrade indefinitely. Once the response to the last in-progress call from the old version of the proxy is serviced, the server can go off line.

When the new server starts up, it advertises itself in the service directory and it makes itself available to clients. The advertisement publishes the new proxy (if needed). The proxy wrappers on clients which have the old version of the proxy react to the new advertisement available in the service repository and upgrade the proxy if necessary. It is important to note that the server is required to preserve backward compatibility with previous versions of proxies. The reason for this is twofold. In the first case, the old server may have ignored some calls from clients during its shutdown process. The new server, when it comes up, finds these calls waiting to be addressed. Until these calls are addressed, the wrapper on the client side keeps the client application blocked, waiting for the method to return. While the wrapper may react to the presence of a new proxy in the new server's ad in the service directory, the most the wrapper can do is fetch the new proxy on the client host and then block again, waiting for the call to return. The server therefore has to be able to execute this call, which necessitates that the server be able to read and understand the request, even if it was formulated by an older version of the proxy.

Figure 4.7 shows the sequence of interactions between different parts of the system. In the initial state, the client has already discovered the service and installed its proxy. The first round trip of calls shows a complete path of interactions starting with the client issuing a call, intercepted by the wrapper which obtains the lock from the synchronization logic and then forwards the call to the proxy which sends it to its server. The call return follows the same way back and releases the lock as it goes through the wrapper to the client.

The second call (shown in Figure 4.7 below the dashed line) occurs at the same time that the server is upgraded. In the scenario depicted, the proxy update request arrives at the wrapper after the method call from the client already went through,

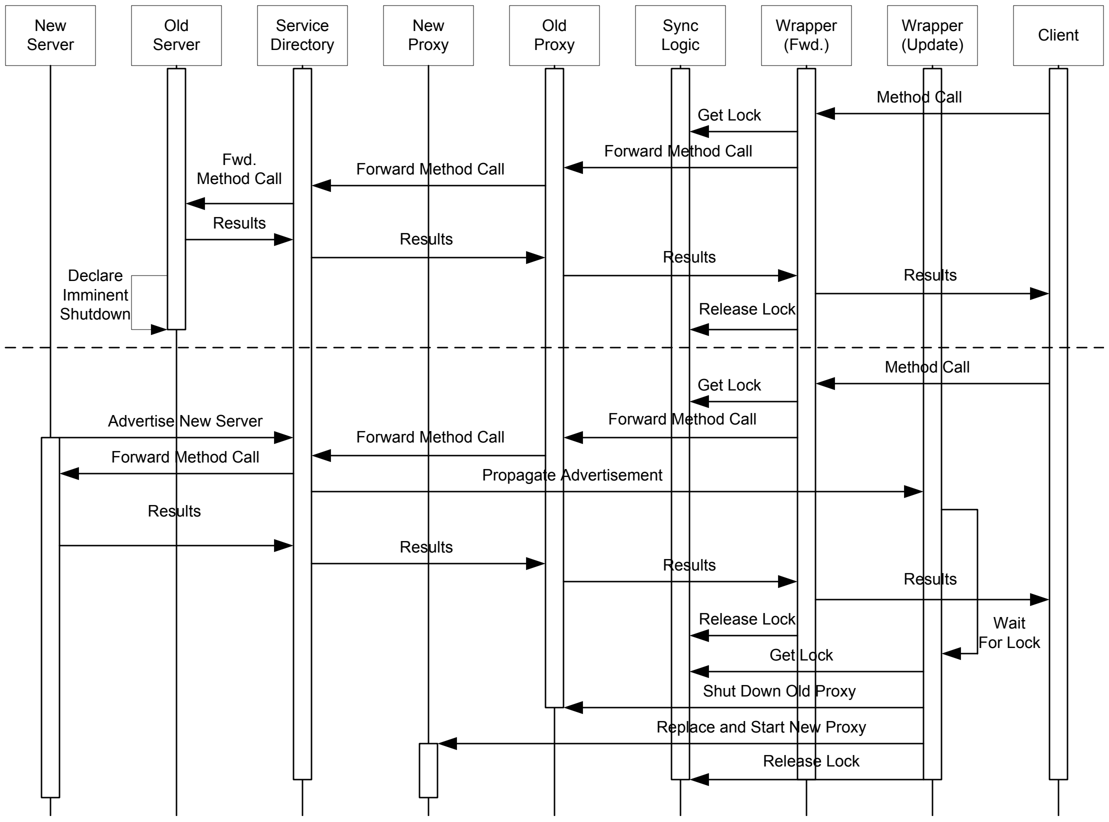


Figure 4.7: Interactions between components of the service upgrade mechanism.

towards the server. The wrapper therefore can only discover the new proxy and fetch it locally, but has to wait for the method to return before it can proceed with the proxy replacement process. Once the result of the call is returned and the lock is released, the wrapper obtains the lock and swaps in the new proxy. Once the new proxy is in place, the wrapper releases the lock which guarded the proxy replacement and once again returns to the default operating mode of simply forwarding client calls and results.

4.4 Follow-me Sessions

4.4.1 Services In the Presence of Mobility

In the dynamic environment of ad hoc networks, an interaction (defined as a bounded sequence of message exchanges) between two hosts may need more time to complete than the interval of connectivity between them. For reliable service provision, it is desirable that an interaction between the client and the service provider, once begun, reaches completion. In MANETs, physical movement of hosts is independent of application semantics and I consider it undesirable for the application to impose mobility restrictions. My solution is to have the client partially complete the task with the help of some host, pause its work, and resume it on another host. This stretches the processing of a task over multiple hosts as they fall within the client host's communication range, each contributing pieces of computation towards finishing the entire task.

The additional code required to support this kind of *pause-transfer-resume* computing can make programming for mobile environments prohibitively complex. It is in the interest of keeping programmer effort low that I introduce a layer of abstraction called a ***follow-me session*** (FMS), defined as a mechanism that preserves the sense of interaction between a client application and a service by masking the disconnections between intervals of connectivity. In essence, "follow-me" sessions offer, within limits, continuity of service provision despite changes in connectivity.

Observe that traditionally, a session is a lasting connection between a client and a server host during which several packets of data are exchanged. Usually, a session is not interrupted by a disconnection. However, in FMSs, the lasting connection is between a client and a service as explained above, rather than a specific service process or volunteer host. Since hosts can move in and out of communication range due to physical mobility, the FMS must have the capability to span multiple connectivity intervals without severely disrupting the client-service interaction. I also impose the restriction that all FMS functionality be built into the middleware layer with the purpose of abstracting complexity associated with a decoupled and transient computing environment from the application programmer. Realization of such functionality requires that the middleware be able to move the data and/or computation state of the client-service interaction from one host to another.

This section describes a set of components that support the delivery of "follow-me" sessions by using one or a combination of the following three strategies: 1) switching on the fly to an alternate provider which offers a similar service on the fly, continuing rather than restarting the execution on the new provider, 2) allowing the client to temporarily disconnect from the service provider while the provider continues processing, or 3) having the client take back the partial results temporarily until a suitable alternate service provider can be found. The use of these mechanisms ensures that the process offering a service can migrate from host to host, so that it is always on a host that is in proximity to the client as the host on which the client resides moves through space.

A summary of the basic mechanisms needed to deliver FMSs are presented in the scenario depicted in Figure 4.8.

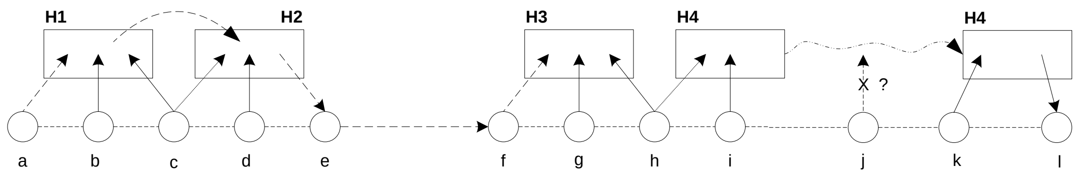


Figure 4.8: Follow-me session components.

The circle represents the client application's host. The horizontal dashed line represents the client hosts's trajectory as it moves through space from location a to l.

As the client approaches host H_1, it pushes the implementation of a service it needs onto H_1, while holding on to the proxy object which will be used to manipulate this server remotely. The dashed arrows indicate how the service moves; a variation of this scenario might allow the client to simply discover the service already running on H_1. The solid arrows denote the client's interactions with the server dedicated to carrying out the task, and are not related to session management. When the client is in position c, the service migrates from H_1 to H_2 since the client will soon lose connectivity to H_1 but will remain connected to H_2. The transfer from H_1 directly to H_2 is possible because H_1 and H_2 are within communication range of each other. When the client reaches location e, there is no host to which the service could jump and therefore the client will "take back" the service (computation state and binary code if the interaction started with the client *discovering* the service; if the client *pushed* the service, the code transfer is not necessary as the client has it) and partial results (the dashed arrow from H_2 to the client). The client cannot run the service on its own host because the resources available on that host do not allow it and therefore the client will only transport the service until a new host is found. At location f the client pushes the server onto H_3 where the client will manipulate it remotely until the client reaches location h. The disconnection from H_3 becomes imminent but H_4 advertises the same service. The client will continue its job using the service advertised by H_4, since it is much cheaper to migrate the computation state than the entire service process, and have the new server *resume* from a predefined intermediary progress point. While working with H_4, the server's host moves out of range but only *temporarily* and the client *knows it.* If the application allows it, H_4 can continue to work on the task while the client crosses landmark j and the two are disconnected. At k the client reconnects with H_4. When the client interacts with H_4 at location l the task is completed, and the results are shipped back to the client.

4.4.2 Building Blocks

Before I address the strategies listed above, I will first introduce a few building blocks that are necessary in the delivery of the FMS middleware.

Checkpointing

Checkpointing is a mechanism introduced to improve the fault tolerance of software (an important reference on checkpointing related resources is [2]). It entails saving the current state of a program and its data, including intermediate results to non-volatile storage, so that if interrupted the program can be restarted at the last checkpoint. If a program run fails because of some event beyond the program's control (e.g. hardware or operating system failure) then the processor time invested before the checkpoint will not have been wasted. I address failures triggered by hosts' disconnection while a task is processed in a distributed manner.

If a client-service process interaction does not run to completion during a window of connectivity, the task has to be completed during a subsequent window of connectivity or in collaboration with another volunteer host. Using checkpoints, the interaction can resume from an intermediary point (i.e., from the last checkpoint the execution flow went through), and does not have to be restarted from the beginning.

At each checkpoint I record the state of the thread, including its program counter and call stack, which are known as the execution state. Note that for the Java implementation, I had to introduce an artificial program counter because the JVM does not allow outside access to its program counter. This artificial program counter is updated at each checkpoint (it actually tracks which checkpoint was cleared last). The value of the artificial program counter is transferred to the destination host and is used to resume the execution of the server process. As in any native programming language, every time a method is invoked, a new frame (or entry) is created and pushed to the top of the call stack. The frame, which contains the parameters of the method call and local (temporary) variables, is destroyed when the method returns. The data state is composed of the values of instance variables (live objects). These are easier to transfer as serializable objects (I do require that these objects are all serializable).

When the service process migrates from one volunteer host to another, the state recorded at the last checkpoint visited is transferred. Any further computing is lost as the session and state information resumes from that last checkpoint, e.g., if the checkpoint is placed just before a `for` loop, the loop will be started from the beginning when the execution is resumed at the destination host. If the checkpoint is added immediately inside the loop, the execution resumes with the last iteration of the loop executed on the initial volunteer host.

I approached these problems by choosing to instrument the bytecode of applications rather than trying to manipulate them at runtime. This instrumentation process adds bytecode to applications to add support for strong code migration, including code to work around these technical limitations.

The programmer defines checkpoints by calling the addCheckpoint() method. While appearing to the programmer to be an ordinary method call, it serves as a marker in the bytecode to indicate the location of checkpoints. After compiling the Java source code, the resulting bytecode is passed into the instrumenter, which converts the code so that it is capable of being strongly migrated. This instrumenter is implemented using the BCEL class file manipulation library [24]. Details about when happens inside a checkpoint are available in the next subsection.

State Saving and Restoring

Checkpoints are places where instantaneous snapshots of the execution are taken. Such a snapshot represents the *computational state* of the application when the execution goes through the checkpoint. These places are potential interruption places, the execution resuming from the last checkpoint cleared (the last computational state known). The computational state can be saved and transferred such that the execution can resume on another machine, or can be resumed after a crash.

To do this, the instrumenter first collects a list of all the local variables in the current method. It then adds a field for each of these local variables; these fields are used later to store the state of the local variables. The instrumenter also inserts a field to store an artificial program counter. The instrumenter then searches for all calls to addCheckpoint(). At each checkpoint, the instrumenter inserts code to check a do_pause field, which indicates whether or not the application thread is being paused so it can be migrated. If this field is set, then the method immediately returns. If it is not, then the method copies all of the in-scope local variables to the fields described above and then sets the artificial program counter to some unique value. Finally, the instrumenter removes the call to addCheckpoint(), since it only serves to mark the bytecode. The instrumenter also appends code at the end of the method to copy these fields back into the corresponding local variables and jump to the checkpoint; these "restoration points" provide a place for the thread to restore its state and return to the last checkpoint it passed before being migrated. The bytecode instrumenter then adds code to the beginning of the method to see if the paused field is set. If this is the case, then the application jumps to the appropriate restoration point based on the contents of the artificial program counter field. This has the indirect effect of restoring the thread's local variables and the JVM program counter.

Code Migration

An essential contributor to FMS is strong process migration. Strong migration entails data state migration, execution state migration, and binary code transfer. Data state migration transfers the state of an object's instance variables when the binary code for that object is already available on the destination machine. Execution state migration entails transferring the (artificial) program counter and the call stack content. Binary code transfer downloads the binary code of the process on the target machine, where the loader can find it and load it into memory.

During migration, the serialization process wraps only the content of an object (values of member variables) and not the bytecode from which the object was created. This includes all objects inside the initial object. To transfer all the binary code needed on the remote machine, I use a combination of reflection and exceptions to build the closure of the classes that need to be migrated. I developed a custom class loader [52] that is able to capture on the destination host all exceptions caused by missing bytecode. After catching such an exception, I use reflection to build the list of dependencies (instances of other classes the object that triggered the exceptions) and send it to the the source volunteer host, requesting the missing bytecode. On the source volunteer host, I inspect the advertised proxy using reflection when I publish the service advertisement. I publish the code of all classes the proxy instantiates and which are not part of the standard JDK or of the middleware (i.e., they may not be available on a client machine). This code is downloaded, installed and loaded by the custom class loader on the destination host.

Location Agnostic Protocols

A key element in the delivery of the context-aware session management is the communication protocol between the client (proxy) and the server applications. All these interactions logically belong to the same FMS. Therefore, the client side should not be impacted by disconnection from service process, service process migration, or rebinding to a new service process, as long as the session is in progress, i.e., it is preferable for changes to be transparent. While the disconnection cannot be completely hidden from the client application (when the client expects an immediate answer from a service process on a disconnected volunteer host, there is nothing the middleware can do to help), it can be masked as a delayed response until the two hosts in question reconnect.

To deliver this, I employ location-agnostic communication protocols. The client obtains a unique session identifier which is used to stamp all messages exchanged in a working session. The client only knows that at the other end there is a service process handling the requests belonging to this session. Similarly, the service process knows to pick up and serve only messages marked with the appropriate session ID. Hence, messages do not have to be addressed to a specific host or network location. If a process migrates from one host to another, it will receive a message intended for it if it uses the same criteria (in terms of content of the message). Similarly, if a client uses context-sensitive binding (described in Section 4.4.3 to connect to a similar instance of a service, it is not unreasonable to assume that it will have the same criteria as the original instance of the service that was used, allowing messages to be propagated automatically to the new instance of the service. This leads to a communication protocol based on the content of the messages rather than the explicit destination stamped on each message (content-based communication rather than point-to-point communication).

This concept is illustrated in Figure 4.9. The producer (lower-left corner) publishes data in tuples with the first field identifying the session to which they belong, and

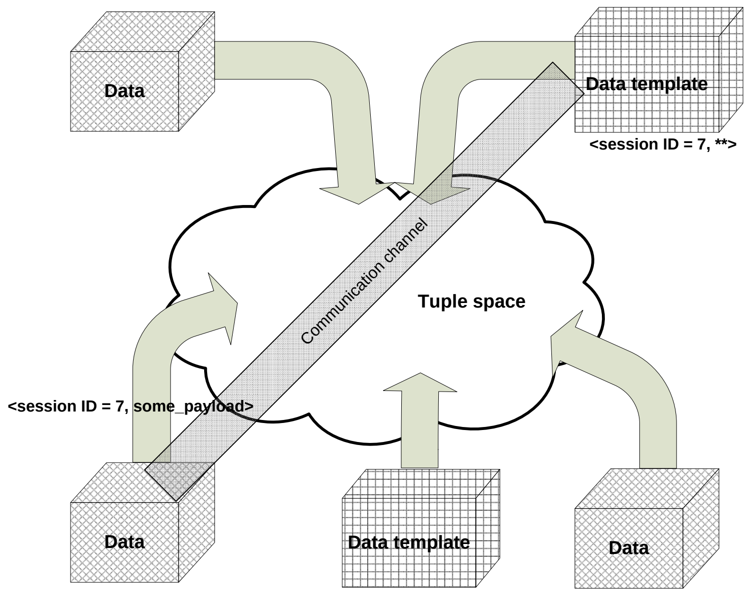


Figure 4.9: Content-based communication supported by tuple spaces helps deliver location agnostic protocols.

some payload in the following field(s). A consumer (upper-right corner), reads data belonging to the session identified by session ID = 7. Tuple space communication is a natural environment for this type of interaction. Note that any other accessor of the tuple space could pick up this message if interested in session 7. This allows the client to change providers and not be impacted other than by the overhead induced by the communication channel reconfiguration.

4.4.3 Mechanisms Employed for Continuous Service Provision

In this section I will present multiple mechanisms that can be employed to provide continuous service to a client, despite the connectivity changes induced by the tost mobility. Since not all mechanisms are always usable or some may yield better results than others, a decision algorithm that helps choose the best option at a certain moment is presented after the above mentioned mechanisms are described.

Context Sensitive Binding

Having presented the building elements needed for dynamically changing the service provider, context-sensitive binding (CSB) is the first mechanism that uses them for achieving the functionality of FMSs. CSB is a mechanism that decouples the interface of a service from its realization. The realization of the interface is provided by a changing set of service processes such that the best service process provides the realization at any given time, where best is defined (for now) as the service process on the volunteer host that is likely to remain connected to the client host for the longest duration. This is a common sense metric that yields good results most of the time, but is not the only metric possible and is not always the best. For example, switching to a provider to which the client stays connected for a shorter period of time, but which has a much faster progress rate might prove beneficial to the client.

Switching between service processes and volunteer hosts occurs dynamically and transparently in a context-sensitive manner. Context-sensitive binding offers several features:

- *Policy based selection:* the set of qualifying volunteer hosts is chosen based on policies, e.g., "ensure that the client is always connected to a volunteer host running the required service that is not more than 25m away." When choosing a volunteer host, the policy is the first filter that is applied after choosing a set of service providers that already offer the needed functionality (i.e., this step executes after the service query/discovery step) or that offer to run code provided by the client.

- *Metric based evaluation:* once the set of candidates is determined, the best volunteer host is chosen according to client-specified metrics (e.g., the client works with the volunteer host that will stay connected to it for the longest time).

- *Transparent binding maintenance:* context-sensitive binding provides for dynamic switching between volunteer hosts to provide the best available service at any given time. Except for a small interval of time when the mechanism is switching between volunteer hosts, continuous binding between the client and the server is maintained. However, the switching of the volunteer host is masked from the client by the middleware, so from the client's perspective, the binding appears continuous for the time interval it uses the service.

While CSB can be guided by any kind of policy, in the interest of a targeted and complete treatment, I focus on spatiotemporal policies, as these are among the most relevant in MANETs. CSB is responsible for changing the volunteer hosts and service providers such that the momentary provider is the best available given a context. In MANETs, the primary reason for context changes is physical mobility. As hosts move in space, they encounter a changing set of hosts over a period of time. Once an initial volunteer host is chosen, the client begins its interactions with the service process on that volunteer host.

Periodically, a snapshot of the interaction is taken (using the checkpointing mechanism described earlier). In parallel, the CSB mechanism monitors the context to check if the current volunteer host is still the preferred one. Changes of interest in the context are changes in the knowledge base managed by the current host (e.g., the current host learns more about its own future evolution in space, or learns more about some other host's motion profile). Unpredicted changes in connectivity are also of interest (these can only be unpredicted encounters with some mobile host whose motion profile the current host doesn't know; all other (dis)connections are anticipated and accounted for in the decisions made about the FMS thus far). These changes can lead to a re-evaluation of the future of the follow-me session.

Depending on whether or not the migration includes execution state migration, process migration can be split in two types: weak and strong migration.

Weak migration requires that the process can run, but not resume, on a destination host. This involves making the code available on the destination machine and restarting the process from the beginning, losing any progress the process may have made before migration. The execution state is not transferred during weak migration. Examples of weak migration are [9], [60]. In some cases, some initialization data can be transferred along with the process but that does not account for complete execution state transfer. The process is restarted, except that the memory is initialized to contain potential partial results (e.g., μCode [94]).

Strong migration [91], [43], [19] entails the migration of the execution state as well. Processes can be stopped, transferred, and resumed on a new host. To deliver the desired semantics, I developed a strong migration mechanism, implemented for Java threads. While capturing and transferring the execution state, the program counter of the Java virtual machine (JVM) and the call stack are captured and transferred in a serializable format to the new destination. Strong mobility is more powerful but it is also more expensive to deliver. It does, however, offer the pause-transfer-resume semantics required by the FMSs.

I deliver strong migration in two forms: lightweight and heavyweight. Lightweight migration entails only the transfer of state information. This is the preferred scenario, as it incurs less overhead. This fortunate scenario requires the binary code of the executing process to be available on the destination host. If this is the case, a thread is stopped in a checkpoint, the state is saved and transferred in serializable format to the target host, is restored there, and then the computation is resumed. When lightweight migration is not possible, heavyweight migrations is needed, i.e., the binary code has to be transferred as well. All mechanisms required to achieve lightweight/heavyweight migration have been presented in the previous section. Figure 4.10 illustrates the CSB idea and shows the usefulness of location-agnostic communication protocols.

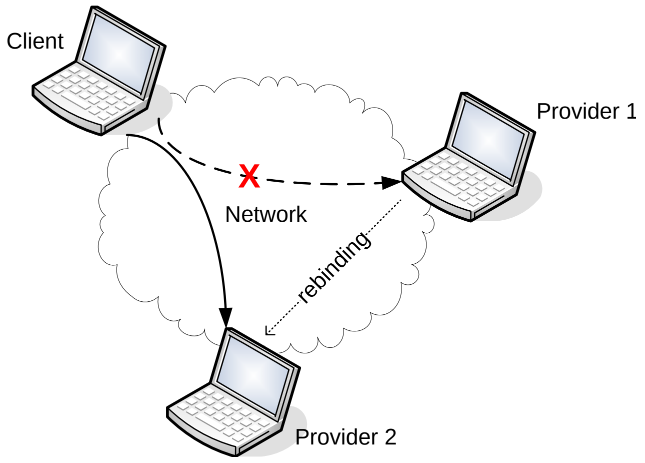


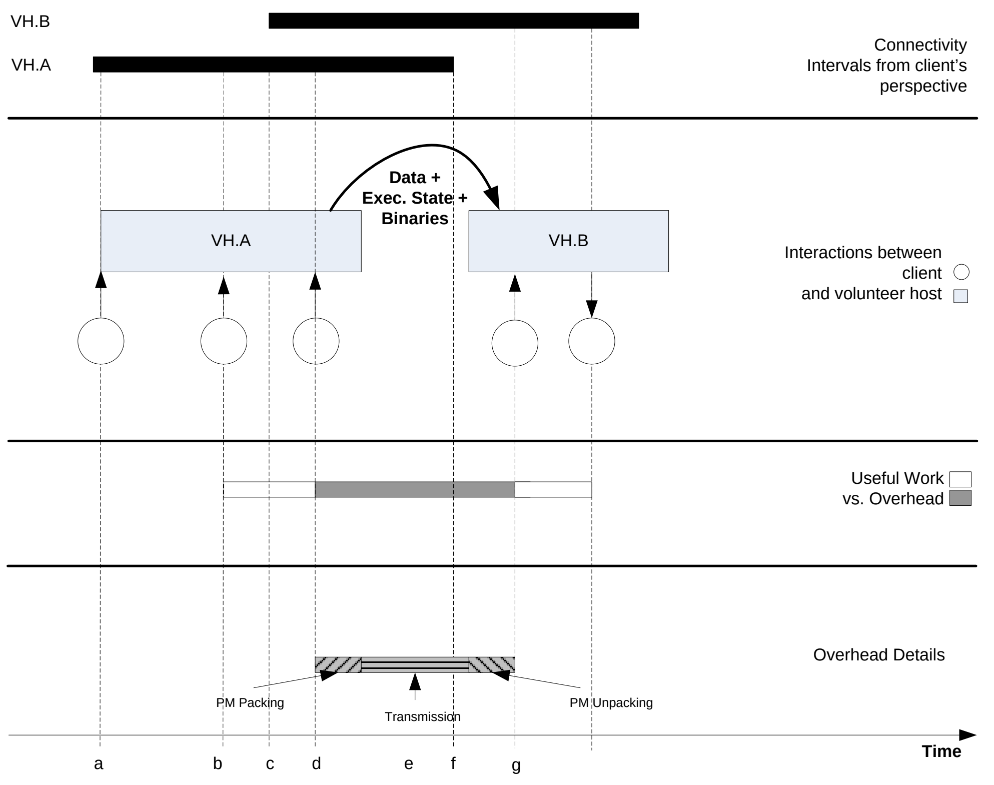
Figure 4.10: Context sensitive binding and location agnostic protocols.

To transfer the control from one host to another, whether code migration is required or not (i.e., CSB needs a heavyweight migration or a lightweight migration suffices), the application programmer creates mobile applications by extending the MobileThread class, which adds several methods and fields to Java's standard Thread class. The programmer defines checkpoints by calling the addCheckpoint() method, which serves as markers for allowed interruption points, as described before. The MobileThread class also adds another two important methods to the standard Thread class: pause() and unpause(). The pause() method sets the do_pause (also introduced above) and paused fields to true; the former tells the thread that it should stop execution as soon as it reaches the next checkpoint, and the latter tells the thread that it should restore its state when it is restarted. The unpause() method simply resets the do_pause field to false and restarts the thread; since the pause() call set the paused flag, the thread will jump to the appropriate restoration point and return to the last checkpoint passed before pausing. This way, rewritten applications can be migrated across hosts by pausing the application thread, serializing it on the original host, deserializing it on the new host, and unpausing it. This entire trick is needed because a thread cannot be safely stopped from *outside* (i.e., all Java's methods for stopping a thread without killing it are deprecated, as they might leave the JVM in an inconsistent state). The entire procedure described above simulates an external thread stop while in reality the thread is requested to stop and it then stops by itself (i.e., from *inside*). At each checkpoint the do_pause flag is examined and this flag tells the thread whether to stop or not and the thread does it itself.

The sequence of events during a process migration can be observed in Figure 4.11.

Temporary Disconnect

If the imminent disconnection between the client and the volunteer host is temporary, i.e., they will reconnect in the near future, the system can opt for a temporary disconnection rather than moving the computation to another host. Under this scheme, the middleware allows the client and the service process on the volunteer host to disconnect, but the service process continues executing for the duration of the temporary disconnection. If the task completes while the hosts are disconnected, the results are saved by the service process on the volunteer host. When the client reconnects, it can retrieve the result, if available, or continue with direct interaction.


VH.B
VH.A
Connectivity Intervals from client's perspective
Data + Exec. State + Binaries
VH.A
VH.B
Interactions between client ○ and volunteer host ▢
Useful Work ▢ vs. Overhead ■
Overhead Details
PM Packing
Transmission
PM Unpacking
a
b
c
d
e
f
g
Time

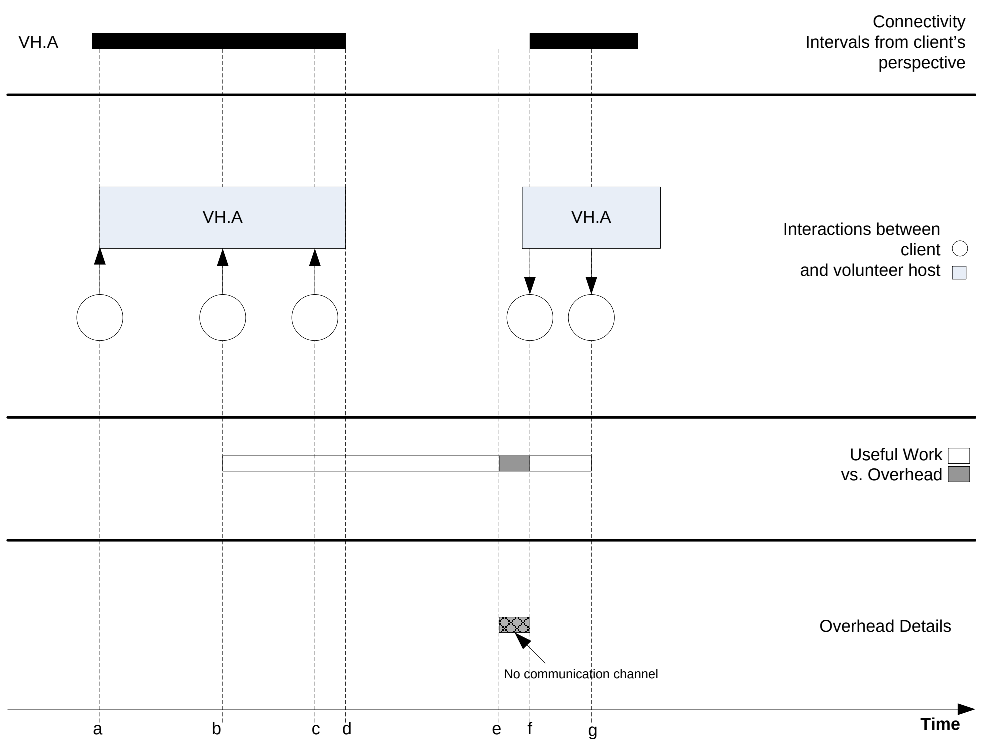
Sequence of Events

a.	Client connects to volunteer host VH.A
b.	Client starts interaction with VH.A
c.	Volunteer host VH.B moves into communication range
d.	Sensing imminent disconnection, client directs VH.A to migrate data + execution state + binaries to VH.B using strong process migration.
e.	Migration to VH.B completed (Note it takes longer than CSB). Client disconnects from VH.A. VH.B takes up processing
f.	Client begins to interact with VH.B
g.	VH.B completes task and ships results to client. Interaction ends.

Figure 4.11: Process migration.

There are two issues with this approach. The first relates to determining whether the disconnection is temporary or not. This is determined by an analysis of the motion profiles of the two hosts. Details will be covered in Section 4.4.4. The second issue relates to potential communication between the client and service process during the temporary disconnection. Clearly, no such communication can occur when the client is disconnected from the volunteer host on which the service process is resident. Hence, this option can only be used effectively for a special class of client-service process interactions which consist of a single communication to initiate the interaction and a single communication (with the result) at the end of the interaction. If additional communication is required, then the processing will block until the two hosts reconnect. While this approach is restrictive, it is attractive in that it results in lower overhead (if the client reconnects before the service process finishes the task or there is no communication required resulting in the process not blocking). If the result is held on the volunteer host or some communication is required during the disconnection however, there is an overhead from the time at which the process finishes or is interrupted, until the time the hosts reconnect.

The sequence of events during a temporary disconnection are presented in Figure 4.12.


VH.A
Connectivity Intervals from client's perspective
VH.A
VH.A
Interactions between client and volunteer host
Useful Work vs. Overhead
Overhead Details
No communication channel
a
b
c
d
e
f
g
Time

Sequence of Events

a.	Client connects to volunteer host VH.A
b.	Client starts interaction with VH.A
c.	Sensing imminent disconnection, client starts temporary disconnection protocol
d.	Client disconnects from VH.A. Processing continues
e.	Processing is complete. Results are held locally. Remaining time until reconnection is overhead
f.	Client reconnects to VH.A. VH.A begins transmitting results
g.	Transmission of results completed. Interaction ends.

Figure 4.12: Temporary disconnection.

The client could begin its interaction with the service process on volunteer host VH.A. However, when a disconnection from volunteer host VH.A is imminent, the client can switch to the service process on volunteer host VH.B. The partially computed results and other data and execution state are migrated automatically from VH.A to VH.B by the middleware.

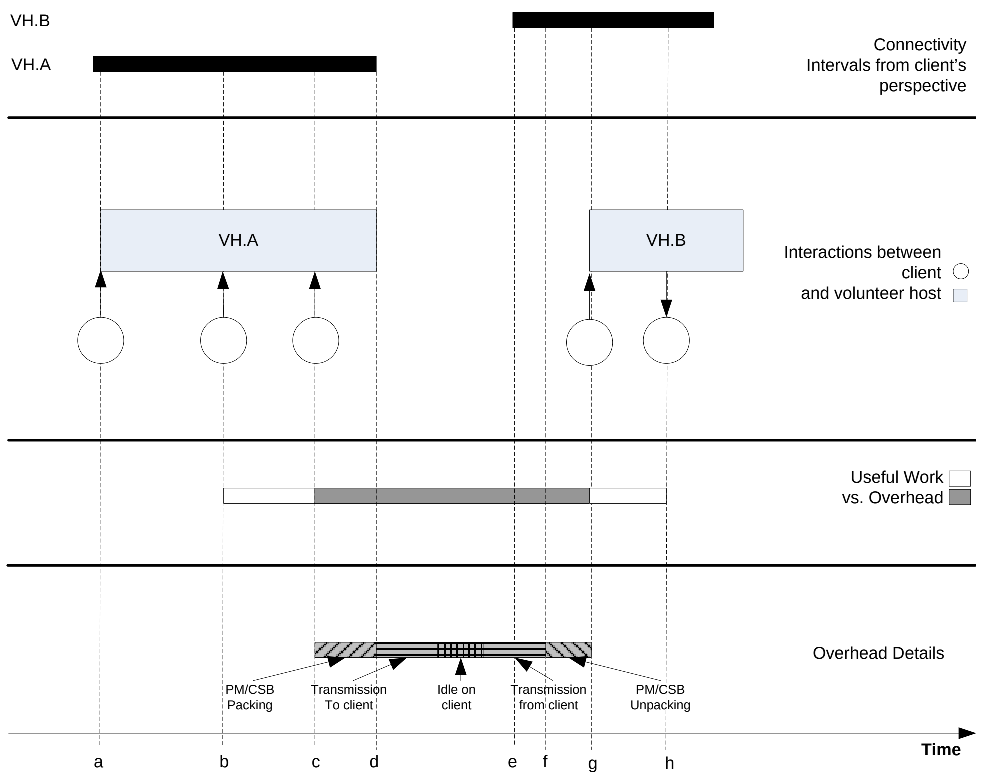
Take Back

In the case that there are no neighboring volunteer hosts at the time of disconnection, resulting in none of the above options being available, the middleware can opt to temporarily suspend the interaction between the client and service process. The data and execution state of the service process on the current volunteer host is shipped to the client and stored on it. The client in the meantime continues to search for alternate volunteer hosts. When such a volunteer host becomes available, the stored state is transferred onto that volunteer host and processing is resumed. Note that I do not consider the possibility of the client executing the code because if this were an option, the client would not have needed to use a volunteer host in the first place.

This option is the most expensive in terms of overhead because it requires two migrations: one to recover the process from the host it is executing on and another to resume it later; and it offers no progress between those migrations because processing is suspended until an alternate is found. Depending on how the interaction with the service started (i.e., the service was *discovered* by the client or *pushed* by the client), the transfer of binary code might be needed. The advantage of this option is that it is always available, i.e., it is independent of other hosts in the MANET but it is used only in the worst case scenario due to its high overhead. The sequence of events during a take back can be observed in Figure 4.13.

4.4.4 Decision Algorithm

In the face of a disconnection, depending on the configuration of the mobile hosts around the host which executes the client application, a decision has to be made as to which of the available options from the ones described above should be followed and which mobile host should be the next partner. The best choice is always desired, but this may vary depending on the particularities of the application in discussion.



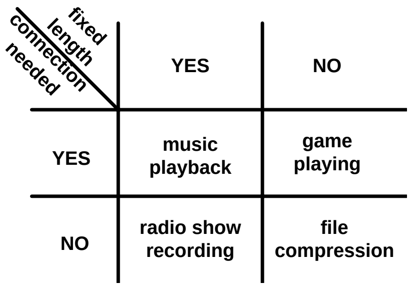
Sequence of Events

a.	Client connects to volunteer host VH.A
b.	Client starts interaction with VH.A
c.	Disconnection is imminent and no other host is in communication range. Begin migration of execution state to client.
d.	Transfer of execution state completed. Client disconnects from VH.A. Execution of task is suspended.
e.	Volunteer host VH.B moves into communication range. Client begins transferring execution state to VH.B
f.	Client finishes transmitting execution state. Execution resumes on VH.B
g.	Client interacts with VH.B.
h.	VH.B completes interaction and ships results to client. Interaction ends.

Figure 4.13: Take back.

Figure 4.14 shows four sample applications, each of them representative for its class, as defined by two parameters: fixed_length and connection_needed.

The fixed_length parameter dictates whether the application has a runtime imposed "from outside". For example, playing a song will take as long as the song is. In this case, any machine whose performance is above the minimum necessary to play the song is just as good and therefore more processor power or more memory will not do the application any good. On the other hand, compressing a file can benefit from more processor and/or memory. Similarly, some applications need the service provider to always be in touch with the proxy manipulated by the client, such as listening to the music. If the service server process has access to the music repository



connection needed \ fixed length	YES	NO
YES	music playback	game playing
NO	radio show recording	file compression

Figure 4.14: Types of applications.

but the client is disconnected and therefore cannot play the music for the human user carrying the client's machine, the service is useless. By contrast, recording a radio or TV show (e.g., TIVO) doesn't require this permanent connection. Once the service is started, it can continue without tight monitoring. I will continue the presentation assuming the situation when the task's length is not fixed and the connection is not strictly required. This is the most general case and all others are just biases of the decision mechanism and will be mentioned at the appropriate moments.

Figure 4.15 shows a sample dynamic configuration of mobile hosts and a part of a FMS spanning the $[\mathsf{t}_0, \mathsf{t}_8]$ interval.

The top half of the picture (above the time axis) shows the FMS as it happens in time. Each horizontal line represents an interval of connectivity between two hosts (underneath the time axis are listed all pairs of hosts that are in direct contact during each time interval) and how the FMS behaves during it (see the legend beneath the picture for more details).

The FMS is captured while the client C works with H_1 at t_0. At t_1 the scenario indicates it starts to migrate to H_3. From t_2 to t_6 it executes on H_3 (including $[\mathsf{t}_4, \mathsf{t}_5]$ which represents a "temporary disconnect" interval). My follow-up ends at t_8 where FMS migrated meanwhile. The question is, *"why did the FMS go to H_3 and not to H_2 which was another option at t_2?"*

Zooming in on two consecutive collaboration intervals with two different servers, the details look like those depicted in Figure 4.16. Each interaction interval is mainly formed of three sub-intervals: the setup - A zone (when the FMS comes from another

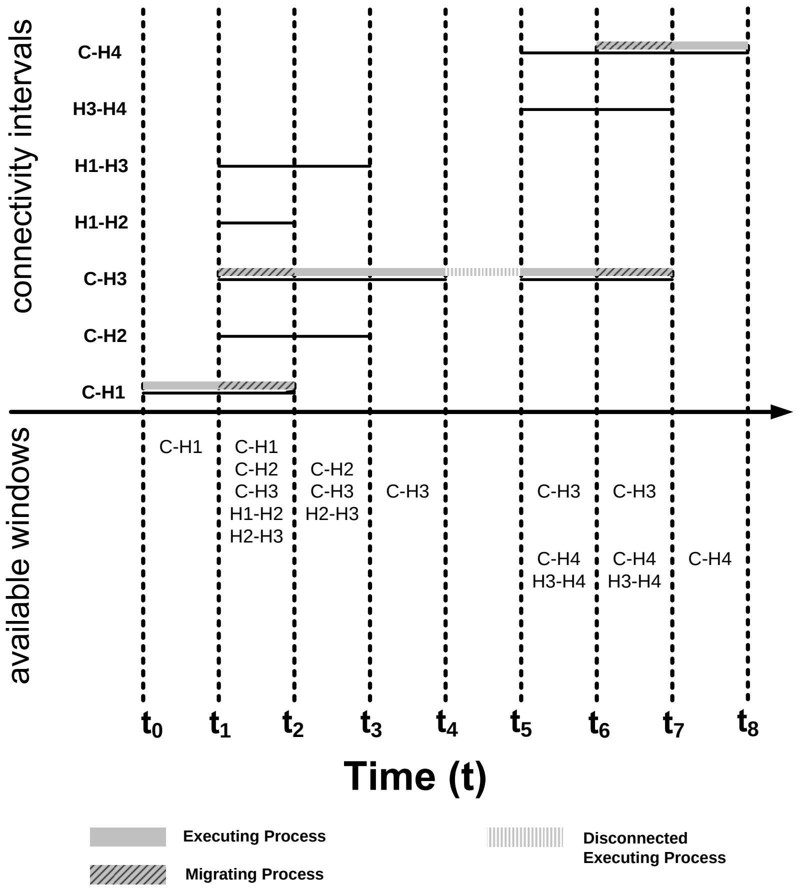


Figure 4.15: Sample FMS and mobile hosts configuration.

host and the computation is resumed), the useful work - B zone (the the service does useful work for the client), and the cleanup - C zone (the FMS prepares to move on or bring the final results back to the client). The thickness (vertical) of the bars in the picture represents the performance while the length (horizontal) represents the time needed to perform the operation.

The overhead is split in two parts: prepare to migrate (A2) /recover from migration (C1) and the transfer itself (A1/C2). During A2 (/C1), the state saving (/restoring) and serialization (deserialization) of transferables is being performed. A2 and C1 are computational efforts and therefore are performed by each machine at the same speed as the B region (giving the same performance). A1 and C2 overlap in time and have the same progress rate as they represent the transfer between the two hosts. This

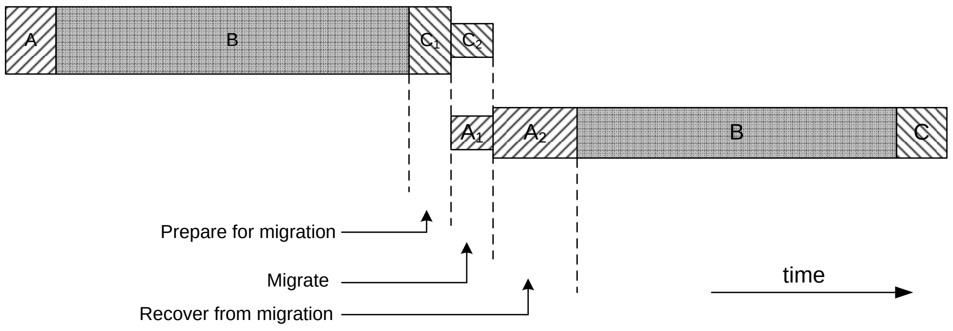


Figure 4.16: FMS transfer from one host to another.

is determined by the quantity of data to be transferred and the bandwidth available between the two hosts. It is obvious that A1 is the same size (performance-wise and duration) as the corresponding C1.

The bandwidth available between two machines can be evaluated at runtime by real time tests. The quantity of the transfer can also be evaluated at runtime, by measuring the size of the byte array to be transferred. This applies only to the very next transfer and may not be accurate for the scheduling of the FMS behavior more than two hops away (in the future). While the size of the code to be transferred can be considered almost constant (and therefore figured once and reused), the computational state and especially partial results are more difficult to estimate. Depending on the type of application, they may be constant (e.g., a service that plays an mp3 file will drag along the same file while migrating, and therefore its size is known). In other cases, this can be roughly evaluated based on the application's progress. For example, a file being compressed has an initial size and decreases (roughly uniformly) in time as the application makes progress. This way, the migration becomes cheaper with every second of useful processing done on a volunteer host, the rate also being computable. In other applications, the transfer becomes more expensive with time, such as in the case of a radio show encoding service. The more the service records, the larger the result file is. In this case, the size can also be estimated (e.g., given the broadcast and compression parameters, the file grows by x kb per second of useful work). There are also situations where this estimation is completely out of question, as the dynamics of the data and computational state are untraceable.

Since this is not central to my research, details of such estimates are abstracted away by using upper bounds on estimates as constant overhead penalties. E.g., for the tests I developed, I measure how much time it takes to do a lightweight/heavyweight migration using a representative sized data file. The computational power of a machine is also a parameter that influences the performance of a FMS. This effect is even more difficult to estimate. In my tests I used the advertised CPU speed as reference data, ignoring the instantaneous workload (which could make a 5GHz processor do worse than a 100MHz processor with less workload). As described before, some applications only require a minimum of resources (e.g., to be able to (de)compress mp3 files in real time, a minimum system configuration requires a processor faster than n MHz and a minimum of m MB of RAM memory; many applications have had their system requirements advertised on their boxes for many years now) to even work, others work under any conditions, just at different progress rates. For my decision algorithm, I need the speeds of various volunteer hosts I can choose from (assuming the workload takes a comparable toll from each of them). I use this to make a simple decision as to whether a host can do a minimum-requirements-specified job and which of two hosts can be expected to the faster one. Work related to evaluating the execution time of some software can be found in [12], [31], [95], [71].

To present the algorithm, I will use the configuration depicted in Figure 4.17. On the vertical axis are all pairs of hosts. The entire universe only contains four hosts for this example. Host a is the reference host, where the client executes. Each pair of hosts has its direct connectivity intervals depicted. A simple line means just that the two hosts could talk in a peer-to-peer fashion. A box indicates that the FMS could go there (e.g., the box that indicates that a and b could interact from t_1 to t_3, including overhead). The black boxes represent the FMS in action, while the empty (white) boxes, show where the FMS could go but it doesn't. The arrows indicate how the FMS transfers from one host to another under the client's machine supervision.

The scenario starts with the client pushing the service onto host b. This is marked as "discovery" on the picture and entails a heavyweight migration of the service. At moment t_2 the client has the option to continue working with b or to switch to c, which has approached meanwhile. Host c already offers the same service, so a lightweight migration suffices in this case (marked on the picture with "csb"). The a-b box has become white as the action moved to the a-c box. The direct transfer from b to c was

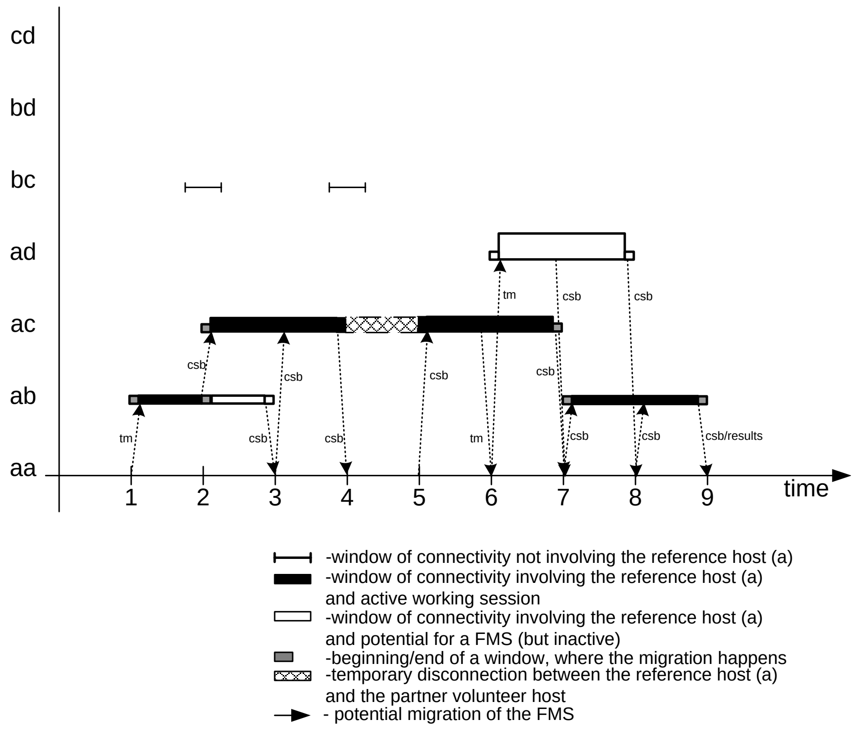

cd
bd
bc
ad
ac
ab
aa
1
2
3
4
5
6
7
8
9
time
tm
csb
csb/results
-window of connectivity not involving the reference host (a)
-window of connectivity involving the reference host (a) and active working session
-window of connectivity involving the reference host (a) and potential for a FMS (but inactive)
-beginning/end of a window, where the migration happens
-temporary disconnection between the reference host (a) and the partner volunteer host
- potential migration of the FMS

Figure 4.17: Sample mobile host configuration to illustrate the decision algorithm.

possible because of the window of opportunity b and c have directly, around moment t_2. Otherwise, the transfer would have evolved as though a had continued with b. At moment t_3, in the latter case, a would have to do a "take back" (a csb to itself) and then another csb to c, because a direct transfer from b to c is not possible at moment t_3. At moment t_4, host a could take back or could allow a temporary disconnect. This is an example of a situation where the need_connection flag advertised by the application makes a difference in the decision process. Also at moment t_2, the only reason for having switched to host c was that c works faster than b (see the thicker line). If the job had had a fixed length and b was obviously good enough to service the client until moment t_2, there wouldn't have been any reason to switch to c as the extra performance would not make any difference. The job is finished at moment t_9 (when the knowledge about the other hosts' motion profiles ends as well, "by coincidence").

The decision algorithm operates on the graph in Figure 4.18. The graph is built from the depiction of the windows of opportunity depicted in Figure 4.17, obtained from the analysis of the known motion profiles (hosts exchange motion profile information when they meet; more details are available in the next chapter).

Each node in the directed acyclic graph is a host with which the reference host (host a in this example, where the client executes) could be working at a certain moment. A directed edge represents either a transition from one host to another or useful work done for the client. Each such edge is labeled with its action to it left and the amount of progress made to it right. The thick arrows follow the FMS described.

The best FMS path is the one that gives the result as fast as possible and as completely as possible. That is the path that achieves the most useful work done per unit of time. The algorithm will therefore search for the path with the *greatest overall average progress rate.* For a fixed progress rate (e.g., for a real-time job such as live sound feed recording) the highest overall progress rate is achieved when there are as few migrations ar possible (and thus the gaps are few and the useful work accounts for more of the time spent on the job). In the case where the job is flexible in terms of performance, the highest progress rate is achieved by always choosing the host with more computational power (of course, the extra performance has to offset the cost of an otherwise unnecessary migration).

The times marked on the time axis represent important moments when a window of opportunity opens or closes. This is when a migration can or has to happen and choices have to be evaluated. Even though they are shown at the same distance, they need not be.

Following the execution of the FMS on the decision graph in Figure 4.18, the first decision has to be made at t_2. To be able to make this decision, the algorithm looks ahead to the point where the two branches converge in the same node again (the node representing host c, just before t_4). From all branches that start at t_2 the amount of progress yielded before the join is the decisive factor. In the example, $p_1+p_3+p_4$ means more useful work done until just before t_4 (where the c node spawns again two edges) than $p_1+p_2+p_5$ and therefore the path to the right (b-c-c-c) is chosen over the one to the left (b-b-a-c-c). To achieve this, the algorithm eliminates the edges that enter a join and lose the competition (such as the “Use c, p_5” edge between t_3

and t_4). Working backwards in the graph all edges and nodes on losing branches are "eliminated" until the node where the split happened is reached.

The same mechanism applies on the interval (just before) t_4 - (just after) t_5. The same approach indicated the path to the right yields more progress. In this particular split point, the need_contact parameter takes precedence though, if specified. If the service cannot be left unsupervised and permanent contact is required (which was not the case in the example above), the FMS would have been taken back by a and the execution path in the graph would have gone to the left.

If one of the branches that split up in a node then splits even further, the algorithm is applied recursively on that section of the graph, the local best result is integrated in the overall performance evaluation of the branch and then compared against the scores reported by other branches. An interesting situation develops between t_6 and t_8. A path splits to the right from c (going to c) and one to the left (going to a). The left branch then splits again at node d just before t_7. Applying the branch evaluation/elimination selection process described above, the branches c-a-d-d-a and c-c-c-a can be evaluated, and the latter wins (because, given the particularities of the scenario and cabailities of the hosts involved on the two paths, c-c-c-a path records more progress than the alternative path until t_7 when they merge). During the edge and node elimination step, only the d-a "CSB, 0" branch can be eliminated because the node d in which originates is a split node. The real reason behind this maneuver is that the branch this node is on (the one that started from c just before t_6) meets with the other one after t_8 and the prefix of the branch evaluated so far (c-a-d-d) is part of the rest of the potential execution trace until the next join. Eliminating the d-a edge mentioned above, the two branches remaining are c-c-c-a-b-b-b (going right from node c just before t_6) and c-a-d-d-d-d-a-b (going left from node c just before t_6). The righthand side path wins again (as above, I assume the branch on the righthand side achieves more progress until t_8). By the end of the scenario, the useful work invested to finish the task was $p_1+p_3+p_4+p_6+p_7+p_8+p_9+p_{10}+p_{13}++p_{15}+p_{16}$ and the overall progress rate is the sum divided by (t_9-t_1), which is the best progress rate that could have been achieved.

An interesting observation is that the graph is a lattice (because the nodes are moments in time and there is a partial order relation over time) with only one source and only one sink. This is because it is always the client who starts the FMS and it is always the client who collects the results at the very end. The information on which the construction of the graph is based is always considered accurate. This means that it is not possible to expect a window of interaction between two hosts because that is what their motion profiles indicates and the window not to happen. The information may be incomplete in the sense that it ends before the job can be finished (e.g., imagine the graph up to t_7). In this case, the *"take back"* option is always available. When there is no other option, the client can take back the FMS, even if the job is not finished.

As time goes by and hosts move, the client's host learns more about the motion profiles of the hosts it meets and thus updates its knowledge base. The changes incurred on the graph are of two types. First, the knowledge can expand the horizon in the future. That means the graph can be extended downward, as time flows top to bottom in Figure 4.18. Another way the graph can expand is sideways. There cannot be discovered new windows of opportunity among hosts whose motion profiles the current host already knows, for the part of the motion profile known (such windows can be discovered after learning more about the motion profiles and expanding the knowledge base in to the future). New hosts can come and create new interaction opportunities. In Figure 4.19 there are two new windows of opportunity (shown by the dashed outlines) discovered after a motion profile update (assume the a-d window didn't exist before).

The graph is updated by adding the corresponding branches in the right places. While these can get quite complicated, they essentially represent alternate paths that can be reduced as described before. Additionally, the entire optimal path computation has the prefix propriety which means the optimal path does not need to be re-evaluated before the moment when the new branch attaches to the graph. It also means that when the new branch is accounted for, if it loses the bid, the effect does not propagate downstream. The graph update has, therefore, a very localized effect. As time passes, expired nodes in the graph are garbage collected to avoid an infinite growth.

Migratory services resemble distributable threads [] [] from Real-Time CORBA 1.2 [83]. A distributable thread is mapped to the execution of a local thread in a distributed system. A distributable thread has a unique ID similar to the unique ID of a

FMS. Each distributable thread has a single execution point in the entire system, like a client interacts with a single service provider host. The distributable thread does not migrate itself. It rather transfers the control to various remote processors where parts of the code is executed similarly to a remote procedure call which behaves as a local operation invocation. A mobile thread assisting a client in a FMS migrates itself from machine to machine and does not perform remote calls and thus transferring the execution to other threads.

4.5 Summary

In this chapter I have presented a complete architecture for service oriented computing in mobile ad hoc networks. I addressed issues that impact all aspects of SOC in MANETs. I presented a completely distributed design for *service directories* that guarantees the consistency of data offered to inquiring clients even with disconnections. I also presented a complete solution for service *advertisement and discovery.* As my solution to SOC in MANETs entails code migration, I developed *a complete automated code management system.* Binary files supporting the serialized objects that are transferred between hosts are managed entirely transparent to the programmer that manipulates the serialized objects. As one such object might use more than its own binary file, the entire closure of binary dependencies is handled in the same transparent manner. The contribution also includes a complete solution for *runtime service upgrades.* Both the server and the proxy can be swapped while the client is making use of the service with little or no impact on the client (the impact should be understood as being at most a delay). While these are issues one can encounter in wired networks too, the dynamic character of MANETs inspired the development of *follow-me sessions.*

The *migration of services* is a novelty I developed to address the issue of short connectivity intervals in the new, dynamic environment. It eliminates the limitations otherwise imposed by the duration of a window of opportunity, supporting (within limits) continuous service provision in the presence of disconnections. A *strong migration mechanism* was employed so that execution can continue on the new host of the server process rather than restarting it, offering a *pause-transfer-resume* behavior.

A decision algorithms using knowledge about host movement in space helps schedule and optimize the future evolution of FMSs.

Research dissemination. This chapter is based on material published in *"Service Oriented Computing Imperatives in Ad Hoc Wireless Settings"*, a book chapter published in "Service-Oriented Software System Engineering: Challenges and Practices" (Zoran Stojanovic and Ajantha Dahanayake, editors, Idea Group Publishing, Hershey, PA, USA, April, 2005); *"Service Provision in Ad Hoc Networks"* published in the Proceedings of the 5th International Conference on Coordination Models and Languages (8-11 April 2002, York, England); *"An Architecture Supporting Run-Time Upgrade of Proxy-Based Services in Ad Hoc Networks"* published in the Proceedings of Pervasive Computing Conference (21-24 June 2003, Las Vegas, NV, USA); *"Knowledge Driven Interactions with Services Across Ad Hoc Networks"*, published in the Proceedings of Second International Conference on Service Oriented Computing (15-18 November 2004, New York, NY, USA); *"Automated Code Management for Service Oriented Computing in Ad Hoc Networks"* published as Washington University technical report number WUCSE-2004-17. *"Context Aware Session Management for Services in Ad Hoc Networks"* published in the Proceedings of the IEEE International Conference on Services Computing (11-15 July 2005, Orlando, FL, USA - **IEEE best student paper award**). An integration of the security mechanisms described in Chapter 3 and the service provision middleware described above was described and published in *"Secure Service Provision in Ad Hoc Networks"* in Proceedings of the First International Conference on Service Oriented Computing (15-18 December 2003, Trento, Italy).

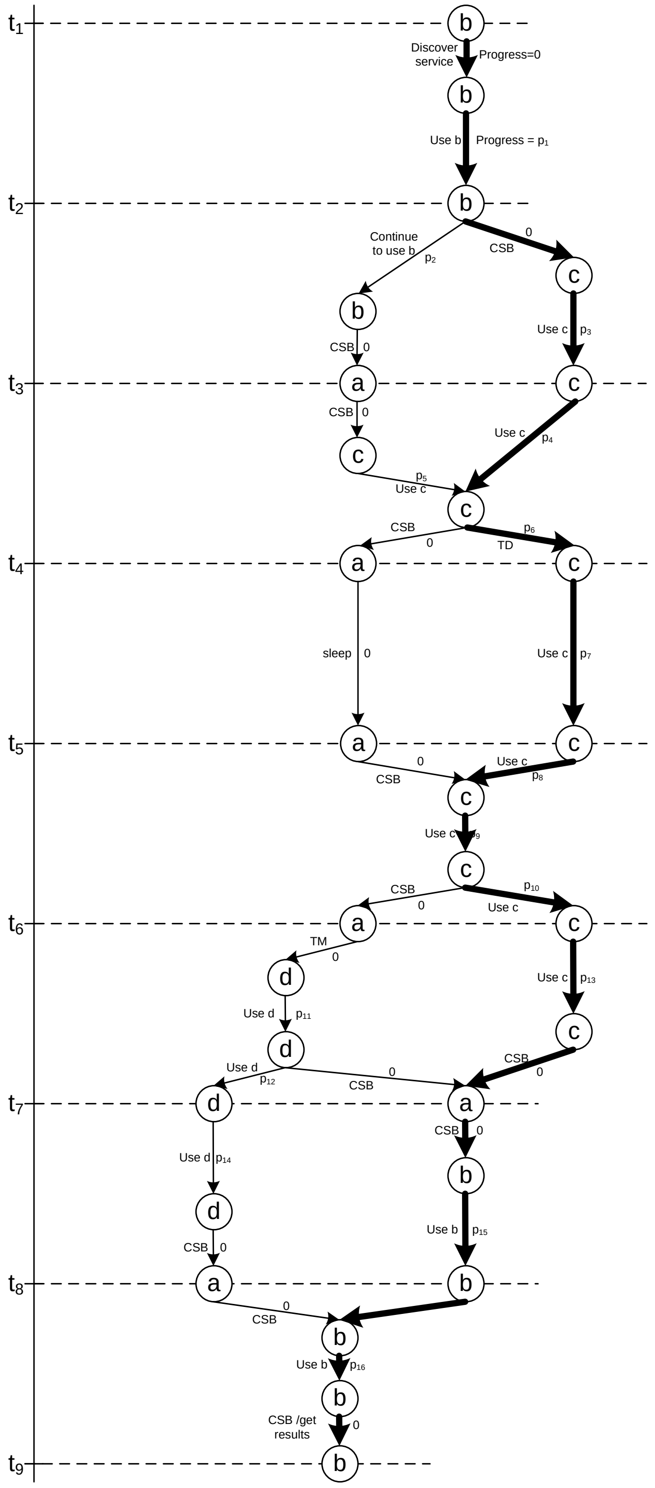


Figure 4.18: The decision graph for the mobile hosts evolution depicted in the figure above.

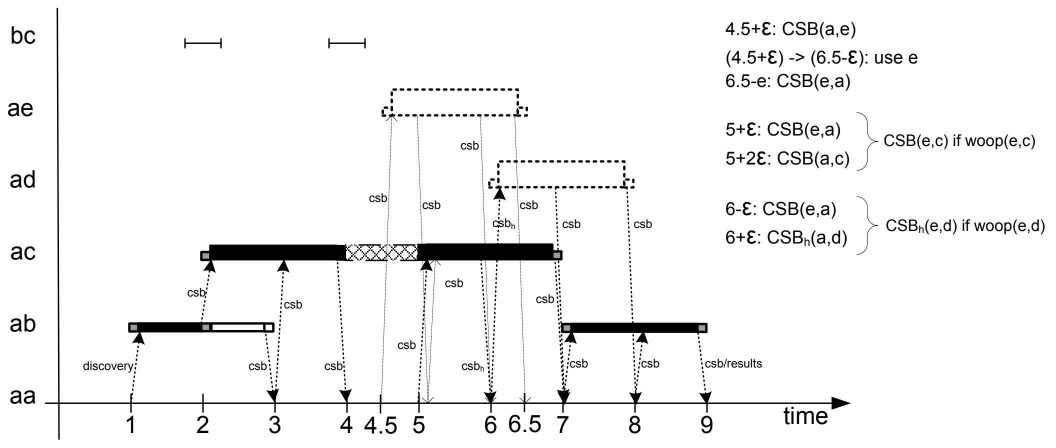


Figure 4.19: As motion profile information is learned, more woops can be discovered.

Chapter 5

Disconnected Routing

5.1 Introduction

The service oriented computing infrastructure described in the previous chapter is designed to help improve the interaction among hosts at the application level. In order for a distributed computing infrastructure to work, routing algorithms must ensure inter-host reachability. Conventional wired networks are increasingly being extended to include wireless links. It is important to distinguish between wireless links that are essentially stable in time and space, and those that are not. With wireless technology claiming an ever increasing share of the market, the interaction patterns between hosts need to be re-examined to address additional issues raised by this new environment. Former assumptions that made things easy or even possible in the wired setting may simply cease to hold in a wireless world. Wireless technology allows devices to become mobile. The topology of the network can change while applications are running and while interactions between such mobile hosts are in progress.

Modern wired computer networks provide a high degree of reliability and stability. The interaction between two different hosts in a wired network is expected to succeed and it is not believed that the normal operation of a host can make it, even temporarily, unavailable to the rest of the network. Security and other considerations may cause an application residing on a host to become unavailable during its normal functioning but even such intervals tend to be reasonably well bounded in duration and frequency. Furthermore, the interaction between hosts is stable: the party a host interacts with is always found at the same IP address or under the same name, even when the name is resolved to different addresses at different times (e.g., a host may obtain its address from a DHCP server).

The stability of such an environment allows for efficient configuration of the traffic in the network. The total set of possible routes is mostly static, *hardcoded* in configuration files and scripts that do not change very often. While on-the-fly adjustment of information about the set of available routes is possible, such adjustments generally have a reactive flavor, e.g., in response to network congestion. Fundamentally, modern routing protocols in wired networks depend at least indirectly on a stable set of physically reliable routes.

Nomadic networks, like the cellular telephony infrastructure or the pervasive wireless networks at universities, ensure communication between mobile devices such as phones, laptops, and PDAs. Nomadic networks do this by maintaining connectivity between each device and some part of a fixed infrastructure that acts as a communication liaison between such devices. Wireless devices connect to the infrastructure via access points connected to other access points connected to other wireless devices as they travel through physical space. Connectivity among devices is maintained by routing algorithms that deliver messages between mobile devices over the fixed infrastructure. The physical position of each device accessing the fixed infrastructure is important for routing packets to the access point closest to the device. The devices are always connected, but move in space.

In ad hoc networks the infrastructure for communication is made entirely of mobile hosts that interact in a peer-to-peer manner and route traffic for each other. There is no fixed infrastructure on which the mobile hosts can rely for message delivery. As long as two devices are connected (in direct contact or via multiple hops), they can exchange information and interact as if they were wired. In ad hoc networks, however, a stable multi-hop connection has severe temporal limitations, as well as spatial limitations due to the ad hoc routing algorithms that have to be carried out by the hosts themselves.

In this chapter I present an approach to increasing communication capabilities in mobile ad hoc networks. This chapter describes three main contributions to the state of the art in ad hoc networks. First, it formalizes the problem of message delivery in the presence of disconnections between hosts, and illustrates the formalism with examples. Second, it gives an algorithm for reliable message passing in the face of temporally discontinuous connectivity, exploiting host motion and availability profiles. Third, it offers an analysis of and refinements to that algorithm, based on the amount of information needed about the temporal patterns of connectivity.

5.2 Motivation and Related Work

Ad hoc routing protocols currently available can be split into four categories, as shown in Figure 5.1. In *proactive protocol*, nodes continuously search for routing information which they store in tables such that every route is already pre-computed before it is needed. The continuous table maintenance is a constant overhead and consumes significant bandwidth. When using *reactive protocols* nodes start looking for a route only when needed. While the bandwidth usage is significantly reduced (no continuous network pinging and polling is needed to keep the table entries updated), there is a delay every time a message has to be sent, as the entire route search is performed reactively when a message is ready to go. The *hybrid* approach uses a combination of proactive and reactive mechanisms in an effort to obtain better performance. All protocols in these three categories discover fully connected routes, from the sender of a message to the intended recipient.

Disconnected routing protocols send messages from source to destination without requiring a fully connected route. They pass messages from host to host, and use host mobility to their advantage, exploiting the continuously changing set of neighbors of the host that has the message at each moment as an opportunity to deliver the message to the destination host or at least to get the message closer to it.

DSDV [93] (Destination Sequenced Distance Vector) is based on the Bellman-Ford algorithm. It maintains a list of all destinations and the number or hops to each of them. In addition to Bellman-Ford, DSDV avoids loops. The periodic update's frequency can be customized, as well as the number of update intervals allowed to transpire until a route is considered stale.

PROACTIVE	REACTIVE	HYBRID	DISCONNECTED
DSDV	AODV	ZRP	Epidemic
WRP	TORA		Message relay
CSGR	ABR		
	DSR		

Figure 5.1: Routing protocols in mobile ad hoc networks.

WRP [81] (Wireless Routing Protocol) enforces consistency checks of the information obtained by each host from its predecessor, to avoid the *count to infinity* problem [1]. WRP is a loop-free algorithm. It maintains four tables: a distance table, a routing table, a cost table and a retransmission table.

CSGR [17] (Cluster Switch Gateway Routing) groups mobile nodes into clusters, with each cluster having a leader. The routing is hierarchical. Inside each cluster, CGSR uses DSDV. Each node maintains a list of cluster members and a routing table. Cluster leaders have a higher resulting communication and computation load.

AODV [92] (Ad Hoc On Demand Distance Vector Routing) is a modification of DSDV where the routes are discovered on demand. It uses broadcast for route requests and sends failure notifications to upstream neighbors. It reduces bandwidth usage and it is a loop-free algorithm. It uses cached information to respond to route requests (which may be out of date until a failure notification is received).

TORA [89] (Temporary Ordered Routing Algorithm) offers localization of control messages to a small set of hosts near the location of a topological change. This algorithm requires that hosts clocks be synchronized. In its operation the algorithm attempts to suppress, to the greatest extent possible, far-reaching propagation of control messages. In order to achieve this, TORA does not use a shortest path

[1]Host B looses link to A (and sets the local routing table entry for A to infinity), but C still thinks it has a route to A over B. C propagates this to B and B believes it can reach A over C. B now adds its delay to C to the received value and saves the new (believed) connection to its routing table (because it is smaller than infinity), creating a loop between B and C for all messages sent to A. Step by step B and C now update each other's routing tables, increasing the entry for A each time until it reaches the value defined as infinity, causing the link to be recognized as broken.

solution. TORA instead builds and maintains a directed acyclic graph rooted at a destination. Nodes are assigned heights in the spanning tree consisting of routes between hosts and no two nodes may have the same height (the heights are calculated using a combination of link failure times, host IDs and a reference level which can be updated as the networks changes). Information may flow from nodes with higher heights to nodes with lower heights. Information can therefore be thought of as only flowing "down" the tree. By maintaining a set of totally-ordered heights at all times, TORA achieves loop-free multipath routing, as information cannot "flow uphill" and thus re-visit a node.

ABR [111] (Associativity Based Routing) is a loop-free algorithm that defines a metric based on the degree of association stability. Each node broadcasts a beacon with a certain periodicity. When the beacon is received by the host's neighbors, they update their associativity tables. Association stability is defined as the stability of the connection between a node and another node over time and space. A high degree of association indicates a low degree of mobility. The effect of the algorithm is thus to discover long lived routes in the mobile ad hoc network.

DSR [61] (Dynamic Source Routing) is based on the concept of source routing. For this protocol, mobile nodes are required to maintain route caches that contain the source routes of which the mobile host is aware. Entries in the route cache are continually updated as new routes are learned. There are two major phases of the protocol - route discovery and route maintenance. Route discovery uses *route request* and *route reply* packets. Route maintenance uses *route error* packets and *acknowledgements.* Like AODV, DSR uses cached information for routing, which results in uncontrolled replies. It also suffers from scalability issues.

ZRP [46] (Zone Routing Protocol) is a hierarchic protocol. Mobile nodes are logically separated into zones, based on proximity. Each zone has a leader. ZRP uses a proactive approach inside each local zone, and is reactive between zones. Since hierarchical routing is used, the path to a destination may be suboptimal. Since each node has higher level topological information, host memory requirements are greater.

Epidemic routing doesn't build routes. Random pair-wise exchanges of messages among mobile hosts promote eventual message delivery. It's a trivial protocol over-credited for its marginal contribution by too many. By simply spreading a message around to any host that comes within range, it achieves a form of disconnected broad-cast in a desperate hope that the destination is eventually reached.

Message relay [69] is an approach that entails changing the hosts' desired movements so that they can build routes as needed. The goal of the algorithm is to cause as little inconvenience as possible when relocating hosts while trying to maximize their connectivity and to find routes.

A more detailed presentation of each of the protocols above, as well as presentations of many others, can be found at Secan Labs [1]. In mobile ad hoc networks, com-munication paths are created, changed and destroyed at a much greater rate than in conventional or nomadic networks. The message routing infrastructure is composed of mobile participants, and therefore is itself subject to disconnections. A new approach is therefore needed to describe the connectivity and communication paths between mobile hosts in ad hoc networks.

In ad hoc networks, the mobility of hosts introduces the question of whether two hosts can be connected at all. The attention shifts from maintaining *stable connectivity paths* (i.e., paths that route the traffic between two hosts that are connected to the network at the same time), to whether two hosts *can communicate* in an environment with *transient connectivity* (i.e., a message can be delivered from one host to another over a sequence of hosts which may or may not form a *continuous path* from source to destination at all times during the delivery).

In conventional networks, two hosts can exchange messages if and only if a stable path exists between them in the network topology. The ability to exchange messages in that setting is reflexive, symmetric and transitive, *at each hop and along the entire path.* However, in mobile ad hoc networks paths are not stable over time, as the connectivity between any two hosts can vary due to the motions of those hosts or due to their intervals of availability (e.g., due to power-saving modes).

When two hosts are in direct contact, messages can flow between them symmetrically, i.e., as peer-to-peer communication. Ad hoc routing extends this symmetry across multiple hops, but requires all hosts along the path to be connected throughout some continuous interval of time.

However, there are reliable message delivery paths in ad hoc networks that are not symmetric, and thus cannot be exploited by either peer-to-peer or ad hoc routing approaches. In particular, messages that traverse a link before it goes down may be delivered on that path, while messages that reach that link after the disconnection will not. Therefore disconnection introduces non-symmetric message delivery semantics for any path longer than one hop.

The following routing techniques leverage characteristic profiles of the motion and availability of the hosts as functions of time to accomplish disconnected message delivery. Thus, these techniques do not need to rely on the availability of a reliable end-to-end communication route during the process of communication. The only thing these techniques must rely is pairwise connectivity of hosts, which can be inferred from the hosts characteristic profiles.

5.3 Transient Connectivity

5.3.1 Connectivity Intervals and Message Paths

Figure 5.2 shows an example of temporally discontinuous intervals of direct connectivity between hosts a, b, and c. At time t_0, hosts a, b, and c are disconnected. From t_1 to t_2, hosts a and b are connected and can exchange messages. From time t_2 to t_3 all hosts are disconnected, and so forth.

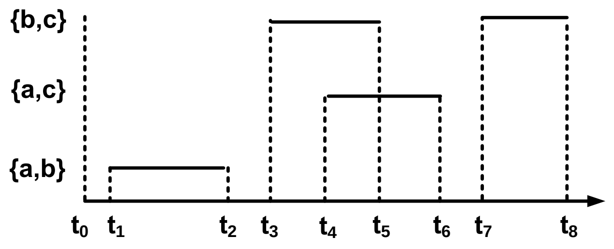


Figure 5.2: Connectivity intervals.

I define a *connectivity interval* for any two hosts to be a continuous time interval they can use for effective communication. In practice, a connectivity interval is likely to be shorter than the total time the two hosts can communicate directly, since the beginning and the end of the direct connectivity period may incur overhead spent to establish contact and to disconnect without loss of data or inconsistency of state. However, because such overhead is a function of the connection and disconnection protocols used, for the sake of generality in this paper I assume the connectivity interval for two hosts is identical to the interval during which they can communicate.

Figure 5.3 shows the possible message exchanges that can take place between hosts a, b, and c in intervals in which the connectivity is available as shown in Figure 5.2. When two hosts are connected, a message can hop from one to the other, and the sequence of connectivity intervals over time can be transformed into a directed graph.

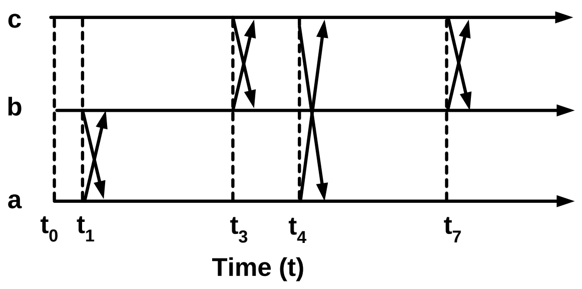


Figure 5.3: Connectivity intervals.

I define a *message path* between any two hosts to be any sequence of hops that a message can follow over time from the source host to the destination host. This should be understood differently from the ad hoc routing definition of a path. In ad hoc routing the entire path has to be defined from start to end at a given moment in time. My approach allows ad hoc routing to occur when hosts are directly connected but also allows message delivery, even when (potentially all) hosts are disconnected during an interval between when a host receives a message and when it transmits it to the next host.

5.3.2 Discontinuities and Non-Symmetric Paths

Two hosts that are directly connected have symmetric message passing capabilities. For example, in Figure 5.2 hosts a and b can exchange messages during interval [t_1, t_2], as b and c can during [t_3, t_5], and so on. However, if no direct connection exists between two nodes then messages may be able to pass between them in one direction but not the other.

Figure 5.4 illustrates two disjoint connectivity intervals between three hosts d, e, and f, where d and e are connected over [t_0, t_1], and e and f are connected over [t_2, t_3].

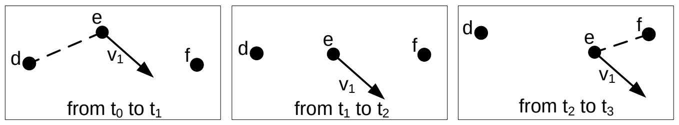


Figure 5.4: Mobile disjoint connectivity. The arrow indicates how the host moves, while the dashed line indicates a wireless connection between two hosts.

Figure 5.5 shows the non-symmetric end-to-end message passing capability resulting from the connectivity intervals shown in Figure 5.4, i.e., messages can be delivered from d to f but not from f to d.

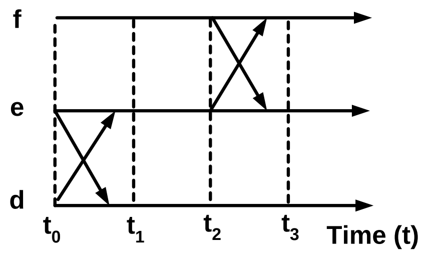


Figure 5.5: Non-symmetric message paths.

5.4 Problem Formalization

The problem addressed by this chapter is to determine whether given

- a set of hosts N,
- a source host $\mathsf{p} \in \mathsf{N}$,
- a destination host $\mathsf{q} \in \mathsf{N}$,
- an initial moment t_0, and
- *characteristic profiles* (here, of mobility and availability) over all hosts in N,

I can construct a path so as to ensure message delivery from p to q. In Section 5.5 I provide an algorithm based on this formalization, to decide whether for two hosts, a message can travel from the first to the second under the restrictions imposed by the characteristic profiles of the hosts. I also seek to build the set of paths a message could follow from source to destination, if any.

I express the *mobility* of a host in terms of a function that maps time $\mathsf{t} \geq \mathsf{t}_0$ to its physical position: $\mathsf{m}_\mathsf{p}(\mathsf{t})$ characterizes the mobility of host p. I formalize the *availability* of a host (e.g., due to power saving modes) as a boolean function over time: $\alpha_\mathsf{p}(\mathsf{t})$ is true if and only if host p is willing and able to communicate. Finally, I formalize a *characteristic profile* $\pi_\mathsf{p}(\mathsf{t})$ for host p as a function from time to the pair consisting of p's mobility and its availability: $\pi_\mathsf{p}(\mathsf{t}) = <\mathsf{m}_\mathsf{p}(\mathsf{t}), \alpha_\mathsf{p}(\mathsf{t})>$.

I formalize the direct connectivity of any two hosts p and q in N as a time-dependent boolean relation $\Gamma_\mathsf{pq}(\mathsf{t})$ abstracted from their characteristic profiles $\pi_\mathsf{p}(\mathsf{t})$ and $\pi_\mathsf{q}(\mathsf{t})$. $\Gamma_\mathsf{pq}(\mathsf{t})$ is true if and only if at time t both hosts are available and their Euclidian distance is less than $\mathsf{R_c}$, the communication range of a host (assumed to be known, constant, and the same for all hosts), and false otherwise. I express this as:

$$\Gamma_\mathsf{pq}(\mathsf{t}) = |\mathsf{m}_\mathsf{p}(\mathsf{t}) - \mathsf{m}_\mathsf{q}(\mathsf{t})| < \mathsf{R_c} \land \alpha_\mathsf{p}(\mathsf{t}) \land \alpha_\mathsf{q}(\mathsf{t})$$

I now formalize an end-to-end path relation κ between hosts p and r over the intervals of connectivity *generated* by the relation $\Gamma(\mathsf{t})$. In doing so, I remove the requirement that the entire communication path be defined from source to destination uninterruptedly. I say that hosts p and r are in relation κ if and only if a message leaving from p can be delivered to r (directly or over a path that spans intervals of disconnection). The formal expression of this definition is:

$\mathsf{p}\ \kappa\ \mathsf{r} \Leftrightarrow \exists\ \mathsf{t_0}...\mathsf{t_k}$ monotonically non-decreasing moments in time, with $\mathsf{t_0}$ representing the current moment, and $\exists\ \mathsf{n_1}...\mathsf{n_k}$ hosts chosen[2] from the set of hosts N, such that $\mathsf{n_1} = \mathsf{p}$, $\mathsf{n_k} = \mathsf{r}$ and $\Gamma_{\mathsf{n_1},\mathsf{n_2}}(\mathsf{t_1}) \wedge ... \wedge \Gamma_{\mathsf{n_i},\mathsf{n_{i+1}}}(\mathsf{t_i}) \wedge ... \wedge \Gamma_{\mathsf{n_{k-1}},\mathsf{n_k}}(\mathsf{t_{k-1}})$.

At any given instant t, the direct connectivity relation $\Gamma(\mathsf{t})$ is reflexive, also symmetric and transitive, in that (1) a host is always connected to itself and retains messages across intervals of unavailability, (2) if two hosts are connected they can communicate bidirectionally, and (3), as in ad hoc routing, messages can flow from any member of an instantaneously connected set of hosts to any other. As Figure 5.5 illustrates however, the end-to-end path relation κ is reflexive but is neither symmetric nor transitive.

5.5 Solution

5.5.1 Basic Global Oracular Algorithm

Figure 5.6 shows a more complex scenario involving eight hosts: a, b, c, d, e, f, g, and h. The discussion begins with the moment $\mathsf{t_0}$ when host a wants to send a message to host h. The top half of Figure 5.6 depicts the connectivity intervals graphically, while the bottom half shows the pairs of hosts in $\Gamma(\mathsf{t})$ during each interval (for readability I show only one of every two symmetric pairs).

The timeline in Figure 5.6 is divided into discrete intervals according to the times when at least one pair of hosts connects or disconnects. I assume sufficient time during each interval for ad hoc routing to occur, i.e., messages can be exchanged among all pairs of hosts in the transitive closure of Γ during that interval - the widths of the intervals appearing in any of the figures in this section are not assumed to be proportional to the duration of the interval.

[2]A given host in N may be chosen repeatedly in the sequence.

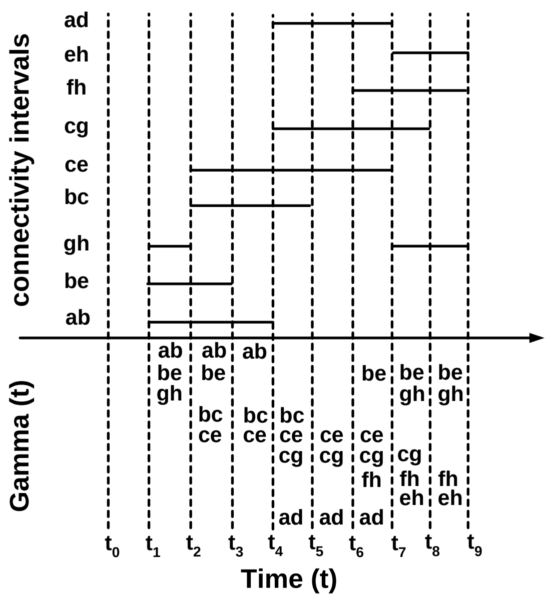


Figure 5.6: Sample pairwise connectivity (global view).

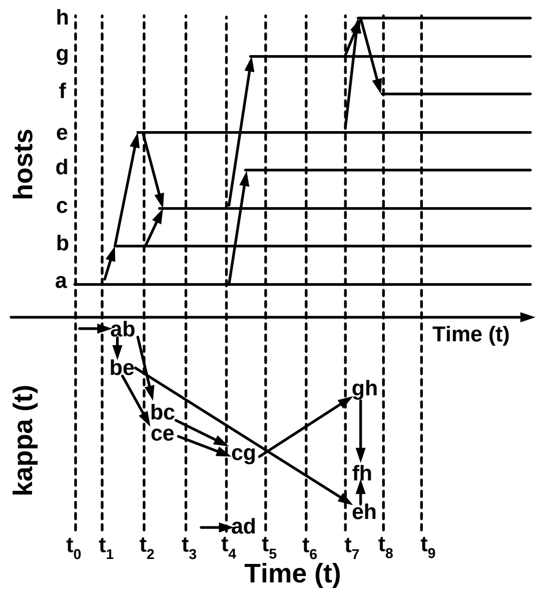


Figure 5.7: Candidate paths for a message sent by host a.

If host a has a message to send, the paths the message could follow in the mobile hosts initial configuration and subsequent evolution depicted in Figure 5.6 are illustrated in

Figure 5.7. A horizontal line in the upper half of Figure 5.7 means the corresponding host has (could have) the message (one might infer that once a message touches a host, the host will have it forever, which is not the case; Figure 5.7 only indicates that once a message reaches a host, that host can pass it on at any later moment). The arrows indicate how and when the message is transferred from a host to another. An algorithm to build a tree of message delivery paths runs on the graph obtained from the motion profiles known at a given moment, for various intervals for each host. The graph is represented as a set of nodes and their respective adjacency lists. The motion profiles lead to windows of communication opportunity across temporary links as can be observed in Figure 5.6. The bottom part of Figure 5.7shows the Γ relations, which identify dynamic links in the graph.

Using the connectivity intervals depicted in Figure 5.6, the following algorithm describes how to build a tree of message delivery paths from source a to destination h. In the path tree, the root is the source host for the message and any path from the root to a node in the tree represents a path a message *could* follow until it is delivered to the destination or gets stuck without any chance for further progress.

The algorithm starts building the tree from the root at time $\mathsf{t_0}$. Each node contains the ordered sequence of hosts the message has visited as of time $\mathsf{t_i}$. The algorithm adds children to each node for each previously unvisited host the message can reach next from that node until there are no more children to add to any node. Using the previously defined formalization, this is expressed as follows. For example, assume a node contains the marking $<\mathsf{a,b,e}>$, because nodes a, b, and e, have already received the message. Because during the interval $[\mathsf{t_2}, \mathsf{t_3}]$ Γ_{bc} holds, I add a child node to the tree, attached to the $<\mathsf{a,b,e}>$ node: $<\mathsf{a,b,e,c}>$. I avoid adding a previously seen host (which could result in undesirable looping of messages or unbounded recursion of the algorithm itself) by recording the sequence of hosts seen so far. I also label each edge in the tree with the earliest point in time when that transition can be taken, again based on the ordering of connectivity intervals in relation κ.

Once the tree is built, a depth-first search ending in a leaf of the tree containing the destination node will reveal the path the message needs to follow from source to destination. Further analysis of the entire set of paths in the fully constructed tree can allow for optimizations in terms of the number of hops or the set of hosts the message will visit (e.g., the latter can be important for security reasons). If I label each edge of the tree with the interval during which the extension can take place (when the child node is added to the tree), then the end-to-end delivery time or the time spent on each host can also be considered as constraints in the path selection decision.

Figure 5.7 illustrates how my algorithm runs on the example input data, which consists of an initial time t_0, a set of hosts a, b, c, d, e, f, g, h, a source host a, a destination host h, and the characteristic profiles of the hosts, $\pi_a(t)$, $\pi_b(t)$, $\pi_c(t)$, $\pi_d(t)$, $\pi_e(t)$, $\pi_f(t)$, $\pi_g(t)$, and $\pi_h(t)$.

During the interval $[t_1, t_2]$ hosts a, b, and e form a temporary ad hoc routing network, and the message can be passed from a to b and from b to e. During the interval $[t_2, t_3]$ the message can be passed from either b or e to c. During the interval $[t_4, t_5]$ the message can be passed from a to d and from c to g. During the interval $[t_7, t_8]$ the message can be passed from either e or g to h and then from h to f.

The resulting tree of all message delivery paths from source host a is shown in Figure 5.8. If a particular destination host such as h is specified, the subtree of interest is one in which all leaves are labeled with a sequence of host names ending in h, illustrated by the dotted line in Figure 5.8.

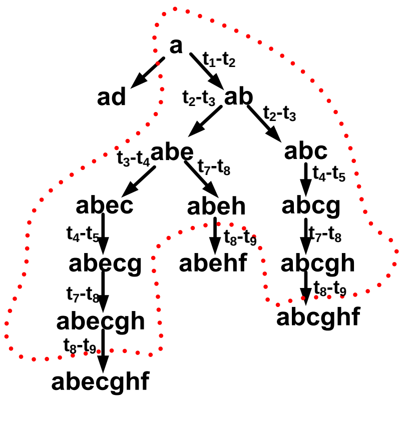


Figure 5.8: Route-building tree.

5.5.2 Tree Algorithm Complexity Analysis

Message Complexity. Because edges in a global path tree (such as the one shown in Figure 5.8) map one-to-one to message transmissions by the global algorithm, the message complexity along any distinct path in my global algorithm is bounded by one less than the height of the tree, i.e., |N|-1. The number of transmissions needed to propagate a message from a source host to *all* other reachable hosts is also bounded by |N| - 1 because in the presence of global information, transmissions are only made to hosts that do not have the message, and after a transmission the number of hosts that do not have the message is reduced by one.

Storage complexity. The principal issue for storage complexity is that of maintaining characteristic profiles on the hosts. The best case occurs when a closed form can be found for equations describing the characteristic profiles. For example, the characteristic profiles of a set of small solar-powered monitoring satellites in orbit around a space station might be described as a function of their orbits and the phases of those orbits when they are in sunlight. In such a case, the storage complexity is effectively linear in the number of profiles stored.

A similar argument can be made about the complexity of profiles that repeat over time, but for which a closed form may not exist or for which storage needed for the expression exceeds the storage needed to represent the profiles explicitly. In these cases, the profiles can be stored as tables of values with entries for each relevant segment of time in the repeated period. Although the storage requirements for these profiles are again effectively linear in the number of profiles, the overhead per profile is expected to be higher than for closed form equations.

The worst case storage complexity occurs when no temporal structure of the characteristic profiles can be used to represent them efficiently. In that case, it may not be possible to represent the entire future of the hosts, because the timeline is potentially unbounded. In some cases, *e.g.*, for factory robots that are powered down at the end of a product assembly run, it may be possible though expensive to store the profiles over a meaningful segment of time, though the storage required is linear in the number of physical movements and availability transitions. Similarly, if characteristic profiles evolve over time due to environmental influences or independent decisions by hosts, *e.g.*, for coordinated mission re-planning among mobile infantry squads, it may be appropriate for each host to store profiles for each of the other hosts only up to the time of some pre-defined future meeting.

Computational complexity. Computing even a single reliable path between source and destination hosts a and h in the basic global algorithm can be factorial in the number of hosts |N|. Specifically, because paths may not be symmetric, the number of nodes in the path tree that must be computed is a function of the permutations of hosts in the subset of N with a and h removed, N - {a,h}, plus one for the path tree node containing only a.

If it is necessary to compute the *best* path between each pair of hosts, an upper bound on the computational complexity to compute all paths in my global algorithm is given by the formula $(|N| - 2)! + 2$. Fortunately, the global nature of the information means that if it is in fact possible to obtain global and accurate information about the hosts characteristic profiles, then it is also possible to compute the path tree *off-line*, choose a path for each message, and simply give the paths to the appropriate source hosts.

This basic algorithm was a good learning exercise and proved that a solution to context aware disconnected routing is possible. The poor performance of this first algorithm motivated further research and the development of the improved algorithm presented next.

5.5.3 Advanced Global Oracular Algorithm

The solution entails a search algorithm in a dynamic graph where edges appear or disappear in a predictable manner over time, following a schedule obtained from the motion profile analysis. Each node in the graph represents a host. An edge between two nodes represents a link that has a specific lifetime. Because message delivery has a temporal dimension, it is necessary to account for the correct succession of links in a path from a temporal perspective. That means it is necessary to ensure that a message that goes from a to c via b traverses the link ab *before* it traverses the link bc. For this reason, it is possible to split the nodes from the tree generated by the previous algorithm into "sub-nodes". If the link ab is connected from t_1 to

t_3, I consider this to be a single link. Assuming the link breaks down at t_3 and is re-established at t_5 until t_7, I consider this to be a second link and therefore split node b into two sub-nodes (b, t_1) and (b, t_2). This way, I can search for paths that follow a correct succession in time. The expense is that the search space grows from the total number of nodes, to the number of sub-nodes, determined by the temporary links. This is an acceptable price to pay, as it accounts for the temporal aspect of the problem without increasing the time complexity of the algorithm at a storage cost bounded by the size of the space to explore, defined by the number of windows of communication opportunity.

The result of the algorithm is a routing table for the current node. The table has the structure illustrated in Figure 5.9. The table has a row for each host the current host could send a message to. For example, considering s as the reference host, the first row in s's routing table contains two entries s can use to send a message to v. Each entry represents an opportunity for s to send the message. The first field in an entry is the ID of the host to which the message is forwarded first. For example, to send a message to v, host s will forward it to a. Since this algorithm runs on every host, a will have a similar entry for v and will forward the message to its next first hop. The second field in an entry is the time when the message has to be sent. In the example, s can send a message to v at send_time_1 and again at send_time_2. To send a message to x, s will forward it to b at send_time_4. After that moment, the next delivery has to go through c at send_time_5. The third field represents the delivery time. This is the guaranteed latest moment when the message reaches the destination. According to s's calculation, the path will deliver the message sent at send_time_1 to v at receive_time_1. It is possible that from the time the route is found until the moment the message traverses a certain link on the way to destination, new information acquired by the intermediary hosts can help deliver the message on a shorter/faster path, unknown to the sender. If this does not happen, receive_time from the routing table entry is still a guarantee of worst case delivery. The last record shows the number of hops from source to destination. Note that the entries in a row of the routing table can be sorted for a variety of criteria, including earliest send time, earliest delivery time, or lowest number of hops.

The algorithm described in Figure 5.10 was inspired by Dijkstra's breadth-first search algorithm, applied to dynamic graphs with temporal traversal dependencies, which

```
v:  [a, send_time$_1$, receive_time$_1$, hops$_1$], [a, send_time$_2$, receive_time$_2$, hops$_2$]
w:  [b, send_time$_3$, receive_time$_3$, hops$_3$]
x:  [b, send_time$_4$, receive_time$_4$, hops$_4$], [c, send_time$_5$, receive_time$_5$, hops$_5$]
```

Figure 5.9: The routing table built for reference node s.

builds the routing table for a reference node, to support connection-less message delivery.

```
for $\forall$ (v, t) $\in$ Adj(s, t)
  start_node $\leftarrow$ v
  start_time $\leftarrow$ t
  enqueue(Q, (v, t, 1)) <1>
  while Q $\neq \Phi$
    u $\leftarrow$ head_node(Q)
    recv $\leftarrow$ head_time(Q)
    hops $\leftarrow$ head_hops(Q)
    for $\forall$ w $\in$ Adj(u, recv) <2>
     enqueue(Q, w, max(recv, time$_{uw}$), hops+1) <3>
     insert_routing(w, start_node, start_time, max(recv, time$_{uw}$), hops+1)
    end_for
    dequeue(q)
  end_while
end_for
```

Figure 5.10: The route table building algorithm.

The search starts from the current node, at the current moment. In the outer loop, the algorithm explores one by one all sub-nodes the current node is (or will be) connected to. This is necessary because each host reached from a certain first-hop sub-node from the source will carry this first node in the routing table as the starting point for message delivery. The parameters start_node and start_time are the same for all nodes reached from a certain first-hop sub-node. This first sub-node is added to the queue Q at the line labeled [3] in Figure 5.10.

As long as the queue is not empty, the algorithm explores hosts that can be reached over paths starting from s through the current sub-node. The algorithm takes the first node in the queue, the time it can be reached and the number of hops traveled by a message to reach this node. It explores all sub-nodes that can be reached from the current sub-node and therefore are adjacent to the current sub-node (line labeled $< 2 >$ in Figure 5.10). Adjacent sub-nodes are sub-nodes that *are* or *will be* in contact with the current sub-node. The algorithm adds any such sub-node w that is adjacent to the current sub-node u to the queue for later exploration and also marks in s's routing table that it can reach w from a certain start sub-node, with a message sent at start_time, over a certain number of hops (incremented every time a link is traversed, when the algorithm reaches a sub-node that is added to the queue and to the routing table). The time the sub-node is reached is $\mathsf{max}(\mathsf{recv}, \mathsf{time}_{\mathsf{uw}})$. The reason for this is illustrated in Figure 5.11.

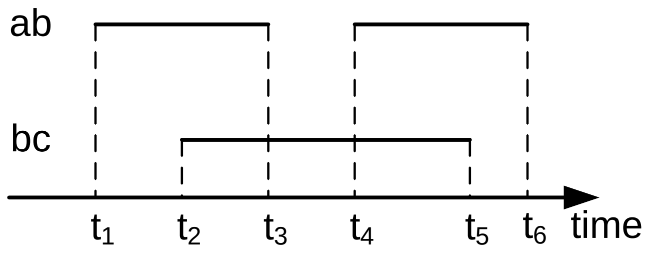


Figure 5.11: Reachability time.

Host a can send a message to c through b. If a needs to send the message at t_1, the message is received by b at t_1 and b can forward it to c at t_2. If a sends out the message only at t_4, from b's point of view the message is received at t_4, after t_2. For situations like this, when the message reaches b while b is connected to its next hop sub-node, it is necessary to consider the delivery time to c as $\mathsf{max}(t_4, t_2)$.

In the end, the first element of the queue is removed, as it represents the sub-node that has just been explored.

5.5.4 Exchanging and Using Partial Information About the Future

The knowledge base a host uses to find a route is made up of other hosts' motion profiles. This information is assumed to be accurate and that once learned, it doesn't change. The knowledge about the other hosts' motion ends at different moments in the future, depending on how much the current host was able to discover from other hosts. This information can grow (extend at the far end of the timeline into the future) but cannot change.

The message delivery paths that *precede* the moment when a host wants to send a message are especially important, as they represent prior opportunities for hosts to have exchanged information about their future behavior. At the point of wanting to send a message, the source host can then exploit the information available up to that point and plan its message delivery path accordingly, if possible. Several optimizations can be done by a host learning information that is not known to other hosts, including the ability to shortcut a path planned by another host when a better plan is discovered – I call this particular optimization a *path update*, and describe its use in greater detail in Section 5.5.5.

A key feature of this algorithm is that a host can send not only information about itself, but information that it has learned from other hosts. When two hosts meet, in addition to exchanging messages whose message delivery paths go from one to the other, each host also sends all motion profiles it has learned. While useless information can be exchanged, it's still likely to be cheaper than calculating the differences and sending only updates. At any point in time, a host can compute message delivery paths over the set of hosts whose characteristic profiles it knows *at that point.* The algorithm to construct those paths is thus the same as in Section 5.5.1, although the domain of hosts over which the algorithm operates is restricted to the subset of hosts whose profiles are known locally.

Assuming the same intervals of connectivity depicted in Figure 5.6, Figure 5.12 illustrates how the hosts learn each others' profiles. For the sake of this discussion, I start at time t_0 with each host knowing only its own profile, and show how characteristic profiles can be learned as time passes and interactions between hosts occur. Profiles are passed between hosts during all intervals from t_1 to t_8 except $[\mathsf{t}_3, \mathsf{t}_4]$ and $[\mathsf{t}_5, \mathsf{t}_6]$. It is also interesting to observe how the non-symmetric message delivery paths in κ result in non-symmetric profile information across the hosts. For example, f, g, and h all learned the profile for a, but a did not learn their profiles.

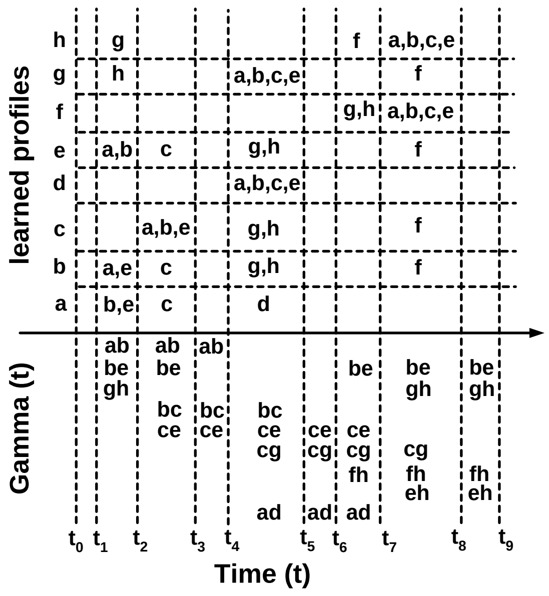


Figure 5.12: Learning profile information.

It is particularly interesting to note that in Figure 5.12 even if a message can start from a at time t_0 and be delivered to h as early as t_7, a is not able to discover that information. Thus, it is important to consider not only which message delivery paths exist between the hosts, but which hosts learn those paths and when. Only paths that are learned prior to when a message needs to be sent can be exploited by this version of my algorithm. The partial information held by the source node when it needs to send a message determines which paths the source node can compute. If a source node cannot compute a valid path due to incompleteness of its information, even if that path should exist, then the only alternative is for that source node to flood messages speculatively.

5.5.5 Recording the Past for Future Efficiency

With only local information, a source host may fail to compute any message delivery path to a particular destination. In that case, the source host may resort to epidemic routing, passing the message labeled only with the desired destination host (and not a planned path to that destination host) to each new host it encounters. In other cases, even if the message has a fully computed path, intermediary hosts may know a better route (e.g., earlier delivery, fewer hops, etc.) to the destination than the route initially computed by the sender.

For this reason, in addition to maintaining and exchanging sets of characteristic profiles, it can be useful to record the history of hosts each message has visited. In this section I extend the algorithm presented in Section 5.5.4 as follows. I store the sequence of hosts a message has transited within the message itself. When a host forwards a message that does not have a planned message delivery path, it adds its identity to the set of hosts recorded in the message. When the message is delivered, the receiver also learns its history. All hosts share a common decision function to rank such paths. A host that knows of a better path than the one in a message they receive (I assume a planned path is better than not having one) will always perform a path update, recording the new plan in the message and sending it to the next host in the plan when it meets it.

Three kinds of information can be provided by recording a message's history. First, if hosts record the set of hosts a message has transited, a host that receives the message can learn of other hosts having the message even though it may know nothing about their characteristic profiles. Second, from the same information, a host can infer at least partial information about which other hosts have been in contact previously, i.e., if combined with hosts exchanging profiles, the message history can give each host that receives it at least a partial representation of the information each previous host knows about the others. Third, if hosts also record the time at which a message traverses each host, and similarly stamp other information such as when host profiles were exchanged, then information about the past can be correlated directly with information about the future. Recording the past history of messages can thus increase the scope of information at each host, and offer future performance improvements, particularly in message complexity.

As Figure 5.12 illustrates, hosts may know different subsets of the global set of characteristic profiles. Using additional information which it has but which the sending host does not have, an intervening host may be able to route a message along a better path it computes, instead of the path originally given to it by the sender. Hosts using only local information can therefore select paths that are closer to optimal if, as described above, they are allowed to update path plans for messages sent from other hosts.

For example, assume the governing path selection heuristic is "fewest hops" and source host a computes its best path to destination host e through the set of hosts whose profiles it knows, {b, c, d, e}, as abcde. Suppose that just after receiving the message from host a and then disconnecting from host a, host b encounters a new host, f, which knows it can deliver the message directly to e. In that case, host b could update the planned path to be abfe and route the message through host f. Note that an intervening host can only learn of a better path by meeting a host whose profile was unknown to the sender (otherwise the sender would have already computed that better path).

Recording the past can further improve future efficiency by eliminating double delivery if the two message delivery paths intersect. The two paths will intersect at the destination but earlier intersection points (if any) can help cancel one of the redundant delivery paths before the message reaches the destination thus limiting unnecessary use of resources.

While dropping duplicates can potentially improve efficiency, it can also be a pitfall leading to deadlock in the message delivery protocol. For example, consider the scenario in Figure 5.13. Host a is the source of a message that leaves once through b and once through c (e.g., through Epidemic Routing), on two message delivery paths towards a common destination h. The two path intersect at d and f. It is possible that the message coming on the route a-b-d-e-f-h reaches e when the message coming on the route a-c-f-g-d-h is at g. In this situation, d will drop the message from g because d has already seen it. Similarly, f will drop the message from e. This leads to deadlock in the message delivery procedure.

It is possible to avoid the deadlock exemplified in Figure 5.13 by partially ordering all message delivery paths from a given source to a given destination, and dropping messages only on paths where another path from the source to the destination appears *earlier* in the partial order. The greatest potential for savings occurs when the paths are totally ordered, so that the message is delivered only along a single path, and the

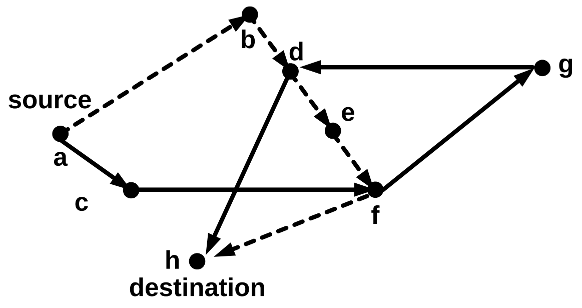


Figure 5.13: Potential deadlock in message delivery.

message coming along the highest ranked route passes first through an intersection point.

It is possible to achieve a partial ordering by simply assigning a non-negative integer to each host, concatenating the host integers along each message delivery path to form an $|\mathsf{N}|$-ary numeric value for each path with earlier hosts in the path representing higher-order digits, and having a host only drop the message if it knows a lower-valued path can deliver it reliably. If the host numbers are made unique, then the paths become totally ordered.

Unfortunately using this scheme alone unfairly causes traffic to be routed preferentially through lower-numbered hosts which could lead to overloading particular hosts or sub-networks. Such an arbitrary host numbering should therefore only be used if needed to break ties between message delivery paths in other more suitable partial orders. Two such partial orders are reasonably evident, one based on host bandwidth and one based on times at which messages are transmitted.

First, hosts could be numbered according to decreasing bandwidth and then arbitrarily among hosts with equivalent bandwidth. The host numbers at each hop would be then concatenated as before to form the path numbers. Even though the message routing is still unfair in this case, it is distributed more proportionally according to the capabilities of the hosts. As with the arbitrary host numbering scheme, shorter paths are preferred to longer ones, and higher-bandwidth hosts are preferred to lower-bandwidth ones at each hop.

Second, path numbers could be generated by concatenating the times of the message transmissions, but with the *later* transmission times representing higher-order digits in the concatenation and the assembled digits placed to the right of the radix as a floating point fraction, rather than as an integer. Although unlikely, it is possible two message delivery paths could be temporally equivalent, in which case they can be arbitrarily distinguished to form a total order by concatenating a unique lowest order digit to the end of each. This construction does not prefer any hosts in particular, nor does it prefer paths according to the number of hops they take. Rather, this construction prefers paths where the delivery of the message to the destination is earliest, then paths where the delivery time to the penultimate host in the path is earliest, and so forth.

5.6 Complexity

The search space of the problem is a graph where nodes are hosts and links are connections between nodes, established in a peer-to-peer fashion, as nodes come within communication range while they move in space. Due to the temporal aspect of the problem, I split a node in sub-nodes such that each node is represented by a different sub-node for each moment when the node can be reached. This trick simplifies the search that has to account for the temporal order in which links are traversed, without increasing the complexity of the problem (it actually reduces it as it is not necessary to verify which links can be considered for an outgoing message that has reached the reference node at a certain moment). In this section I analyze the complexity of the algorithms used to build a routing table, to lookup information in the table and to maintain the routing table. For the rest of this analysis I will assume the routing table entries are ordered by the send_time field of the entries.

5.6.1 Building the Routing Table.

The algorithm for building the routing tables at start-up is a breadth-first search algorithm, in which each link is traversed once. As each link represents a window of communication opportunity (hereafter referred to as a “woop”) between two nodes, the link is traversed during exploration only *once* because of the sub-nodes I introduced. The complexity of the problem is thus O($\|$ *woops* $\|$), where $\|$ *woops* $\|$ represents the number of woops between all sub-nodes in the graph (the number of edges in the graph). $\|$ *woops* $\|$ is a parameter that depends on the number of hosts and on their mobility patterns, which is why it cannot be split further into tributary components.

The insert_routing routine adds an entry to an ordered list, which takes logarithmic time. The worse case complexity is O($\|$ *woops* $\|$), when all entries are added to only one list in the routing table, e.g., when all messages that originate on host a can reach the rest of the world only after a first jump to host b (i.e., there is only one host that ensures the communication between the reference host and the rest of the world). Assuming a mobility model that allows a uniform distribution of communication opportunities, the complexity of the insert_routing routine becomes $O(log\frac{\|woops\|}{number\_of\_hosts})$. This means the complexity of the algorithm becomes $O(\| woops \| \; log\frac{\|woops\|}{number\_of\_hosts})$. Enqueueing and dequeueing elements into/from Q takes O(1) per enqueue or dequeue operation.

5.6.2 Using the Table.

To send a message to a host h, a reference host a needs to retrieve information from the row for h in its the routing table. The routing table is organized as a hash table with entries representing routing paths being stored in lists for each host. The access to such a list is O(1) because the hash table is indexed by the host name. In the list of entries specific to a target host h, the head of the list offers the first opportunity to send the message from the reference host (a), to the target host (h). I remind the reader that I considered the lists to be sorted by the send_time field of each entry, which yields an O(1) access to the head of the list (first opportunity to send). If the semantics of lookup are different (i.e., having earliest delivery time), the lists are built to obey a different sorting (based on another field in each record), which takes the same complexity as described in Subsection 5.6.1, and the lookup is also O(1). For the case when the routing table is sorted on `send_time` but there is a need to lookup a route that delivers the message the earliest (e.g., the earliest receive_time), the average complexity is $O(\frac{\|woops\|}{number\_of\_hosts})$ inside a routing list (which takes O(1) to find, because of the hash table). The worst case complexity is $O(\| woops \|)$, when all routes from the reference host have the first hop through the same intermediary host (e.g., host b is the only one that connects the reference host a with the rest of the world).

5.6.3 Maintaining the Table.

Once the initial routing table is populated, as hosts move in space and encounter other hosts, they learn more about other hosts' mobility and thus can enrich their knowledge base and expand their routing tables. As a reference host meets another host, the newly acquired information leads to the discovery of new woops. Running the entire path search algorithm is too expensive for this incremental motion profile learning. I identify the following possible scenarios:

New window of opportunity. Let's assume that host a meets host b and the result of motion profile information exchange is the discovery of a new window of communication opportunity. For example, assume a discovers that c and d will be able to talk in the in the interval (t_m, t_n). If both c and d are unknown to a, that means they are not reachable from a and therefore this information can be ignored. If one of the two is reachable from a, there must be a path that can take a message from a to some host h who can talk to c (assume c is reachable from a). In this case d is also reachable from a via a chain of nodes that reaches c. This means d is reachable from a the same way c is, plus the link c-d. This means that a's routing list for d is the same as the routing list for c, plus the changes induced by the link c-d, e.g., the number of hops is incremented by 1, and the delivery time is updated to indicate t_m. The operation takes $O(\frac{\|woops\|}{number\_of\_hosts})$ in average and $O(\| woops \|)$ in the worst case.

If both c and d are known and reachable from a, the appearance of a new window between them allows the hosts reachable from a via c *after* t_m to be reachable via c too and vice versa. As can be observed in Figure 5.14, the d host can be reached from c and the c host can be reached from d at t_m. Subsequently, a can reach all hosts d can reach after t_m via c (and vice versa). In Figure 5.14, the c-d woop is shown to be inserted in a's d routing list. Additionally, all grey records from a's c routing list are

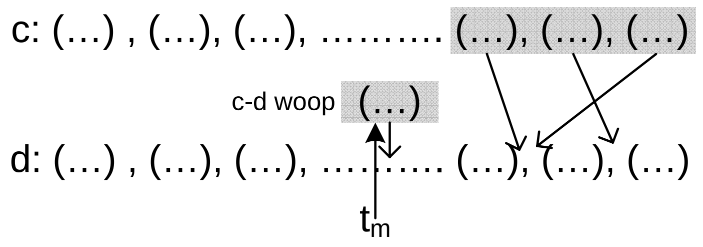

c: (...) , (...), (...), ... (...), (...), (...)
c-d woop (...)
d: (...) , (...), (...), ... (...), (...), (...)
tm

Figure 5.14: The new window between c and d is discovered to take place at tm.

added to the d routing list, as they are reachable from d. The needed adjustments are made at the insertion time (e.g., updating the number of hops). Although Figure 5.14 doesn't show it, a similar construction exists for inserting records from the d list into the c list.

The worst case complexity for this operation is O($\parallel woops \parallel log \parallel woops \parallel$), because the algorithm needs to consider each element in a routing list (after tm) and insert it (binary insertion) into the other ordered routing list. This worst case happens when the $c-d$ window comes early in the send_time-ordered lists such that there are many entries after tm and the routing table is unbalanced such that most/all entries are in the c and d lists. On average, the complexity is O($\frac{\parallel woops \parallel log \parallel woops \parallel}{number\_of\_hosts}$).

Merging two routing tables. As two hosts meet, everything reachable via one of them becomes reachable from the other one too. The operations described above repeat for all elements in a routing table. The complexity is the square of the complexity for a single window of opportunity, as described above, both for the average and for the worse case scenarios.

Deleting expired entries. As time passes, certain entries in the routing tables become obsolete (represent links that cannot be used anymore) and will be garbage-collected. The deletion expired entries entails the removal of the first entries in each list in the routing table, whose send_time value is older than the moment when the garbage collector inspects them. This yields an average complexity of O($number\_of\_hostslog \parallel woops \parallel$) - for each host (i.e., list in the routing table), the garbage-collector needs to search in the ordered list for the cut point. All records from that point to the beginning of the list can be collected.

5.7 Summary

In this chapter I have presented a novel routing protocol suitable for mobile ad hoc networks. While most ad hoc routing protocols today treat mobility as a source of challenges, I use additional context information in the form of motion profiles, to use host mobility to my advantage. This is a significant improvement over simply spreading the message to everybody and hoping that the intended recipient will eventually receive it. My algorithm controls each route, beyond simply identifying each host the message visits, but also indicating when and even where (physical location) the message is located. On-the-fly route updates are presented as a variation of the algorithm, where intermediary hosts may re-route a message in transit if they know a better path to the destination. This may have side effects and should be used with caution, especially if security is a concern.

Research dissemination. The chapter presented a basic algorithm that does the job but at a potentially exorbitant cost. An improved algorithm and a detailed complexity analysis are presented as well. The research presented in this chapter was in part published in *"Accommodating Transient Connectivity in Ad Hoc and Mobile Settings"* in the Proceedings of the Pervasive Computing Conference (Vienna, Austria, 2004).

Chapter 6

Coordination Across Time and Space

6.1 Introduction

Developing software for mobile devices is significantly more difficult than traditional software development that targets fixed hosts. Significant effort is spent dealing with issues not directly related to the application semantics, but rather induced by the target operating environment. The service oriented computing infrastructure described in Chapter 4 addresses issues specific to the distributed computing level of abstraction. How messages are sent from one host to another is addressed by the message routing layer and should not be a concern at the SOC layer. For this reason, the message routing layer (Chapter 5 takes care of managing the motion profile information, discovering routes, and forwarding messages. A coordination layer between the SOC layer and the routing layer provides a level of abstraction that offers the SOC layer a minimal interface consisting of powerful coordination primitives, while taking care of all the details related to implementing those primitives on top of the message routing layer and using the contextual information offered by the motion profiles.

To address the complexities of ad hoc networks and attempt to hide them from the application developer, coordination models stand in contrast to other alternatives such as enhancements to the operating system, or specialized languages, in that coordination models provide a clean separation between individual software components and the interactions within their overall software organization. Coordination can benefit the design, development, debugging, maintenance, and reuse of all complex distributed systems. Coordination models delivered in the form of middleware platforms provide high-level abstractions while leveraging the existing software infrastructure. When designed properly, middleware can divert attention from mundane concerns like protocol development, to more fruitful areas involving application-specific goals.

Coordination models designed for distributed computing in wireless ad hoc networks have been deployed in the past [80], [32], etc. Currently, coordination happens among processes on the same host, on neighboring hosts that are in direct communication, among processes on hosts members of a connected group, or among members of a reachable group. Connectivity beyond immediate neighbors is facilitated by ad hoc routing protocols which make the management of connected groups possible. A route is valid as long as all segments are connected along the entire path between two hosts. This makes distant coordination transfer transparent to multi-hop coordination.

However, not all mobility patterns offer the luxury of an uninterrupted multi-hop path between every two end points. Since hosts are mobile, time and space play an important role in their ability to coordinate. Previous coordination models do not account for time and space explicitly. While all coordination models decouple the computation from inter-component interactions, they restrict interactions to *now* and *here*. The coordination happens now because all hosts are "present" here, i.e., all involved parties are in contact when the coordination happens because they run on the same host, on nearby hosts, or are connected via a *continuous* multi-hop communication path. The coordination model I introduce in this chapter moves concerns of time and space management from the application level down into the coordination layer. The application may make calls that have to be executed at a certain time, at a certain location (physical area or host) but doesn't have to worry about how the calls travel to their destinations, nor does it have to keep track of time or handle scheduling explicitly so that each call is issued at the appropriate moment. This decoupling further enhances the separation between each agent's computation and the intercommunication between agents by managing the spatio-temporal coordination needs of applications. For example, my coordination model allows the applications to say "turn off the lights in the garden at 22:00" and the application will no longer have to be either available or present at 22:00 to issue the call. The application developer will thus focus more on the application semantics related to the time and space, and less on the low level mechanisms needed to deliver them.

To accomplish this, I make time and space explicit parameters of the coordination primitives exposed to the application programmer, while hiding the complexities introduced by the dynamic character of the mobile ad hoc networks. Particularly, I consider that all relevant hosts are reachable from (but not necessarily connected to) the reference host from the time and place where, at the moment when the coordination procedure has to be invoked, to the user-specified time and place where the coordination should take place. For reachability, I include the possibility of a host sending a message to another host. In most cases this is synonymous with having a route from source to destination. While a route is generally considered to be an uninterrupted communication path between the two end points, I focus on a particular type of route, that may not be fully connected at all times, as described in Chapter 5. For example, Mom can tell Alice to ask Bob to turn off the light in the garden at 22:00, after he walks the dog. Mom can talk to Alice *now* and *she knows* Alice *will talk* to Bob who *will walk* the dog in the garden at 22:00. Mom doesn't have to talk to Bob directly, doesn't have to go to the garden to talk to Bob (or turnoff the lights herself), and she doesn't need to wait until 22:00 to send the request to Bob. All these are possible because Mom knows that Alice, who is in the same room now, is on her way to Bob's room where she will meet Bob and that Bob will go to walk the dog in the garden at 22:00. The path from Mom to Bob is a disconnected path. Alice is "connected" to Mom, picks up the message, then "disconnects" from Mom and later "connects" with Bob and delivers Mom's message. My model thus acknowledges time and space as important elements of the coordination mechanism in a mobile ad hoc network. It provides coordination among hosts that are reachable but not necessarily through direct or even transitive immediate connection.

6.2 Motivation and Contribution

Applications that have a spatio-temporal dimension span a wide variety of domains. Generally speaking, any application that runs on a mobile host and interacts with another (mobile) host, or even a distributed application whose components run on different hosts and need to communicate, can leverage the coordination model to be presented below.

Coordination middleware facilitates application development by providing high-level constructs such as channels [78], events [7], blackboards [27], or tuple spaces [37], in place of lower-level constructs such as sockets. Tuple spaces and blackboards are both shared-memory architectures in which nodes may insert and remove data. Tuple spaces differ from blackboards in that they use pattern-matching for retrieving data; in a blackboard, the data is stored in a global database and is generally accessed by type alone. Channels are similar to sockets in that data is inserted at one end and is retrieved from the other. They differ from blackboards or tuple spaces because the latter two provide persistent storage.

Channels and events are the main means of interaction for remote entities using *direct coordination.* Direct coordination relies on message passing and directly identifies the collaborating partners. This presents several drawbacks in mobile wireless networks where repeated interactions require a stable network connection and therefore are highly dependent on network reliability. Examples of such coordination models are Sumatra [3] and Odyssey [85]. They exhibit a typical client-server model that can be used for all sorts of interactions, while requiring a very precise specification of the communication protocol. Agent TCL [43] and Ara [91] exhibit a rendez vous behavior by establishing synchronous meetings (Agent TCL also supports asynchronous messaging). Event-based *direct coordination* can be considered to be a particular type of channel coordination where entities send and listen for events that control their coordination. Examples of such models are JEDI [22] and IWIM [7]. Event-based coordination models offer a publish-subscribe paradigm where nodes interact by exchanging events through a logically centralized event dispatcher. Nodes subscribe to events using various representations of the event (e.g., regular expressions on the event name). More details on event-driven coordination can be found in [70] and [79].

Backboard or tuple space coordination are forms of *indirect coordination.* Tuple space associative coordination is probably the most popular approach for wireless mobile environments. I only mention a few from the large set of tuple space coordination models. MARS [15] provides logically mobile agents that migrate from one node to another. Each node maintains a local tuple space that is accessed by agents residing on it. An agent can coordinate only with agents that reside on the same node and thus agent migration is required for inter-node coordination. Lime [80] entails logically mobile agents executing on physically mobile hosts. Lime offers a group membership paradigm where neighboring hosts form a group that shares one or more logically centralized Linda-like tuple spaces. Reactive programming allows the system to notify an agent when a particular tuple is in the tuple space, which eliminates the need for polling. Other models that deliver tuple space coordination are Limbo [25], TuCSoN [84], Jada [18], etc. *Indirect coordination* achieved through blackboards or tuple spaces has the distinctive feature that it decouples in time and/or space the agents that coordinate one with the other. For example, in Ambit [16], an entity attaches a message to a system and another entity can retrieve and read the message without associative access. In ffMAIN [90] agents interact via information spaces using messages identified by a unique key, also with no associative mechanism involved.

While some of these models are applicable to ad hoc networks, their coordination capabilities have a certain scope. We can observe how the scope of coordination can range from the local host (e.g., in MARS) to the vicinity of the current host defined by direct connectivity (peer-to-peer 1-hop communication) to other hosts (e.g., in Lime) or the immediate transitive closure of direct connectivity, spanning a subset of hosts connected via multi-hop paths.

Consider the example illustrated in Figure 6.1. On the westbound side of highway I44 there is an accident that blocks traffic completely. Given that the accident alone blocked all traffic lanes (due to debris, oil/gas spills, and/or injured people that need on-site medical assistance), the closing will take a significant amount of time to clear. Meanwhile the traffic backs up behind the accident as cars approach the accident site. Once on the highway, these cars to not have any other option but to wait for the road to reopen. As more cars add to the road block, police, ambulance and towing vehicles have an even harder time trying to reach the accident.

Fortunately, a traffic monitoring application that runs “on streets” can help reduce the overall cost of the accident. The application involves vehicles in traffic and roadside smart devices such as intelligent display panels or a smart scheduler embedded in the traffic lights of an intersection. This distributed application may run all over the

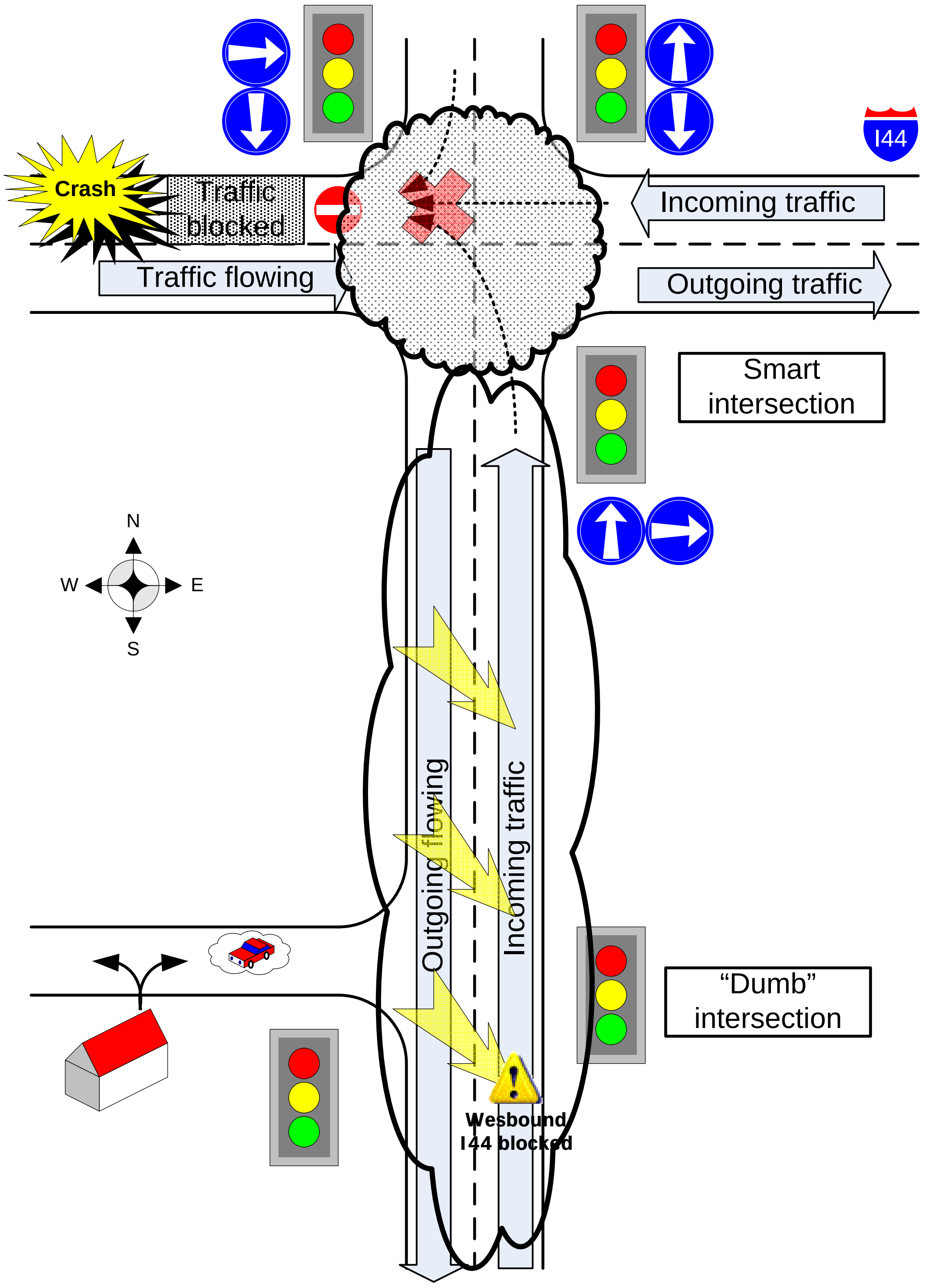


Figure 6.1: Traffic jam and smart intersection

city, providing drivers with traffic information similar to the traffic updates broadcast every 10-15 minutes on radio stations. Eastbound traffic on I44 “sees” the accident and delivers adequate notification messages to the smart intersection and beyond, to incoming traffic, warning vehicles not to enter the highway. The smart intersection will update the traffic light schedule and divert the traffic away from the blocked stretch of highway. The vehicles passing through the smart intersection will pick up the warning and spread it away from the smart intersection, warning all incoming traffic such that a vehicle going North with the intention to continue West on I44 will be informed about the accident and can make a left turn on the last street before the highway.

At the same time, Joe, who lives in the house in the lower left corner of Figure 6.1, has to decide every morning when he leaves for work on the route to follow to his office. Do do this, he needs information about traffic in certain areas, at specific moments (time interval(s)) in the morning, prior to his departure. While he has a preferred route, due to traffic jams he may have to choose alternate routes and make decisions several times on the way to office, similar to the one he makes when he leaves the driveway. His preferred route would have him turn right and then left to go North on 141 to take I44 West. The application that draws Joe’s route each morning can also calculate alternate routes if needed, based on traffic flow information it receives from. The application performs a path search in a weighted graph, where the weights of edges are dynamically updated to reflect the flow of traffic during the time interval of interest, in the area of interest. The values are input data for the path search application which should not be concerned with the mechanics by which these values are obtained, as the the mechanism has little in common with the application semantics. The coordination layer underneath the application layer should expose a high level interface which offers the application developer powerful constructs that allow him to focus on the application semantics, while hiding the low level implementation details from him. To do this, the coordination layer needs contextual knowledge that helps proactive interaction scheduling.

The spatio-temporal aspect of information retrieval (and dissemination) can be abstracted from the application and encapsulated as a novel capability of the coordination model. To provide this I use additional contextual information that allows us to reach and interact with other hosts. Information about host motion profiles helps me develop the spatio-temporal dimension of the coordination model. In the first (simpler) example, Mom *knows* that Alice, who is now *next to* Mom, *will soon be next to* Bob and therefore Alice could relay a message from Mom for Bob. Thus, Mom *reaches* Bob through Alice. The communication route Mom-Alice-Bob is a disconnected communication path because when Alice is “connected” to Bob, she is no longer “connected” to Mom. The information Mom has about Alice’s mobility and Bob’s whereabouts is enough for her to determine that she can send Bob a message.

The notion of reachability over disconnected routes allows one to define a set of hosts with which a reference host can interact and it also allows reasoning about moments and/or places where interactions can take place. Previous coordination models have used similar knowledge in a much more simple form. “Reachability” was defined using the implicit knowledge that the processes run on the same host or on neighboring hosts that can interact in a peer-to-peer manner. Ad hoc routing helped to extend transparently the concept of reachability to a set of transitively connected hosts. For example, for simple forms of data delivery, a one-way route is enough, while for reading some data a looping path is needed (a forward path to carry the request and a return path for the result). In the next section I present the coordination model I developed on top of context-aware disconnected routes.

6.3 Spatio-Temporal Coordination

In this section I present CAST, a Linda-like spatio-temporal coordination model. I assume a set of hosts, each with one active entity (an *agent*) and a local tuple space. For the sake of simplicity, I will assume that only one tuple space exists per host, although multiple tuple spaces identified by different names can co-exist without affecting the coordination model. The same goes for multiple agents on a host. All local tuple spaces form a universal tuple space, even if they are not all connected. The universal tuple space is simply the collection of all (potentially isolated) local tuple spaces.

Coordination operations are performed by a the agent on a host on the tuple space. Operations can be performed on the tuple space, which makes them insensitive to the network topology. They can also be performed on the local tuple space of another host or on a projection of the universal tuple space defined by hosts that are reachable from the host that initiates the operation. A reachability relation is defined over disconnected communication routes, and is specific to each type of operation according to the mechanics needed to implement the operation. Different operations exhibit different patterns of interaction between hosts. For example, for a simple message delivery, destination host b needs to be reachable from source host a. If a confirmation for delivery is needed (or the result of a remote method invocation, for example, is expected at the initiator), host b needs to be reachable from host a *and* host a needs to be reachable from host b *after* b received the message from a (a loop in space and time representing a round trip of a message from source to destination and back).

In order for a host to use this kind of reachability relation, it needs to have information about its own motion profile and the motion profiles of other hosts with which it develops such relations. This knowledge about host mobility helps discover both connected and disconnected routes among hosts. These motion profiles are assumed to be accurate, even if they are known only for a limited interval into the future. The extent to which a host can take advantage of these relations depends on the amount of knowledge it has about its own as well as the other hosts' evolution in physical space. Given the extension of these motion profiles in the future, the reachability relations can also be extended in the future. Additionally, the reachability relations can be defined between a source host and a target area (i.e., hosts that are or will be in a designated physical space).

The spatio-temporal dimension of my coordination model also impacts the tuples in the tuple spaces. My tuples can have a precisely defined lifetime. They can also be bound to certain hosts (i.e., once they are written into some host's local tuple space, they stay there regardless of how their carrier host moves) or to a specific location/area (e.g., they try to remain in a designated region by migrating from one host to another if their current host is leaving the target area of interest).

6.3.1 Mobile Tuples

Tuples are arrays of objects that have types and values. Coordination is mediated by tuple spaces, the tuples being the means by which hosts interact. A host can write, read, or remove one or more tuples in/from the tuple space. In traditional tuple space models, deployed mostly for environments where access to the tuple space is not an issue, a host places a tuple in the tuple space (which can be easily managed at a centralized location) and all other hosts have access to it (ignoring potential security concerns that might filter the access to a tuple). In more recent models, such as Lime [80] and Limone [32], the tuple space is implemented in a distributed fashion. Each host has its own local tuple space which is shared with other hosts to which the reference host is connected (peer-to-peer or over multi-hop routes). These local tuple spaces are logically merged into a global federated tuple space. A tuple in a local tuple space is accessible to all hosts that participate in the federated tuple space. By default, a tuple is written in the local tuple space of the host that produced it, until it is removed as a result of a coordination operation. It is possible for the producer host to specify an explicit destination different from the local tuple space for the tuple, and in that case the tuple will migrate to that destination upon being written to the tuple space. The tuple only becomes available for coordination when it has reached its destination. Such a tuple is bound to the host where it was deployed. It remains on that host until a coordination operation removes it explicitly.

In CAST, the universal tuple space is a union of (potentially disconnected) local tuple spaces. Local tuple spaces that reside on (peer-to-peer or transitively connected) hosts form federated tuple spaces, restricted to islands of connectivity. Coordination can occur among hosts that share a federated tuple space but it can also occur among hosts that will never come in direct (peer-to-peer) or transitive (over multi-hop fully connected routes) contact.

While traditional tuples are still available in CAST, my model also exhibits new types of tuples. The tuples in my model are enriched with spatio-temporal qualifications that describe the mobility, lifetime, and the location of each tuple. These qualifications are defined by the parent host of a tuple, when it writes the tuple to the tuple space. The spatial qualifications determine the location where a tuple is written, as well as the dynamics of the tuple during its lifetime. Based on their spatial qualifications, tuples can be **static** or **dynamic** (the details are presented below).

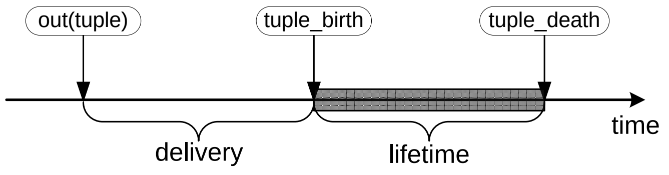


Figure 6.2: The temporal qualifications of a tuple.

The temporal qualifications of a tuple define its temporal existence and entail two parameters in the time domain: tuple_birth and tuple_death. The tuple is required to be delivered prior to tuple_birth and is available for coordination in the time interval [tuple_birth, tuple_death], unless a coordination operation removes it during its lifetime. Even if the tuple is delivered prior to tuple_birth, it stays invisible in the tuple space and therefore not accessible for coordination until tuple_birth. At tuple_death, the tuple is removed from the tuple space and destroyed so it is again not available for coordination (Figure 6.2).

Static Tuples

Static tuples are bound to the hosts where they are written. As was mentioned before, when written to a tuple space, tuples can migrate at parent's request to another host's local tuple space. This assumes that the parent of the tuple knows the name or ID of the destination host and can address the tuple accordingly. CAST tuples can also migrate to a specific physical location: CAST supports the migration of tuples to a specific area, regardless of which hosts may be present in the designated area. The motion profile analysis can reveal the path to a host that is in or will enter that region. Combined with the tuple lifetime explicitly specified by the tuple's creator, the motion profile analysis and tuple delivery can be optimized to target the host that spends as much as possible of the tuple's life in the designated area of interest. When delivered to a specific host, once the tuple reaches the destination host it stays there until it is removed by a coordination operation or until the end of its defined lifetime (until tuple_death or until the host leaves the designated area, whichever occurs first), and will *not* relocate to a different host that continues to stay in the designated area.

Dynamic Tuples

CAST also supports space-constraint tuples. These tuples are delivered to a certain physical location and will do their best to stay within the designated area for as long as they are alive, which gives them a dynamic character. While a host may leave an area before the tuple is scheduled for destruction, the tuple may hop to another host that will stay within the designated area longer. If there is no such host to pick up the tuple, it will simply disappear when the last host that could have had it leaves that area (note that while there may be hosts in the target area after a reference host leaves, they may not be reachable from the host that needs to leave a tuple behind and therefore the tuple cannot be transferred).

The spatio-temporal dimensions of the tuples presented so far can be found in the coordination primitives the model exposes and are presented in the following sections. Before I present the coordination primitives provided by my model, I first introduce the notion of reachability, which helps me define the semantics of the said operations.

6.3.2 Reachability

It is essential for a host to be able to reach another host for coordination to take place (or any other interaction for that matter). Therefore all coordination models entail a certain notion of reachability, addressed in different ways. For simplicity, I consider the coordination to happen among hosts, assuming one agent per host. In MARS [15], an agent is said to be "reachable" and the reference agent can coordinate with it only if they both run on the same host. In Lime [80] or Limone[32] "reachable" means that agents run on hosts in direct (peer-to-peer) contact. This extends transparently to hosts in transitive contact, where an ad hoc routing algorithm establishes a multi-hop path between hosts, entailing mobile hosts that bridge the communication between the two ends involved in coordination. It can be observed that the notion of reachability is in strong correspondence with the existence of a communication route between the entities involved in the process of coordination (be it a 0-hop route - agents on the same host; or a 1-hop route - peer-to-peer communication - or multi-hop path; transitive connectivity bridged by a chain of hosts in contact that can build a fully connected path).

Figure 6.3 shows hosts a through d and the transient connections they establish as they roam in space and come within each other's communication range (e.g., a and b can communicate directly from t_1 to t_2, from t_3 to t_4, etc). While a and c or d are never in direct or transitive contact, they are in *disconnected transitive* contact. The example depicted in Figure 6.3 will be used as the reference example for all remaining discussions in this paper.

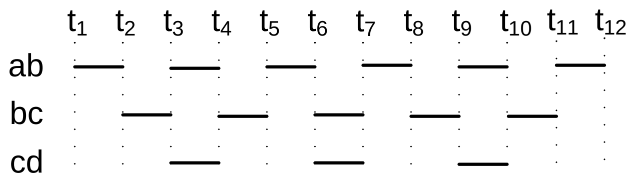


Figure 6.3: P2P interactions between a, b, c, and d.

The windows of peer-to-peer interaction opportunity depicted in the figure above are extracted by each host based on the analysis of the set of motion profiles each host knows. Figure 6.4 shows how each host learns the motion profiles of the others, based on the mobility pattern depicted in Figure 6.3. While motion profiles can have a certain extent (i.e., can describe the future movement for a certain interval in the future), for this example I assume all four motion profiles are defined until at least t_{12}. Obviously, each host can use only the motion profiles known at a certain moment, even if they terminate at different moments for different hosts. For the example used in this paper all hosts learn the motion profiles of the others until at least t_{12}.

As shown in the table in Figure 6.4, a and b exchange motion profiles at t_1 when they meet for the first time (each column in the table shows how a host learns the motion profiles of other hosts.) At t_3 they meet again and exchange motion profiles. Meanwhile, host b met host c and learned its profile at t_2. At t_3, a learned c's profile from b. By t_5 each host had the opportunity to learn the motion profiles of all others, at least until t_{12}. I assume the motion profiles are accurate and take an insignificant

	a	b	c	d
t_1	ab	ab	c	d
t_2	ab	abc	abc	d
t_3	abc	abc	abcd	abcd
t_4	abc	abcd	abcd	abcd
t_5	abcd	abcd	abcd	abcd

Figure 6.4: Motion profile information dissemination.

amount of time to transfer between hosts, compared to the period of time a pair of hosts is connected during an interval of direct connectivity.

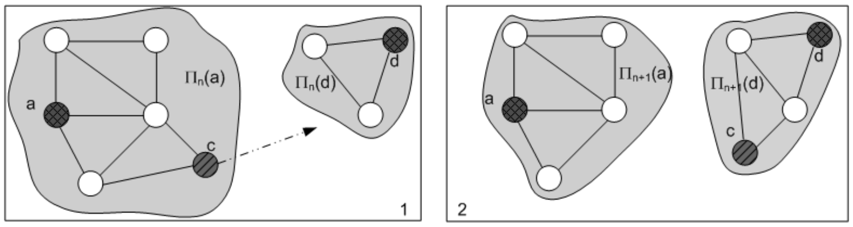


Figure 6.5: 1. Hosts a and d in their disjoint configurations at moment n. Host c is connected to a at this time. 2. At moment n+1 host c is no longer in (transitive) touch with a but it is with d.

At moment t_n, a host a is in a configuration $\Pi_n(a)$, where it is connected directly (P2P) to some hosts and indirectly (over multi-hop fully connected paths) to other hosts. To a host d, the configuration $\Pi_n(d)$ represents an "island of connectivity" at time t_n (Figure 6.5).

Two hosts a and d are defined to be in relation ρ_n if the configuration $\Pi_n(a)$ contains a host that will be part of configuration $\Pi_{n+1}(d)$. This means the host that bridges $\Pi_n(a)$ and $\Pi_{n+1}(d)$ could deliver a message from a to d (host c in the example in Figure 6.5).

$$a \ \rho_n \ d \ \equiv \ (\Pi_n(a) \ \cap \ \Pi_{n+1}(d) \ \neq \ \phi) \tag{6.1}$$

This means that d is *one-reachable* from a (at most one configuration change is needed). A *multi-reachable* relation can be defined formally as:

$$\mathsf{a}\ \overline{\rho_\mathsf{n}}\ \mathsf{d}\ \equiv\ (\ \exists\ \mathsf{c}_0,\ \mathsf{c}_1, \dots\ \mathsf{c}_\mathsf{k},\ \textit{such that}\ \mathsf{c}_0 = \mathsf{a} \\ \wedge\ \mathsf{c}_\mathsf{k} = \mathsf{d}\ \wedge\ \mathsf{c}_\mathsf{i}\ \rho_\mathsf{i}\ \mathsf{c}_{\mathsf{i}+1}) \tag{6.2}$$

Note that $\mathsf{a}\ \overline{\rho_\mathsf{n}}\ \mathsf{d}$ does not imply $\mathsf{d}\ \overline{\rho_\mathsf{n}}\ \mathsf{a}$. Also, $\mathsf{a}\ \overline{\rho_\mathsf{n}}\ \mathsf{b}$ and $\mathsf{b}\ \overline{\rho_{\mathsf{n}+\mathsf{k}}}\ \mathsf{c}$, implies $\mathsf{a}\ \overline{\rho_\mathsf{n}}\ \mathsf{c}$. It is always true that $\mathsf{a}\ \overline{\rho_\mathsf{n}}\ \mathsf{a}$.

Using the reachability notion defined in (6.1) and (6.2), I will define the semantics of tuple space operations offered by my coordination model. Different operations on the tuple space have different patterns of interaction with the tuple space (i.e., the host that issues the operation may execute different protocols involving one or multiple other hosts, depending on the type of operation). For each type of operation, from motion profile analysis and inter-host patterns of interaction, I build an acquaintance list. I populate this acquaintance list with hosts that could participate in that type of operation, initiated from the current host. For example, assume an operation entails the current host asking another host for an integer value, receiving that value, and then sending back the integer multiplied by 2. This means the remote host has to be 3-reachable from the current host: the current host needs to send the request; then it needs to obtain the integer; then it needs to send the doubled integer back. Note that n-reachability entails n trips between two hosts and that each of the n trips can start only after the previous one has ended (they are serialized in time).

Given the *n-reachability* problem (different hosts have different values for n with respect to the current host, values that can change over time), each operation can be described by a specific reachability constraint. That operation can be performed by a current host only on other hosts that exhibit an n-reachability from the current host, where n is a minimum imposed by each operation (n=3 in the example above where I double the integer). Based on the n-directional reachability specific to each operation, each host builds and maintains an acquaintance list that contains hosts that can participate in specific types of operations from the current host's perspective. Thus, as along as a host d is 3-reachable from host a, d will be in a's *double-integer-acquaintance-list*:

$$\mathsf{d} \in \mathsf{AQ}_{\mathsf{double\_integer}}(\mathsf{a}) \tag{6.3}$$

This list is dynamically updated, as more knowledge can be extracted from the motion profiles (which are updated continuously at run time). While the maintenance of such acquaintance lists may seem expensive, it is the goal of a coordination model to offer a small yet powerful set of primitives. As we will see below, my coordination model requires a maximum reachability of 3 for the primitives it offers.

6.3.3 The OUT Operations

The Linda out operation places a tuple in the tuple space. My general form for the out operation is:

```
out(Tuple tuple, Mode mode Time tuple_birth,
                     Time tuple_death, Target target);
```

The default values for these parameters are "now" for tuple_birth and "never" for tuple_death, and "current host" for target. These defaults help to emulate a traditional tuple which is active as soon as it is placed in the tuple space, does not expire, and stays in the local tuple space of the host that created it until it is explicitly removed.

The mode and target parameters affect tuple delivery. The target is a set of constraints or properties the target host(s) need to have to be considered for delivery. By default, the value of the target parameter is "current host", meaning that the tuple is written in the local tuple space of the host that created it. The parameter can also identify a set of hosts based on the enumeration of their IDs or by describing a geographic restriction of their physical location. The values accepted for mode are anycast or multicast. Anycast delivery ships the tuple to any *one* host that satisfies the constraints (if a specific host ID is specified, the delivery is equivalent to unicast). Multicast delivery ships the tuple to *all* hosts that satisfy the filter constraints. The default value is multicast (note that this value combined with the default value for target still identifies only one host, the current host). In all cases, the selection process applies only to hosts that are reachable from the parent of the tuple (are in the *out-acquaintance-list* of the caller).

As was mentioned before, a host a is always reachable from itself, which is why a local out is always successful. When the target information identifies anything but the local host, the tuple needs to migrate to the final destination. The final destination has to be 1-way reachable from the tuple producer.

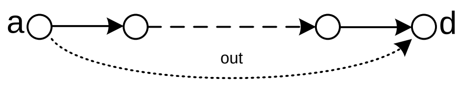


Figure 6.6: A one-way path is needed for a successful out from a to d.

In Figure 6.6, host a can write a tuple to host d if d is in a's *out-acquaintance-list* at the moment when the operation is issued:

$$(\mathsf{d} \in \mathsf{AQ}_{\mathsf{out}}(\mathsf{a})) \equiv (\mathsf{a}\ \overline{\rho_{\mathsf{n}}}\ \mathsf{d}) \tag{6.4}$$

If the tuple is written to a remote location, then the tuple is migrated as soon as a disconnected path is found to the desired destination. CAST enables access to the tuple in its destination local tuple space at the end of migration or at tuple_birth, whichever comes last.

If the tuple is a dynamic tuple, the target is a region, not a particular host. Depending on the mode parameter, the destination is *one* host chosen non-deterministically among the hosts in $\mathsf{AQ}_{\mathsf{out}}(\mathsf{a})$. Policies that bias the selection can be easily implemented, e.g., pick the host that spends most of the tuple's lifetime in the the designated area, or *all* hosts from $\mathsf{AQ}_{\mathsf{out}}(\mathsf{a})$ in the designated area (these hosts can forward copies of the tuple to other hosts in the designated area which couldn't be reached by the source host of the tuple). As these hosts move around, when they leave the designated area they delete the tuples bound to that space and as they (re-)enter the area will receive copies of the tuples bound to that space.

A variant of the out operation is the outg operation. The semantics are the same with the only difference that outg writes multiple tuples at the same time, as opposed to only one tuple. All tuples have the same lifetime, the same mode, and the same target.

6.3.4 The RD Operations

The Linda rd operation reads from the tuple space a tuple that matches a given template. If no such tuple is found, the operation blocks the caller until a matching tuple appears. My general form for the rd operation is:

```
rd(Tuple template, Time op_start, Time op_stop,
                    Time op_report, Target target);
```

The template represents a description of the tuple the operation is intended to return. A template is an array of fields, like a tuple, each field containing a description of the tuple field that is expected to be matched. The comparison is made pairwise, each field in the template being compared against the corresponding field in the tuple. For example, the tuple in Figure 6.7 matches the template below it. The template requires that the first field in the tuple is of type Integer and has the value 25, which matches exactly the first field in the tuple. The second field is required to be of type String, but the value of that field is not enforced by the template. The second field in the tuple matches this request, as it is of type String and has the value "Boat". The third field is not important from the tuple's perspective. This is implemented using Java's Object.class which works as a wildcard that matches any type and does not enforce any value.

<Integer(25), String("Boat"), Location("Pacific") >

<Integer(25), String.class, Object.class >

Figure 6.7: A tuple (top) and a template (below) matched by the tuple.

The template has to be explicitly specified in the method call, with all other parameters having default values. The op_start and op_stop parameters define the lifetime of the call. After the call is issued, it becomes effective only at op_start and searches for a matching tuple until op_stop. The rd operation is synchronous. This means that it blocks the agent that issued the operation until it returns. The process that invokes the rd operation is blocked as soon as it issues the operation, even if the operation is scheduled to be activated later (e.g., there's some time between when the operation is issued and op_start). The process is unblocked when the operation finds and retrieves a matching tuple or at op_stop (when the operation returns NULL). The default values are "now" for op_start and "never" for op_stop, which help emulate the traditional Linda rd operation which blocks the caller process until the tuple is found (potentially forever). The op_report parameter defines the time limit until which the reference host is willing to wait for results. The value of op_report has to be greater than the value of op_stop. Even if this is the case, it may be possible that the last opportunity for a target host to send back a message that reaches the reference host before op_report is before op_end. If this opportunity is missed, the result from this target host's perspective will be null, even if a tuple does become available before op_end.

The target parameter filters the hosts to which the call is addressed in a manner similar to the target parameter used with the out operation. The set of target hosts verify a certain property such as have certain IDs, or be at a certain location. By default, the value of the target parameter is "all hosts," which means "no restriction." The rd can be bound to a set of hosts, which makes it a *static* rd, e.g., I'm only interested in the tuple that describes the temperature of a particular host, or to a specific physical location, which makes it a dynamic rd, e.g., I need to read the temperature in a certain room, no matter who is there to service my call. For a *static read* operation, the set of hosts targeted by the call is known at the moment the call is issued (it is an identifiable subset of the hosts in the *rd-acquaintance-list* of the reference host at the moment the call is issued). Thus, the reference host can compute when the last reply will come back from the target hosts and thus set the default value of the op_report parameter large enough to include that last reply message. For a *dynamic read* operation, the set of target hosts is not known a priori by the reference host. Furthermore, the set of target hosts changes dynamically, while the read operation is active, as hosts enter or leave the designated target area. The default op_report value in this case is set to op_stop plus some constant.

The rd operation does not have a mode parameter. The operation is forwarded to all hosts (from the *rd-acquaintance-list*) that pass the target filters. If there is only one such host, the operation has *unicast* semantics. This means the call has only one target host where it is sent and from which a result is expected. If multiple target hosts pass the target filter, the operation has *anycast* semantics. This means the operation is forwarded to all those target hosts but only one result is expected in return. Any host can send back a matching tuple. The first tuple that reaches the reference host is returned to the application and the others are ignored. My model does not support *multicast* semantics as the rd operation returns only one tuple by definition, while multicast entails at least one tuple from each target host.

The operation blocks until a target host returns a matching tuple, or until the operation expires. When the first result tuple (maybe the only one) comes back, the operation forwards it to the application and discards the others (if any). As there may be hosts that still look for matching tuples after another host has returned a result, the rd requests they have received will be deleted if (1) their lifetime elapses, or (2) the last opportunity for the target host to report back a result has passed (i.e., there is no known path back to the reference host in the known future), or (3) the caller requests the call to be canceled. After obtaining a matching tuple, the initiator sends cancelation requests to hosts that have not responded yet. The presence of a target host in the *rd-acquaintance-list* of the initiator guarantees the return of a result (for a period of time, if found) but does not guarantee the delivery of the cancelation request. This request can be sent opportunistically and only helps where it is possible to terminate rd operations on those hosts that may need more time to delete it under circumstances (1) or (2).

The interaction pattern between two hosts participating in a rd operation entails a round trip from caller to the target and back. The first segment of the trip (from caller to the target or the rd operation) carries the request for the tuple (i.e., the rd operation and its template parameter). The return segment of the trip brings back the result (if multiple results come back from multiple hosts, the caller retains the first of them and discards the others - as rd returns a copy of the tuple, the acceptance

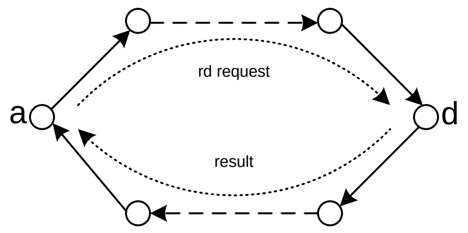


Figure 6.8: A round trip path is needed for a successful rd from a to d.

or the rejection of a tuple as a result of a rd does not affect the other hosts). Given the hosts' mobility, only hosts that can be part of such a round trip can be considered as potential targets. Each host maintains a *rd-acquaintance-list* with hosts that can participate in a rd operation initiated by the reference host. The filters specified by the target parameter are applied to the following set of hosts.

$$(\mathsf{d} \in \mathsf{AQ}_{\mathsf{rd}}(\mathsf{a})) \equiv (\mathtt{a}\ \overline{\rho_{\mathtt{n}}}\ \mathtt{d}) \wedge (\mathtt{d}\ \overline{\rho_{\mathtt{n+k}}}\ \mathtt{a}) \tag{6.5}$$

For example, a rd with all parameters set to their default values will be forwarded to all hosts that are in the current host's *rd-acquaintance-list* at the moment the operation was issued, will be host-bound, will be effective immediately, and will never expire. A host that is the target of a rd operation will be subject to the call for as long as a potential result could return to the reference host. Location-bound operations are forwarded among hosts as they enter/leave the designated region. If at some moment the last chance for a message to make a trip back to the reference host is lost (e.g., knowledge about motion profiles does not indicate the availability of a disconnected path in the future from a target host to the reference host), the target host will discard the rd operation. Thus, "never" is limited to the known near future. If there is no known chance for the host to send back the tuple, there is no reason for the target host to keep looking for the tuple on behalf of a beneficiary that cannot be reached anymore. While the reference host may become reachable again in the future, if the momentary knowledge does not reveal this detail, the operation is dropped at the expiration of the last known route back.

A variant of the rd operation described above handles groups of tuples. While rd retrieves one of the tuples that match its template, rdg returns all matching tuples from a single host. This restriction is needed for consistency reasons. If I send the request to multiple hosts *and* expect answers back from *all* of them it means I need to keep the reference host blocked for as long as the operation is alive and collect responses from all target hosts, as matching tuples become available. The operation would unblock and return the control to the application that issued the call only after that. This leads to inconsistent semantics as the result of such a rdg does not represent the state of the system at a certain moment in time. A host could reply immediately and send back a number of tuples while another target may need to wait for a while before it finds any tuple to report. Meanwhile, the host that reported first may have already removed some of its matching tuples. Thus, the result would be a union of snapshots collected at different moment, but which appear as a consistent and uniform result to the reference host. The snapshots *logically* capture a plausible state of the system at a single moment, while in reality the system may have experienced transformations during the collection of the results. The rdg operation can therefore be sent to only one host. Both *unicast* and *anycast* semantics are available. The call is synchronous. It blocks until op_end or until at least a matching tuple is found. If matching tuples are already available in the target tuple space, all matching tuples are returned. Once a set of tuples is returned, the operation terminated (i.e., does not continue to search for tuples until it's lifetime expires in an attempt to return more tuples - same consistency reasons described above). If no matching tuple is found by op_end, the operation returns NULL.

Modern coordination models inspired by the traditional Linda model (e.g, Lime, Limone) also offer a probe variant of the traditional rd Linda operation. This variant, known as a *read probe* (rdp), is a non-blocking read that returns immediately a matching tuple is available or NULL otherwise. My model does not offer a rdp operation explicitly. Since my rd operation has a very well defined lifetime, at op_stop the operation returns NULL if no tuple is found. If I make this op_stop parameter equal to the op_start parameter, my rd operation behaves like a rdp that probes the tuple space at op_start. For this reason, I do not need to provide a separate rdp operation explicitly.

6.3.5 The IN Operations

The Linda in operation reads and removes a tuple from the tuple space. This operation is also synchronous, i.e., it blocks the reference host until a matching tuple becomes available. The only difference from rd is that in also removes the tuple. My general form for the in operation is:

```
in(Tuple template, Time op_start, Time op_stop,
                    Time op_report, Target target);
```

The interaction pattern between two hosts participating in an in operation entails a request message from the initiator, a response message back from the target host and an acknowledgement message from the reference host to the target see Figure 6.9. Potential targets that could run this *3-reachable* protocol are stored in the *in-acquaintance-list*, managed by each host locally. All additional filters specified by the target parameter are applied to this set of hosts.

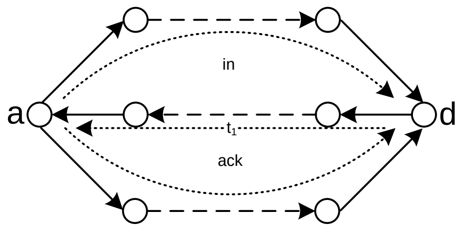


Figure 6.9: Three trips are needed for a successful in from a to d.

The three paths are needed for the following reason: suppose a host a issues an in operation that spreads to hosts *b*, *c*, and d, which are in a's *in-acquaintance-list* (Figure 6.10). Differently from the rd operation (where the acceptance of some host's tuple and the rejection of the other hosts' tuples did not impact the hosts that provided the tuple, in this situation, the in operation has to remove the tuple from the host that provided it and therefore it matters which host's tuple is accepted). All target hosts that have matching tuples report back the tuple availability. They all

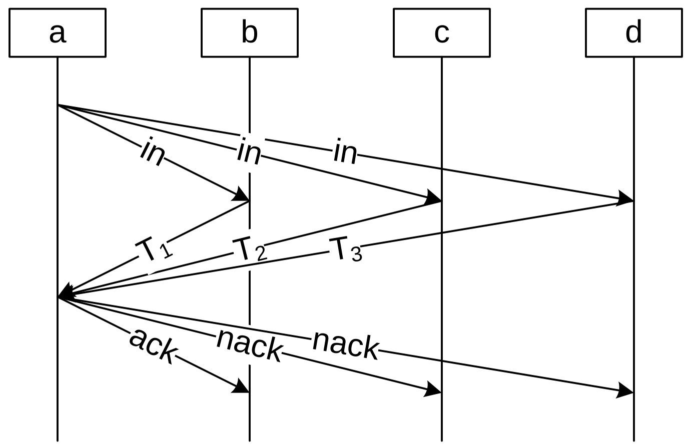


Figure 6.10: Acknowledgements and negative acknowledgements.

delay access to their respective tuples by all other coordination operations that might use those tuples (they essentially hold the operations until a decision can be made about whether the tuple is or is not available to those operations). When a receives notifications from hosts b, c, and d, a will pick one of them (the first one) and send a positive acknowledgement to the host that sent the chosen notification (e.g., host b) and negative acknowledgements back to all other hosts. When c and d receive the NACK they unlock the operations that targeted the same tuple and allow them to access those tuples in the tuple space (as they know a chose some other host's tuple). If such calls are on hold on b, they will continue their normal behavior as if the tuple wasn't there (remain blocked or return NULL, etc).

$$(\mathsf{d} \in \mathsf{AQ}_{\mathsf{in}}(\mathsf{a})) \equiv (\mathsf{a}\ \overline{\rho_{\mathsf{n}}}\ \mathsf{d}) \wedge (\mathsf{d}\ \overline{\rho_{\mathsf{n+k}}}\ \mathsf{a}) \wedge (\mathsf{a}\ \overline{\rho_{\mathsf{n+k+l}}}\ \mathsf{d}) \tag{6.6}$$

Both *unicast* and *anycast* semantics are available depending on the target parameter, similar to the rd case. The in operation also has a group variant, called ing, with semantics similar to rdg. A *probe in* can be achieved the same way I described the *probe read.*

6.3.6 Reactions

The coordination primitives presented in sections 6.3.3, 6.3.4, and 6.3.5 are all synchronous coordination mechanisms, i.e., the process that issued a call is blocked until a response comes back. For a rd operation, the caller is blocked until one of the target hosts sends back a matching tuple (if the expiration time of the call is "never"), or until the time expires without finding a tuple, when the result is NULL. The calling process is thus blocked for various lengths of time, depending on the type of the rd (or in) operation and on the availability of matching tuples.

Reactions are a mechanism for asynchronous coordination. If the calling process cannot afford to block, it can register a reaction that will look for a tuple that matches a given template and that will execute a provided piece of code when the tuple is found. The process continues execution as soon as it registers the reaction and does not wait for the reaction to fire and/or for a matching tuple to appear. The general form of a reaction is:

```
reaction(Tuple template, Time r_start, Time r_stop,
Mode mode, Type type, Target target, Handler code)
```

The reaction is active between r_start and r_stop, searches for tuples matching the template among hosts that pass the target filters and executes the code when a tuple is found. In the implementation, Handler is a type of object that has a run(...) method which is called when the reaction is fired. The run(...) method contains the code that is to be executed when the tuple is found, in a thread different from the thread that registered the reaction (and that continued its execution after registration).

The mode parameter identifies *any* host (*anycast* semantics) or *all* hosts (*multicast* semantics) that pass the target filter. The target filter can also identify hosts or locations (i.e., the reaction can look for tuples on host x or "in the garage"). *Unicast* semantics can be obtained using the *anycast* mode and ensuring that only one host passes the target filter (e.g., enforcing a certain value for hostID).

The type parameter can take one of the the following two values: once (which is the default value) or once-per-tuple. A once reaction deregisters itself automatically after it fires once. A once-per-tuple reaction remains active as long as it is alive and fires every time is finds a tuple that matches its template. The semantics for the various combinations of values for mode and type can be best observed in Figure 6.11.

	once reactions	**once-per-tuple reactions**
anycast	- sent to all hosts that qualify - fires once and deregisters itself automatically - only the first tuple is considered, the others are ignored	- sent to all hosts that qualify - fires once for each matching tuple - the first tuple that reaches the reference host selects the target host to be considered - tuples from other hosts are ignored - reactions on other hosts are de-registered if possible
multicast	- sent to all hosts that qualify - fires once for each host and deregisters itself locally	- sent to all hosts that qualify - fires once for each matching tuple on all target hosts

Figure 6.11: Types of reactions.

Installing a reaction is similar to a rd operation from the reachability point of view. The target has to be *2-reachable* from the source. The reference host needs to send the template for the desired tuple and needs to be able to receive the notification when the tuple appears at the remote location (it needs to be able to complete a round trip to the target and back). A reaction can be removed explicitly by the host that installed it. This means the host that installed the reaction needs to be able to send a cancelation request to a target, which entails a one-way trip from the current host to the target (a one-reachable relation). This comes *in addition* to the *2-reachable* relation for installation (and is not required by the interaction pattern that defines the minimum reachability for a successful installation). A reaction can be removed implicitly, when its lifetime expires or when the target host loses any chance to communicate with the initiator (i.e., does not know of any future disconnected routes that might allow it to notify the source of the reaction that a matching tuple has become available).

When a matching tuple is found, a copy is sent back to the host that installed the reaction and the code is run on this host. This code might perform its own coordination operations on the tuple space, for example to attempt to remove the tuple that triggered the reaction. All these operations are subject to their respective reachability constraints and may or may not be able to be executed, even if the reaction did fire.

6.3.7 Acquaintance Lists

Now that I have described the semantics of the operations supported by the coordination model I can better describe the meaning and usefulness of the acquaintance lists. Each host computes its own $\mathsf{AQ}_{\mathsf{out}}()$, $\mathsf{AQ}_{\mathsf{rd}}()$, and $\mathsf{AQ}_{\mathsf{in}}()$ lists (implemented as sets) based on the motion profiles it knows. These sets evolve over time as motion profiles expire and new information is acquired. The table in Figure 6.12 shows how the 3 acquaintance lists host a manages evolve over time, from t_1 to t_{12}, as new information is acquired and paths are formed or lost. The motion profiles of the hosts a, b, c, and d are assumed to generate the windows of opportunity depicted in Figure 6.3 in Section 6.3.2.

	$AQ_{out}(a)$	$AQ_{rd}(a)$	$AQ_{in}(a)$
1	ab	ab	ab
2	ab	ab	ab
3	abc	abc	abc
4	abc	abc	abc
5	abcd	abcd	abcd
6	abcd	abcd	abc
7	abcd	abcd	abc
8	abc	abc	ab
9	abc	abc	ab
10	ab	ab	ab
11	ab	ab	ab
12	a	a	a

Figure 6.12: Acquaintance lists maintained by a.

If an operation is issued with a specific target host, that host must be present in the operation's respective AQ() list of the host that issues the operation at the moment the request is sent. If no target host is specified, the operation will be addressed to all hosts in the AQ() list. For example, in the table in Figure 6.12, host a cannot write a tuple to host d at t_3 because a hasn't learned about the existence of host d and it's motion profile yet. Similarly, a can perform an in on host d at t_5 but could not do it anymore at any later moment (there will not be enough paths between a and d for a successful in.)

6.4 Formal Specification of Key Semantic Constraints

In any system it is important to present the semantics of the primitives offered to the user in a concise, correct, and complete form. In this section I present the formalization of the key semantic constraints of the primitives offered by CAST.

Central to my model are the motion profiles. Knowledge about host motion profiles allows me to investigate in the future, predict and schedule operations that happen at specific moments, over specific periods of time, or at specific locations. The formalization of host motion profiles allows me to develop the formal specification of all the operation semantics exhibited by my model. Using the formalization of motion host profiles I describe the semantics of the coordination operations by deriving a tuple's motion profile and specifying properties of a tuple (or template) in concordance with the semantics of the operation that uses it.

Motion profile analysis at the host level allows me to discover whether there are paths available from a reference host to one or more target hosts, paths that would allow coordination primitives to be completed as expected. By associating a motion profile with a tuple and/or template I obtain a method of specifying the semantics of all coordination primitives supported by my model.

6.4.1 Static Primitives

Static primitives are coordination primitives offered by CAST, where the target is statically identified when the operation is issued. A tuple is sent to a specific, predefined set of hosts by a static out operation. A template also goes to a precisely determined set of target hosts when it is sent out in search of matching tuples by static in or rd operations, or by reactions.

Static OUT

Assuming a host a calls a static out:

a.out(value, b_time, d_time, mode, target)

value is the tuple that a writes to the tuple space. b_time is the birth time when the tuple is required to become active and available for coordination. d_time is the tuple's death time when it is destroyed by its host, if it hasn't been previously removed by an in-like operation. mode is the known anycast/multicast parameter. target identifies the recipients of the tuple. There is another important parameter which is implicit: c_time represents the creation time of the tuple. This is the moment when a calls the out method to write the tuple to the tuple space. This parameter is important to the formal specification of out and is different from b_time, when the tuple becomes active. The tuple is inactive, meaning it can travel to its target and, but cannot participate in any coordination operations while in transit (and before it's scheduled b_time).

The set of parameters that accompany a tuple is initialized when the tuple is created with values readily available (e.g., its value) or computed by the middleware (e.g., its profile):

- $Tuple.value = value$

- $Tuple.b\_time = b\_time$
- $Tuple.d\_time = d\_time$
- $Tuple.mode = mode$
- $Tuple.target = Dest$ (set computed by the middleware)
- $Tuple.profile = \mu$ (profile computed by the middleware)
- $Tuple.c\_time = c\_time$
- $Tuple.active = False$
- $Tuple.ID = new\ ID$
- $Tuple.location = current\_host$ (the host that created the tuple)

The ID of a tuple is a unique identifier which every tuple has. If the mode of the operation is *multicast*, each copy of the tuple going out to a different target host has a separate ID.

The tuple's profile is defined as of c_time and at that moment is the same as the creator host's profile:

$$Tuple.profile(Tuple.c\_time) = a.profile(Tuple.c\_time) \tag{6.7}$$

At any moment before creation or after destruction of the tuple, its profile is undefined:

$$\begin{aligned} \forall t \,:\, t < Tuple.c\_time \vee t > Tuple.d\_time ::& \\ Tuple.profile(t) = \bot& \end{aligned} \tag{6.8}$$

At any moment when the tuple's profile is defined, the tuple's location must be a unique host (for multicast, there are multiple copies of the tuple sent along different routes and therefore the specification stands, as each copy is a different tuple and thus I do not have a single tuple travel along multiple paths simultaneously):

$$\begin{aligned} \forall t \,:\, Tuple.profile(t) \neq \perp &:: \exists! b \, such \, that \\ & Tuple.location(t) = b \end{aligned} \tag{6.9}$$

During the tuple's lifetime (i.e., when the tuple is active) its profile is the same as the profile of the target host where it was send by its parent and the tuple is active (available for coordination), unless the tuple is removed before the end of its programmed lifetime. In this case, the profile becomes undefined again:

$$\begin{aligned} \forall t \,:\, Tuple.b\_time \leq t \leq Tuple.d\_time &:: \exists\, d \in Dest \, such \, that \\ & Tuple.profile(t) = d.profile(t) \\ & \wedge Tuple.active = True \qquad (6.10) \\ \vee \quad \forall t' > t,\, t' \leq Tuple.d\_time\; & Tuple.profile(t') = \perp \qquad (6.11) \end{aligned}$$

At any time, the profile of the tuple is the profile of the host the tuple resides on. For example, if hosts a and b travel together (i.e., one next to the other, significantly closer than the communication range), if host b carries the tuple, the tuple's profile will be that of host b's, even though both a and b have equal access to the tuple:

$$\forall t \, Tuple.profile(t) = (Tuple.location(t)).profile(t) \tag{6.12}$$

A tuple transfers from one host to another only when the two hosts are within communication range. Assuming time to be discrete, the transfer can take place by having the tuple change hosts between moments t and t+1. This is possible if the profiles of the two hosts involved intersect at those moments:

$$\begin{aligned}\forall\, a,\, b,\, t,\ such\, that\, Tuple.location(t) = a\ \wedge \\ Tuple.location(t+1) = b,\ then \\ a.profile(t) \cong b.profile(t) \\ \wedge\, a.profile(t+1) \cong b.profile(t+1)\end{aligned} \tag{6.13}$$

The boolean operator $\cong$ used above means that the two profiles *intersect* but are *not* equal. The two hosts come within communication range, can travel as close as touching each other, but still maintain their own separate motion profiles.

Obeying the formal specifications from above, the tuple travels from the host that creates it (the reference host) to its destination host(s). The destination is prepared in the Dest set by the middleware:

$$\begin{aligned}Dest = \{a\,|\,a = host \\ \wedge\, P(a) = True\, for\, \forall\, predicate\, P \in target \\ \wedge\, a\, reachable\, from\, reference\, host\, before \\ Tuple.b\_time\}\end{aligned} \tag{6.14}$$

With the Dest set built as specified above, the target is picked from Dest as follows: if Tuple.mode is unicast pick *any* host from Dest; if Tuple.mode is multicast pick *every* host from Dest.

The outg operation's formal semantics are the same as the formal semantics described above for out, with the only mention that value stands for a group of tuples instead of only one. All tuples in the group go where a single tuple would have gone, and live like a single tuple would have lived.

Static RD

In the rd call I have an extra parameter:

a.rd(value, b_time, d_time, s_time, mode, target)

The s_time parameter represents a time limit until the operation is willing to stay blocked and wait for a response to return. This has to allow for an answer to propagate back from a target to the reference host. Since the propagation time is affected by the connectivity of the hosts, I have to explicitly account for the delay incurred by host mobility, after a result was found and sent back. For the static case, this parameter is redundant as the exact configuration or paths can be determined and the precise return moment of the result is known. As will be shown in the formal specification of the dynamic call, since the target and/or the return paths may not be fully known to the reference host, this parameter becomes very important.

The first part of a rd operation is very similar to an out operation. It entails sending a template to a certain destination. The template "survives" at the destination for a predetermined period of time or until it matches a tuple. My formal specification begins after the template has reached its target.

Each template has a Template ID, similar to how tuples have Tuple IDs. The difference is that all templates sent out by an operation have the same ID. For example, two successive rd operations will send out templates with different IDs, even if their values are equal. A *multicast* rd targeting multiple hosts sends out multiple templates, all of them having the same ID.

Once at its destination, a match between a tuple and a template is declared at a moment m_time if the conditions below are met:

$$\exists\, m\_time \,\in [Tuple.b\_time, Tuple.d\_time]$$

$$\begin{aligned}\cap[Template.b\_time, Template.d\_time]\\ such\,that\\ Tuple.profile(m\_time) = Template.profile(m\_time)\\ \wedge Tuple.value \doteq Template.value\end{aligned} \tag{6.15}$$

The $\doteq$ boolean operator above means "matches". The semantics of the $\doteq$ operator are defined as follows: a tuple Tuple matches a template Template if they have the same length (i.e., same number of fields) and each field in Tuple matches (type and value) the corresponding field in the Template:

$$\begin{aligned}(Tuple.value \doteq Template.value) \equiv\\ Tuple.length = Template.length\\ \wedge \forall\, i = 1...Tuple.length\\ \{\,((Tuple.field[i]).value = (Template.field[i]).value\\ \vee\,(Template.field[i]).value = not\_specified)\\ \wedge((Tuple.filed[i]).type \simeq (Template.field[i]).type)\}\end{aligned} \tag{6.16}$$

The $\simeq$ operator means polymorphic type matching, returning True if the left operand is of the same type or a sub-type of the right operand. For example, assuming the Java class hierarchy, (String $\simeq$ String) = True, (String $\simeq$ Object) = True, (String $\simeq$ Integer) = False.

After m_time when the matching happens, *a copy* of the tuple travels from the target host where it was found back to the reference host who issues the rd operation in a similar manner a tuple travels following an out operation or in a similar manner the template traveled from the reference host to the target host where it found the tuple. The copy of the tuple that goes back to the reference host is in inactive state such that it cannot be involved in any coordination primitive on its way back to the reference host where it is consumed by the operation that triggered the copy of retrieval of the tuple:

$$\forall\, t > m\_time\, TupleCopy.active(t) = False \tag{6.17}$$

When the tuple reaches the reference host (i.e., the host that issued the call), the tuple is consumed by the host and does not have a life in the local tuple space of the reference host.

$$\begin{aligned} &\forall\, t > t' \, such\, that\, TupleCopy.location(t') = ref\_host, \\ &\qquad TupleCopy.profile(t) = \perp \end{aligned} \tag{6.18}$$

Once a target host finds a matching tuple, the rd request has to be canceled on the other target hosts, as the reference host only needs one response. All other templates with the same ID are canceled:

$$\forall\, h \in Dest\, send\, \text{``}cancel < Template.ID > \text{''} \tag{6.19}$$

Static IN

The static in operation's formal semantics are the same as the rd operation's with the only difference that Tuple is removed from the target's local tuple space.

Static reactions

The static reaction's formal specification is similar to a rd operation's specification. The only difference is that a reaction does not block the reference host while waiting for a matching tuple, but entails no differences in the manner the template travels, the matching happens, or the result returns.

6.4.2 Dynamic Primitives

The core difference from the static case is that the Dest set is fully determined and fixed in the static case, while in the dynamic case the set can change over time and include hosts the reference host does not know about. The target of a dynamic out operation is an area and not a specific set of hosts, as in the static case. Similarly, templates sent out by dynamic rd and in search for matching tuples in target areas rather than target set of hosts. As the target is an area which can even change over time, the set of hosts included in this area can change without the full knowledge of the reference host.

Dynamic OUT

A *unicast* dynamic out identifies a target host (h) whose motion profile places it inside the target area when the tuple is scheduled to activate at b_time:

$$h.profile(Tuple.b\_time) \in Area(t) \tag{6.20}$$

Once h is identified, the tuple is sent in a manner similar to static out.

a.out(value, b_time, d_time, mode, Area)

The target area (i.e., the Area parameter above) is a function of time which, evaluated at some moment t, yields the area of interest where the tuple is intended to be active. This can change over time or the temporary host of the tuple can move towards the edge of the target area, with the intent to leave it. In this case, the tuple moves to another host, which stays within the designated area for a longer time, if available, otherwise, the tuple is destroyed, as it should not be available outside the designated space.

$$\begin{aligned} \forall t\ :\ t \in [b\_time,\, d\_time]\ :: & \\ \{(Tuple.location(t)).profile(t)\ \in\ Area(t) & \\ \wedge\, Tuple.active(t) = True\} & \qquad (6.21) \\ \vee\, Tuple.active(t) = False & \qquad (6.22) \end{aligned}$$

The above specification shows that a host can continue to carry the tuple even if it goes outside the target area. The tuple is inactive but it becomes active again if the host re-enters the target area. Just like in the static out case, the tuple is destroyed at d_time.

An interesting situation can occur when a host carrying an inactive tuple (e.g., the host has left the target area and the tuple has changed to inactive state) meets a host headed for the target area. In this case, the tuple is copied to the inbound host, still in inactive state, and will become active when the newcomer enters the designated area:

$$\begin{aligned} \forall t,\, t',\, a,\, b,\ such\,that\,Tuple.location(t) = a & \\ \wedge\, a.profile(t) \notin Area(t),\, b.profile(t) \notin Area(t) & \\ \wedge\, a.profile(t) \cong b.profile(t) & \\ \wedge\, t < t' \leq d\_time,\, b.profile(t') \in Area(t') & \\ Tuple' = Tuple \wedge Tuple'.location(t) = a & \\ \wedge\, Tuple'.location(t+1) = b & \qquad (6.23) \end{aligned}$$

Host b receives a *copy* of Tuple because at t' (which is during the tuple's lifetime) host b will be inside Area. The new tuple is in inactive state.

The transfer above happens only if host a has enough information from the motion profile analysis to find that the tuple can re-enter the target area. If this information is not available by d_time the tuple is destroyed. Also, the tuple is destroyed if the lifetime is "forever" but the host doesn't seem to re-enter the area again (based on the limited motion profile information it has at the moment it leaves the target area), nor it knows any host entering the area and to which it can give the tuple:

$$\begin{aligned}
\forall\, t > t',\, t < d\_time,\, a,\, such\, that \\
\{a.profile(t') \in Area(t') \\
\wedge\, a.profile(t'+1) \notin Area(t'+1) \\
\wedge[a.profile(t) \notin Area(t) \\
\wedge(\nexists b\, such\, that\, a.profile(t) \cong b.profile(t) \\
\wedge \exists t'' > t,\, t'' < d\_time\, such\, that \\
b.profile(t'') \in Area(t''))]\} \\
\\
Tuple.profile(t) = \bot
\end{aligned} \tag{6.24}$$

Dynamic RD and IN

Dynamic rd and in are similar to dynamic out. Their templates behave just like the tuple of a dynamic out operation does. A template is active inside the designated area, inactive outside it, has a b_time and d_time and once a matching tuple is found, the behavior obeys the specifications from the static case, where the result comes back to the reference host and cancelation requests are sent out.

Dynamic reactions

Similar to static reactions, dynamic reactions do not block the reference host while waiting for a tuple, but they can hover around the target area like a dynamic tuple or a template from a dynamic rd or in.

6.5 Summary

In this chapter I have presented a novel coordination model that learns from previously developed coordination model for mobile ad hoc networks Lime and Limone, all inspired by the original Linda. The model provides synchronous and asynchronous, single tuple and group coordination tuple space coordination primitives. It makes the time and space parameters explicitly available to the programmer using the model. CAST provides, like all coordination models, a separation of concerns between the computation that happens at layers above from communication concerns sealed below. CAST is the first coordination model that defines a lifetime motion profile for tuple and templates which can specifically send a tuple to a host or physical geographic area, and specify a pre-determined active period when the tuples are available for coordination and when the templates look for matching tuples.

Chapter 7

Future Work

The research I have described provides a solution to several problems for service oriented computing in ad hoc networks. The SOC architecture is built on top of a disconnected routing algorithm which also provides support for the context aware coordination model presented in this document. One of the key novelties in my work is the use of contextual information in the form of motion profiles to schedule interactions among hosts. All information used from the motion profiles has been assumed accurate and immutable. While these assumptions supported the development of the infrastructure and algorithms presented, they also represent steps away from reality. To make my approach more applicable to real life scenarios, assumptions about the quality of the contextual information have to be relaxed and even eliminated. Further investigations will target the notion of uncertainty in mobility prediction and all the side effects that propagate from the network layer up to the distributed system.

Disconnected Routing. The disconnected route discovery method I presented relies on motion profile information to identify periods of time when two hosts promise to be at a distance shorter than a certain threshold considered the communication range. In reality, this information is only approximative. For example, buses have an advertised schedule and try to stick to it, but even the most accurate transportation services cannot guarantee precision to the second. While mobile hosts deviate from their advertised motion profiles, this has an effect on the windows of communication opportunity. Some may disappear, some new may be created. While the new windows may be opportunities to taken advantage of, missing windows of communication opportunity can definitely cause serious problems.

As each location information (current host location read from a GPS or projected host location based on future motion profile) is affected by an error, there is a certain probability that the value is true, and a certain probability that the real value is nearby the theoretical value. The real values are sprinkled around the theoretical value, following a certain *error model*. To model this, I plan to use probability density functions which help us describe how far the real values are from the theoretical values, and how they are distributed.

Host A is located somewhere around the point where it thinks it is. In Figure 7.1, the location of host A is somewhere in the gray patch labeled A. The ideal location is at the center of the patch. The darker the color, the higher the probability for A to be there. The probability decreases (uniformly in all directions) as we look further away from the ideal location. The same type of error affects host B, depicted to the right of A. The problem reduces to what are the chances that hosts A and B are "around there", with errors controlled by (1) a distribution model (in Figure 7.1 both hosts' locations are shown as governed by a Gaussian probability density function) (2) a mean equal to the ideal value advertised and (3) standard deviations assumed known. Assuming a model where the distance between the hosts compared to the communication range decides whether the hosts can communicate or not, the new question is *"what is the probability that the distance between A and B is shorter than the communication range r?"*

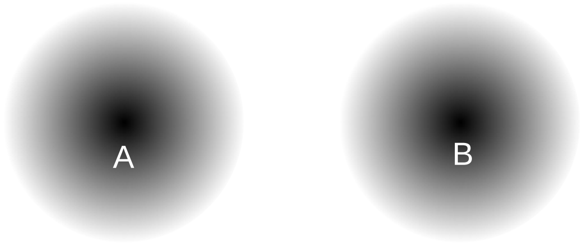


Figure 7.1: Both A's and B's locations can be affected by errors.

Associating a probability with each projected link leads to a probability of delivery along each computed route. The participants above can define their comfort threshold and decide when to risk sending the message and when not to.

The size of a window of communication opportunity can also be affected by the motion profiles and location estimation errors. While throughout the document I silently assumed that a window is comfortably large (or transfers are negligibly short), this is also an assumption to be taken into account. The size does matter. This has to be taken into account in an effort to address scenarios closer to what real life throws at us.

Disconnected Coordination. The ripple effect of introducing the probabilities in the routing layer propagates into the coordination layer. A probability-based spatio-temporal coordination model is the next natural step. Tuple delivery probability, the success chance of a coordination primitive and, more importantly, clearly defined semantics for the situations where a link is lost against the favorable odds. Recovery from link failure is an issue that can and has to be addressed at the coordination layer.

Service Oriented Computing. The uncertainty from the layers below definitely influences all layers above, including the SOC layer. At this level, the SOC architecture could provide the applications with APIs that expose the probabilities in discussion and provide the application developers with the possibility to define a minimum comfort level (communication probability threshold).

Regarding the follow-me sessions, unexplored directions include the study of software composition. The service could be formed of parts originating on different hosts or may even be built of other stand-alone services. The migration and integration of such parts are part of my future research plans.

Also related to follow-me sessions, I intend to research the possibility of expanding the mechanisms that can be employed during a FMS. For example, in Figure 7.2, two new types of interactions are depicted, neither of them currently covered by the researched presented.

When the client (the round circle at the bottom traveling left to right) is about to disconnect from H_1, the options currently available are take back, temporary disconnect or context-sensitive binding (to H_2), while the client passes through location b. The new feature would allow H_1 to continue processing the client's job while using H_2 as an intermediary for (disconnected routing). Obviously, the link between the

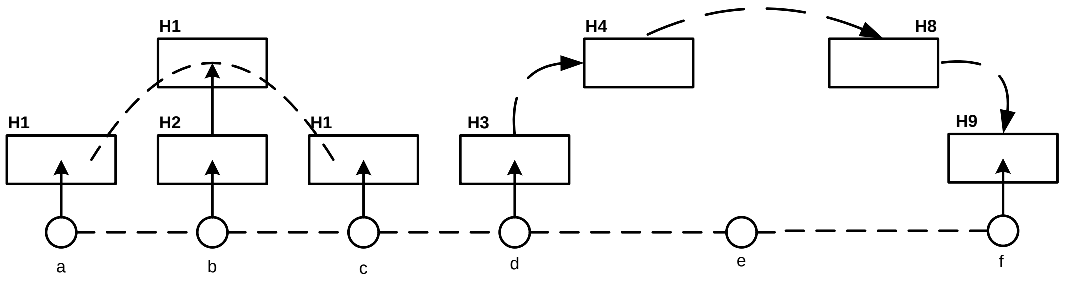


Figure 7.2: Extending the FMS.

client and the server's host could expand more than only one intermediary host. Another part of my future research will involve the scenario depicted on the righthand side part of the same Figure 7.2. The service could follow a loop including multiple volunteer hosts (from H_4 to H_8) with no connection to the client's machine. This is currently not possible, as the client holds a close control over the FMS, without allowing it to "go away".

Chapter 8

Conclusions

This dissertation takes the first steps in studying the spatio-temporal service provision in mobile ad hoc networks. Addressing issues from routing protocols in wireless ad hoc networks through coordination models and service-oriented computing specific challenges, I believe this work provides a glimpse of a compelling new information dissemination paradigm, coordination mechanism, and anytimme/anywhere service provision. It also helps to lay a solid foundation for future research. I believe this work will lead to more interesting and fruitful explorations in the spatio-temporal dimensions of the interactions in mobile ad hoc networks.

References

[1] http://wiki.uni.lu/secan-lab/secan-lab.html. Web Page, 2005.

[2] http://www.checkpointing.org/. Web Page, 2005.

[3] Anurag Acharya, M. Ranganathan, and Joel Saltz. Sumatra: A Language for Resource-aware Mobile Programs. In J. Vitek and C. Tschudin, editors, *Mobile Object Systems: Towards the Programmable Internet*, volume 1222, pages 111–130. Springer-Verlag: Heidelberg, Germany, 1997.

[4] Sameer Ajmani, Barbara Liskov, and Liuba Shrira. Scheduling and simulation: How to upgrade distributed systems. In *Proceedings of the Ninth Workshop on Hot Topic in Operating Systems*, May 2003.

[5] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services: Concepts, Architectures and Applications*. Springer Verlag, 2004.

[6] A. Ankolekar, M. Burstein, J. Hobbs, O. Lassila, D. Martin, D. McDermott, S. McIlraith, S. Narayanan, M. Paolucci, T. Payne, and K. Sycara. Daml-s: Web service description for the semantic web. In *Proceedings of the 1st International Semantic WebConference*, 2002.

[7] Farhad Arbab. The iwim model for coordination of concurrent activities. In *Proceedings of the First International Conference on Coordination Models, Languages and Applications*, number 1061 in Lecture Notes in Computer Science, pages 34–56. Springer-Verlag, 1996.

[8] Joachim Baumann, Fritz Hohl, Nikolaos Radouniklis, Kurt Rothermel, and Markus Strasser. Communication concepts for mobile agent systems. In *MA '97: Proceedings of the First International Workshop on Mobile Agents*, pages 123–135. Springer-Verlag, 1997.

[9] Joachim Baumann, Fritz Hohl, and Kurt Rothermel. Mole - concepts of a mobile agent system. In *Proceedings of the 2nd ECOOP Workshop on Mobile Object Systems*, 1997.

[10] Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web. Scientific American, May 2001.

[11] Premysl Brada. Component change and verification in sofa. In *Proceedings of SOFSEM 1999*, number 1725 in LNCS. Springer Verlag, 1999.

[12] Maciej Brzezniak and Norbert Meyer. Evaluation of execution time of mathematical library functions based on historical performance information. In *Proceedings of 5th International Conference on Parallel Processing and Applied Mathematics*, LNCS, pages 161–168. Springer Verlag, 2004.

[13] Matthew Burnside, Dwaine Clarke, Todd Mills, Srinivas Devadas, and Ronald Rivest. Proxy-based security protocols in networked mobile devices. In *Proceedings of Selected Areas in Cryptography*, 2002.

[14] Giacomo Cabri, Letizia Leonardi, and Franco Zambonelli. The impact of the coordination model in the design of mobile agent applications. In *Proceedings of the 22nd International Computer Software and Application Conference*, pages 436–442, 1998.

[15] Giacomo Cabri, Letizia Leonardi, and Franco Zambonelli. MARS: A programmable coordination architecture for mobile agents. *IEEE Internet Computing*, 4(4):26–35, 2000.

[16] Luca Cardelli and Andrew D. Gordon. Mobile ambients. In *Foundations of Software Science and Computation Structures: First International Conference, FOSSACS '98*. Springer-Verlag, Berlin Germany, 1998.

[17] C. Chiang, H. Wu, W. Liu, and M. Gerla. Routing in clustered multihop, mobile wireless networks. In *IEEE Singapore International Conference on Networks*, pages 197–211, 1997.

[18] Paolo Ciancarini and Davide Rossi. Jada - coordination and communication for java agents. In *Proceedings of the Second International Workshop on Mobile Object Systems*, number 1222 in Lecture Notes in Computer Science, pages 213–226. Springer-Verlag, 1997.

[19] Mark Claypool and David Finkel. Transparent process migration for distributed applications in a Beowulf cluster. In *Proceedings of the International Networking Conference*, July 2002.

[20] J. Cohen and S. Aggarwal. General event notification architecture. http://www.globecom.net/ietf/draft/draft-cohen-gena-p-base-01.html, July 1998.

[21] Jonathan Cook and Jeffrey Dage. Highly reliable upgrading of components. In *Proceedings of the 1999 International Conference on Software Engineering*, pages 203–212, 1999.

[22] Gianpaolo Cugola, Elisabetta Di Nitto, and Alfonso Fuggetta. The JEDI event-based infrastructure and its application to the development of the opss wfms. *IEEE Transactions on Software Enggineering*, 27(9):827–850, 2001.

[23] Steven E. Czerwinski, Ben Y. Zhao, Todd D. Hodes, Anthony D. Joseph, and Randy H. Katz. An architecture for a secure service discovery service. In *Mobile Computing and Networking*, pages 24–35, 1999.

[24] Markus Dahm. Byte code engineering with the bcel api. Technical Report B-17-98, Freie Universitat Berlin, Institut fur Informatik, 2001.

[25] N. Davies, S. Wade, A. Friday, and G. Blair. Limbo: A Tuple Space Based Platform for Adaptive Mobile Applications. In *Proceedings of the International Conference on Open Distributed Processing/Distributed Platforms (ICODP/ICDP '97)*, pages 291–302, Toronto, Canada, May 1997.

[26] Carl Ellison, Bill Frantz, Butler Lampson, Ron Rivest, Brian Thomas, and Tatu Ylonen. Simple public key certificates. Internet Draft http://world.std.com/cme/spki.txt, 1999.

[27] Robert Engelmore and Tony Morgan. Blackboard systems. Addison-Wesley Publishing Company, 1998.

[28] Pasi Eronen, Christian Gehrman, and Pekka Nikander. Securing ad hoc jini services. In *NordSec2000*, 2000.

[29] Pasi Eronen, Johannes Lehtinen, Jukka Zitting, and Pekka Nikander. Extending jini with decentralized trust management. In *The Third IEEE Conference on Open Architectures and Network Programming (OPENARCH)*, 2000.

[30] Pasi Eronen and Pekka Nikander. Decentralized jini security. In *Network and Distributed System Security*, 2001.

[31] Meik Felser, Michael Golm, Christian Wawersich, and Jrgen Kleinoder. Execution time limitation of interrupt handlers in a java operating system. In *Proceedings of 10th ACM SIGOPS European Workshop*, 2002.

[32] Chien Liang Fok, Gruia-Catalin Roman, and Greg Hackmann. A lightweight coordination middleware for mobile computing. In *Proceedings of the 6th International Conference on Coordination Models and Languages*, volume 2949 of *Lecture Notes in Computer Science*, pages 135–151. Springer Verlag, 2004.

[33] Chien-Liang Fok, Gruia-Catalin Roman, and Chenyang Lu. Rapid development and flexible deployment of adaptive wireless sensor network applications. In *Proceedings of the* 24^{th} *International Conference on Distributed Computing Systems (ICDCS'05)*, pages 653–662, 2005.

[34] Chien-Liang Fok, Gruia-Catalin Roman, and Chenyang Lu. Software support for application development in wireless sensor network. In *Mobile Middleware*, chapter 7H. CRC Press, 2005.

[35] Alfonso Fuggetta, Gian Pietro Picco, and Giovanni Vigna. Understanding Code Mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, 1998.

[36] L. G.DeMichiel, L. U. Yalcinalp, and S. Krishnan. Enterprise java beans specification. Sun Microsystems, 2000.

[37] David Gerlenter. Generative communication in linda. *ACM Computing Surveys*, 7:80–112, January 1985.

[38] Christopher Gill, Yamuna Krishnamurthy, Douglas Schmidt, Irfan Pyarali, Louis Mgeta, Yuanfang Zhang, and Stephen Torri. Enhancing adaptivity via standard dynamic scheduling middleware (to appear). *Journal of the Brazilian Computer Society*, 2005.

[39] Silvia Giordano, Ivan Stojmenovic, and Ljubica Blazevic. Position based routing algorithms for ad hoc networks: a taxonomy. citeseer.ist.psu.edu/496653.html, July 2001.

[40] Y. Goland, T. Cai, P. Leach, and Y. Gu. Simple service discovery protocol. http://www.upnp.org/download/draft_cai_ssdp_v1_03.txt, April 1998.

[41] Y. Goland, T. Cai, P. Leach, Y. Gu, and S. Albright. Simple service discovery protocol/1.0: Operating without an arbiter. http://www.upnp.org/download/draft_cai_ssdp_v1_03.txt, 2001.

[42] Li Gong. A secure identity-based capability system. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 56–63, 1989.

[43] R. S. Gray. Agent Tcl: A flexible and secure mobile-agent system. In *Fourth Annual Tcl/Tk Workshop (TCL 96)*, pages 9–23, 1996.

[44] Matthias Grossglauser and David N. C. Tse. Mobility increases the capacity of ad-hoc wireless networks. *IEEE/ACM Transactions on Networking*, 10(4):477–486, August 2002.

[45] E. Guttmann, C. Perkins, J. Veizades, and M. Day. Service location protocol. IETF Internet Draft, RFC 2608, 1999.

[46] Zygmunt J. Haas, Marc R. Pearlman, and Prince Samar. The zone routing protocol for ad hoc networks. IETF Internet Draft, July 2002.

[47] Radu Handorean, Chris Gill, and Gruia-Catalin Roman. Accommodating transient connectivity in ad hoc and mobile settings. In Alois Ferscha and Friedemann Mattern, editors, *Proceedings of the Second International Conference on Pervasive Computing (Pervasive 04)*, number 3001 in Lecture Notes in Computer Science, pages 305–322. Springer-Verlag, 2004.

[48] Radu Handorean, Jamie Payton, Christine Julien, and Gruia-Catalin Roman. Coordination middleware supporting rapid deployment of ad hoc mobile systems. In *Proceedings of the ICDCS Workshop on Mobile Computing Middleware*, pages 362–368. IEEE Computer Society, 2003.

[49] Radu Handorean and Gruia-Catalin Roman. Service provision in ad hoc networks. In *Proceedings of 5th International Conference on Coordination Models (COORDINATION 2002)*, number 2315 in LNCS, pages 207–219. Springer-Verlag, 2002.

[50] Radu Handorean and Gruia-Catalin Roman. Secure service provision in ad hoc networks. In *Proceedings of The First International Conference on Service Oriented Computing (ICSOC 03)*, number 2910 in Lecture Notes in Computer Science, pages 367–383. Springer-Verlag, 2003.

[51] Radu Handorean and Gruia-Catalin Roman. Secure sharing of tuple spaces in ad hoc settings. In Riccardo Focardi and Gianluigi Zavattaro, editors, *Electronic Notes in Theoretical Computer Science*, volume 85. Elsevier, 2003.

[52] Radu Handorean, Rohan Sen, Greg Hackmann, and Gruia-Catalin Roman. Automated code management for service oriented computing in ad hoc networks. Technical Report WU-CSE-2004-17, Washington University in Saint Louis, 2004.

[53] Radu Handorean, Rohan Sen, Gregory Hackmann, and Gruia-Catalin Roman. Context aware session management for services in ad hoc networks. In JD Cantarella, editor, *Proceedings of the 2005 IEEE International Conference on Services Computing*, volume I, pages 113–120. IEEE Computer Society, July 2005.

[54] Michael Hicks, Jonathan Moore, and Scott Nettles. Dynamic software updating. In *Proceedings of the ACM SIGPLAN Workshop on Types in Compilation*, September 2000.

[55] I. Horrocks. Daml+oil: A description logic for the semantic web. *IEEE Bulletin of the Technical Committee on Data Engineering*, 2002.

[56] Qingfeng Huang, Christine Julien, and Gruia-Catalin Roman. Relying on safe distance to achieve partitionable group membership in ad hoc networks. *IEEE Transactions on Mobile Computing*, 3(2):192–205, 2004.

[57] Qingfeng Huang, Chenyang Lu, and Gruia-Catalin Roman. Mobicast: Just-in-time multicast for sensor networks under spatiotemporal constraints. In *Lecture Notes in Computer Science*, number 2634. Springer-Verlag, April 2003.

[58] Jean-Pierre Hubaux, Levente Buttyan, and Srdan Capkun. The quest for security in mobile ad hoc networks. In *ACM MobiHOC Symposium*, 2001.

[59] Tomasz Imielinski and Julio Navas. Gps-based addressing and routing. http://rfc2009.x42.com/, 1996.

[60] D. Johansen, R. van Renesse, and F. B. Schneider. An introduction to the TACOMA distributed system–version 1.0. Technical Report 95-23, University of Tromso, Norway, 1995.

[61] David B. Johnson and David A. Maltz. Dynamic source routing in ad hoc wireless networks. *Mobile Computing*, 353, 1996.

[62] G. Karjoth, D.B. Lange, and M. Oshima. Mobile agents and security. *Lecture Notes in Computer Science*, 1419:188–205, 1998.

[63] Brad Karp and H. T. Kung. GPSR: greedy perimeter stateless routing for wireless networks. In *Mobile Computing and Networking*, pages 243–254, 2000.

[64] Ralph Keller and Urs Hölzle. Binary component adaptation. *Lecture Notes in Computer Science*, 1445:307–324, 1998.

[65] Ralph Keller and Urs Hölzle. Binary component adaptation. *Lecture Notes in Computer Science*, 1445:307–324, 1998.

[66] Young-Bae Ko and Nitin H. Vaidya. Geocasting in mobile ad hoc networks: Location-based multicast algorithms. 1999.

[67] Butler Lampson. Protection. In *5th Princeton Conference on Information Sciences and Systems*, volume ACM Operating Systems Rev. 8, pages 18–24, 1971.

[68] Vincent Lenders, Polly Huang, and Men Muheim. Hybrid Jini for limited devices. In *Proceeding of the IEEE International Conference on Wireless LANs and Home Networks*, 2001.

[69] Qun Li and Daniela Rus. Communication in disconnected ad hoc networks using message relay. *Parallel and Distributed Computing*, 63:75–86, January 2003.

[70] Theophilos Limniotes, Costas Mourlas, and George A. Papadopoulos. Event-driven coordination of real-time components. In *Proceedings of the 22nd International Conference on Distributed Computing Systems Workshops*. IEEE Computer Society, 2002.

[71] Bjorn Lisper. Fully automatic, parametric worst-case execution time analysis. In *Proceedings of the 3rd International Workshop on Worst-Case Execution Time Analysis*, pages 77–80, 2003.

[72] Marco Mamei, Franco Zambonelli, and Letizia Leonardi. Tuples on the air: a middleware for context-aware computing in dynamic networks. In *Proceedings of the 2nd International Workshop on Mobile Computing Middleware at the 23rd International Conference on Distributed Computing Systems (ICDCS)*, pages 342–347, 2003.

[73] Martin Mauve, Jorg Widmer, and Hannes Hartenstein. A survey on position-based routing in mobile ad hoc networks. *IEEE Network Magazine*, 15(6):30–39, November 2001.

[74] Vladimir Mencl, Zuzana Petrova, and Frantisek Platil. Update description language, June 1999.

[75] Microsoft-Corporation. Universal plug and play device architecture. http://www.upnp.org/download/UPnPDA10_20000613.htm, June 2000.

[76] SUN Microsystems. Javaspace specification.

[77] S. Miller. The autoip publisher page. http://www.autoip.net, October 2003.

[78] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, parts i and ii. *Information and Computation*, 100(1):1–40, 1992.

[79] Gianluca Moro and Antonio Natali. On the event coordination in multi-component systems. In *Proceedings of the* 14th *International Conference on Software Engineering and Knowledge Engineering*, pages 315–322. ACM Press, 2002.

[80] A.L. Murphy, G.P. Picco, and G.-C. Roman. Lime: A middleware for physical and logical mobility. In *Proc. of the 21st Int'l Conf. on Distributed Computing Systems*, pages 524–533, April 2001.

[81] Shree Murthy and J. J. Garcia-Luna-Aceves. An efficient routing protocol for wireless networks. *Mobile Networks and Applications*, 1(2):183–197, 1996.

[82] Julio C. Navas and Tomasz Imielinski. GeoCast – geographic addressing and routing. In *Mobile Computing and Networking*, pages 66–76, 1997.

[83] Review Draft of the 1.2 revision to the Real-Time CORBA Specification. http://www.omg.org/docs/ptc/01-08-34.pdf.

[84] Andrea Omicini and Franco Zambonelli. The TuCSoN coordination model for mobile information agents. In *Proceedings of the 1st Workshop on Innovative Internet Information Systems*, Pisa, Italy, June 1998.

[85] General Magic Odyssey Page. http://www.genmagic.com/agents/odyssey.html.

[86] M. Paolucci, T. Kawmura, T. Payne, and K. Sycara. Semantic matching of web services capabilities. In *Proceedings of the 1st International Semantic Web Conference*, 2002.

[87] George Papadopoulos. Models and technologies for the coordination of internet agents: A survey, 2000.

[88] George A. Papadopoulos and Farhad Arbab. Coordination models and languages. In *761*, page 55. Centrum voor Wiskunde en Informatica (CWI), 31 1998.

[89] Vincent D. Park and M. Scott Corson. A highly adaptive distributed routing algorithm for mobile wireless networks. In *Proceedings of INFOCOM'97*, pages 1405–1413, 1997.

[90] P.Domel, A.Lingnau, and O.Drobnik. Mobile agent interaction in heterogeneous environments. In Springer Verlag, editor, *Proceedings of the 1st International Workshop on Mobile Agents*, number 1219, pages 136–148, Stuttgart, Germany, April 1997.

[91] Holger Peine and Torsten Stolpmann. The architecture of the Ara platform for mobile agents. In Radu Popescu-Zeletin and Kurt Rothermel, editors, *First International Workshop on Mobile Agents MA'97*, volume 1219 of *Lecture Notes in Computer Science*, pages 50–61, Berlin, Germany, April 1997. Springer Verlag.

[92] Charles Perkins. Ad-hoc on-demand distance vector routing. In *MILCOM '97 panel on Ad Hoc Networks*, 1997.

[93] Charles Perkins and Pravin Bhagwat. Highly dynamic destination-sequenced distance-vector routing (DSDV) for mobile computers. In *ACM SIGCOMM'94 Conference on Communications Architectures, Protocols and Applications*, 1994.

[94] G. P. Picco. μCode: A lightweight and flexible mobile code toolkit. In *Proceedings of the 2^{nd} International Workshop on Mobile Agents*, volume 1477 of *LNCS*, pages 160–171, September 1998.

[95] Peter Puschner and Alexander Vrchoticky. An assessment of task execution time analysis. In *Proceedings of the 10th IFAC Workshop on Distributed Computer Control Systems*, pages 41–45, 1991.

[96] Marija Rakic and Nenad Medvidovic. Increasing confidence in off-the-shelf components: A software connector-based approach. In *Proceedings of the 2001 Symposium on Software Reusability*, pages 11–18, 2001.

[97] Ronald L. Rivest and Bulter Lampson. Sdsi - a simple distributed security infrastructure. Presented at CRYPTO'96 Rumpsession (http://citeseer.nj.nec.com/ rivest96sdsi.html).

[98] Gruia-Catalin Roman and H. Conrad Cunningham. Mixed programming metaphors in a shared dataspace model of concurrency. *IEEE Transactions on Software Engineering*, 16(12):1361–1373, 1990.

[99] Manuel Roman and Roy Campbell. Gaia: Enabling active spaces. In *Proceedings of the 9th ACM SIGOPS European Workshop*, pages 229–234, 2000.

[100] Salutation-Consortium. The salutation consortium homepage. http://www.salutation.org, October 2003.

[101] Tomas Sander and Christian F. Tschudin. Protecting mobile agents against malicious hosts. *Lecture Notes in Computer Science*, 1419:44–60, 1998.

[102] Douglas Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture*, volume 2, chapter 2, pages 109–141. John Wiley and Sons Ltd., 2000.

[103] Douglas Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture*, volume 2, chapter 2, pages 47–75. John Wiley and Sons Ltd., 2000.

[104] Rohan Sen, Radu Handorean, Gregory Hackmann, and Gruia-Catalin Roman. An architecture supporting run-time upgrade of proxy-based services in ad hoc networks. In Hamid R. Arabnia and Laurence T. Yang, editors, *Proceedings of Pervasive Computing Conference*, pages 689–696. CSREA Press, 2004.

[105] Rohan Sen, Radu Handorean, Gruia-Catalin Roman, and Chris Gill. *Service-Oriented Software System Engineering: Challenges and Practices*, chapter Service Oriented Computing Imperatives in Ad Hoc Wireless Settings, pages 247–269. Idea Group, 2005.

[106] J. Srinivas. Open network computing remote procedure call protocol specification. http://www.ietf.org/rfc/rfc1831.txt, August 1995.

[107] Ivan Stojmenovic. Position-rased routing in ad hoc netowkrs. *IEEE Communications Magazine*, 2002 2002.

[108] Sun-Microsystems. Java remote method invocation page. http://java.sun.com/products/jdk/rmi/, October 2003.

[109] Clemens Szyperski. *Component Software, Beyond Object-Oriented Programming.* ACM Press - Addison-Wesley, 1997.

[110] Clemens Szyperski. *Foundations of Component-based Systems*, chapter Component Software and the Way Ahead, pages 1–20. Cambridge University Pres, 2000.

[111] Chai-Keong Toh. A novel distributed routing protocol to support ad-hoc mobile computing. In *Fifteenth Annual International Phoenix Conference on Computers and Communications*, pages 480–486, 1996.

[112] UDDI-Organization. Uddi technical white paper. http://www.uddi.org/pubs/Iru_UDDI_Technical_White_Paper.pdf, 2000.

[113] A. Vahdat and D. Becker. Epidemic routing for partially connected ad hoc networks. Technical Report CS-200006, Duke University, 2000.

[114] Marco Vettorello, Christian Bettstetter, and Christian Schwingenschlgl. Some notes on security in the service location protocol version 2 (slpv2). In *Proc. Workshop on Ad hoc Communications, in conjunction with 7th European Conference on Computer Supported Cooperative Work (ECSCW'01)*, 2001.

[115] W3C-Metadata-Activity. Resource description framework schema specification 1.0. http://www.w3.org/TR/2000/CR-rdf-schema-20000327/, March 2000.

[116] W3C-Semantic-Web-Activity. Worldwide web consortium page on resource description framework. http://www.w3.org/RDF/, October 2003.

[117] W3C-XML-Activity-On-XML-Protocols. W3c recommendation: Web services description language 1.1. http://www.w3.org/TR/wsdl, October 2003.

[118] Jim Waldo. The Jini Architecture for Network-Centric Computing. *Communications of the ACM*, 42(7):76–82, 1999.

[119] Palm Source Website. http://www.palmsource.com/palmos/, January 2005.

[120] P. Wyckoff. Tspaces. *IBM System Journal*, 37(3):454–474, 1998.

[121] XML-Core-Working-Group. W3c recommendation: Xml version 1.0 second edition. http://www.w3.org/TR/2000/REC-xml-20001006, October 2000.

[122] XML-Protocol-Working-Group. W3c recommendation: Soap version 1.2 parts 0-2. http://www.w3.org/TR/SOAP/, June 2003.

[123] Yuanfang Zhang, Bryan Thrall, Stephen Torri, Christopher Gil, and Chenyang Lu. A real-time performance comparison of distributable threads and event channels. In *Proceedings of the 11th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 497–506, 2005.

[124] Wenrui Zhao and Mostafa H. Amma. Message ferrying: Proactive routing in highly-partitioned wireless ad hoc networks. In *Ninth IEEE Workshop on Future Trends of Distributed Computing Systems*, 2003.

Vita

Radu Handorean

Date of Birth	19 December 1975
Place of Birth	Brasov, Romania
Degrees	B.Sc. Politehnica University, Bucharest, Romania, June 1999 M.Sc. Washington University, St. Louis, USA, May 2003 D.Sc. Washington University St. Louis, USA, December 2005

December 2005

Short Title: SOC in AHN Handorean, D.Sc. 2005