The Dissertation Committee for Andrew Erich Stovall
certifies that this is the approved version of the following dissertation:

Easing Software Development for Pervasive Computing Environments

Committee:

Christine Julien, Supervisor

Sarfraz Khurshid

Scott Nettles

Dewayne E. Perry

William J. O'Brien

Gruia-Catalin Roman

Easing Software Development for Pervasive Computing Environments

by

Andrew Erich Stovall, B.S.; M.S.E.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

December 2009

Dedicated to my wife and parents.

Thanks.

Easing Software Development for Pervasive Computing Environments

Publication No. _____

Andrew Erich Stovall, Ph.D.
The University of Texas at Austin, 2009

Supervisor: Christine Julien

In recent years pervasive computing has enjoyed an amazing growth in both research and commercial fields. Not only have the number of available techniques and tools expanded, but the number of actual deployments has been underwhelming. With this growth however, we are also experiencing a divergence of software interfaces, languages, and techniques. This leads to an understandably confusing landscape which needlessly burdens the development of applications. It is our sincere hope that through the use of specialized interfaces, languages, and tools, we can make pervasive computing environments more approachable and efficient to software developers and thereby increase the utility and value of pervasive computing applications.

In this dissertation, we present a new method for creating and managing the long-term conversations between peers in pervasive computing environments. The *Application Sessions Model* formally describes these conversations

and specifies techniques for managing them over their lifetimes. In addition to these descriptions, this dissertation presents a prototype implementation of the model and results from its use for realistic scenarios. To address the Application Sessions Model's unique needs for resource discovery in pervasive computing environments, we also present the *Evolving Tuples Model*. This model is also formally defined in this dissertation and practical examples are used to clarify its features. A prototype for both sensor hardware and software simulation of this model is described along with results characterizing the behavior of the model. The models, prototypes, and evaluations of both models presented here form the basis of a new and interesting line of research into support structures for pervasive computing application development.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The increasing availability of ubiquitous computing technology has enabled a new class of pervasive computing applications. They interact with a dynamic environment and a spectrum of collaborating applications and devices to provide adaptive, responsive, personalized, and intuitive computing experiences. As the deployment and capabilities of these technologies continue to grow, they will provide an environment that will foster a new generation of highly connected computing applications. These applications will no longer use remote information and services occasionally, but will instead incorporate them into their core functionality. Applications will derive their intrinsic value from interactions with each other and the environment and embed new utility into our everyday experience.

With this shift in computing paradigms, our computing applications take on new characteristics, such as the need to support extreme heterogeneity of devices, unpredictability of connectivity between devices, the increasing scale and distribution of the network, and the need for geographically specific information. Applications cannot even assume compatible capabilities from peers in the network. Resources in pervasive computing environments may

be simple environmental sensors and actuators controlling aspects of the environment, or complete ecosystems of highly capable devices. Additionally, the nodes in the network may be physically moving and thus creating the need for adaptive data delivery mechanisms. All of these features contribute significantly to the complexity of designing, programming, and deploying pervasive computing applications.

Much of the past and current research for pervasive computing environments concentrates on the physical, data link, and network layers of communication. While these elements are vital to pervasive computing, we must take care when exposing these features to application developers. If these features are modeled properly, developers can delegate low-level concerns to the appropriate technologies and focus their attention on the high-level concerns of their application's domain. However, the *currently available conceptual and concrete models for pervasive computing environments are too complex for software development by application domain experts.* The work in this dissertation focuses on providing a model for application interactions with resources in these environments that ease the task of software development.

Several well designed interfaces have successfully realized a separation of concerns in traditional networking environments, among them *sockets*, *RMI*, and *WebServices*. We believe that a big part of the success of these technologies is the support for *autonomous long-term conversations*. In these approaches, the application developer supplies only essential information to the implementing technology which creates and maintains an artifact representing a durable

conversation. Once created, the application interacts with its peer on the other end of the conversation until the application terminates it. At no point between its creation and termination must the application interact directly with the supporting technologies such as DNS, data serialization, or SOAP.

We believe that application developers in pervasive computing environments should also be afforded a similar convenience, specifying a few parameters when beginning a conversation but otherwise delegating setup and maintenance to underlying system. However, the additional complexities introduced by the dynamic and heterogeneous nature of pervasive computing networks inserts several new characteristics to the application level conversations. For example, the loss of a routing path in a traditional network will simply terminate a conversation. In a pervasive computing environment, the loss of a routing path is so common that it is expected. Certainly the loss of a routing path will interrupt communications, but applications may reasonably expect to be reconnected at some point in the future. When possible, applications should expect interruptions in *communications* to merely suspend the longer-term *conversation*, not terminate it. More generally, technologies supporting conversations in pervasive computing environments must contend with the following complications:

- *unpredictable connectivity*: due to mobility and other outside forces (e.g., unreliable wireless communication), supporting technologies must help applications handle intermittent connectivity.

- *unpredictable and dynamic availability*: unpredictable connectivity results in sporadic and volatile availability of the peers in a pervasive computing network, requiring opportunistic leveraging of resources when they are available.

- *heterogeneous devices*: pervasive computing mechanisms must support a variety of devices including laptops, PDAs, sensors, and lightweight domain specific devices (e.g., small medical tags [56] in a triage deployment.

- *compatibility*: owners of devices in pervasive computing networks will make different technology and configuration decisions. Rather then ignoring incompatible resources entirely, applications should be provided some basic support for interactions.

- *need for application coordination*: pervasive computing applications do not operate autonomously and must coordinate to gather data and access digital resources.

- *large scale and wide distribution of the network*: pervasive computing networks can grow very large, requiring coordination through multi-hop connections and scalable communication protocols.

- *locality of information and interactions*: applications' interactions are commonly defined by some abstract characterization of locality.

We address these concerns by rephrasing the pervasive computing environment into metaphors that are comfortable to an application developer in Chapter 2 of this dissertation. These metaphors allow conversations to become intuitive, first-order components of the application. This work takes special care to specify techniques for the long-term maintenance of the conversations by the underlying system. However, this maintenance is not the only challenge to providing the same conveniences available in traditional networks. The challenges listed above are also a hindrance to just *finding* the resources for a client application.

The approach taken by the traditional technologies noted above (*sockets*, *RMI*, *WebServices*) relies on the notion of ids in some global namespace. While these ids are not necessarily globally unique to each host in the network, they incorporate the notion of a logical location or address (e.g., IP addresses or domain names). When locating resources in pervasive computing environments, the identity of a host (when it is assigned one) is often less important than the capabilities that it provides. Unsurprisingly, systems have been proposed and studied for searching resources by the context they provide [2, 71].

While the field remains a very active area of research, discovery systems often focus only on the context provided by the resource itself. We believe that the discovery in a network should also include attributes describing the nodes and connections that support the inevitable long-term conversation that will be established between peers. We also believe the value of discovery increases with access to a greater variety of attributes, especially when values are from

multiple abstraction layers in a host system. Furthermore, we believe that a client should be able to specify combinations of attributes at runtime that were not (and perhaps could not be) envisioned by designers of the networks and the embedded resources. This is a much more general approach to defining dynamic context than current literature considers (e.g., geographical or logical distances).

Unfortunately, the very nature of pervasive computing environments can impede the evolution of discovery protocols to provide new attributes and even more the addition of new ways of combining attributes. Specifically, once nodes are deployed, altering or updating protocols and applications can be very difficult, if not impossible. In these cases, clients of shared pervasive computing deployments are restricted to the interactions envisioned by the original developers and deployed on that network.

For example, an algorithm that anticipated the need to conserve battery power may become useless when it is altered to prioritize based on location or temperature metrics. Recent research has provided techniques for updating software so long as all clients are synchronously updated with the network [22, 63]. In cases when resource and network providers cannot require clients to adapt simultaneously to network updates, complex backward-compatibility functionality is then required. This additional logic limits the progress of new applications and can result in unstable and brittle environments.

Discovery mechanisms for pervasive computing environments must be durable in the face of these challenges. They must allow access to broad swaths

of cross-layer information, and allow for the evolution of protocols and algorithms. In Chapter 3 we introduce a collaboration model that allows nodes to use cross-layer information in novel ways without requiring updates or additions to the nodes already deployed in a network. This allows developers to evolve the techniques used for finding resources in the pervasive computing network and ensures that application developers can establish the conversations they desire. The model also allows many different protocols to be used concurrently by embedding behavior into the exchanged messages themselves.

We feel the challenges addressed by our models are not part of pervasive software's *essence* as described by Brooks [12]. They are *accidents* – merely difficulties that hamper its production. The proper application of abstractions and tools can reduce their impact and lead to higher quality, and ultimately higher value, applications.

1.1 The Intelligent Construction Site

Many scenarios and case studies for pervasive computing research have been proposed in the past. These include support for first responders [56, 84], aware homes [49], taxi cab reservations [83], and electronic tourist guides [18]. However, these applications address scenarios that are point-deployments, allowing (and requiring) software developers to be intimately familiar with the details of the software and hardware of all the participating devices. Developers are also in control of all of the devices in the deployment, allowing them to redeploy software or replace hardware as the application requires. But most

importantly, each deployment scenario is focused on performing a single task with all participants pre-configured to support it. In contrast we use a scenario focused on performing many discrete and unrelated tasks at the same time in the same environment.

In this section we will briefly describe the "Intelligent and Automated Construction Job Site" case study[1], a previously used example for pervasive computing technologies [33, 42, 44, 48, 81]. This scenario envisions the deployment of sensing, instrumentation, communication, and decision support onto an active construction site. Members of the civil engineering and construction fields explicitly recognize that coordination and automation provided by these devices promise to drastically improve the effectiveness and productivity of many facets of construction execution [26]. To realize these benefits, the hardware and software deployed to construction environments is expected to participate in a variety of tasks and collaborate with peers on an ad-hoc basis.

The construction site is typical of pervasive computing environments and presents many of the challenges discussed in above. For example, the construction site in Figure 1.1 shows users, vehicles, and even raw materials that communicate through wireless and cellular networks. In the lower right corner of this figure, sensors attached to a crane are able to determine its load and monitor its movement. On the right, materials are delivered and entered into inventory automatically. At the top, a detector is installed to monitor

[1]Sometimes referred to as just "Intelligent Construction Site"

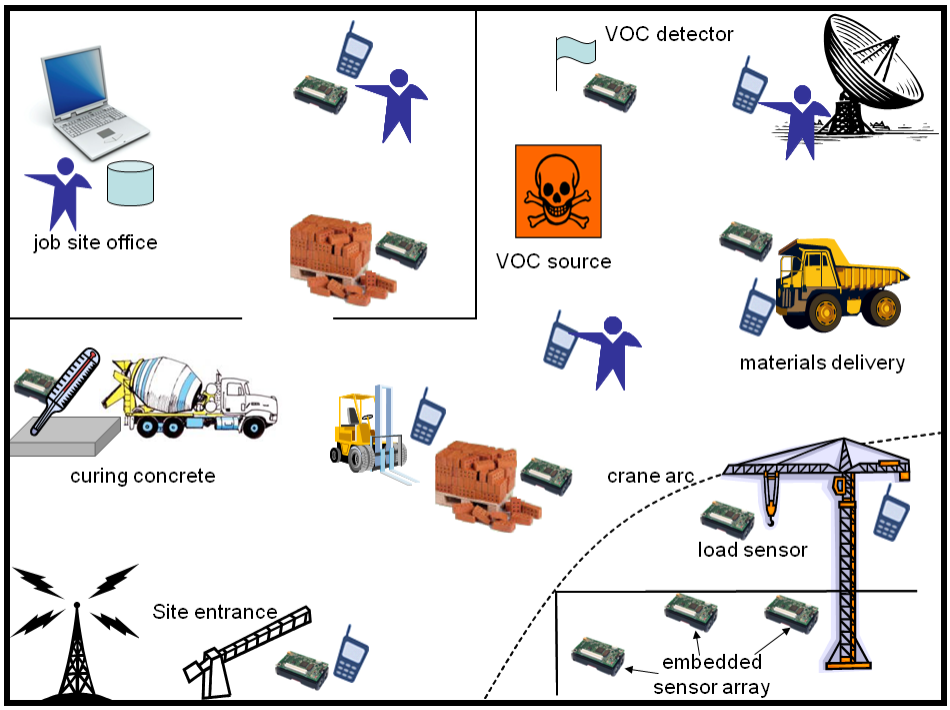

Figure 1.1: Intelligent and Automated Construction Job Site

the release of any dangerous volatile organic compounds (VOCs). Inside the office at the upper-right, managers coordinate the movement of materials and personnel on the site, and on the left a concrete pad is monitored while it cures.

Many construction sites are adopting the level of instrumentation shown in this figure, but the lack both the low-level and high-level communication and coordination technologies prevents the fulfillment of the Intelligent Construction Site promise. In this dissertation, we pull examples from this case study to explain the motivations, models, and evaluations of the work presented.

1.2 Goals

In this dissertation, we address the challenges of creating and maintaining long-term conversations between peers in a pervasive computing environment. As we have already discussed, there are significant differences in the conversations occurring in traditional networks and those in pervasive computing environments. Among these differences is the use of contextual information to select peers, the number of peers involved in any one conversation, and the instability of the underlying connections over its lifetime.

Our approach to confronting this task is to propose new models for these interactions. We present the *Application Sessions Model* in Chapter 2 to give a concrete form to conversations. This model defines the configuration that an application must supply and the methods of maintaining conversations once they are established. As this model was derived, it became clear that a new method for dramatically flexible discovery was also needed. The *Evolving Tuples Model* presented in Chapter 3 was derived to fulfill this purpose. After formally describing and documenting the models, we show they are not only useful, but potentially successful by building prototype implementations and applying them to exemplar applications taken from the Intelligent Construction Site domain discussed above.

In this dissertation, we claim these models are *promising solutions* to manage the complexities of pervasive computing conversations. The work required to make these comprehensive conclusions is far outside the scope of this work. The explicit aim our work is to show the proposed models and

designs are *feasible* solutions to address these challenges. We do this through the following six explicit contributions:

1. **The Application Sessions Model:** This contribution defines the nature of *conversations* in pervasive computing applications, lending formal semantics to several conversation types. This model can then be used both to document the behavior of these conversations for informing application developers and to reason about the applications built using the model.

2. **Application Sessions Middleware Design and Prototype:** The second contribution is a practical realization of the Application Sessions Model. They are used as the basis of our feasibility study and enable the development of pervasive computing applications outside of those described in this dissertation.

3. **An Application Sessions Feasibility Study:** In this contribution we demonstrate the feasibility of both the Application Sessions Model and Middleware through case study applications from the Intelligent Construction Site domain

4. **The Evolving Tuples Model:** This contribution defines a method for allowing messages exchanged in pervasive computing applications to impact their own content by calculating abstract values. The model serves to document the structure and behavior of these messages to guide

developers interacting with them and to reason about the operation of pervasive computing networks that process the messages.

5. **The Evolving Tuples Model Prototype:** In this contribution we realize the Evolving Tuples Model for both real pervasive computing devices and for software simulation. This prototype is used as the foundation for our feasibility study as well as the groundwork for protocols under development elsewhere.

6. **An Evolving Tuples Model Feasibility Study:** The final contribution validates the model's anticipated characteristics and the model's feasibility through both theoretical examples and practical experiments on actual hardware and in software simulation.

The remainder of this document is organized into three chapters. The next chapter focuses primarily on the Application Sessions Model and contributions 1, 2, and 3. Chapter 3 concentrates on the Evolving Tuples Model and contributions 4, 5, and 6. The final chapter summarizes our findings and discusses several avenues for future research.

Chapter 2

Application Sessions Model

This work is motivated by an increasing demand for pervasive computing applications, and the ensuing need to enable application programmers to create them. As we described in the previous chapter, one of the key abstractions that is missing from the suite of current tools is support for conversations in pervasive computing environments. In this chapter we specify a model for representing and managing these long term conversations for the applications. We specifically design this model to remove the application programmer's need for intimate familiarity with the details of communication. Instead, applications declare the characteristics of required conversations and delegate construction and maintenance tasks using the *Application Sessions Model*.

The Application Sessions Model addresses the challenge of long-term conversations by creating intuitive metaphors that conceal the complexities of establishing and maintaining connections over long periods of time. These metaphors then become the application developer's view of the environment. This shifts his focus and attention away from the network details and back to problems in his domain of expertise. Application developers no longer need

to learn the details of discovery and routing protocols, nor how to deal with the arrival and departure of network nodes, nor the libraries that implement them. This separation of concerns both architecturally and organizationally will lead to more robust, successful, and ultimately *valuable* software for pervasive computing environments.

The primary metaphor that our model uses to characterize the environment is the application session. We define an **application session** to be *a collection of robust logical connections to one or more networked devices over which application data is exchanged*. To create the connections that are members of the application session, a set of remote applications, devices, sensors, and actuators (collectively referred to as "resources") must be selected. The model defines three components to manage this selection, the *resource predicate*, the *resource preference*, and the *session limit*. The resource predicate describes how an application defines *interesting* for a resource. The *resource preference* describes how an application defines *more* interesting. The *session limit* is used to cap the number of resources assigned to an application session. These elements are typically based on the application's non-functional requirements or dynamic environmental characteristics. Once selected, the application's choice of *connection maintenance strategy* determines how the system should manage the short-term connections to the resources. For example, we may or may not wish to attempt to reconnect to resources that have become unavailable.

Any framework designed for pervasive computing environments must

support a broad array of application domains with widely varying requirements. An analysis of all application domains is impossible as we anticipate widespread adoption of pervasive computing to affect nearly all aspects of our lives. Instead, in this chapter we use the Intelligent Construction Site scenario to exemplify the unique challenges of building pervasive computing applications.

In the next section, we describe hypothetical applications to provide more specific motivations. Sections 2.3 and 2.4 present the Application Sessions Model and a design for implementing the model. The hypothetical applications are then used in Section 2.5 to show how the middleware implementation is used, and to evaluate the model's usefulness as a pervasive computing programming construct. A discussion of work directly related to this model is included in Section 2.2.

2.1 Exemplar Applications

The fundamental tenants of the "Intelligent and Automated Construction Job Site" case study have been outlined previously. In this scenario, people and equipment are assembled on a site for a wide variety of tasks. These tasks differ broadly in a number of aspects such as time-scales and the number of participants. For example, administration offices are often present on the site before the site preparation begins and may remain until well after the project is complete, interacting with hundreds of devices and individuals during this time. Conversely, delivery vehicles may only be on site for minutes

or hours and may only directly interact with a receiving clerk. To improve efficiency, the Intelligent Construction Site uses pervasive computing applications to monitor and mediate interactions between the people and equipment present. Some of these applications involve short duration interactions with a simple device, as in downloading a file or punching the clock. Other tasks such as environmental monitoring require more open ended interactions with multiple resources. In this section, we select three specific hypothetical applications to serve as exemplars of the large variety of applications envisioned for this scenario.

Application 1 - Monitoring the Curing of Concrete

To ensure that it is strong and durable, concrete must be cured for three to seven days. During this process, the bond between the concrete paste and the aggregate becomes stronger. To properly cure, concrete must be kept at the proper level of moisture; to ensure proper curing, uniform moisture levels must be maintained throughout the batch. Typically one of two techniques is used to control these levels: a fine mist of water or a covering of plastic sheeting. In the Intelligent Construction Site, a few humidity sensors are placed to measure moisture levels, and many temperature sensors to monitor the progress of reaction through the heat it releases. Our hypothetical application will monitor curing conditions by periodically collecting values from these sensors and synthesizing the results locally.

The results provided by this application may be passed to control software to adjust water valves or notify staff when conditions exceed certain thresholds. The readings could also be used by building inspectors or insurance companies to ensure compliance, or by architects and engineers to analyze any failures that occur later in the process. For this version of our hypothetical application, we will assume that readings need only be taken once every ten minutes or so, a very low sampling rate by intranetworking standards. Given the low rate of data acquisition, a constant datastream from the sensors would needlessly consume network resources to transmit the data and maintain network routing information in the dynamic networks. Streaming data from the sensors to the application is a poor choice for this application. Instead a state-less query-and-respond approach should be used. Periodically, each sensor is contacted for its current reading, which it sends back immediately. There is no need to maintain any information about the query or the source of the query after the sensor has responded. We avoid needlessly allocating system resources on the sensors to maintain state while also realizing a drastic reduction in network traffic overall.

To ensure the correct data is collected by the application, it may be necessary to select sensors subject to constraints on a variety of attributes. The selection, or *discovery*, of resources must leverage both low-level data (e.g., network addresses) and application-level data (e.g., building level), or even dynamically inferred data (e.g., proximity to a specific sensor). Sensors may even be specified by a combination of their attributes. A model must be

able to perform the discovery task using any of these data types to support this application.

Application 2 Crane Monitor

Monitoring the usage of equipment can be an important component of resource- and risk-management plans. The "Crane Monitor" application seeks to record usage information reported by sensors attached to cranes on the construction site. Once the data is recorded, it can be analyzed off-line to schedule routine servicing and other maintenance activities. To be effective, this application requires much finer-gained information from the sensors than the previous application. A constant connection to each sensor is justified for this purpose.

When the application is started, we first select the sensors on the site whose data is to be recorded. We then constantly monitor all of the reported values as long as the crane is on the site. In effect, we are carrying on a long-term conversation with these sensors. This application, and thus the conversation, will run the entire time a crane is on the construction site. This time period may range from hours to days, and even months. As factors contributing to connection and link quality vary, this application must reestablish broken connections. Some sensors may be offline for long periods of time (over-night or longer) preventing routing layers from providing route-repair for the application. While reconnecting to sensors, the application must also be careful to avoid connecting to newly arrived sensors providing similar data.

Application 3 Monitoring Danger

Safety is an ongoing concern for workers on the Intelligent Construction Site. Despite design foresight, the conditions considered to be dangerous change over time. Arrival of new components on the site, the progression of the project, changes in codes, and simple changes in perception all contribute to the dynamic definitions of "danger". On the site, some dangerous conditions are explicitly sensed by purpose-built hardware (e.g., volatile organic compound sensors, fire detectors), while other conditions must be inferred through the cooperation of various sensors (e.g., trucks in each other's path). Inferred dangers are specified by defining formulas to be calculated using current readings from various sensors. Formulas can be set up by the site management or by individuals who wish to define their own "dangerous conditions". No matter how the dangerous condition is defined, our application is only interested in conditions that apply to its user. Additionally, the application must provide appropriate alerts based on the type of danger. Alerts to routine conditions such as passing trucks and working cranes should not interfere with the worker's normal activities. However, the detection of fire should be given immediate attention.

Unlike the query-and-respond technique used in the previous application, the application will monitor the status of devices at all times. More specifically, it will monitor devices reporting danger in geographical proximity to the user. Over time, dangerous conditions and the sensors reporting them

will come and go from the network. Therefore it is the job of the supporting components to provide mechanisms for retrieving the data that is most relevant to the user at any given time.

2.2 Related Work

It has previously been shown that adopting a coordination approach to handling the unpredictability inherent in mobile computing can lead to solutions that simplify programming [68]. Several middleware solutions have taken this approach [27, 45, 61] but focus on exchanging data items in dynamic conditions and not on generic resource usage in pervasive computing situations. As pervasive computing has come to the forefront, projects have increasingly focused on providing dynamic access to a changing set of resources. Many efforts mediate quality of service requirements by leveraging object mobility to enhance application responsiveness and network-wide performance metrics [31, 37, 69]. These approaches focus on bringing objects closer to clients instead of on mobile clients that require inherently location-dependent resources.

Projects closer to our goals update bindings between clients and services as processing or environment dictates [10, 50]. A *follow-me session* [35] provides constant connectivity to services by transferring a connection from one provider to another. Context-Sensitive Bindings [35, 67] implement the follow-me session by defining a *context* and selecting resources from that context that match an application's specification. This approach decouples the remote resource's implementation from its realization at the client, and it

handles many of the migration concerns that apply in our framework's type session. Finally, this approach favors complete transparency and assumes that a resource binding should always be transferred, subject to an application's specified policies. It does not allow for the other types of sessions we described nor does it enable group interactions.

Service Oriented Network Sockets [70] provide support for selecting new services based on an application's configuration. This approach, however, addresses client concerns at the lower level of network sockets as opposed to application objects. In addition, the approach assumes the availability of well-accepted service discovery mechanisms that it uses to gather *all* matching services locally before deciding which services to connect to. This can incur significant amounts of overhead in networks that are dynamic, large in size, or contain numerous satisfactory services. iMash [6] addresses migration of user connections as services are discovered but does not provide a convenient client interface for specifying resources or enforcing session semantics as defined in this paper. In addition, iMash relies on knowledgeable intermediaries that handle service switches on behalf of clients and resources. Similarly, Atlas [20] uses a central server to mediate the transfer of a service binding from one provider to another.

Scenes [3] and *Network Abstractions* [66] provide alternate mechanisms for selecting resources, but they do not allow the client to specify limits or orderings on the returned set of resources. In the case that networks implement these constructs to provide discovery support, it may be possible to translate

our resource predicate into the proper input for these mechanisms. This would be just one example of leveraging existing functionality in the network to optimize our middleware's operations.

In summary, our approach differs from these projects in several ways. First, we seek not to limit an application's sessions to a single type but to adapt to an application's needs, including simple queries, lasting connections, and transparent resource migration. Second, while we aim to decouple the semantics of application sessions from the implementation supporting the session, we recognize that the extreme scale and device constraints necessitate communication protocols tailored to particular session requirements. As discussed previously, in the application sessions middleware, it is possible to support a suite of communication protocols that efficiently support a variety of coordination semantics.

2.3 The Application Sessions Model

From an architectural view, our model sits directly below the domain-specific application and just above the wide variety of resource discovery and routing protocols used to find and access remote resources. In implementation terms, the application sessions model replaces the use of *sockets*, *RMI*, and other middleware solutions conceived for traditional networks. In this section we provide the operational semantics of the Application Sessions Model. These semantics demonstrate the model's ability to transparently adapt its functionality to a mix of application requirements and changing operational

environments. We will first discuss our representation of resources and then define and discuss the three specifications that define the resources in an application session: predicate, limit, and preference. With these three components in place we will show *how* resources are selected, and then discuss of how connection maintenance strategies effect *when* resources are selected.

2.3.1 Resource Representation

As part of the Application Sessions Model, we specify a model of the resources themselves. This is only used to stipulate how our components interact and should not be construed to imply a complete model for resource implementations. Actually, this model seeks to declare a minimal interface which may be expanded by individual resource providers to support other technologies.

Resources are viewed as a set of (potentially nested) values. Each value is associated with a key, or *name*, that describes the nature of the value and a *type* that describes the representation of the value. The *name* of a value is simply a character string, while the *type* is an element of some large set of enumerated data types available to all participants. Like a programming variable, the values are shared between different processes and may change at any given time. This results in a semi-structured data model [1], leaving the definition of value dependence and interaction to the applications. For example, a "location monitor" resource might be represented in this model as shown in Figure 2.1. Here the model defines which three fields must appear

Name	Type	Value
latitude	real	30.2867
longitude	real	-97.7364
elevation	real	152.1
expires	integer	1190935202938

Table 2.1: Sample "location monitor" resource

in each tuple, but not how many tuples nor the value of any field.

Obviously there is a need for a standardized structure for names, types, and units to ensure unambiguous definitions; an issue also affecting the Evolving Tuples Model discussed in Chapter 3. In both models it is necessary for users to share an understanding of how data will be named. In this example, clients of the resource assume the meaning of the field named "latitude." Distributing shared naming schemes in dynamic heterogeneous environment is a well-studied problem, and not one we intend to undertake here. Instead, we simply assume the presence of this shared knowledge and use it to semantically tag data.

2.3.2 Resource Predicate Function

In our model, the client application provides the application session framework with a *resource predicate* that is used to filter available resources. This predicate, $pred(x)$, is simply a conditional statement that evaluates to $true$ or $false$ for any given input resource x. Since the client-supplied predicate only allows for a boolean result, this function should be viewed as describing the minimal acceptable characteristics of the target resources. A predicate will

typically include the type of resource (such as *thermometer* for the *Curing Concrete* task) but may also filter on attributes such as the sensor's *owner* or *location*.

When supported by the underlying discovery mechanisms, the resource predicate could specify connection or environmental attributes as well. For example, the client could specify a maximum hop-count to prevent the underlying discovery protocols from searching in distant areas of the network. Even though measurements like maximum hop-count are associated with the connection between a client and host, they can be modeled as attributes assigned to each resource on the host.

Metrics such as the latency and bandwidth of a physical connection could also be measured and assigned to the remote resource as meta-attributes in the same way. While some of these connection specific attributes are available in some discovery protocols [41, 43, 47], the Evolving Tuples Model described in Chapter 3 is specifically designed to support this type of attribute evaluation.

2.3.3 Resource Limit

Since we are designing a model with the pervasive computing environment in mind, the scope of a client's request must be limited to avoid overloading the network, the hardware, or the client application. Part of this limitation can be provided by including time-varying attributes in the resource predicate function described above. However, this technique is not always sufficiently

restrictive. We include another parameter to explicitly limit the number of resources that should be provided by the infrastructure for a particular session. The *resource limit* parameter simply specifies the maximum number of resources that will be included in the session at any given time.

While the primary motivation for specification of a limit is to restrict the load imposed on the network and remote resources, thought it can also be used to intimate an application limitation. For example, the relatively small screen size of hand-held devices may limit the amount of data that can be presented. The application can use the resource limit to limit the number of resources that are in the session as a way of reducing the data passed to the user interface.

2.3.4 Resource Preference Function

Using a client's predicate function, the model is able to determine which of the available resources are acceptable for a particular session. However, we would like to return not just acceptable results, but the *best possible results* to our client. This is particularly important when the client has specified a maximum number of resources that is smaller than the number of acceptable resources. In this case, we must make *some* decision as to which physical connections are established and which are omitted. Simple algorithms for this selection would include non-deterministically choosing resources, or perhaps choosing the first n resources that are returned by the discovery protocols. These methods certainly have their place and serve as the default algorithms

for choosing between otherwise equal resources. However, we allow the client to specifically dictate a selection algorithm through the *resource preference function*.

This function allows the model (and the discovery protocols if they support it) to reduce the set of resources that are returned by simply providing a means for comparison. This comparison need not provide a detailed analysis, just an indication of preference between two resources, if it exists. With this function, the model can return an ordered list of resources that are "better" than all other available resources, where the definition of "better" is provided by the client. For example, the client application in the Danger Monitor example above would prefer to use resources that are geographically closer to the user. We define the *resource preference function*, $pref(x, y)$, to provide this functionality.

$$
\begin{aligned}
pref(x,y) > 0 &\iff \text{x is ``better'' then y} \\
pref(x,y) < 0 &\iff \text{y is ``better'' then x} \\
pref(x,y) = 0 &\iff \text{no preference}
\end{aligned}
$$

As we discussed in Section 2.3.2, discovery and routing protocols may be able to calculate connection specific information such as hop-count and latency. Since we are modeling these values as attributes of the remote resources, these values can also be used for preference functions in addition to predicate functions. For example, a client could use hop-count in the resource predicate to limit the scope of discovery to nearby (in the network) resources,

and then use latency in a preference function to select the "quickest" of those resources.

2.3.5 Resource Selection

Using the client's resource predicate function (*pred(x)*), resource preference function (*pref(x,y)*), and a resource limit (n), we can now create a function $findBest$ that returns a list of resources ordered by preference. To correctly support all of the connection maintenance strategies discussed in the next section, the list of resources returned by this function is padded with null values ($\emptyset$) to the size specified by the argument n.

$$
\begin{aligned}
&findBest(pred, pref, n) = R[0..(n-1)].(\\
&\qquad \forall r \in R :: r.reachable \wedge \\
&\qquad \forall r \in R :: pred(r) \wedge \\
&\qquad \forall r \in R, r' \notin R, r'.reachable :: pref(r) \geq pref(r') \wedge \\
&\qquad \forall i \in 0..(n-2) :: (R[i+1] = \emptyset) \vee (pref(R[i], R[i+1]) \geq 0) \\
&\quad)^{1}
\end{aligned}
$$

This function returns a list of reachable resources $R[...]$ such that its size is equal to n, all of its members satisfy the client supplied predicate function, and the members are ordered according to the client-supplied preference function. The exact order may be non-deterministic since resources considered equal by the preference function can appear in any order relative to each other.

[1] The construct $x.$(`condition`) non-deterministically returns any value that matches the `condition` and will return $\emptyset$ (null) immediately if no value can be found.

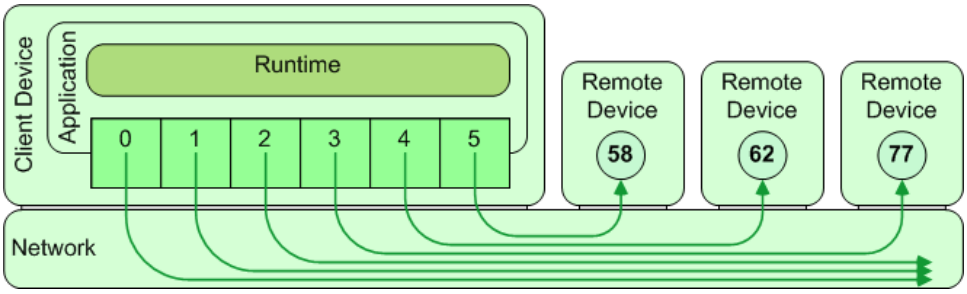

Figure 2.1: High-level view of Application Session in runtime architecture

2.3.6 Connection Maintenance Strategy

Using the three specifications above (predicate, limit, and preference) and the $findBest(...)$ function, the model can now select an initial set of resources to satisfy the client's request. However, in a pervasive computing environment, the client's conversations may last longer than the underlying best-effort connections can support, especially in environments including physically mobile nodes. Thus, rather than using the results of *findBest(...)* directly, applications call a new function *newSession(...)* which returns a *session*. Applications interact with a session in much the same way as a list, so we adopt a similar notation $S[i]$ to denote the i^{th} element of the session S. Once an application has created a session, each of the session's elements provides a connection to a remote resource as shown in Figure 2.1. In this figure, for example, when the application accesses $S[5]$, a connection to the remote resource with the value 58 is returned. The elements of this list can be read by the client application sequentially $(S[0], S[1], S[2]...)$ or randomly $(S[3], S[2], S[4]...)$, but cannot be modified by the application.

Though similar to the `findBest(...)` function, the `newSession(...)` function requires an additional parameter, the *connection maintenance strategy*. This argument specifies an invariant to be maintained on the connections in the session. To preserve the invariant, the session's contents may be removed, reorganized, and even replaced over its lifetime. This shuffling of resources allows the model to efficiently manage system and network resources in support of the applications' long-term conversations.

The Application Sessions Model allows the client to select one of three connection maintenance strategies. Below we describe each strategy, define its invariant, and give an example operational definition which satisfies the invariant. Since the invariants and definitions below describe the state of the session over time, we must introduce a bit of notation: a subscript attached to S to denote the logical time of the session. Thus S_0 refers to the session at time 0 (the instant it is created) and S_τ refers to the same session at some time τ. Combined with the aforementioned index notation, $S_\beta[5]$ designates the fifth element of session S at time β.

We first describe the *Query Session Strategy*, used when the client needs only a short-term connection without the overhead of maintaining a stable connection. We then describe the two cases for long-lived connections. The first of these, the *Provider Session Strategy* is used when the client wants to maintain the connection with the **same** resources throughout the session. The *Type Session Strategy* is used when the client wants to maintain a connection to the most preferable resources throughout the session.

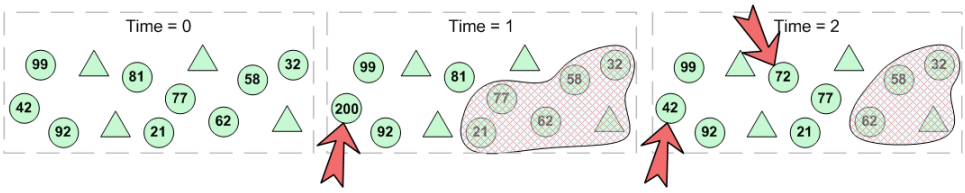

Figure 2.2: Example network configurations at three moments in time

To give the reader a more concrete feeling for the strategies, a graphical representation of the short-term network connections is provided with each description. In each, the session is depicted at three different (logical) times with the three different network configurations shown in Figure 2.2. At time 0, all nodes in the network are reachable in the network. At time 1, six of the nodes have become disconnected and the left-most node has changed values from 42 to 200. At time 2, two of the disconnected nodes have rejoined the network, the left-most node has reverted it value to 42, and another node has changed values from 81 to 72.

To emphasize the distinctions between the strategies, each of the examples below passes the same parameters (excepting the strategy) to `newSession(...)`: the predicate selects only circle-shaped resources, the preference function favors larger values, and the session is limited to six resources. Using these values, the initial resources selected for each session are shown in Figure 2.3.

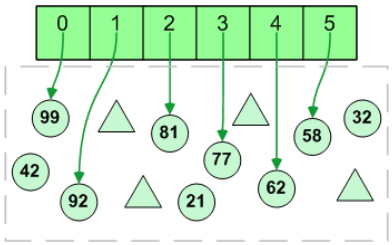

Figure 2.3: Initial resource selections in example network for all sessions

2.3.6.1 Query Session Strategy

The *Query Strategy* is used when an application will only use each remote resource once. This implies that the infrastructure is free to dispose of any system resources associated with the connection to the remote resource once it has been used by the application. Additionally, the Query Strategy also frees system resources for the connections to resources that have left the network since the session's creation. This aggressive management of system and network resources is particularly useful when the client or remote devices have limited system resources.

Figure 2.4 depicts this interaction over the three network configurations discussed above. This figure shows the connections at $S[3]$, $S[4]$, and $S[5]$ are released when the associated resources leave the network at time 1. Even when one of those resources rejoins at time 2, the connection is not reestablished. The same connection release can be seen as the client uses the remote resources $S[0]$ and $S[1]$ at times 0 and 1, respectively. Once these resources are used, the connections are released and are never reestablished.

The formal definition of the fundamental invariant of the Query Strat-

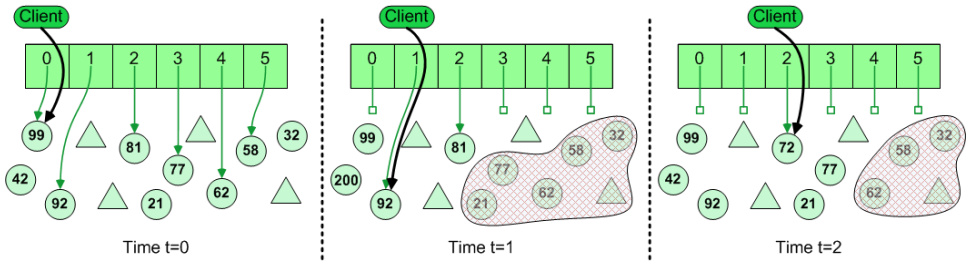

Figure 2.4: Resource assignments over time using Query Session Strategy

$$\Phi(t) = \langle \text{ set } \forall i, \tau : \\ i \in [0..n) \ \wedge \ 0 < \tau \leq t \wedge \\ (S_\tau[i].used \vee \neg\, S_\tau[i].reachable) :: i \ \rangle$$

$$\underline{\textit{Query Strategy Invariant}} \\ S_0 = \mathit{findBest}(\mathit{pred}, \mathit{pref}, n) \ \wedge \\ (\forall i, j, t : \\ 0 < t \wedge i \in \Phi(t) \wedge j \notin \Phi(t) :: \\ S_t[i] = \emptyset \wedge S_t[j] = S_0[j])$$

Figure 2.5: Query Connection Maintenance Strategy Invariant

egy is shown in Figure 2.5. In this figure, the function $\Phi(t)$ serves only to simplify the rest of the definition. Given a time t, $\Phi(t)$ returns the indexes of S that hold resources that were either used or unreachable at some time τ between 0 and t. In other words, $\Phi(t)$ returns indexes of resources that should be disconnected. The *Query Strategy Invariant* says that at time 0, the session S_0 is equal to the results returned by the *findBest(...)* function. At times t after 0, the elements of the session that were assigned to used and unreachable resources (indicated by $\Phi(t)$) should now contain the `null` value ($\emptyset$). The elements representing unused and still reachable resources (those not returned by $\Phi(t)$) should still contain the original connections.

$\langle S[...] \Leftarrow \mathit{findBest}(\mathit{pred}, \mathit{pref}, n)\rangle$
co
$\quad \langle \mathbf{await}(S[i] \neq \emptyset \wedge S[i].\mathit{used}) \rightarrow S_t[i] \Leftarrow \emptyset\rangle$[2]
$\quad \|$
$\quad \langle \mathbf{await}(S[i] \neq \emptyset \wedge \neg\, S[i].\mathit{reachable}) \rightarrow S_t[i] \Leftarrow \emptyset\rangle$
oc

Figure 2.6: Query Connection Maintenance Strategy Operational Model

To give a sense of how this invariant can be satisfied in practice, we present an operational model of the Query Session Strategy in Figure 2.6. Here $S[...]$ is initially (atomically) assigned the values returned by *findBest(...)*. Then, any time a non-null resource in $S[...]$ is either used or unreachable, its entry in the session is immediately and atomically assigned the value $\emptyset$. Once this assignment is made, any system resources associated with the connection to the resource may be released.

In the model we purposefully avoid narrowly defining how a resource transitions to the "used" state. Our prototype system (described below) implements this condition as *any access to a resource's values*. However, this definition may be too constraining when dealing with the variety of devices that must be supported. The definition of "used" may even be a technology-specific function. This particular aspect of the Application Sessions Model will be revisited as deployment experience is gathered.

[2]The $\langle \mathbf{await} A \rightarrow B\rangle$ construct [4] allows a program to delay execution until the condition A holds. When A is true, the statements in B are executed in order. The angle brackets enclosing the construct indicate that the statement is executed atomically, i.e., no state internal to B is visible outside the execution of B.

The Query Session Strategy correlates with the Curing Concrete example application described in Section 2.1. For this application, a query session would be established for temperature and humidity resources in a particular area. The application would then access each resource in turn (i.e., "using" it) to download each sensor's readings. After each use, the underlying resources associated with the connection to the remote resource are immediately reclaimed. This illustrates the key benefit of the query session — the framework has managed these resources as efficiently as possible without any effort or interaction from the client (save the initial session creation). When the application needs another reading ten minutes later, a new session is created and the process repeats.

2.3.6.2 Provider Session Strategy

The *Provider Strategy* is designed to support client tasks that require long-term conversations with the same resources, even if "better" resources become available. This is useful, for example, when a task produces state at the remote endpoints that is necessary for subsequent operations. In a pervasive computing environment this can be a complex goal to achieve due to the inherent instability of short term connections. This instability may cause resources to be unreachable for long periods of time. In these cases, the Provider Strategy is responsible for reconnecting to any disconnected resources and maintaining a consistent view of the resources that are available.

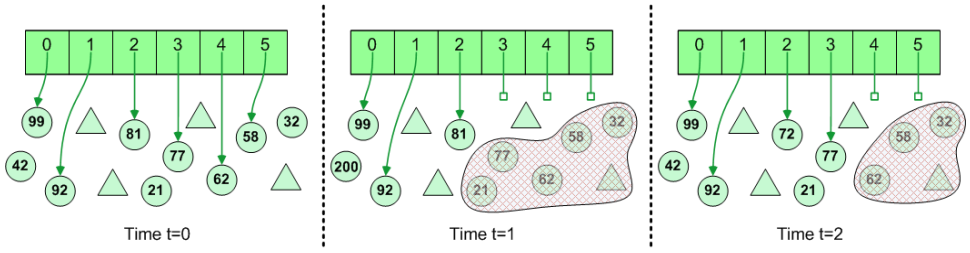

Figure 2.7: Resource assignments over time using Provider Session Strategy

Figure 2.7 continues our running example by showing how the application of this strategy effects resource connections. The initial connections are the same as the query session's at time 0. At time 1, when three of the originally included resources are no longer available, the session's connections are broken. However, the session reconnects to one of the original resources (node with value 77) when it becomes available again at time 2.

Figure 2.8 states the Provider Strategy's goal more exactly as the *Provider Strategy Invariant*. The invariant first specifies that the session at time 0 (S_0) is equal to the results of *findBest(...)*. The list Γ is a copy of $S_0[...]$ and always contains references to the original resources. When the initial resources are reachable at time t after 0, the session S_t holds a reference to them, and when they are not, the associated session element is set to `null` ($\emptyset$).

As with the query session, we provide an operational model in Figure 2.9 to show how the invariant can be maintained on the session $S[...]$. It specifies a private list $G[...]$ which holds the original remote resources of $S[...]$. Using the `await` construct, the framework waits until there is a element in S that is assigned to $\emptyset$ while its counterpart in G is reachable. When this occurs, the

$$
\begin{array}{l}
\underline{\textit{Provider Strategy Invariant}} \\
\quad S_0 = \mathit{findBest}(\mathit{pred}, \mathit{pref}, n) \ \ \wedge \\
\quad \Gamma = S_0 \ \ \wedge \\
\quad (\forall i, j, t : \\
\qquad 0 < t \wedge i \in [o..n) :: \\
\qquad\quad (\Gamma_t[i].\mathit{reachable} \wedge S_t[i] = \Gamma_t[i]) \\
\qquad\quad \otimes \\
\qquad\quad (\neg\, \Gamma_t[i].\mathit{reachable} \wedge S_t[i] = \emptyset))
\end{array}
$$

Figure 2.8: Provider Connection Maintenance Strategy Invariant

$$
\begin{array}{l}
\langle S[...] \Leftarrow G[...] \Leftarrow \mathit{findBest}(\mathit{pred}, \mathit{pref}, n) \rangle \\
\textbf{co} \\
\quad \langle \textbf{await} (S[i] = \emptyset \wedge G[i].\mathit{reachable}) \Rightarrow S[i] = O[i] \rangle \\
\quad \| \\
\quad \langle \textbf{await} (S[i] \neq \emptyset \wedge \neg G[i].\mathit{reachable}) \Rightarrow S[i] = \emptyset \rangle \\
\textbf{oc}
\end{array}
$$

Figure 2.9: Provider Connection Maintenance Strategy Operational Model

element in S is immediately and atomically updated to the value in G. In parallel, the model specifies the opposite transition: an element that is not $\emptyset$ which is unreachable is assigned the value $\emptyset$.

The Provider Strategy is a good fit for the Crane Monitoring task described in Section 2.1. In this application the client collects data from sensors attached to a crane over the course of many days, or even months. As network conditions vary over the course of the session and resources become unreachable, the `null` value is substituted for the resource in the shared list $S[...]$. When connections are reestablished, its entries in the list are switched back to the original resource references. In this way, the application is shielded from the short-term concerns of reestablishing broken connections. The application

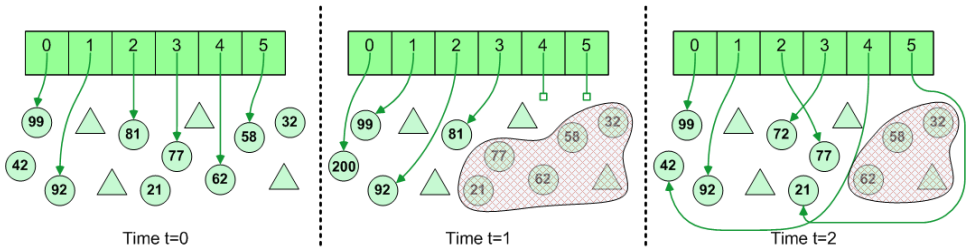

Figure 2.10: Resource assignments over time using Type Session Strategy

must acknowledge that these problems do exist by testing each element in the session for `null` before using it, but is not involved with remedying the problems.

2.3.6.3 Type Session Strategy

The Type Strategy is intended for an application task in which the exact identity of a resource is less important than how it compares to other resources. Stated another way, the client application wants the "best" resources at all times and does not mind if the order or identity of those resources changes over time. If we establish a session with the Type Strategy for our example network, the connections will be updated as shown in Figure 2.10. When the left-most node changes values at time 1, the first element of the session is updated to refer to this new "best" resource. The connections to resources that are no longer available are dropped, and the connected resources are assigned to the elements of the session in decreasing values.

At time 2, the left-most element changes values again and becomes the "worst" resource that is still connected in the network. As a result, the last

$$\frac{\textit{Type Strategy Invariant}}{(\forall i, j, t : 0 \leq t :: S_t = \mathit{findBest}(\mathit{pred}, \mathit{pref}, n))}$$

Figure 2.11: Type Connection Maintenance Strategy Invariant

$\langle S[...] \Leftarrow \mathit{findBest}(\mathit{pred}, \mathit{pref}, n)\rangle$
co
$\langle$**await**$(S[i].\mathit{valueChange}) \Rightarrow (S[...] = \mathit{findBest}(\mathit{pred}, \mathit{pref}, n))\rangle$
$\langle$**await**$(\neg\, S[i].\mathit{reachable}) \Rightarrow (S[...] = \mathit{findBest}(\mathit{pred}, \mathit{pref}, n))\rangle$
$\langle$**await**$(r \notin S[...].\mathit{reachable}) \Rightarrow (S[...] = \mathit{findBest}(\mathit{pred}, \mathit{pref}, n))\rangle$
oc

Figure 2.12: Type Connection Maintenance Strategy Operational Model

element of the session is connected to this resource. The updated value, 72, on the node previously assigned the value 81 causes the connections originally assigned to elements 2 and 3 to be reversed. The result of all these updates is to assure that the references are connected to the session elements in order of decreasing value. We express this formally as the *Type Strategy Invariant* in Figure 2.11. This invariant allows the client to simply iterate through the elements of the session to find the best resources *at any given time*.

The simplest way to model this strategy is to periodically update the entire list with the results of a full reevaluation of available resources (i.e., the `findBest(...)` function). We can avoid needlessly reevaluating `findBest(...)` by waiting until there is a change that could affect the elements of the session. Specifically, we wait for one of three conditions: a change in the value of a resource in the session, the departure of a resource in the session, or the arrival of a resource not in the session. When one of these three events occurs, we re-execute our selection function `findBest(...)` and

update the session with the results.

The Type Strategy would be used to implement the danger monitor example above. Here the original sessions would be configured to choose the closest resources that represent dangerous conditions. As network conditions fluctuate, and as sensor values change, the list of resources is constantly updated to present the client application with the most current readings.

2.3.6.4 Session Strategies

The Connection Maintenance Strategies defined in the Application Sessions Model are designed to support the majority of applications. There will inevitably be applications that require semantics that lie outside those provided above. In these cases, it may be necessary for the application to provide its own strategy definition. However, even in the absence of a formal survey we believe that these strategies cover a large majority of potential pervasive computing applications. Furthermore, we believe any subset of these strategies will not be sufficient to provide the semantics required to implement a large number of applications. Therefore, the Query, Provider, and Type strategies presented here form a necessary and nearly complete set of strategies to implement applications for pervasive computing environments.

2.4 A Middleware Design

In this section, we describe a middleware based design to support the application sessions model. This is not meant to imply that the application

sessions model is best implemented with a middleware based architectural design. As software engineers, we are acutely aware that any definition of "best" can be quite volatile. Implementations based on Mobile Agents and peer-to-peer architectures have also been considered and may prove to be superior architectures for other purposes. However, our goal here is not a traditional metric such as speed or resource usage, but rather the architecture's usefulness as an explanatory tool. To this end, we require our design and implementation to (1) minimize the need for deep knowledge outside the particular pervasive computing application domain and (2) reflect the components of the model itself. We feel that the middleware pattern addresses both of these goals rather succinctly.

The middleware architectural pattern is specifically designed to simplify the details of distributed communication and coordination into high-level primitives presented to the application developer [23]. The choice of high-level primitives is left to the designer which, in this case, we choose to be the primary components of the Application Sessions Model. To different software designers, this definition (and others like it) can take many different forms. For example, some definitions of middleware attempt to *hide* the distributed nature of the environment as much as possible and create what appears to be a "single integrated computing facility"[59].

When parts of the distributed computing problem are hidden from the application, designs typically adopt one aspect to be the primary dimension of the middleware. This leads to several families of software designs such

as "Transaction Oriented," "Message Oriented," "Object / Component Oriented," "Real-time Oriented," and so on. While the following design is clearly a member of the "Object Oriented" family of solutions, our approach differs from the traditional goals of middleware. We choose to hide only the mechanics of the distribution and present all of the potentially interesting properties of the environment, including its many independent nodes, to the application programmer as *cleanly and clearly* as possible. We feel that this gives the application developer more control and flexibility while developing applications and postpones preemptive optimization.

Since our middleware design and implementation directly reflect the model, our discussion here mirrors the organization of Section 2.3. We begin this section with our model of resources and an introduction to the predicate and preference specification languages. The middleware's analog for *findBest(...)* is presented in Section 2.4.4 and it's approach to ensuring the strategy invariants is discussed in Section 2.4.5.

2.4.1 Resource Model

In a middleware design, resources must be modeled with a uniform interface that balances the needs of applications and the typical capabilities of the discovery and routing protocols available in the environment. The model in Section 2.3.1 calls for resources to be modeled as sets of $\langle key, type, value \rangle$ triples. Our design is targeted for a strongly typed language, allowing us to infer the *type* at runtime from the *value* of each tu-

Resource

```
Object get(key)
```

Figure 2.13: Pseduo-code for the Resource Interface

ple. This reduces the design and implementation of resources to a set of $\langle key, value \rangle$ pairs, a very common structure referred to as an "associative container" or (more commonly) a "Map". Once a resource is acquired, its values are typically accessed as `resource.get(key)`, and the type can be inferred by `resource.get(key).getType()`. A pseudo-code version of the `Resource` interface is shown in Figure 2.13.

This brings us to the problem of acquiring the resource. Since most resources are accessed through a network, the components that we directly interact with are typically local proxies that forward messages to remote resources. Each proxy's implementation is specific to the particular combination of network and discovery protocols used to deploy the resource. To unify this diversity, we define an interface for our middleware to communicate with these protocols. The `Discovery` interface in Figure 2.14 separates the deployment-time specifics of protocol interactions from the development-time concerns addressed by our middleware design. Implementations of the `Discovery` interface provide a mechanism to supply resources (local, proxied, or otherwise) through the member method `Discovery.find(...)`. Once we discuss the middleware's versions of the predicate and preference functions, Section 2.4.4 below discusses how the `find(...)` method is used in our design.

In addition to providing resources on-demand, many discovery and

```
Discovery
    Collection<Resource> find(...)
    addListener(listener)
```

Figure 2.14: Pseduo-code for the Discovery Interface

```
DiscoveryListener
    onConnect(resource)
    onDisconnect(resource)
    onValueChange(resource)
```

Figure 2.15: Pseduo-code for the Discovery Listener Interface

routing protocols also provide support for asynchronous notification of node arrivals, departures, and updates to remote values. To use this functionality, a client typically registers a *listener* with the discovery mechanism or with the resource itself. Figure 2.15 presents the `DiscoveryListener` interface used by the `Discovery` implementations to make these asynchronous notifications. Like the `Discovery` interface, the primary purpose of this interface is to provide a single coordination mechanism between the middleware and the discovery protocols available at a specific deployment. The notification events passed to the middleware are used to maintain the session invariants, a process detailed in Section 2.4.4. When direct support for listeners is not provided by a discovery protocol, it can be approximated by a `Discovery` component by periodically polling the network's membership and values to detect changes.

2.4.2 Resource Predicate Specification

The resource predicate acts as a filter to limit resources considered for inclusion in an application session. Traditional network applications use

$$
\begin{array}{ll}
A & \rightarrow \textit{attribute name} \\
V & \rightarrow \textit{value} \\
O & \rightarrow \ == \mid != \mid > \mid >= \mid < \mid <= \\
T & \rightarrow \emptyset \mid T\ T \mid V \\
U & \rightarrow (A\ O\ V) \mid (A\ \textbf{in}\ \{T\}) \\
S & \rightarrow \emptyset \mid U \mid (S\ \textbf{and}\ U) \mid (S\ \textbf{or}\ U)
\end{array}
$$

Figure 2.16: Predicate Specification Language Grammar

addresses or unique identifiers to limit the potential resources. In pervasive computing networks, these addresses and ids are not always well known, and often we rely on attribute-based descriptions of remote resources. Some discovery and routing protocols support this type of selection, but ultimately it is the responsibility of the middleware to ensure the resources returned to the client meet the requirements of the predicate. Thus, the simple language defined here is designed for use by the middleware, but may be translatable to some discovery protocols. Furthermore, this simplistic language is intended to support the first generation middleware implementations. We expect the development of more elaborate structures to support more elaborate formulas to be proposed as the technique is field tested.

The grammar of the current language is shown in Figure 2.16. Here we define two comparison methods (within U), one for each of the allowed value types in our simple tuple-based model. To limit a resource based on a numerical value, the programmer needs only to specify an attribute name (A), a comparison operator (O), and the target value (V). For example, the specification `"altitude > 2000"` will restrict resources to those that have a

numerical value for `altitude` that is greater then `2000`.

To limit a resource by a character string value, the **in** operator is used with a set of possible values. A common use of this comparison operator is to limit resources to a particular type. For our concrete curing example above, the programmer would use a predicate of `"type in {"humidity"}"` to limit the search to resources whose `type` attribute has the string value `"humidity"`. We anticipate the need for other operators and other value types in the future (for example the **not-in** operator), but the current language design is limited to just this functionality.

Numerical and string value restrictions can be combined using the logical **and** and **or** operators, which are subject to standard rules for order of operations, allowing a full range of predicates to be formed. In this design, nested expressions are not yet supported (`"(A or B) and C"` would have to be expanded to `"A and C or B and C"`), but this feature will be incorporated into future versions of the design.

2.4.3 Resource Preference Specification

The current resource preference specification language is designed to be very similar to the predicate language above. It too is specified with limited capabilities while the design is being developed and will be extended as experience guides us. As with the predicate language, the preference specification may be passed along to underlying technologies if they support discovery based on application-level concerns.

$$
\begin{aligned}
A &\rightarrow \textit{attribute name} \\
O &\rightarrow + \mid - \\
U &\rightarrow A\ O \\
S &\rightarrow \emptyset \mid U \mid S\ U
\end{aligned}
$$

Figure 2.17: Preference Specification Language Grammar

The preference specification language shown in Figure 2.17 is made up of a series of attributes, each of which should be evaluated for any pair of resources, until a preference for one resource is established. The preference attributes are evaluated in the order they are specified, giving precedence to the attributes appearing earlier in the expression. Since some attributes are "better" when they are larger (i.e., bandwidth) and some attributes are "better" when they are smaller (i.e., latency), each attribute is annotated with the sorting parity to be used.

As mentioned above, the user specifies his preference for larger or smaller values by using a plus (+) or minus (-) sign after the attribute name. For example, the specification $(latency-, bandwidth+)$ would indicate a preference for resources that are quicker to respond (smaller latency), and a secondary preference for larger bandwidth in the case of equal latency.

The application session determines preference by comparing pairs of resources. The evaluation of the preference function retrieves the attribute values for each of the resources. For numerical values, a subtraction is sufficient to detect differences. The specification of a domain-specific ordering for non-numeric values is a future research task. Currently, the default string

comparison operator provided in the programming language is used.

2.4.4 Resource Selection Algorithm

With the definitions of the resource predicate and preference languages above, the client can now describe his definition of "best" to the middleware. In this section we describe how a middleware should implement the $findBest(...)$ function defined in Section 2.3.5. In the next section we will discuss how the various sessions are built and maintained.

To describe our algorithm and implementation of $findBest(...)$, we begin with a very basic case and then build on it to incorporate more complex configurations. In this basic case, let us assume all resources we wish to communicate with are local to the middleware. That is, all resources are components in our memory space and we can retrieve references to them by calling `Discovery.find(...)`. Given this, the "best" resources are selected by applying the predicate specification to filter the resources, applying the preference specification to order the selected resources, and then pruning the remaining resources to the right number. This process is outlined in pseudo-code in Figure 2.18. In this figure, the parameters `predicate` and `preference` are executable versions of the predicate and preference specifications provided in the languages described above.

If we wish to use resources provided over a network, we must consider the performance implications of this algorithm. Fortunately, many existing discovery protocols support some of the selection features required by the

```
List findBest(ResourcePredicate predicate,
              ResourcePreference preference,
              Integer maxSize,
              DiscoveryListener listener) {

    List temp = new List();

    List connectedResources = discovery.find(...);

    for (Resource resource :  connectedResources)
      if (pred.evaluate(resource))
        temp.add(resource);

    sort(temp, pref);

    return temp.subList(0, maxSize);
}
```

Figure 2.18: Pseduo-code for the basic algorithm to implement $findBest(...)$

model. For any discovery mechanism used by the middleware, there are three basic cases to be considered:

- **Full Support:** When `Discovery` supports the use of predicate, preference, and limits, the `findBest(...)` implementation simply delegates entire requests directly to `Discovery`. Returned resources are wrapped by proxy components to provide the `Resource` interface, but are otherwise untouched. To support these implementations, the `ResourcePredicate` and `ResourcePreference` artifacts that are passed to `Discovery` provide the raw (text) expressions provided by the client.

- **Partial Support:** If `Discovery` supports only *some* of the selection behavior (e.g., predicates but not preferences), the remaining func-

tions must be implemented locally. By wrapping the resources returned by `Discovery` after applying the selections that are supported, the `ResourcePredicate` and/or `ResourcePreference` can be applied to complete the process.

- **No Support:** When `Discovery` does not support any selection of resources, all remote resources must be wrapped with a proxy providing the `Resource` interface and passed through the same basic algorithm shown in Figure 2.18. A naïve proxy component which forwards individual requests for resource values could easily lead to large amounts of network traffic as sessions are created. However, `Discovery` implementors must already be familiar with the discovery, routing, and data-exchange properties of the network and can play to its specific strengths to minimize costs.

Using the techniques outlined above, we can now select the best resources given any one implementation of `Discovery`. However, we expect to find many situations when multiple network technologies are present in the same environment (e.g., RMI and CORBA). In these situations, multiple implementations of `Discovery` can be provided to the middleware representing and interfacing with the available technologies. To merge the results of m `Discovery` components, a single façade component is created to call each instance in turn, generating m lists of resources. The Façade then selects the top *maxSize* elements from these lists using the `ResourcePreference` to compare

```
Session
    init(...)
    Resource get(int i)
    int size()
```

Figure 2.19: Pseduo-code for the Session Interface

`Resources`.

Regardless of the `Discovery` implementation and its providence, the middleware may also supply a `DiscoveryListener` to monitor changes in the network. Using this mechanism, the middleware can be notified when there are changes in the network that potentially invalidate the "best" property of the returned list. We will see how this feature can be used to implement the strategy invariants.

2.4.5 Session Creation and Maintenance

In the previous section we have shown the middleware analog to the model's *findBest(...)* construct. In this section we present the analog to the model's session S[...], the `Session` interface, shown in Figure 2.19. The `Session`'s `get(i)` method returns an object supporting the `Resource` interface described in Section 2.4.1. The session's `size()` method returns the size limit defined when the session was created. If there are fewer resources then the session's limit, the session is padded with null values. For example, if the session was created with a limit of 10, but only 1 resource could be found to match the predicate, `get(3)` will return `null`.

Like the operational models of Section 2.3.6, each of the middleware's

strategy components calls on `findBest(...)` to provide the initial resources. Continuing with the operational model, the middleware then updates these assignments to ensure the strategy's invariant. To implement the `await` construct, the component maintaining the assignments must be able to intercept both the application's use of resources and any connectivity or value changes in the networks. By implementing both the `Session` and `DiscoveryListener` interfaces, a single component can receive all the events that trigger the `await` conditions of the operational models. In our design we specify three of these components: `QuerySession`, `ProviderSession`, and `TypeSession`. As their names imply, each component is imbued with the logic to support a single connection maintenance strategy *and* serve as the actual implementation of the session component that is returned to an application.

The remainder of this section is devoted to summarizing how each of these components implements the respective operational model. For brevity, synchronization and mutual exclusion concerns are omitted from this discussion. In Section 2.5 we will present some details of an actual implementation of this design. The source code for the evaluation implementation [75] does provide the necessary guarantees and can be consulted for more details.

2.4.5.1 Query Session

In this section we describe the `QuerySession` component, an implementation of the Query Session Operational Model (Figure 2.6) that guarantees the Query Session Invariant. As its name implies the `QuerySession` compo-

nent not only ensures the invariant but models the session $S[...]$ as well. A pseudo-code version of the `QuerySession` is shown in Figure 2.20. When a new instance is created, the results of the `findBest(...)` method are stored in an internal list, `resources`. To protect this collection, the client's access to these resources is mediated by the `Session.get(i)` method. Here, the content of the `resources` list at index i is returned to the client and replaced with the `null` value. This behavior effectively models the first `await` statement in the operational model. The second `await` statement is triggered by calls to the `onDisconnect()` method inherited from the `DiscoveryListener` interface. When this event is fired, the appropriate element of the resources array is also set to `null`.

2.4.5.2 Provider Session

Like the `QuerySession` component, the `ProviderSession` component's implementation is drawn directly from the operational model for the session strategy of the same name. An outline of its implementation is shown in pseudo-code in Figure 2.21. As with the `QuerySession`, we initialize an internal list of resources with the results returned by `findBest(...)`. At the same time, we also create a list of boolean variables `connected` to represent the "connected-ness" of the resource with the same index in `resources`. The `size` of the session is always the same and set from the initial list of resources. Since there may be fewer resources available, size will range between zero and the limit specified by the client.

QuerySession `implements Session, DiscoveryListener`

```
List resources;

    init(...)  {
      resources.addAll(findBest(...));
      size = resources.size();
    }

    get(int i) {
      tmp = resources.get(i);
      resources.set(i,null);
      return tmp;
    }

    onDisconnect(resource) {
      i = resources.indexOf(resource);
      resources.set(i,null);
    }
```

Figure 2.20: Pseduo-code for the Query Session

When clients access the session's elements via `Session.get(i)`, the status in `connected` is checked. If the resource is connected, it is returned from the `resources` list. Otherwise, `null` is returned. The values in the `connected` list are maintained by intercepting the events that are indicated by the `await` statements in the operational model in Figure 2.9 by implementing the `DiscoveryListener` interface. When resources are connected to, or disconnected from the network, the index of the resource is found by searching the `resources` list. The matching value in `connected` is then updated to reflect the new status.

2.4.5.3 Type Session

The `TypeSession` component represents a session implementing the Type Strategy for connection maintenance. As with the other `Session` components, we begin our pseudo-code implementation of the `TypeSession` in Figure 2.22 with a call to `findBest(...)`. However, in this invocation, we ignore the session limit specified by the client and greedily select as many resources as there are available. We then use the implementation of `get(i)` to restrict the client from accessing elements of the `resources` list with indexes larger then the size limit.

Using this technique, the internal list `resources` can contain more resources then are allowed by the session size limit. Thus, when a resource is removed, the remaining elements are still in order, and the `get(i)` method still returns the correct resources for every index. The implementation of the

ProviderSession `implements Session, DiscoveryListener`

```
List resources;
List connected; // list of booleans

init(...)  {
  resources.addAll(findBest(...));
  connected = new List(resources.size());
  connected.setAll(true);
  size = resources.size();
}

get(int i) {
  if (connected.get(i)) {
    return resources.get(i);
  } else {
    return null;
  }
}

onConnect(resource) {
  i = resources.indexOf(resource);
  connected.set(i,true);
}

onDisconnect(resource) {
  i = resources.indexOf(resource);
  connected.set(i,false);
}
```

Figure 2.21: Pseduo-code for the Provider Session

third await statement in the operational model, the method `onDisconnect`, relies on exactly this property.

The second `await` statement of the operational model (Figure 2.12) is implemented in the body of `onConnect`. Here the resource is checked against the session's predicate and added to the internal list which is then sorted. The final `await` statement is implemented in the `onValueChange` method and blends the techniques used by the other methods. If a value change causes a remote resource to be included in the session, then it is added to the internal list `resources`. If it is already there, this addition is skipped, but either way the list is re-sorted to maintain the invariant. If the resource fails the predicate, it is removed from the internal list and the invariant is naturally preserved.

2.5 Model and Middleware Evaluation

The Application Sessions Model proposes a new way of interacting with the remote resources available in pervasive computing environments. In this section we qualitatively evaluate the feasibility of the model and the middleware design by building a prototype middleware implementation of the model and using it to implement the sample applications described in Section 2.1.

However, before introducing the middleware implementation, we first describe how applications would be implemented if the data were available locally and the Application Sessions Model was not needed. We refine this design in the following sections to describe how the middleware's components are incorporated. We begin in Section 2.5.2 by showing how sensors are exposed

TypeSession implements Session,Listener

```
List resources;

init(...)  {
  resources.addAll(findBest(...));
}

get(int i) {
  if (i < min(resources.size(), maxNum)) {
    return resources.get(i);
  } else {
    return null;
  }
}

onValueChange(resource) {
  if (predicate.evaluate(resource)) {
    if (! resources.contains(resource)) {
      resources.add(resource);
    }
    sort(resources);
  } else {
    resources.remove(resource);
  }
}

onConnect(resource) {
  if(predicate.evaluate(resource)) {
    resources.add(resource);
    sort(resources);
  }
}

onDisconnect(resource) {
  resources.remove(resource);
}
```

Figure 2.22: Pseduo-code for the Type Session

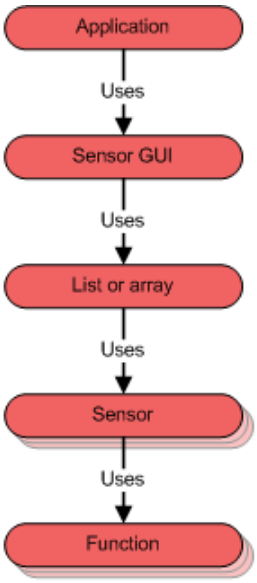

Figure 2.23: Component overview for application with local sensors

as `Resources`. This is followed in Section 2.5.3 with a very brief description of how predicate and preference functions are parsed and evaluated. Using predicates and preferences to locate resources is discussed in Section 2.5.4 which also discusses how the connection maintenance strategies are implemented. Since this document is a poor medium for a detailed tour of the source code, we offer only an annotated overview in the following sections using figures and pseudo-code. Complete documentation of the source is available the code [75] and should be referred to for a detailed inspection.

2.5.1 A Basic Application

An overview of the target applications is given in Figure 2.23. The raw data to be presented in this configuration is provided by `Function` components.

These components generate values based on the system's clock (aka wall-clock time). Several `Functions` are gathered together and assigned attribute names to create `Sensor` components. `Sensor`s serve as synthetic representations of physical sensors deployed in a hypothetical pervasive computing environment. The canonical example of a `Sensor` in this section is a location-aware temperature sensor. The `Sensor` component to model this example would incorporate four `Function` components, one for each of four attributes: *latitude*, *longitude*, *altitude*, and *temperature*.

When `Sensor` components are directly accessible and managed directly by the application without the Application Sessions Model, they are simply collected into a `List` or `Array` component held by a `Sensor GUI` component. This component is responsible for transforming the data for human consumption. To maintain a consistent view of the sensors, this GUI component must periodically query or register as listeners of the `Sensors` to collect new data and update the display. In this architecture, the `Application` component merely serves as a container for the GUI component and will be reused in the sections below.

The `Application`, `Sensor`, and `Function` components in this section are used primarily as surrogates for their counterparts in an actual deployment. Implementations of these components provide the most basic functionality and should not be considered as part of the model's evaluation.

2.5.2 Sensors as Resources

The first adaptation made to the application design above is the removal of the glaring assumption of local resources. To simulate network access of remote resources, the `Sensor` component is wrapped by a `ResourceSensor` component. The `ResourceSensor` encodes all data provided by the target `Sensor` as character strings which are in turn encoded as binary data. This data is returned through the `Resource` interface discussed in Section 2.4.1. The deconstruction of the `Resource` interface into a byte-oriented network protocol is a simple matter that is not germane to the evaluation and omitted for brevity.

As `ResourceSensor` components are created, they are added to a `ResourcePool` component. The `ResourcePool` approximates the network on which remote resources would normally be found. In this capacity, the `ResourcePool` supports the `Discovery` interface to expose its `Resource`s to clients. By using the `Discovery` and `Resource` interfaces, the client imitates the separation of components that exists in a real deployment. The simulation is simplistic but effective and sufficient for the purposes of evaluation discussed above.

With these substitutions, our example application takes the form shown in Figure 2.24 and loosely models traditional remote resource interaction systems. From this point we are able to apply the Applcation Sessions Model to provide high-level metaphors to the client.

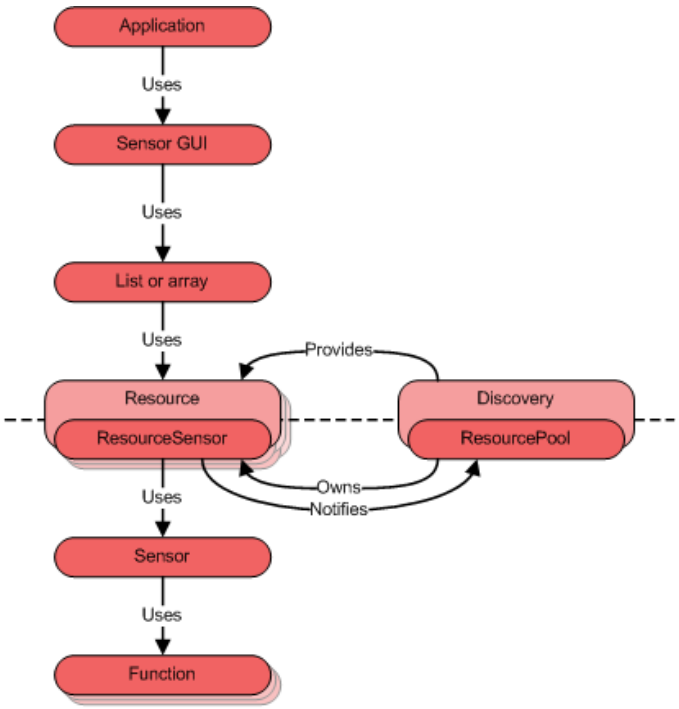

Figure 2.24: Direct access to `Sensors` replaced by `Resources` and `Discovery`

2.5.3 Evaluating Predicates and Preferences

The ability to support complex predicates is a fundamental component of the overall structure of the Application Sessions Model. For example, a predicate requiring a temperature reading from a sensor east of some latitude can be specified as `"exists(temperature) && latitude > 8"`. To provide robust support, we incorporated an external library [29] for parsing and evaluating these mathematical expressions. Once parsed, variables named in the preference expression can be gathered from the predicate under evaluation (e.g., "temperature", "latitude"). The parsed expression is then evaluated with these values, and the boolean result used to determine the resource's inclusion in the session.

The majority of the middleware components involved in evaluating the preference function for each resource is dedicated to choreographing interactions with the external library and handling any exceptional cases. When an exceptional case is encountered (i.e., an expression cannot be evaluated for a particular resource), the resource is considered to have failed the predicate and will not be included in the session. This case can occur when a resource's attribute is missing or uses an incompatible type (e.g., using a string instead of a number).

When evaluating the client's preference function, the current implementation only supports numerical attributes in the preference function. While somewhat limiting, this restriction also simplifies the evaluation of the preference function. The preference specification language in Section 2.4.3 is easily

parsed, and the appropriate attributes are gathered from the candidate resources. The values supplied by resources are parsed as numbers and then compared to produce the proper ordering. When a resource's attribute cannot be parsed, it is regarded as "equally preferred" to resources with the same unparseable attribute and "less preferred" than resources that are successfully parsed.

2.5.4 Selecting Resources

The selection of resources in this prototype implementation focuses on the most basic case, described as "No Support" in Section 2.4.4. In this case, the `Discovery.find(...)` implementation is not able to perform any of the selection functions and simply returns a complete enumeration of available resources. Clearly this is sub-optimal for network performance, but it serves as a baseline solution that can be improved upon.

The middleware's selection component draws directly from the pseudo-code description in Figure 2.18. The resulting `findBest(...)` method is then used by each of the session implementations to initialize the session elements. The complete implementations [75] include additional code to implement necessary exclusion and ordering properties.

To make the middleware library a bit more intuitive, a `SessionFactory` component is also introduced. The `SessionFactory` is charged with allocating system and network resources. It also provides the `newSession(...)` method to the client, accepting the four application session parameters and returning

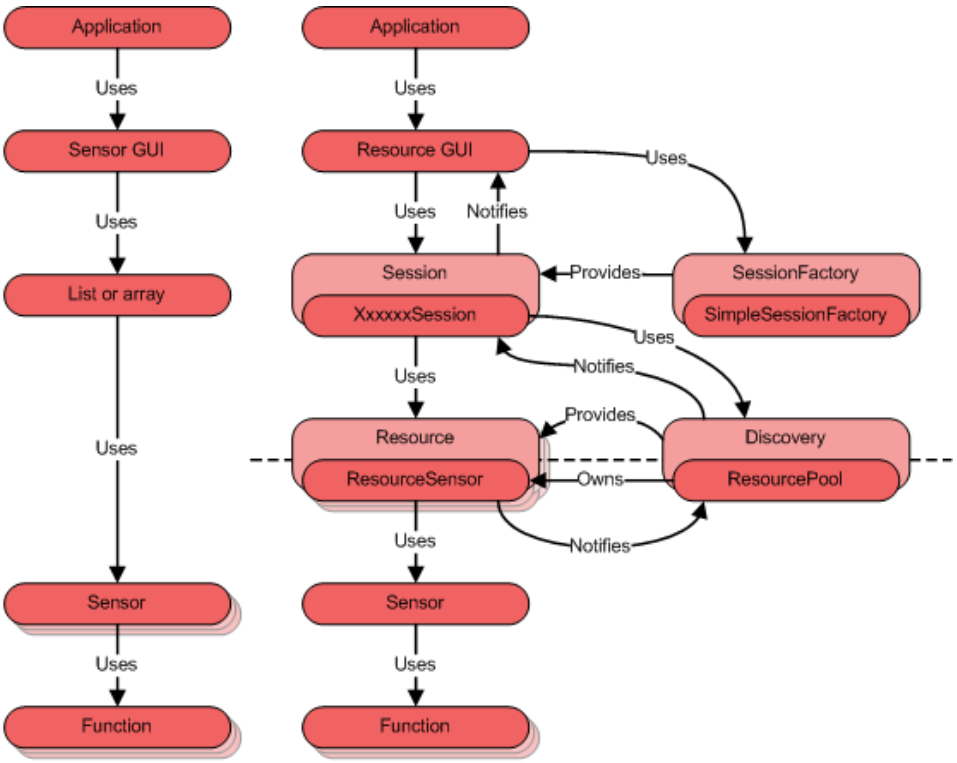

Figure 2.25: Overview of an application using Application Sessions

a completely configured `Session` component. Once the underlying components are configured, the `SessionFactory` and `Session` interfaces constitute the entire client-facing API. Figure 2.25 shows how the original application architecture has been adapted to the middleware implementation along with the original components for comparison.

In the current prototype implementation, the `newSession(...)` method requires a `Map` parameter to explicitly declare the data types of each of the parameters referenced in the predicate. This parameter is an artifact of early implementations and has not yet been removed. A full implementation should

not require this information in order to correctly evaluate resource predicates.

2.5.5 Intelligent Construction Site Applications

Using the Application Sessions Middleware described above, the Intelligent Construction Site applications introduced in Section 2.1 were prototyped to test the feasibility of the model and middleware. As the reader might expect, the applications do indeed benefit from the abstractions provided by the Application Sessions Model, and the middleware design proved to be well suited for the model. In this section we will show how the core functionality of each hypothetical application was abstracted for the evaluation, and show the resulting artifacts.

For each application, we create a suite of sensors to represent a hypothetical Intelligent Construction Site. We define five sensor types to provide the required data for the applications. We create 25 instances of each sensor type and assign them to locations on the five-by-five grid shown in Figure 2.26. The locations on this grid are labeled (A, B, C...) for discussion purposes only. In addition to a sensor specific value (e.g. `humidity` for the `Humidity` sensors), each sensor also provides its location as `latitude`, `longitude`, and `altitude`. For the sake of simplicity, we use some unnamed unit of latitude and longitude to measure these values using short numerical values. Table 2.2 summarizes the attribute names and types provided by each sensor.

The `VOC Detector` sensor type models a sensor designed to detect dangerous concentrations of volatile organic compounds (VOCs). For our current

Type	Values	Type
Temperature	`celcius`	real
Humidity	`humidity`	real
VOC Detector	`safe`	boolean
Crane Load	`loaded`	boolean
Crane Movement	`moving`	boolean
all of the above	`latitude`	real
	`longitude`	real
	`altitude`	real

Table 2.2: Synthetic Sensor Types and Attributes

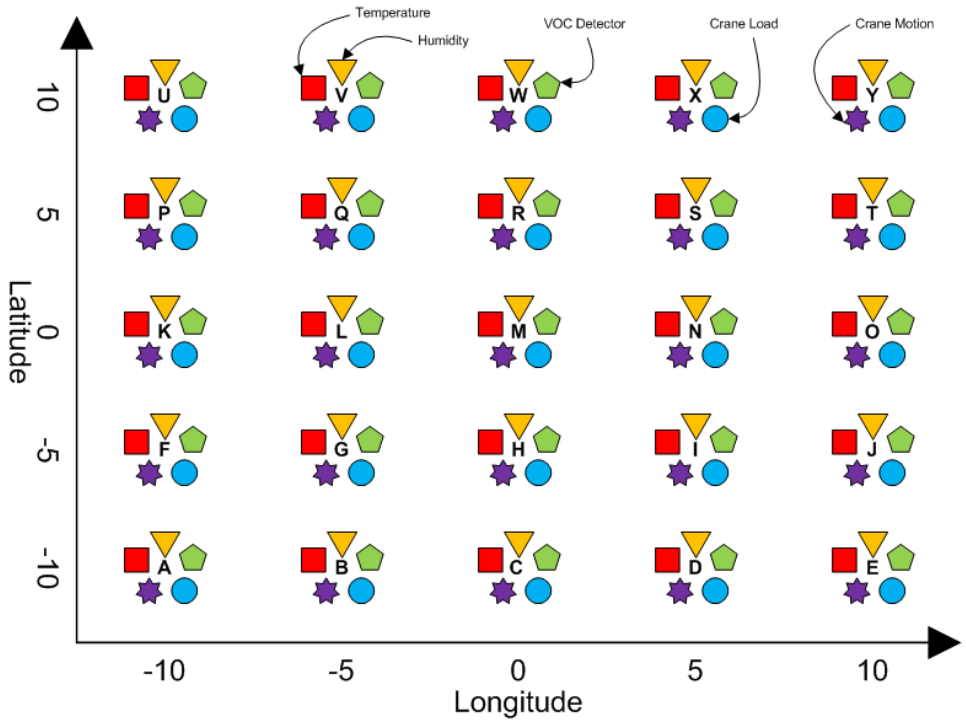

Figure 2.26: Sensor locations on hypothetical construction site

example, these sensors only report a boolean value for the attribute "`safe`" which is consistent with commercially available sensors. The `Crane Load` sensor returns true when the associated hypothetical crane is carrying a load. Likewise, the `Crane Movement` sensor returns true when the crane is moving a load. Since a crane cannot move a load it is not carrying, these sensors are linked in the simulation. Specifically, the `Crane Movement` sensor can only return true when the `Crane Load` sensor is also returning true.

The raw data provided by the sensors is generated by `Function` components as discussed in Section 2.5.2. To model stationary sensors, the three attributes representing location are always provided by constant functions. The other attributes (`celcius`, `humidity`, `safe`, `loaded`, and `moving`) cycle between values on a periodic basis. To ease the visualization of the global sensor view, shown in Figure 2.27, we use sinusoidal function based on wall-clock time for the real-valued attributes. To a large extent, attributes' functions are chosen for convenience and are not meant to represent actual events on a construction site.

Experiment 1 - Query Session / Curing Concrete

The first application we will examine is "Monitoring the Curing of Concrete". The core requirement of this application is to periodically request Temperature and Humidity values from a specific region in space. Using the Application Sessions Model to perform each request, we repeatedly establish Query Sessions using the code in Figure 2.28, and pass the results to a GUI to

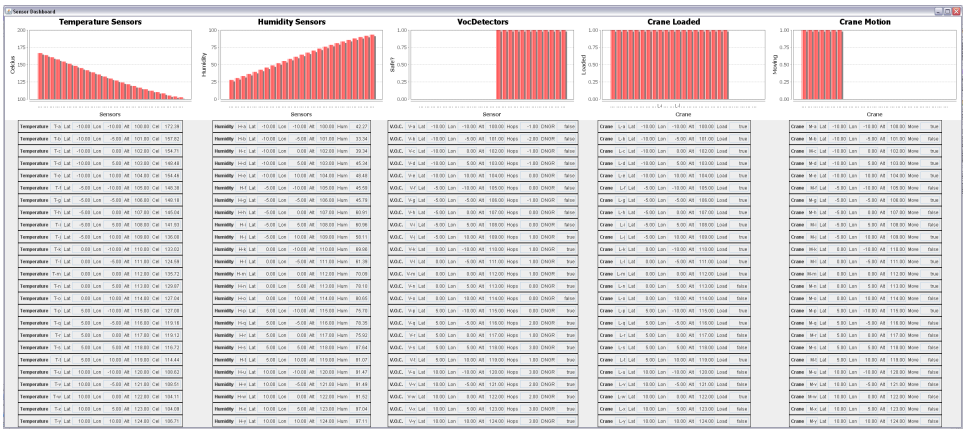

Figure 2.27: View of all sensors on hypothetical construction site

be displayed. Here the predicates specify `"longitude > 8"` which limits the session to sensors east of 8 degrees longitude. Both calls to `newSession(...)` limit the session to 10 resources and pass a `Map` of attribute names to their expected data types as the parameter "`types`". This map is an implementation artifact discussed in Section 2.5.4.

The `humiditySession` in this example uses the preference function `"humidity+"` to sort the humidity values from smallest to largest. This assures us that the lowest humidity readings, those that indicate a problem, will appear first in the session. In the snapshot of the application shown in Figure 2.29, the humidity session is plotted on the right half of the window. As expected, the values are arranged from smallest to largest as a result of the preference function. Likewise, the `"temperatureSession"`'s function indicates a preference for larger values. Thus the temperature readings shown on the left half of the application window are sorted from largest to smallest.

DoQueryButtonListener

```
Session humiditySession =
  sessionFactory.newSession(Strategy.Query,
    "longitude > 8",
    "humidity+",
    10, types);

Session temperatureSession =
  sessionFactory.newSession(Strategy.Query,
    "longitude > 8",
    "celcius-",
    10, types);

gui.setHumidities(humiditySession);
gui.setTemperatures(temperatureSession);
```

Figure 2.28: Establishing the Query Sessions for the Curing Concrete Monitor application

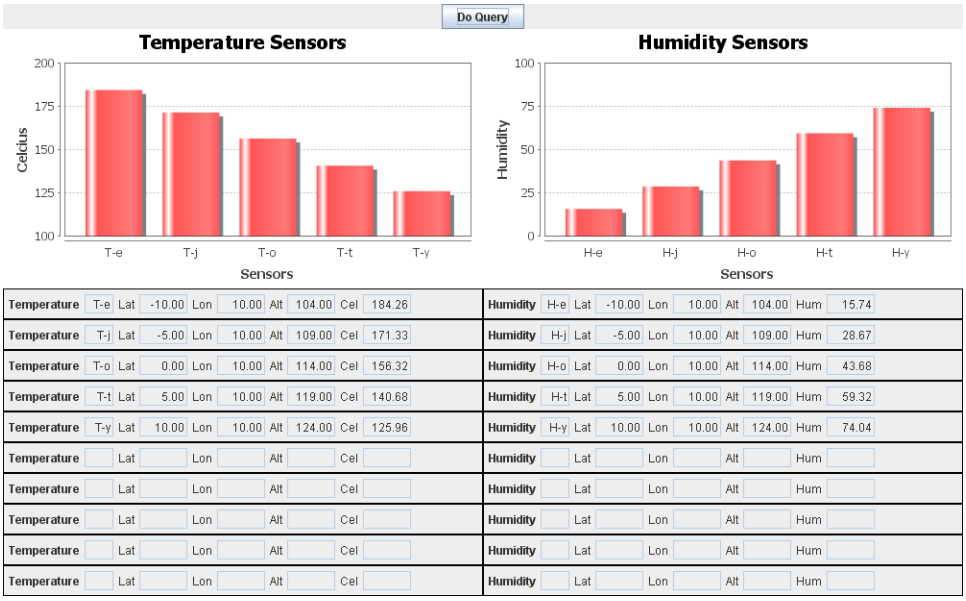

Figure 2.29: Snapshot of Curing Concrete Monitor example application

We should note that in our hypothetical construction site, this predicate selects 25 sensors, five of each type. In this case, it is our GUI component that selects only the resources with the correct attribute. There are a number of predicate variations that can make this selection before the resources are passed to the GUI. Furthermore, rather then plotting values in a GUI, a commercial application for monitoring curing concrete might alert personnel when certain thresholds are violated. In this case, an alternate predicate could include the thresholds to limit the session to *only* the sensors reporting troublesome conditions. The application would take action when the session contains *any* non-`null` entries.

Experiment 2 - Provider Session / Crane Monitor

In our second study, we model the "Crane Monitor" application. Like the curing concrete application above, this application retrieves values from sensors on a periodic basis. However, the interval between samples is much shorter, and the application's time frame much longer. For this study we implement the application using the Provider Session in Figure 2.30. By using the `exists(...)` function, these predicates select resources explicitly by an attribute they provide. The preference function sorts the sensors from north to south and then east to west for convenience.

In addition to the variations in sensor readings, the sensors also experience variations in connectivity. In our hypothetical scenario, each sensor is

DoQueryButtonListener

```
Session craneLoadSession =
  sessionFactory.newSession(Strategy.Provider,
    "exists(loaded)",
    "latitude+, longitude+",
    10, types);

Session craneMotionSession =
  sessionFactory.newSession(Strategy.Provider,
    "exists(moving)",
    "latitude+, longitude+",
    10, types);

gui.setCraneLoads(craneLoadSession);
gui.setCraneMotions(craneMotionSession);
```

Figure 2.30: Establishing the Provider Sessions for the Crane Monitor application

randomly disconnected from the network 20% of the time. This leads to sporadically missing values in our application's interface. In the snapshot shown in Figure 2.31, the first five resources of the `craneLoadSession` are shown in the two left-most columns while the first five resources of the `craneMotionSession` are displayed in the right-most columns. The missing values for the fifth crane-load resource is evidence that the sensor was disconnected when the snapshot was taken.

Experiment 3 - Type Session / Danger

The third application, "Danger Monitor", uses the Application Sessions Model to vigilantly monitor for dangerous conditions that are declared when the application starts. For this purpose we create two `TypeSessions`, one

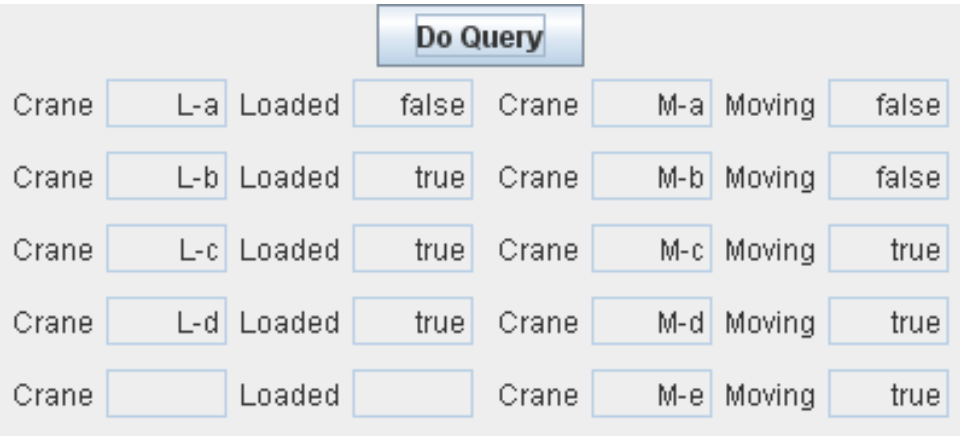

Figure 2.31: Snapshot of Crane Monitor example application

to monitor VOC Dectors and another to monitor cranes. We establish both sessions with a predicate to select sensors in the general vicinity of the user. For example, if the user is located at 3 units of latitude and -3 units of longitude in Figure 2.26, we establish the sessions using the code in Figure 2.32. Here, the first four terms of the predicates select sensors within 5 units of latitude and longitude from the user. This will select the sensors at the locations marked M, R, Q, and L in Figure 2.26. The last term in each predicate selects only sensors reporting conditions of which the user needs to be aware. In this example, the preference function is left blank for simplicity.

Once the sessions are created, they are passed to a GUI to be displayed. At the time the snapshot shown in Figure 2.33 was taken, the user was in particular danger: all four cranes and all four VOC detectors were indicating dangerous conditions. In a more realistic deployment, the sessions would be empty most of the time, making for a less interesting example. Again, a

InitDangerApp

```
Session vocSession =
  sessionFactory.newSession(Strategy.Type,
    "latitude > -2 && latitude < 8 &&
     longitude > -8 && longitude < 2 && danger",
    "", 10, types);

Session movingSession =
  sessionFactory.newSession(Strategy.Type,
    latitude > -2 && latitude < 8 &&
     longitude > -8 && longitude < 2 && moving",
    "", 10, types);

Container appPanel = new DangerGui(vocSession, movingSession);
```

Figure 2.32: Establishing the Type Sessions for the Danger Monitor application

commercial application would likely not chose bar-graphs to report the results of sessions, instead choosing a more intuitive mechanism. For example, an unobtrusive visual alert for moving cranes, but an unavoidable buzzer and audio alert for VOC warnings.

2.6 Summary

In the examples describing the Application Sessions Model and the middleware prototype, we have focused on attributes that describe the remote resources themselves. However, the pervasive computing environment provides us with a wealth of other information about itself that could be used for selection. For example, the network latency between the host and resource is not necessarily descriptive of the resource itself; rather it is a trait of the link(s)

Figure 2.33: Snapshot of Danger Monitor example application

between them. In ad-hoc networks, latency if often dominated by the number of links in each multi-hop route. The desire to access this information can be directly or indirectly satisfied by some routing and discovery protocols. Alas, complex and application specific characterizations of the environment are not often available. For this functionality we have developed a solution in the Evolving Tuples Model described in the next Chapter. Using this technology, resources can be selected based on latency and hop-count as well as metrics such as "average latency per hop" or even "reachable without traversing a cold node." Exposing these rich metrics to the application developer's predicate and preference functions considerably promotes the value of the resulting Application Session and in turn, the application itself.

Chapter 3

Evolving Tuples Model

In the previous chapter we presented a model to help application developers reason about their pervasive computing environments. The Application Sessions Model provides a smooth and intuitive mechanism to manage the long-term conversations with the resources in these environments. One of the key features in this technique is the declarative definition of acceptable resources via the *resource predicate*. Not only does the predicate allow the model to select the initial set of resources for a session, but it enables the potential addition of newly available resources (in the case of the type session strategy). The utility of a session is thus highly dependent on the expressiveness of the predicate that supports it.

Using a centralized technique for predicate evaluation would likely result in a simple and straightforward solution. For example, a client simply gathers all data values from the environment and applies the predicate locally. Unfortunately, centralized approaches are poor choices for pervasive computing networks. The dynamic nature of both the connectivity and the values makes the process of gathering a consistent snapshot of the network difficult. Additionally, pervasive computing environments are particularly likely to con-

taining power-restricted devices. Needlessly transmitting data from these devices would exhaust batteries prematurely. For these reasons a centralized approach is not a viable solution for the general case. Instead, a distributed solution for predicate evaluation is required for the majority of environments to apply the Application Sessions Model.

There is a wide range of existing research focused on the general field of distributed resource discovery. Attempts have been made to adapt traditional (centralized) techniques to pervasive computing environments [10, 25, 53, 54, 78]. These approaches either push resource information preemptively or wait for queries to be flooded through the network. Some solutions use overlay networks and specialized messages to bolster performance while others piggyback information on existing routing or data messages to reduce power usage. These approaches typically require remote resources to be identifiable through the use of unique identifiers, network addresses, or technology specific service interfaces.

The exchange of identifying information amongst loosely-related organizations and their devices is too high an expectation for field deployments. For example, participants will not be likely to negotiate and agree on service interfaces and network addresses before arriving at the Intelligent Construction Site. We feel that approaches which only expect semi-structured data are more likely to succeed in environments where the relationships between the participants are semi-structured themselves.

In this chapter, we present the *Evolving Tuples Model* designed for resource discovery in pervasive computing networks. This model is specifically designed to support the resource predicate structure of the Application Sessions Model by evaluating potential resources using a language similar to that proposed in Section 2.3.2. As the Evolving Tuples Model has developed, it has proven to also be applicable outside this scope. Specifically, the model is useful for prototyping certain classes of distributed algorithms such as route discovery.

Before detailing the model in Section 3.3, we first present some background information on previous tuple-based models in Section 3.2. We then show how the the Evolving Tuples Model is used to perform the resource discovery that originally drove the development of the model in Section 3.4. The additional prototyping functions are shown through example in Section 3.5. Section 3.6 discusses an implementation of the model and experimental results collected from feasbility studies performed in both hardware and software simulation. While we believe the current model to be a significant contribution to the field, we also detail several potential model extensions and implementation concerns in Sections 3.7. Related work from the field is discussed in Section 3.1.

3.1 Related Work

The early tuple space designs [30] and implementations [13] for Linda targeted parallel processing environments. Specifically, the atomic insertion and removal operations on tuple spaces relied on locks provided by shared memory. The LIME [62] system introduced distributed tuple spaces that provided the same atomicity guarantees across a truly global tuple space spanning many devices in a mobile ad-hoc network. This adaptation of tuple spaces allows for a very abstract representation of the network underlying a pervasive application, but requires that tuples be delivered to consumer processes without interacting with the "lower levels" of the network. We believe that exposing information from these "lower levels" as cross-layer information in our approach allows for more powerful applications at the cost of a slightly more complex representation.

In a similar approach, mobile agent systems also combine behavior with the data that traverses the network. Often these implementations choose to extend the tuple model, as its minimalist nature makes it particularly well suited for encoding for transmission.

MARS [14], for example, associates a tuple space to each host in a group of physically connected nodes. Mobile agents can roam from host to host, interacting with other agents through the locally available tuple spaces. MARS employs reactivity in its tuple spaces to allow context information to impact some of these interactions. However, the context-awareness is embedded in these reactions, which are coded as separate entities from the data they impact.

In our approach the tuples themselves carry the behavior that creates context-aware adaptation. Like MARS, LIME relies on reactions external to the tuple space to create context-aware adaptation, while we focus on embedding this adaptation directly in the coordination model.

The Agilla [28] system also provides a single tuple space per node to mobile agents which roam from node to node. In addition to local access though, Agilla allows agents to insert and remove tuples from tuple spaces located on other nodes. In this way, Agilla can use tuples as network messages between processes, but any behavior must be transmitted to remote nodes as a separate mobile agent. However, Agilla does target sensor networks and thus most all pervasive computing deployments can support the functionality required to support its agents. While not cited directly, these minimal requirements of node capability make the system attractive to long-lived and mixed technology deployments.

TOTA [57, 58] has a more integrated approach to incorporating context-awareness into tuple spaces. In this system, tuples are automatically moved in a dynamic network according to contextual properties. TOTA subroutines can adapt to external properties in the environment and to the content of the tuples to make decisions regarding, for example, routing. While these subroutines can be carried within the tuple, the tuples effectively become empowered mobile agents. The evolving tuples model, on the other hand, maintains a tuple structure imposed over the data *and* the behavior in combination, maintaining the easy-to-use benefits of traditional tuple space approaches.

In general, these aforementioned tuple based systems provide complex behavior but often place undue burden on either the developer or the hosts. Systems like Agilla require the developer to understand very low-level programming languages, while systems like TOTA and MARS require host to support high-level languages (i.e., Java). We feel that the evolving tuples model strikes a balance between the skills required to use the system, and the capabilities that are required of the network hosts.

Another similar technology is active networking [77], in particular capsule-based systems [80]. The biggest difference with the evolving tuples model is the target environment. Active networking targets relatively powerful routers in wired networks to enhance performance and offer new services while evolving tuples are intended for relatively resource poor nodes in (typically) wireless networks. Additionally, the active networking community has struggled with the security trade-offs associated with allowing mobile code to modify network elements. Since the evolving tuples model only allows formulas to modify the tuple itself, the security issues are reduced significantly.

While rooted in different technologies, there are a number of other designs to reduce the efforts required to develop pervasive computing applications. For example, Weis et al. [79] use visual programming techniques to reduce the learning curve for hobbyist developers. Other approaches [8, 34] provide additional abstract programming interfaces to manage complexities that can be hidden from the developer.

Centralized querying semi-structured data as mentioned above has a

number of established characterizations and solutions [1]. The application of these techniques to distributed resource discovery has also been studied in the literature [7, 17]. Approaches such as mobile agents [11, 21] can distribute semi-structured data queries by embedding calculations into the network messages. We expand on these approaches to arbitrarily combine data elements from multiple sources to satisfy discovery requests.

As we have mentioned above, the evolving tuples model aims to reduce the cost of developing applications through reducing the need for recompilation and redeployment. Other work has tackled these problems as well. For example Dyer et al. [22] deploy nodes in pairs to allow wired access to running applications, but also allow new applications to be more easily deployed. We anticipate that these techniques to be complementary to the evolving tuples model, and might be combined to further ease software development for pervasive computing.

3.2 Background

The evolving tuples model is an extension to the large body of research on the use of tuple spaces for ubiquitous and pervasive computing. Originally introduced as part of the Linda [30] system, a *tuple* is simply a name followed by an ordered list of typed data fields. Tuples are collected in a bag-like[1] data-structure called a *tuple space*. The insertion and

[1]As with a bag (or multiset), a tuple space can contain multiple copies of a tuple, and tuples are unordered.

removal operations, *tuplespace*.`out()` and *tuplespace*.`in()` respectively[2], are atomic operations, making the tuple space a natural mechanism for buffered communication between parallel processes. For example, a process monitoring a sensor can package a temperature reading as the tuple ⟨ *"temperature", (string,"celcius"), (int, 26)* ⟩ and add it to a shared tuple space. Another process that uses temperature sensor readings can simply remove this tuple from the tuple space and extract the relevant data.

In the original tuple space design, a process removing a tuple from the tuple space provides a pattern to which candidate tuples are compared. This mechanism allows a tuple space to store both temperature and wind-speed readings (for example) using the self-describing nature of the data to provide type-safety. These patterns take the form of a name followed by an ordered sequence of *actual* or *formal* values. A tuple that matches a given pattern has the same name and the same number of fields as the pattern, and has equal values for any actuals and has values with the the same type as any formals. To continue our temperature sensor example above, the process using temperature sensor data would specify a pattern of ⟨ *"temperature", (string,"celcius"), (int, ?)* ⟩ where *"celcius"* is an actual that must be matched exactly, and *(int, ?)* is a formal specifying only the type of the second field of a matching tuple.

[2]Tuple space operations are named from the perspective of the process. To remove a tuple from a tuple space, a process is moving the data *into* its own scope and thus the operation is called `in()`. Likewise the operation to move data *from* the process to the tuple space is named `out()`.

While forming a simple mechanism for passing data from one process to another, this tuple space design requires that both the data producer and data consumer are maintained together. Any alteration to the name, format, or types of the generated tuples will cause them to no longer match the pattern used by other processes. Since a pervasive computing environment typically consists of devices that are not under the control of a single administrative entity, we must assume that tuple formats and tuple patterns are likely to change independently over the lifetime of a deployment.

To address this issue some models, such as LighTS [9] and ELights [46], decouple tuple fields from their position in the tuple by assigning unique names to each field. The types and values of a tuple's fields are accessed using these names instead of fixed positions. If a new field is added, or an unused field is removed, consumers of the tuple will continue unaffected by the changes. As we will explain in Section 3.3, our evolving tuples model also adopts this technique to gain the same decoupling between the implementations of producers and consumers.

These data producers and consumers are not only different processes, but are typically located on different nodes of a connected network. Since processes are already exchanging well formated data as tuples, simply allowing nodes to pass these tuples as network messages is a natural mechanism for interprocess coordination.

However, using traditional systems the behavior of the applications that consume the data must still be specified *a priori* so that an application gener-

ating tuples can provide the right data to the consumers of tuples. Evolving tuples reduce this level of coupling by directly embedding some of the behavior we expect from nodes into the tuple itself. Specifically, rather then requiring network nodes to adhere to a pre-specified recipe for manipulating the data as it is transmitted across the network, evolving tuples allow tuple creators to provide this behavior at runtime.

3.3 The Evolving Tuples Model

The evolving tuples model consists of three major components: the tuple format, the evolution process, and the standard deployment. The tuple format describes how data and behavior are specified while the evolution process describes how the behavior is applied. The standard deployment describes the instantiation of the elements that applications can expect from other nodes in the network.

3.3.1 Evolving Tuple Format

The format of an evolving tuple is a simple extension to the original Linda tuple design. The name assigned to each tuple is dropped in favor of names assigned to each of the tuple's fields. Each field is then extended to include a formula that governs the behavior of the field, and indirectly, the behavior of other fields and the behavior of the tuple itself. The *name* and *formula* field elements are each discussed in detail in the following sections.

3.3.1.1 The *name* element

The addition of a unique name to each field in a tuple decouples the implementations of the tuple producer and consumer from each other. Instead of depending on the exact format of a tuple, applications can depend on the data that the tuple should provide. For example, if one application requires tuples with fields A and B while another requires fields B and C, both applications can correctly identify and consume a tuple with fields A, B and C. In addition, tuples with an extra field D can be introduced without affecting either application, and the removal of field A would only affect the first application.

To ensure unambiguous use of data, we constrain the names of the fields to be unique within a tuple. The uniqueness of field names allows us to unambiguously reference a field by using only the field's name. This allows a tuple to be viewed as a look-up table of sorts; given a name, the type and value of a field can be returned by simply inspecting the associated field with a matching name element.

With this addition, we must also alter the format of the tuple templates used by tuple space operations. In this model we choose to adhere to the original Linda specifications and allow the user to match either the exact value (actual) of a field, or the type of the field's value (formal). This small set of predicate functionality suits our application requirements and avoids adding unnecessary complexity to the matching function. The format of a template is

simply the name, type, and value of each of the fields that must be matched:

$$\begin{array}{ll} \langle\ (\textit{name}, \text{type}, \text{value}), & \leftarrow\text{Actual} \\ \quad (\textit{name}, \text{type}, \emptyset), & \leftarrow\text{Formal} \\ \quad \ldots \rangle & \end{array}$$

The template's *name* and *type* elements have the same meaning as in a tuple. The third element can take either the null value ($\emptyset$) or a concrete value. The null value effectively turns the field into a *formal*, indicating that the matching function should only be concerned with the type. If a concrete value is provided, the field is an *actual* and thus the matching function requires the candidate field to contain the same value. In either case, the field formulas (discussed below) are ignored when matching tuples to templates.

The matching function $\mathcal{M}$ used by the `in` and `read` operations is defined for a tuple θ and a template τ as:

$$\begin{aligned} \mathcal{M}(\theta, \tau) \equiv \langle \forall c : c \in \tau :: \langle \exists f : f \in \theta \\ \wedge f.\textit{name} = c.\textit{name} \\ \wedge f.\textit{type} \ = \ c.\textit{type} \\ \wedge (c.\textit{value} = f.\textit{value} \ \vee \ c.\textit{value} = \emptyset) \rangle^{3} \rangle \end{aligned}$$

For each field in the template, the tuple must contain a field with the same name and type. If a template field also specifies an actual, the field must

[3] In the three-part notation: $\langle op\ \textit{quantified\_variables} : \textit{range} :: \textit{expression} \rangle$, the variables from *quantified_variables* take on all possible values permitted by *range*. Each instantiation of the variables is substituted in *expression*, producing a multiset of values to which *op* is applied, yielding the value of the three-part expression. If no instantiation of the variables satisfies *range*, then the value of the three-part expression is the identity element for *op*, e.g., *true* if *op* is $\forall$ or $\emptyset$ when op is set.

have a value equal to the one specified. Note that a template may match a tuple with more fields then the template. Specifically, the fields in a template must only be a subset of the fields in any matched tuple. This flexibility allows applications to use data from different sources, provided a consistent naming scheme for the fields of interest.

3.3.1.2 The *formula* element

In addition to the name element, the Evolving Tuples Model also adds a *formula* element to each tuple field. The formula associated with a field describes how the field's value can be automatically updated or *evolved*. We will discuss the formula element in detail shortly. Combined with the type and value elements of the traditional tuple model, the format of an evolving tuple takes the form:

$$\begin{array}{l}\langle\ (name, type, value, formula),\\ \quad (name, type, value, formula),\\ \quad \ldots\ \rangle\end{array}$$

The tuple's producer specifies the field formulas to manage updates to the values of each field and to impart some amount of behavior to the tuple as it passes through a network. Previous to the evolving tuples framework, tuple values were either immutable, or altered only according to strict pre-existing protocols deployed to network nodes. By moving the logic of value evolution to the tuple, its producer is free to alter his tuple's behavior without altering the protocols already deployed to the network.

Name	Description
$+ \; - \; * \; /$	Arithmetic operators
$< \; \leq \; > \; \geq$ $= \; \neq$	Comparison operators
&& || !	Logical operators (and, or, not).
if(x,y,z)	Conditional statement: takes the value y when x is **true**, the value z otherwise.
exists(x)	returns the value true if a variable by the name x is present, the value false otherwise.
append(x,y)	appends the value y to the end of the value x. The value has the **list** type.
elementAt(x,y)	returns the y^{th} element from the list x
newUuid()	returns a new universally unique id.

Table 3.1: Formula Operators

In an extension to the original Linda model, Carriero and Gelernter described a similar mechanism called *active tuples* [15] which contained fields to be evaluated by the tuple space itself. This evaluation produces just a single value for the tuple field, and is only available after the evaluation is complete. The field values of an evolving tuple are always available and can be repeatedly updated by their associated formula, producing a significantly different effect.

Though it can be empty or null ($\emptyset$), a field's formula is nominally an arithmetic expression. Since tuple fields are uniquely indexed by their names, formulas can reference the values of peer fields through the field's name. In addition to normal arithmetic operators, a few simple logical functions are also provided. Table 3.1 outlines the operators that we will be using in the sections below.

Additionally, we allow expressions to access elements of a dictionary-like construct which we will call the *evolution context*. The evolution context serves as a lookup table for sensor readings, configuration information, and other contextual information related to the tuple's current location. The evolution context is provided by the process which is evolving a tuple. Since this data is indexed by names that are not necessarily unique from the names of tuple fields, formulas use the prefix "EC." to differentiate them from references to peer tuple fields. By combining values taken from the context and values already stored in the tuple, a formula can synthesize raw data values into abstract specific application values.

3.3.2 Evolution

When a tuple is *evolved*, each field's formula is evaluated and the existing value is replaced with the result. Since formulas typically combine both the previous value and the values provided in the evolution context, the new value is viewed as the subsequent evolution of the field's value. Consider the following example in which a field in an evolving tuple periodically encounters new evolution contexts. In this example, the tuple aims to maintain a field that contains the maximum value for the *temperature* field in any evolution context. This field can be defined as:

⟨ ...,
  (*maxTemp*, int, 0, if (maxTemp > EC.temperature,
                          maxTemp, EC.temperature)),
  ...⟩

When this tuple is evolved, the *maxTemp* field's value will be assigned the greater of the evolution context's temperature field (EC.temperature) and the field's current value (*maxTemp*). Each time the field is evolved with a larger *temperature* value, the field's value is updated to reflect its "environment," effectively maintaining the maximum observed value for temperature.

This formula is relatively simple, and is not effected by the order in which it is evaluated relative to its peers. Let us consider for a moment how a peer field *maxTempCount* might be effected by the order of evolution:

$\langle \ldots,$
(*maxTempCount*, int, 0, if (maxTemp > EC.temperature,
maxTempCount, maxTempCount + 1)),
$\ldots\rangle$

The intention of this field is to keep track of how many times a new *maxTemp* value has been assigned. So long as this field's formula is evaluated before the *maxTemp* field is evaluated, the formula works. However, if *maxTemp* is evaluated before *maxTempCount*, the formula will never increment its value.

To determine the order of evaluation for an evolution step, we must create a standard that is universal to all nodes. Various evaluation orders are available which would yield a simple-to-implement standard (e.g., "alphabetically by field name"), but we prefer a standard that is not only implementable, but *intuitive* also.

To derive evaluation order, we use a dependency tree and evaluate fields that are *depended upon* before the fields that *depend on* them. In addition to being intuitive, this technique ensures that the values appearing in the resulting tuple are the values used to compute the other values in the tuple, ensuring a consistent data structure.

This ordering does, however, impose the additional requirement that formulas do not create circular dependencies. We make one exception to this rule to allow a formula to reference itself. In this case, the value used is the field's previous value. We feel that the restriction on circular dependencies is more than offset by the deterministic and intuitive behavior that it provides. Given this restriction, we can formalize the `evolve()` operation using the following definitions.

First, let f refer to a field in a tuple (i.e., one name, type, value, and formula combination). Within a field, $f.formula$ refers to the code that specifies that field's evolution. Within the formalization that follows, a *formula* has three components. The first specifies the names of the sibling fields other than itself that the formula relies on. The second specifies the names of the evolution context fields that the formula relies on. The third specifies the executable behavior. That is, a formula, ϕ can be represented as the triple: $\phi = \langle D, E, behavior \rangle$. D and E are specified simply as sets of *names*. These dependencies are extracted from the formula when it is parsed by the `evolve()` operation and are easy to extract based on notation. We also define the following piece of shorthand notation: $names(\theta)$, which allows us to access

the set of names contained within the tuple θ.

Let θ' be the result tuple that is constructed incrementally during evolution from the original tuple θ. Fields in the tuple evolve one at a time, and as each field evolves, it is added to the result tuple θ'. Initially, θ' contains no fields.

Before formally defining tuple evolution, we define what it means for a single field f in the tuple θ to be *enabled*, i.e., to be capable of being evaluated:

$$\boxed{f.enabled} \triangleq$$
$$f.formula = \emptyset \ \vee \ f.formula.D \subseteq names(\theta')$$

The above states that a field's formula is enabled exactly when either no evaluation is required (the formula is $\emptyset$) or the sibling fields that a formula depends upon have been added to the new tuple θ' (i.e., they have already been evaluated for this evolution step).

We now define evolution of a tuple in terms of single steps that evolve one field at a time, ultimately generating a new tuple (θ') that has exactly the same field names and formulas as the original tuple (θ) but potentially new values:

$\boxed{\theta' := \texttt{evolve}(\theta, \varepsilon)} \triangleq$
  $\theta' = \text{newTuple}()$
  while $f := f'.(f' \in \theta \wedge f'.enabled \wedge f'.name \notin names(\theta'))$[4] $\neq \emptyset$ **do**
    if $(f.formula.E \in names(\varepsilon))$ **then**
      $new\_value := \text{exec}(f.formula, f.value, \theta', \varepsilon)$
      $\theta'.\text{add}(\langle f.name, new\_value.type, new\_value, f.formula \rangle)$
    else
      $\theta'.add(f.name, f.type, f.value, f.formula)$
    fi
  od
  while $f := f'.(f' \in \theta \wedge f'.name \notin names(\theta')) \neq \emptyset$ **do**
    $\theta'.\text{add}(\langle f.name, f.type, f.value, f.formula \rangle)$
  od

Here ε is the evolution context tuple provided by the calling process. The guard on the first loop in this definition requires that there exists a field in the original tuple that is enabled and has not yet been evaluated for this evolution step. As long as such a field exists (i.e., the non-deterministic selection results in a non-null value), the selected field is subjected to a second guard requiring the formula's dependencies on the evolution context to be satisfied. If these dependencies are present, the field is evolved, and the result of the evolution is placed in the result tuple θ'. Since the evolution context tuple ε does not change during evolution, any formula that fails the second guard cannot be evaluated during this evolution, and the field is simply copied with its original value to the new tuple. The evolution of a particular field (either evolved or copied) may enable additional fields in the original tuple whose formulas rely on the new field. When there are no more enabled fields in the

[4]The construct $x.(\texttt{condition})$ non-deterministically returns any value that matches the `condition` and will return $\emptyset$ (null) immediately if no value can be found [5].

original tuple, the second loop copies any remaining unselected fields to the result tuple; their evaluation cannot be enabled in this context.

3.3.3 The Deployment Model

Though evolving tuples and the evolution procedure itself can be used directly by an application, the true power of the evolving tuples model is realized when nodes in a network each provide a consistent processing scheme for tuple-based messages. The model presented in this section is a reference design that represents the conceptual flow of tuples through each node. While the details of any particular implementation may differ, the externally observable behaviors of each should match those of this reference design.

The reference model contains six components: three processes (`Receive`, `Director`, `Send`) and three tuple spaces (*inbound*, *outbound*, *application*). This model is depicted in Figure 3.1 and described below. We first discuss the central `Director` process, which will also introduce the three tuple spaces. We then describe how tuples are sent to and received from the network by the `Send` and `Receive` processes respectively.

3.3.3.1 The `Director` Process

When a host receives a tuple, the tuple is first processed by the `Receive` process. The details of this process are described below, but for now we can assume that this process deposits each tuple into the host's *inbound* tuple space. The evolving tuples deployment model includes a `Director` process

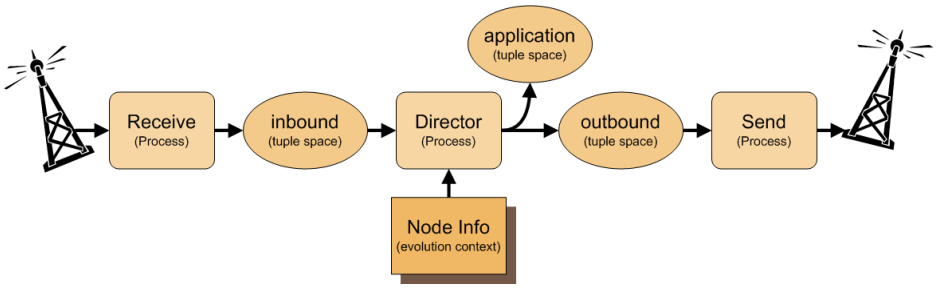

Figure 3.1: Flow chart of standard deployment model

that removes tuples from this tuple space and performs evolution on each. This evolution operation is provided with any data that the node can make available to the tuple, including sensor readings, as the evolution context.

After evolution, the tuple is inspected for a *destination* field. This field is used by the model to guide the tuple to its target. If the tuple's *destination* field contains either the node's address or the broadcast address, it is destined for this node and the `Director` process deposits a copy of the tuple in the *application* tuple space. If the tuple needs to be forwarded (because the *destination* field contains either a different node's address or the broadcast address), the `Director` process deposits a copy of the tuple in the *outbound* tuple space, where another process `Send` (described below) will deliver the tuple to its destination. If the tuple does not contain a *destination* field, it is ignored. The behavior of the `Director` method is formalized as:

```
Director ≜
    while true do
        θ := inbound.in(∅)
        θ′ := evolve(θ, evolution-context)
        if (θ′[destination].value = my-address or
            θ′[destination].value = broadcast) then
                application.out(θ′)
        fi
        if (θ′[destination].value ≠ my-address)[5] then
                outbound.out(θ′)
        fi od
```

In this definition the `Director` process first selects a tuple from the *inbound* tuple space. Here the tuple pattern $\emptyset$ is used to match *any* tuple in the tuple space. The selected tuple θ is then evolved using the node's evolution context, and the new tuple is assigned to θ'. This tuple is then deposited into one of (or both) the tuple spaces as discussed above. Since the tuple space does not maintain order, the `Director` process does not guarantee fairness when processing tuples.

3.3.3.2 The `Receive` Process

The use of the reserved broadcast address (typically -1) in a tuple's *destination* field designates that the tuple should be sent to every node in the network. The only significant complexity of the `Receive` process is a mechanism for filtering out duplicate broadcast messages. That is, a node that receives duplicate copies of the same broadcast message should not reprocess it. Because the evolving tuples model equates messages with tuples, this means

[5] Here we assume *my-address* $\neq$ *broadcast*

that the node should not deposit the "same" tuple into the *inbound* tuple space twice, even if the tuple has been evaluated in the meantime.

In our model, we accomplish this by assigning each tuple a unique identifier, or *tuple-id*. This id should be universally unique and can generally be generated using the traditional combination of some unique node id, (e.g., IP or MAC address) and a monotonic counter, though any procedure that generates universally unique identifiers is acceptable. Access to the identifier generation mechanism must be provided to both the evolution process (e.g., the `newUuid()` function in Table 3.1) and any application creating new tuples.

To suppress duplicate broadcast messages, the `Receive` process simply maintains a collection of *tuple-id*s from recently received broadcast tuples. Any broadcast tuple received that contains an *tuple-id* from this collection is considered a duplicate and simply discarded. The collection of *tuple-id*s should be pruned to prevent unbounded growth. The mechanism used for this function is not defined as part of the Standard Deployment Model.

As our notation indicates, the *tuple-id* is simply another field in the tuple. An implication of this choice is that, just like any other field, the value can actually be changed by its associated formula. In effect, the tuple can be replaced by a "new" tuple which, in all respects other than the id, is identical to the old tuple.

3.3.3.3 The `Send` process

In an evolving tuples network, messages between nodes are simply the tuples themselves. Tuples from the `Receive` process or any application that need to be sent out on the network interface are placed in the *outbound* tuple space. Since messages require a destination, the reference model assumes each tuple has, at a minimum, a *destination* field. The producing application should initialize the value of this *destination* field to the address of a neighboring node, or to the broadcast address. The `Send` process continuously mines the *outbound* tuple space, removing each tuple and transmitting it to the node indicated by the *destination* field. If this transmission fails, the tuple is redeposited into the *outbound* tuple space where it may be selected at a later time for another attempt. Formally, this process can be stated as:

```
Send ≜
    while true do
        template := ⟨(destination, int, ∅)⟩
        θ :=outbound.in(template)
        if θ[destination].value ≠ ∅ then
            success := send(θ, θ[destination])
            if (success = false) then
                outbound.out(θ)
            fi
        fi
    od
```

The `Send` process non-deterministically removes a tuple from the *outbound* tuple space. If the tuple has a null ($\emptyset$) *destination*, the `Send` process ignores (i.e., discards) it. Otherwise, the process delivers the tuple to the

address specified by the *destination* field. If this delivery fails, rather then immediately reattempting to send the tuple, the process deposits the tuple into the *outbound* tuple space from which it will eventually be reselected[6].

Note that throughout the `Director` and `Send` processes, tuples are handled based solely on their *destination* fields. Since the field's value can be updated by its formula, the tuple can provide its own routing information. This *self-routing* functionality can be leveraged by a tuple author to elicit elaborate behavior from a tuple such as way-point based routing, or even request-reply routing without any support from the application layer. We will examine an example of this in section 3.4.

3.3.4 Safety Properties

We have not yet performed a detailed analysis of the safety properties of the Evolving Tuples Model, but a cursory examination yields a few interesting observations. For example, the duplicate message elimination mechanism relies on values provided by tuples' `tuple-id` field. Since this field can be updated by the tuple itself, it is easy to design a tuple that will never be dropped by the `Send` process as a duplicate. Combined with a `destination` field directing the node to always broadcast the tuple, a broadcast-storm can be unleashed accidentally (or maliciously) on a network of evolving tuple nodes.

[6]Here we are making no assertions about *how* the send method delivers the tuple. Specifically, the `Send` process does not limit the delivery to single-hop transmissions but rather uses whatever technique the underlying network interface chooses to support. Control over this decision an area for future investigation.

While the network is exposed to this and other safety issues, we believe that nodes themselves and the evolution process in particular are not subject to these types of attacks. Evolving Tuples must be passed in network messages and therefore must have a finite size. This implies that the functions contained in the tuples must also have finite length. Since the evolving tuples formulas do not allow looping constructs, their evaluation must terminate in a finite number of steps.

Though the evolution process can be considered safe, the risk to the network due to broadcast-storms and related issues implies that any Evolving Tuples network exposed to the public should take precautions. Research on security and safety issues for Mobile Agent and Active Network technologies can provide a basis for this future work.

3.4 Resource Discovery with Evolving Tuples

In this section we describe the use of Evolving Tuples for resource discovery. The primary goal of this application of evolving tuples is to provide support for the Application Sessions Model's predicate functions, which can be used to select resources from remote clients. When nodes share resources with clients, they must be able to receive resource requests and send appropriate replies. Often the process of matching resources to requests requires information from application level processes (e.g., physical location) as well as from various levels of the networking stack (e.g., link latency, buffer saturation). This context information is often restricted to the values originally built into

the resource discovery protocol. If a certain attribute (e.g., manufacturer) was not specified during the protocol's design, a resource request may not be able to restrict resources based on its value. Evolving tuples allow us to collect context information by name from any provided context and allows the tuple author to combine these values using mathematical and logical operators to form a variety of selection and preference behaviors.

In this section, we first show a discovery request expressed as an evolving tuple, explain the various fields, and show how the fields provide context-aware resource discovery. As resource discovery tuples are broadcast across the network, they will make their way into the *application* tuple space. At this point, the task-specific process `Discovery` defined below in Section 3.4.2 is responsible for processing these tuples and generating reply messages. After an explanation of how this process is modeled, we include a step-by-step example of a discovery tuple as it passes through a sample network.

To reduce the complexity of the example, we assume that the `Send` process on the nodes of this network has access to a reliable multi-hop routing mechanism for unicast addresses, and 1-hop neighbor delivery for the broadcast address. We will explore some alternatives to this assumption in the next section when we discuss multi-hop source routing with Evolving Tuples. The selection or restriction of multi-hop routing could be considered an application level concern, and is one of the tasks left for future work noted in Section 3.7.

3.4.1 Crafting a Resource Request

A resource discovery request, like other evolving tuples, is a self-routing structure that gathers data and changes its values as it progresses through a series of evolutions. The tuple will cross a multi-hop network and will be evolved in the context of potentially matching resources. If a suitable resource is found, the tuple switches itself from a request to a reply and sends itself back to the original source.

Table 3.2 shows an example tuple that a requester could create to locate a VOC detector within walking distance of a given location. To support the "Danger Monitor" example application in the previous chapter, a tuple like this could be used to satisfy the resource predicate. This request has embedded within it all of the information necessary to evaluate the request (in the context of potential resources) and to return replies to the source. Because this definition is under the control of the requester, many options can be changed depending on application requirements. For example, the reply could be dispatched to a node other than the requester itself. Below we explain each field and formula in the example, which assumes a broadcast address of -1.

- *source*: The network address of the node sending the resource request. In our example, this value will be used as our destination for the reply sent when the request finds a matching resource.

- *distance*: A simple distance calculation to bound the resource request geographically. This field actually contains the *square* of the difference

Name	**Value**	**Formula**
source	5	$\emptyset$
distance	0	$(30.2867 - latitude)^2 +$ $(-97.7364 - longitude)^2$
match	$\emptyset$	if ($match \neq \emptyset$, *match*, (*distance* $\leq$ 0.00001 **and** EC.*resource-type* = "voc-detector"))
msg_type	"discovery-request"	if (*match* = true, "discovery-reply", "discovery-request")

destination	-1	if (*match* = $\emptyset$,
		if(*distance* $\leq$ 0.00001, -1, $\emptyset$),
		if(*match* = *true*, *source*, $\emptyset$)),

resource-uri	$\emptyset$	if (*resource-uri* = $\emptyset$ **and** *match* = true, EC.*uri*, $\emptyset$)

Table 3.2: Example discovery tuple

in latitude and longitude, which serves as a reasonable approximation for this application (and location). When the field is evolved by processes without location information, the *latitude* and *longitude* fields are not present in the evolution context, preventing the formula from being evaluated. In this case, the field retains its previous value, the best available approximation.

- *match*: Serves as a simple switch to notify other fields that a resource has (or has not) been found. The initial value is set to $\emptyset$ to indicate that no resource has been evaluated. When the formula is evaluated, the value becomes either *true* to denote a suitable resource, or *false* to denote an unsuitable resource. We use an if statement to preserve the field's value if it is non-null.

- *msg_type*: A flag used to retrieve resource discovery tuples from the *application* tuple space. Initially, and any time before the *match* field is set to true, the message type is "discovery-request". Once the tuple has found a matching resource, the message type is changed to "discovery-reply". Since this reply tuple will eventually be returned to the source, a process there can use this value to retrieve a successful resource discovery. The *msg_type* field may also be used by other application level processes, for example a print daemon might use tuples with a *msg_type* of "print-job".

- *destination*: Used by both the `Director` and `Send` processes to guide the tuple. Until a resource has been evaluated (i.e., *match* = $\emptyset$) the *destination* is a function of *distance*. We set the *destination* to the broadcast address if the next hop will not exceed our scope, and set the value to null ($\emptyset$) if it will. While the *destination* field is set to the broadcast address, the `Director` process will deliver a copy to both the *application* and *outbound* tuple spaces. Tuples deposited into the *application* tuple space will be subject to the discovery resolution methods described in Section 3.4.2 below. Tuples deposited into the *outbound* tuple space will be forwarded by `Send` appropriately. After a tuple has been through discovery resolution, the *destination* is a function of *match*. If *match* is *true*, we set our *destination* to the value of the *source* field. If *match* is *false*, we set our *destination* to null, causing the `Send` process to drop the tuple.

- *resource-uri*: originally null, this value is set to the URI[7] of the matched resource. The `if` statement guards the value from being updated once it has been set and only sets it when a match has been found. This information can be used by the application to actually use the resource (e.g., as a member of an Application Session).

As shown above, the data and logic necessary for selecting resources is completely contained within the tuple with the exception of the requisite

[7]Uniform Resource Identifier

resource attributes. This reduction frees the host system from any deep involvement in the process of discovery, and, as we will see next, the interface between the host and tuple is reduced to just the *destination* and *msg_type* fields.

3.4.2 Discovery Resolution

In our model, the `Director` process evolves request tuples using general information about the node in its evolution context. However, this information does not include the specific resources that the node is exposing to external clients. Instead, this information is handled by a higher-level process, `Discovery`. The `Discovery` process mines the *application* tuple space for resource requests, and evolves each tuple using a different evolution context populated with information about a shared resource. As shown in Figure 3.2, each tuple removed from the *application* tuple space is evolved once for each shared resource, and the results are deposited into the *outbound* tuple space.

Instead of directly depositing the new tuples into the *outbound* tuple space, the `Discovery` process could filter the tuples based on their *match* field, only passing on those that are actual matches. However, this introduces additional coupling between the tuple format and the process implementation, which we are trying to avoid. For this reason, we simple let the `Send` process drop the tuple due to a null *destination* field. We formally describe the `Discovery` application process as:

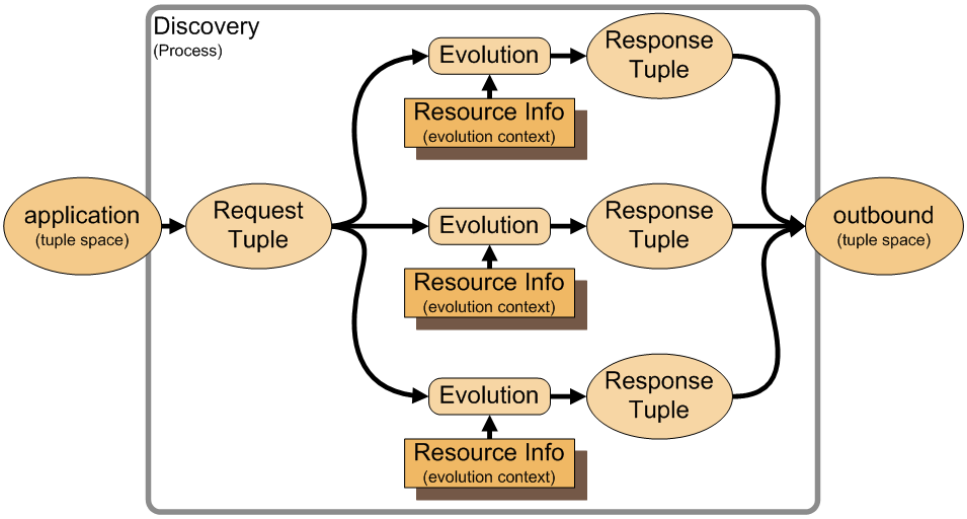

Figure 3.2: The `Discovery` process

Discovery $\triangleq$
  template := $\langle$(*msg_type*, string, "discovery-request")$\rangle$
  while *true* **do**
    θ = *application*.`in`(*template*)
    res[ ] := `getResources`()
    for each *r* **in** *res*
      θ' := `evolve`(θ, *r*)
      outbound.`out`(θ')
    rof
  od

This definition contains only the procedural framework for discovery, while all of the logic of matching resource attributes and determining resource satisfaction is delegated to the formulas of the evolving tuple. In this definition, we explicitly use tuples to provide the evolution context functionality. These tuples representing the properties of each resource are provided by the method `getResources()`, which could in turn retrieve its tuples from a separate tuple

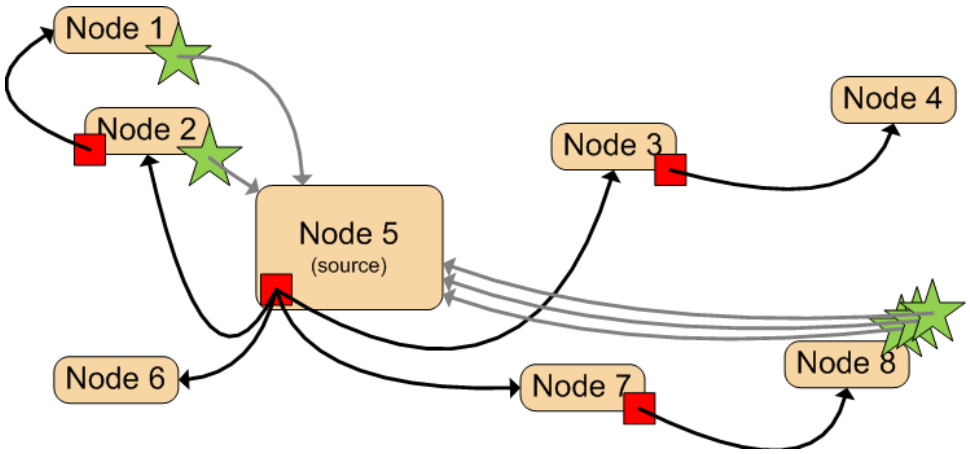

Figure 3.3: The dissemination of *discovery request* tuples followed by the return of *discovery reply* tuples

space (not shown). By using a "probing group read"[8]of this "resources tuple space," the operation would return all of a host's resources in a single step.

3.4.3 Discovery Example

In this section, we will step through the entire process of sending and returning resource discovery tuples. For this example we will use the network depicted in Figure 3.3. The discovery tuple from Table 3.2 is sent by node 5 through the entire network, but we will concentrate on the tuples that pass through nodes 1 and 2.

The process begins when an application on node 5 creates a resource-discovery tuple. This tuple is inserted directly into the `outbound` tuple space

[8]a "probing group read" is a common tuple space extension that atomically returns copies of all tuples in a tuple space that match a template.

where it is removed by the `Send` process described in Section 3.3. The `Send` process reads the value of the *destination* field (the broadcast address -1) and sends the tuple to each of the node's neighbors (2, 3, 6, and 7). When this tuple is received at node 2, it is handled first by the `Receive` process and then by the `Director` process and encounters its first evolution.

This evolution is performed using an evolution context containing values describing the node itself. First the *distance* field's formula shown in Table 3.2 is evaluated which updates the current distance (squared) from our office, for example 0.000001. Since the `Director` process's evolution context does not contain a *resource-type* entry, the dependencies of the *match* field can not be resolved and it retains its current value (∅). The other tuple fields (*destination*, *msg_type*, and *resource-uri*) each depend on *match*, so they too are not evaluated and retain their initial values.

After this evolution step is complete, the new tuple is inserted into both the *application* and *outbound* tuple spaces. From the *outbound* tuple space, the `Send` process will re-broadcast the tuple to all of the node's neighbors, where receptions of duplications of the same broadcast are handled as described in Section 3.3. The tuple in the *application* tuple space is removed by the `Discovery` process described in the previous subsection, which will then evolve the tuple in the context of each of the resources available on the node.

For the purposes of discussion, we will assume that node 2 contains only one resource, and the tuple describing it has a field named *resource-type* with value "voc-detector." The `Discovery` process removes the tuple from the

application tuple space and evolves the tuple with this resource's description as the evolution context. Since the *distance* field depends on context fields that are not available, its value is not changed. However, since this field (*distance*) and the evolution context field *resource-type* are both available, the *match* field is enabled. When it is evaluated, the value for the *match* field is set to *true*. This enables evolution of the remaining fields: *destination*, *msg_type*, and *uri*. The *destination* field is updated to the value of the *source* field (5), the *msg_type* field is set to "discovery-reply," and the *resource-uri* field is set to EC.*uri* (e.g., http://2:83/voc). The `Discovery` process then places this evolved tuple into the *outbound* tuple space where the `Send` process will send it to its new destination, node 5.

As mentioned previously, the `Send` process also sent the first evolution of our tuple to node 2's neighbors, or more specifically, to node 1. In our example depicted in Figure 3.3, node 1 contains a matching resource which causes the `Discovery` process to evolve a *discovery-reply* tuple and place it in the *outbound* tuple space. However, if node 1 contains a resource with a *resource-type* field that is not equal to "voc-detector", the evolution will result in a new tuple with a *match* value of *false* and a *destination* field of $\emptyset$. When this tuple is removed by `Send` from the *outbound* tuple space, it is ignored and dropped due to its null destination. In contrast, node 8 creates multiple discovery-reply tuples for each matching resource hosted on the node. Each of these tuples are inserted into the local *outbound* tuple space and are forwarded individually to node 5.

When the *discovery-reply* tuples arrive at node 5, the application that inserted the original tuple (or potentially another application) will retrieve the reply tuples from its *application* tuple space using `in`($\langle$(*msg_type*, string, "discovery-reply")$\rangle$). In our sample, the application would then use the value in the *resource-uri* field to begin communications with the `voc-detector` resources that were discovered.

3.5 Route Discovery with Evolving Tuples

While experimenting with the Evolving Tuples Model for resource discovery, we have also found the model to be useful for prototyping certain types of pervasive computing applications. In this section we demonstrate how the model can be used for this purpose. Specifically, we incrementally derive a route discovery protocol for pervasive computing in which each step in the derivation adds additional functionality. This is not presented as a new protocol for study, rather we intend to demonstrate that the evolving tuples model enables designers to rapidly implement and deploy functionality to validate new behavior *in situ*.

In this study, we use *source routing*, a technique that has enjoyed significant success in mobile computing due to its robust and autonomous nature [19, 39, 43][9]. In its simplest form, source routing requires the message

[9] A number of other routing techniques have also been shown to perform well, e.g., AODV [65] and DSDV [64]. We do not promote a particular protocol or style, but have selected source routing to demonstrate our approach.

sender to include the message's route through the network by specifying the addresses of each intermediate node. This is typically accomplished by embedding an ordered list of network addresses in the message header. Either by cycling the addresses in this list, or by maintaining an index in the message's header, a relaying node can determine the next recipient in the path.

Before application data can be sent, source routing requires a sender to discover a route to the destination. In mobile and pervasive computing networks, this discovery is usually accomplished by flooding a *route-request* packet through the network. As a node forwards the packet, it appends its own address to an accumulating ordered list of addresses. This list represents the path the route-request message has taken. Assuming bi-directional links when the target node receives this tuple, it can simply reverse the path to send a *route-reply* packet back to the original sender.

Our derivation starts with a simple source routing protocol and demonstrates how it can be quickly and easily prototyped using the evolving tuples model. We then incrementally extend this basic protocol to include additional behaviors that eventually lead to a context-aware source routing protocol. Each derivation of the protocol requires only changes to the routing application's tuples, without requiring any changes to the code already deployed on nodes in the network.

3.5.1 Version 1: A Basic Protocol

To implement basic source routing functionality, we introduce an evolving tuple that performs the basic functions of route discovery. By modifying field values to slightly change behavior, the same tuple is used for both the request and reply. Initially we assume that connections are uni-directional, an assumption supported by empirical results demonstrating that this is often the case in wireless networks [51, 52]. This assumption requires both the route-request and the route-reply message to be flooded across the network to build a route in each direction. For this reason, some route discovery algorithms embed the route-reply message inside a new route-discovery message sent from the target back to the source [43]. In the evolving tuples approach below, the route from the source to the target *and back* is recorded in the single route-request/route-reply tuple.

The original tuple deposited by the application is shown in Table 3.3 (the types of each field have been removed for brevity). In this table and the examples below, we will assume that the initiating application resides on a node with address 0 and it is attempting to discover a route to a node with address 2. These values are stored in the *source* and *target* fields in the evolving tuple. The *onSource* and *onTarget* fields are flags to be used by other formulas in the tuple and signal that the tuple is being evolved upon its return to the source node and the target node respectively. In a final version of the tuple, these flags could be in-lined, but they are useful for demonstrating the framework.

Name	Value	Formula
source	0	Ø
target	2	Ø
onSource	false	EC.address == *source*
onTarget	false	EC.address == *target*
route	0	append(*route*, EC.address)
destination	-1	if (*onSource*, EC.address, -1)
id	`newUuid()`	if (*onTarget*, `newUuid()`, *id*)

Table 3.3: Route Discovery Tuple (Version 1)

The *route* field carries the accumulated route by appending the current node's address, EC.address, to the end of the route field's current value at each evolution. This includes both the trip from the source to the target, and the return trip[10]. The *destination* field, as described in Section 3.3.3, is used to send the tuple to the next node. Since both the request and reply must be flooded across the network, the *destination* field is almost always the reserved broadcast address *-1*. In the case that the tuple is being evolved on the source node itself, we know that the tuple is a route reply and does not need to be forwarded to any other nodes. In this state we set the destination to the source node's address, which prevents it from being deposited into the *outbound* tuple space.

As mentioned in Section 3.3, the broadcast dissemination protocol on a node in an evolving tuples network avoids duplicate transmissions by caching the unique identifiers of the tuples it has recently transmitted. To differentiate

[10]The resulting path must be split into the out-bound and in-bound routes by the application.

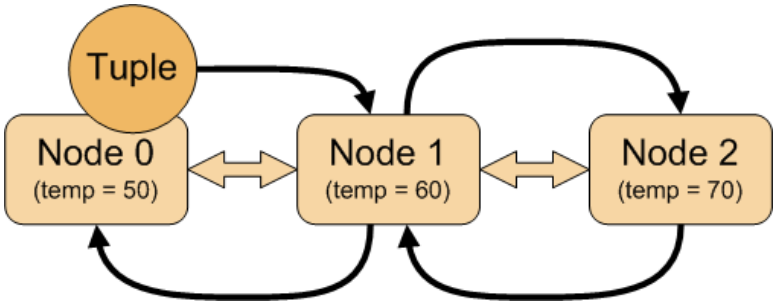

Figure 3.4: An example network for route discovery with Evolving Tuples

the route-reply tuple from the initial outbound route-request tuple, it must have a different value for its *id* field before it is sent back to the requester. This is accomplished by adding a formula to the tuple's *id* field. When the tuple changes from a route-request into a route-reply, the field's formula assigns the tuple a new id. For the purposes of our example, we use tuple ids composed of two parts: the id of the node generating the tuple and a counter. For example, the first tuple id generated by Node 0 in this example is 0.0. The second id would be 0.1. The first tuple id generated on node 2 would be 2.0

Figure 3.4 gives an example. When the tuple in Table 3.3 is deposited in Node 0's *outbound* tuple space, it is broadcast to all neighboring nodes (i.e., Node 1). As the tuple moves through this simple network and is evolved, its fields' values change. The new values are shown in Table 3.4 as they would appear after evolution on the node at the top of the column.

We briefly walk through the tuple evolution, using the evolutions from Table 3.4 as an example. Since evaluation order is governed by dependency assurance (see Section 3.3.2), even if the tuple fields are arranged in a different

Field Name	Node 0	Node 1	Node 2	Node 1	Node 0
source	0	0	0	0	0
target	2	2	2	2	2
onSource	false	false	false	false	true
onTarget	false	false	true	false	false
route	0	01	012	0121	01210
destination	-1	-1	-1	-1	0
id	0.0	0.0	2.0	2.0	2.0

Table 3.4: Route Discovery Tuple (Version 1) values as tuple propagates through the example network

order, the evolution is deterministic, and the results would be the same. For reference, the dependency tree for the formulas in our example is shown in Figure 3.5. Since formulas can be evaluated once the fields they depend upon are evolved, the order of evolution may be slightly different than described here:

- *source* and *target* have no formula and thus depend on no other fields. These values are retained.
- *onTarget* and *onSource* depend on the *source* and *target* fields and the *address* value in the evolution context. If EC.address is present, then the fields are evaluated. If the local node's address matches the *target* or *source* field values, then the *onTarget* and *onSource* fields are set to *true* (respectively), otherwise the fields are set to *false*.
- *route* depends only on EC.*address* (and itself). This formula updates the field value by appending the current node's address to the ordered list already stored there.

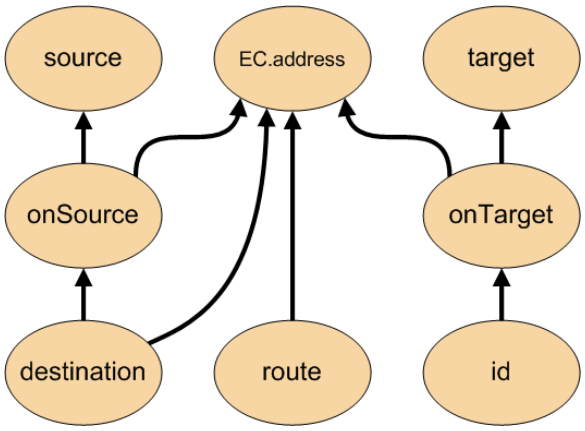

Figure 3.5: Route Discovery Tuple (Version 1) field formula dependency graph

- *destination* depends on the *onSource* field. Once this field is evolved, the destination formula is evaluated. The field is usually set to -1 (the broadcast address) with the exception of when *onSource* is *true*. In this case we set the *destination* field to the current node's address preventing it from propagating any further.

- *id* depends only on the *onTarget* field. When the tuple is being evolved on the target node, it needs to choose a new id to allow it to be re-flooded across the network back to the source. Here the formula calls upon the `newUuid()` function to generate a new globally unique identifier.

3.5.2 Version 2: Bidirectional Links

The first derivation of our prototype protocol targets networks in which unidirectional links are present. When links are bidirectional, the target node can eliminate the use of flooding for the route-reply message by inverting the

discovered path in the route-request message. As in the previous version, the *route* field will store a history of nodes visited. However, in this version of the tuple, there is no need to store addresses in the return trip. To make this determination easier, we introduce the *isReply* field to determine in which phase of route discovery the tuple is currently operating.

When the tuple is operating as a reply message, we use the addresses in the *route* field to direct the tuple back to the source. Since we are no longer broadcasting, the *destination* field must be updated on each evolution to be the address of the next node in the route. The *destination* field formula in this tuple has three cases. As in the previous version, if the tuple has returned to the source, the destination is set to the source's address. In the case that the tuple is in the request phase (still searching for the target), the destination is set to the broadcast address (-1). In the final case, we select the next address from the previously assembled route using the value of the *routeIndex* field, which reverses the route that was received at the target. This is done by setting the index correctly at the target, and then decrementing it after each hop along the route, or more specifically, on each evolution. If the tuple is operating as a route request, the value is uninportant, so we set it to -1. These fields and formulas are shown in Table 3.5.

In Table 3.6 we show how the values for the fields in this tuple evolve as the tuple passes through the example network shown in Figure 3.4. For brevity, we show only the fields that have been changed for this version of the protocol. Note that in this version, the *route* list continues to collect node

Name	Value	Formula
source	0	Ø
target	2	Ø
onSource	false	EC.address == *source*
onTarget	false	EC.address == *target*
isReply	false	*isReply* || *onTarget*
route	0	if (!*isReply* || *onTarget*, append(*route*, EC.address), *route*)

routeIndex	-1	if (*isReply*,
		if (*onTarget*,
		size(*route*)-2,
		routeIndex-1),
		-1)

destination	-1	if (*source* == EC.address,
		EC.address,
		if (*isReply*,
		elementAt(*route*, *routeIndex*),
		-1),
		)

id	`newUuid()`	Ø

Table 3.5: Route Discovery Tuple (Version 2)

Field Name	Node 0	Node 1	Node 2	Node 1	Node 0
...	...	...	...	...	...
isReply	false	false	true	true	true
route	0	01	012	012	012
routeIndex	-1	-1	2	1	0
destination	-1	-1	1	0	0
id	0.0	0.0	0.0	0.0	0.0

Table 3.6: Route Discovery Tuple (Version 2) values as tuple propagates through the example network

addresses as the tuple is returned to the source.

3.5.3 Version 3: Context-Based Discovery

The first two versions of our mobile communication protocol assume that a source node knows the network address of the destination (2). In pervasive computing networks (especially those that include nodes from different administrative domains) applications may not have the benefit of knowing each other's addresses *a priori* [16, 82]. In the absence of this pre-shared information, applications can use centralized or distributed service directories to locate addresses of nodes which host specific information or services [24, 32, 53]. Another approach to service discovery is to use context-based information at the routing layer [47, 82].

The next step in our derivation changes our previous protocol to one that searches for the target based not on its address, but on a description of the node. In the evolving tuples model, the attributes of a node are represented as part of the context that node provides. In evolving tuples language, this

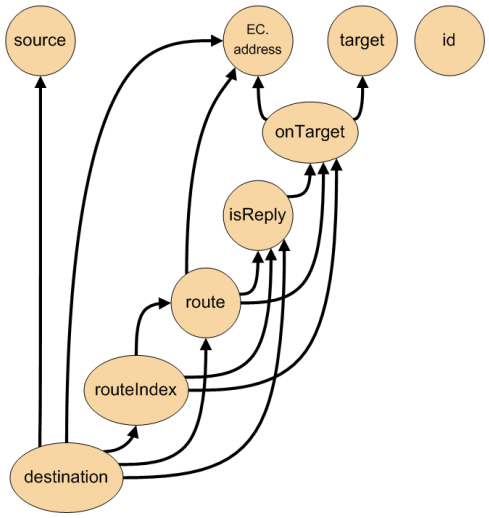

Figure 3.6: Route Discovery Tuple (Version 2) field formula dependency graph

means descriptive attributes are stored in the evolution context. As our route-request tuple travels through the network, the task of determining whether it has found a compatible target must now examine a descriptive attribute rather then a prescriptive attribute. Specifically we will be searching for a node with a temperature reading larger than 65 rather then a node with a pre-determined address.

The evolving tuple that accomplishes this is shown in Table 3.7. We simply update the formula for the *onTarget* field to examine the *temperature* context field rather then the *address* field. We also set the initial value of the *target* field to null ($\emptyset$) to reflect the lack of a concrete target. The formula for *target* is also updated to set its value to the network address of the current node when the *onTarget* field is true. The formulas for the *route* and *destination* fields can be selected from either of the previous tuple versions based on the

Name	Value	Formula
...	...	...
onTarget	false	EC.temperature > 65
target	$\emptyset$	if (*onTarget*, EC.address, *target*)
...	...	...

Table 3.7: Route Discovery Tuple (Version 3)

Field Name	Node 0	Node 1	Node 2	Node 1	Node 0
onTarget	false	false	true	false	false
target	$\emptyset$	$\emptyset$	2	2	2

Table 3.8: Route Discovery Tuple (Version 3) values as tuple propagates through the example network

presence of unidirectional links. Here we store the address of the matching node as *target*. This value can be used to split the accumulated route into out-bound and in-bound segments if we use Version 1's technique. The values of these fields as they pass through our sample network are shown in Table 3.8. In this example, the route-request tuple is searching for a node whose temperature value is greater than a specified threshold (65).

3.5.4 Version 4: Context Collection

Our derivations so far have generated a protocol that can find and return network routes using either unique addresses or context information to designate the route's endpoints. Using elements from these protocol approaches, we can now use the evolving tuples model to derive additional protocols with more complex behavior. For example, applications designed for pervasive computing networks are commonly interested in both the data stored

Name	Value	Formula
numTemps	0	if (exists(EC.temperature), *numTemps*+1, *numTemps*)
totalTemp	0	*totalTemp* + EC.temperature

Table 3.9: Route Discovery Tuple (Version 4)

at the ultimate destination and data from intermediate nodes (e.g., average temperature sensor readings). This is often addressed using protocol schemes that aggregate data along paths in a network [55]. Such approaches have been shown to have significant performance benefits when compared to contacting each node on the route individually and then locally aggregating the results.

Along these lines, our next protocol derivation adds the ability to aggregate contextual information encountered as a tuple traverses the network. In this particular example we collect enough information to determine the average temperature of the nodes along the route. Similar approaches could be used to collect other common aggregates (e.g., minimum, maximum, count, etc.). In addition, the expressiveness of the evolving tuples formula language allows the tuples to compute more complex, application-defined aggregates as well, e.g., such as those defined in [72] and [36].

In the tuple shown in Table 3.9, the sum of all the temperatures encountered is maintained as the value of the *totalTemp* field, and the number of temperature readings is maintained in the *numTemps* field. In this example, the final calculation (*totalTemp*/*numTemps*) is left to the application receiving the returned tuple.

Field Name	Node 0	Node 1	Node 2	Node 1	Node 0
numTemps	0	1	2	3	4
totalTemp	0	60	130	180	230

Table 3.10: Route Discovery Tuple (Version 4) values as tuple propagates through the example network

The values of our two new fields as the tuple propagates through the network are shown in Table 3.10. Variations of this tuple could be created to maintain separate averages for the route-discovery and route-reply phases if the application required by simply using the existing *isReply* field as a guard.

While this is just a simple example of how an evolving tuple can be used to provide in-network context collection and aggregation, it demonstrates the power of the evolving tuples model. By using the evolving tuples model to prototype such sophisticated communication schemes, protocol developers can use and evaluate various context values quickly and effectively without requiring a recompilation or redeployment process. It also allows the developers to deploy and evaluate the protocols on test-bed networks in advance of creating a full-fledged low-level implementation.

3.5.5 Version 5: Context-Based Flooding Optimization

In typical implementations of flooding based protocols, we often find some sort of mechanism to limit the scope of messages to prevent the consumption of valuable resources far from the area of interest. Normally, these mechanisms are based on the number of network transmissions that the message has been passed along (i.e., hop-count or TTL). Other metrics can also

Name	Value	Formula
...	...	...
inFloodRegion	true	EC.temperature > 35
destination	-1	if (! inFloodRegion, $\emptyset$, *old destination formula*)
...	...	...

Table 3.11: Route Discovery Tuple (Version 5)

be used to limit the distribution of flooded messages, for example nodes that are aware of their physical location can incorporate geographical boundaries.[11]

Using evolving tuples, we can not only implement these simple techniques, but we can combine a variety of contextual data to precisely limit flooded messages to regions of interest on a per-message basis. As we have done previously, we present a simple example in Table 3.11, but much more complex variations are clearly possible. The approach here is to simply guard the previous tuple formula for the *destination* field with a check to the *inFloodRegion* field. This field is initially set to true, but is set to false if the tuple encounters a node where the temperature is below 30.

Taken together with the previous versions, our tuple now implements source-based route discovery for networks with unidirectional links to find a node whose temperature is greater then 65 without traversing any links to nodes with temperatures less then 35. Additionally, the route's average temperature is easily calculated from the values collected as the message traversed

[11]The resource discovery example in Section 3.4 used location to define geographical boundaries in this way.

the network.

3.6 Evaluation

Ideally, the evaluation of the model would be performed in a collection of hundreds of heterogeneous devices deployed at a construction site or similar environment. Unfortunately, a full-scale test is outside the scope of this dissertation and modest simplifications must be made. In this section we will detail the methods used to evaluate the use of evolving tuples for prototyping in sensor networks and the results extracted from several experiments.

The first section below describes the Evolving Tuples Model implementation for the SunSPOT[12], a mid-level sensor platform combining a micro controller, an 802.15.4 [40] radio, a rechargeable battery, and several environmental sensors. This implementation was used to validate the model and the examples from Section 3.5. In the following section, we describe how the core libraries from the SunSPOT implementation are re-used for large-network simulations to provide quantitative results.

3.6.1 Implementation for SunSPOT platform

We implemented the Evolving Tuples Model for the SunSPOT [73] platform. The SunSPOT project combines the Squawk Virtual Machine [74] with wireless sensor hardware to create an ideal platform for prototyping new

[12]Small Programmable Object Technology

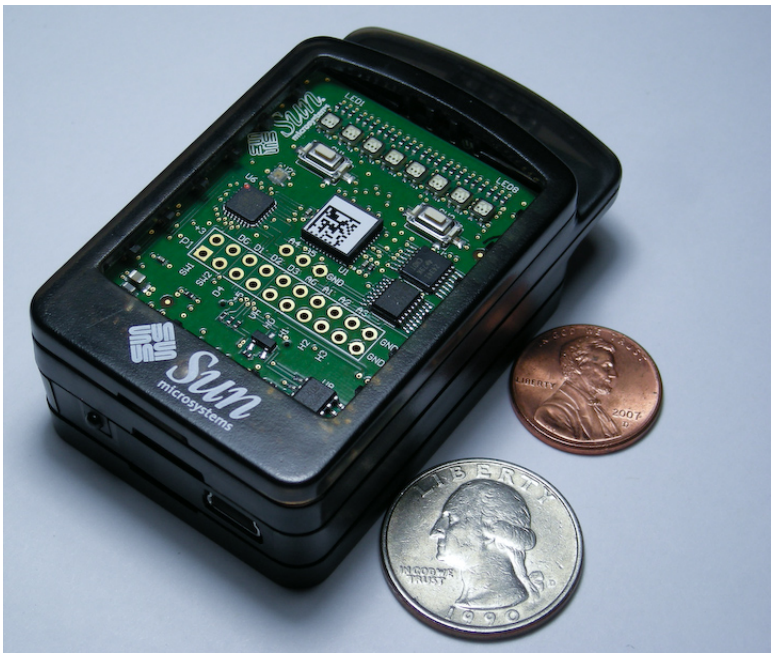

Figure 3.7: SunSPOT Device

pervasive computing applications. For reference, Figure 3.7 shows one of our SunSPOT devices.

As if to prove the difficulty of application development for pervasive computing applications, the first major implementation challenges were associated with building and deploying any working program for the device. This process requires locating and installing development tools from the vendor, updating firmware, and a variety of configuration changes to support the new device. Additional work was required to integrate the tools into a familiar development environment and to port pre-existing code to the SPOT variant of the Java platform. These challenges proved to be major stumbling blocks.

The source code for the Evolving Tuples prototype is divided into three major libraries: Core, Math, and Evolving Tuples. In the sections below we give and overview of the components in each of these libraries. The complete source code and detailed documentation are available on the project website [76].

3.6.1.1 Core Library

The core library delivers general purpose utilities and frameworks. Most fundamentally it provides the `Set` or `Bag` data types which are not included in the standard SunSPOT platform, but are the basis for Tuple Space implementations. In addition to data types, support for daemon threads is also absent, making background services difficult to create. This library also includes classes to support background services by abstracting away most of the concurrency and lifecycle issues and delegating to "work" of the services to clients. Properly synchronized and throughly tested implementations of these classes are the primary focus of the core library.

However, as a secondary focus, we also addressed the biggest productivity challenge: a fast and robust unit test framework. While vendor tools provide some support for unit testing, they assume tests are run only on the SunSPOT devices themselves. As a result, the user is forced to deploy unit tests to the device each time they are to be run, a process that can consume several minutes per device. This effect is magnified by tests that require the passing of tuples across a network which implies deployment to several devices.

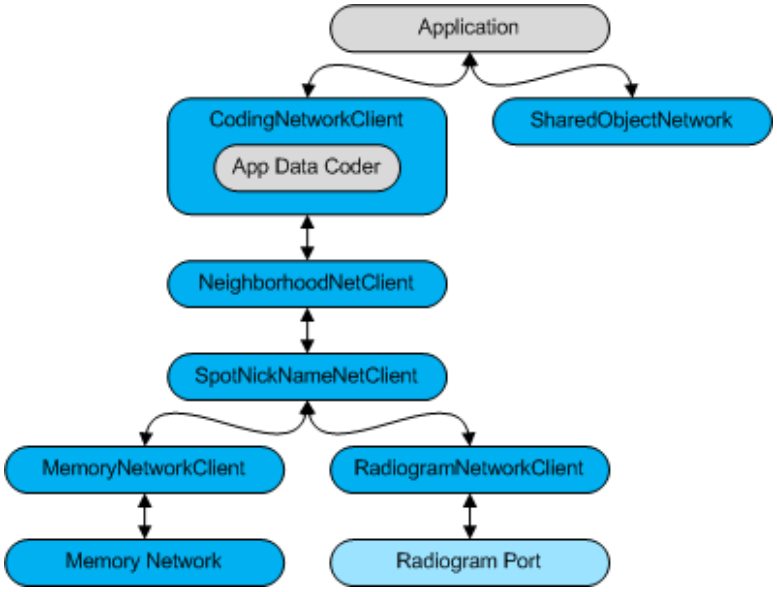

Figure 3.8: Communications Stacks provided by the Core Library for high-level applications on SunSPOT

To address these issues we designed a set of components to emulate a datagram network that can be executed on *either* a desktop computer or the SunSPOT devices themselves. The `SharedObjectNetwork` and `MemoryNetwork` emulators allow clients to exchange *object references* and *byte arrays* respectively. In both implementations network clients operate in separate threads, and the platform's native memory synchronization mechanisms are used to mediate message delivery. The first component (`SharedObjectNetwork`) is much simplier to configure and includes features that make it ideal for high-level unit testing. The latter component (`MemoryNetwork`) is built to closely resemble the SunSPOT `Radiogram` protocol to support low-level and system tests.

As part of the unit test framework, a simple layered communication stack provides features commonly used by network applications. Figure 3.8 shows how these components are typically assembled. Each layer of the stack exports the same API so that they may be interchanged to provide alternate behaviors (except the `CodingNetworkClient` and `SharedObjectNetworkClient` which accept object references rather then byte arrays). A brief description of each component in Figure 3.8 is given below:

- *CodingNetworkClient:* Accepts object references from clients and encodes them into byte-arrays; decodes byte-arrays received from its delegate into object references for consumption by clients.

- *AppDataCoder:* An application specific component provided to the CodingNetworkClient when the stack is initialized; this component handles the details of the object / byte conversions independently of network abstractions.

- *NeighborhoodNetClient:* Filters incoming and outgoing messages according to a client-specified list of neighbooring nodes; this component allows for an artificial topography to be imposed on the network during testing and for neighborhood maintenance in deployments.

- *SpotNickNameNetClient:* Maps short (2 byte) network names to full MAC addresses required by delegate layers; this component allows for easier debugging and smaller routing tables. These optimizations are potentially unnecessary in full deployments.

- *MemoryNetworkClient / RadiogramNetworkClient:* Manages the technology specific details of the connections to neighboring hosts.
- *MemoryNetwork:* Shared memory MAC / PHY implementation.
- *Radiogram Port:* SunSPOT 805.14.5 MAC / PHY implementation provided by the platform.

3.6.1.2 Math Library

The most difficult technical challenge proved to be the parsing and evaluation of arbitrary mathematical expressions. To reduce errors and effort, our final solution was to port a pre-existing open-source library [29] to provide most of this functionality. Much of this source was automatically generated by a parser generator which explains the relatively large size (∼3500 lines) of this library's source code. Unfortunately, the porting process removed many of the original features due to the limited functionality available in the Squawk JVM. The resulting code base was then wrapped in a simplifying API to bring it to the proper level of abstraction for consumption by the Evolving Tuples Library.

3.6.1.3 Evolving Tuples Library

This library (not surprisingly) implements all of functionality specific to the Evolving Tuples Model itself. Tuples, tuple spaces, and the services that move tuples through a node's deployment are defined here. The choreography required to create data structures and services is also provided by this library.

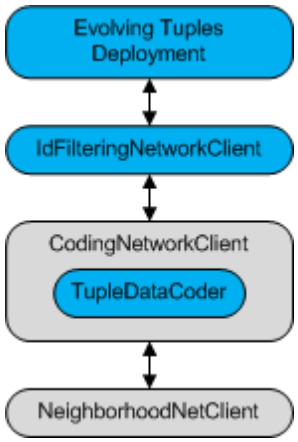

Figure 3.9: Evolving Tuples components added to the core communications stack

The current implementations for the Receive, Director, and Send processes delegate scheduling to the virtual machine through the use of threads. In retrospect, an implementation providing its own process scheduling may have provided more predictable and more testable behaviors.

The broadcast duplicate elimination mechanism described in Section 3.3.3 is implemented in this library through the use of a communication stack layer as shown in Figure 3.9. The `IdFilteringNetworkClient` retains a cache of recently sent or received tuple ids and is able to drop any duplicate messages before they are propagated to the core evolving tuples deployment.

3.6.1.4 Implementation Results

To validate the Evolving Tuples implementation, it was deployed on a network of SunSPOT devices. Several small networks were used for testing and debugging of the code, however for the discussion below we will focus on a spe-

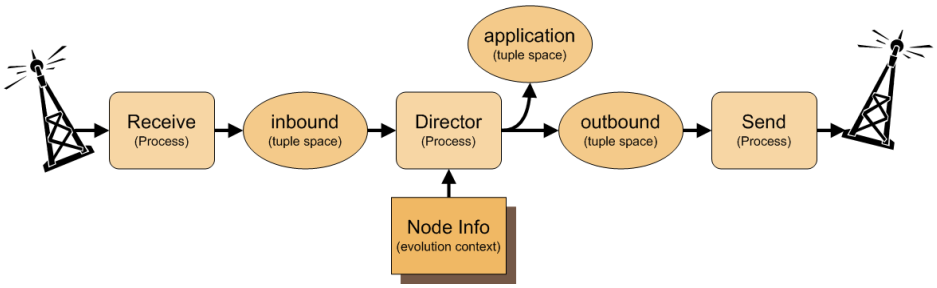

Figure 3.10: Flow chart of standard deployment model (Same as Figure 3.1)

cific arrangement of 10 nodes. In this deployment, the `NeighborhoodNetClient` component discussed above was used to impose a simple topography; each node N_i has two neighbors, N_{i-1} and N_{i+1}. The resulting network is an extended version of the example network depicted in Figure 3.4. However, since the SunSPOTs use real radios unlike the diagram, the nodes all contend with each other for access to the wireless medium.

To validate the implementation, the route discovery tuples given in Section 3.5 were sent from node N_0 to find a route to N_9. As expected, the tuples discovered routes and aggregated data properly. The majority of tuples were returned to the source within 20s with no discernible difference in performance between the versions of the tuples. We believe that this performance on real hardware is within acceptable limits for prototyping the classes of applications described in this dissertation.

3.6.2 Large Network Simulation

In the previous section we have described our evolving tuples implementation for the SunSPOT platform. While this implementation is limited by the

functionality provided by the platform, we took care during development to limit the use of SunSPOT-specific features. Through careful design, the resulting libraries are compatible with the desktop-oriented *Java Standard Edition* (JavaSE). Of particular note, the in-memory network abstractions created to support unit testing (See Section 3.6.1.1) work seamlessly in JavaSE. With the addition of a few support classes, the implementation described in Section 3.6.1 runs unmodified on the desktop. This design not only reduces the complexity of the overall project, but adds significant validity to the simulation results. The primary implication is the expectation that a large deployment of SunSPOT-based evolving tuples nodes would indeed show the same qualities observed on the desktop.

For the large network simulations below, we used `MemoryNetwork` and the other networking components shown in Figures 3.8 and 3.9. The `MemoryNetwork` component uses shared memory to communicate and models concurrent clients with Java threads. This environment does not use elaborate models for wireless channel loss or node mobility. The statically placed nodes have perfect communication to all nodes within the source's circular transmission range.

Additionally, the order of transmission and reception events relies on the virtual machine's servicing of the hundreds of threads in the simulation. This limitation means that it is impossible to re-execute specific event orderings and may even produce otherwise impossible ordering of messages in the network. A different type of simulation (e.g., discrete-events) may provide

more reproducible results but would also suffer a validity penalty since the code would require modification to run on the hardware.

3.6.2.1 Simulation configuration

The large network simulations that we have used below are designed to test the feasibility of the Evolving Tuples Model, and as such do not include the potentially complex interactions of multiple clients. Instead the focus is on measuring high-level trends with which an application prototypist is most likely concerned. In these scenarios, the programmer is less interested in metrics easily optimized in a specific implementation, and more interested in the protocol's intrinsic metrics. For example, we will discuss the number of *messages* sent instead of the number of *bytes*. Carefully designed message formats can greatly reduce the size of each message, but cannot eliminate the need for the message's transmission.

In fact, the values of the metrics themselves can be misleading and we instead focus on the trends that we see in these metrics as a function of network size and connectivity. In our trend analysis below we will vary the number of nodes and the transmission range of those nodes providing the two independent variables N and R respectively. Node positions are assigned randomly in the playing field and given in a unitless length in the range $[0.0, 1.0]$. All nodes in an experiment share the same circular transmission range, and receive any message perfectly when sent by a node within that range. To enable the testing of context-value driven protocols, each node x is also assigned a

unitless temperature T_x in the range $[0, 100]$. Since many context values, and temperature in particular, are observed as gradients in the physical world, T_x is assigned as a linear function of the node's distance from the origin (<0,0>). More realistic and complex assignments of context values is an area of future study.

Experiment 0

Before we continue to the Evolving Tuples experiments, we must first understand the effects of the independent variables N and R. The most fundamental result of changing these parameters is the number of nodes with which each node can directly communicate. In Figure 3.11 we see that the number of neighbors is approximately linear with the number of nodes and the transmission radius. Here $R(0.1)$ represents a transmission radius of 0.1, or a tenth of the length of the square playing field. In all of this section's graphs, 90% confidence intervals are plotted in addition to the mean values.

However, for the route and resource discovery algorithms presented above, we are less interested in the number of neighbors than we are in the likelihood that the node or resource we are looking for is *reachable* from the source node. Figure 3.12 plots the probability that the network is connected, that is all nodes can communicate with each other via multi-hop messaging. For large transmission ranges (0.4), networks are reliably connected when the playing field is populated with at least 40 nodes. For small transmission ranges (0.1), more then 600 nodes are required. Furthermore, Figure 3.13 shows that

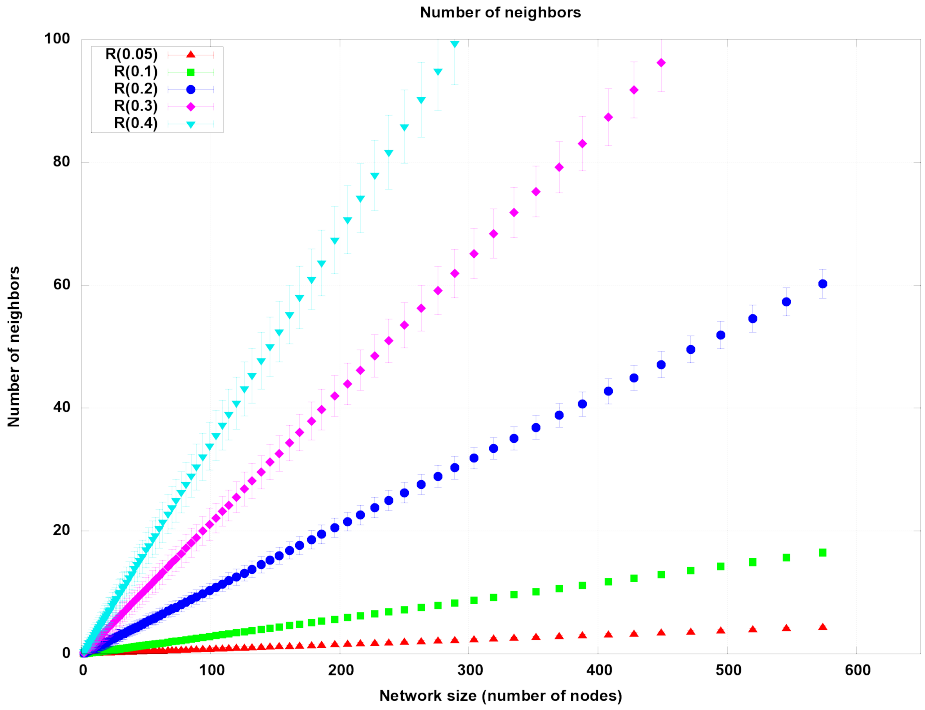

Figure 3.11: Number of neighbors for size and transmission radius

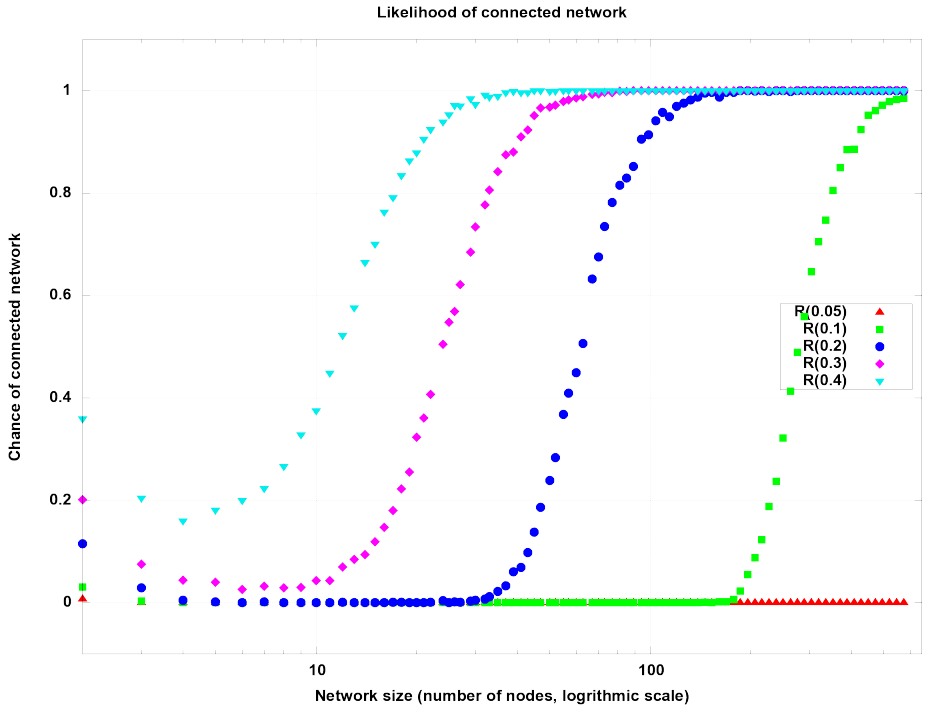

Figure 3.12: Likelihood of connected network for size and transmission radius

networks without sufficient nodes to be connected form a large number of independent network partitions. For our purposes, the $R(0.2)$ data series is particularly interesting. While networks of more then 100 nodes are usually connected, there is a distinct range of networks (10-30 nodes) with a large number of network partitions ($\sim$8). By focusing on this data series we can study both connected and highly partitioned networks within a reasonable range of network sizes.

In addition to the static transmission radius of 0.2, we also examine networks with an adaptive transmission radius. Nodes in these networks all

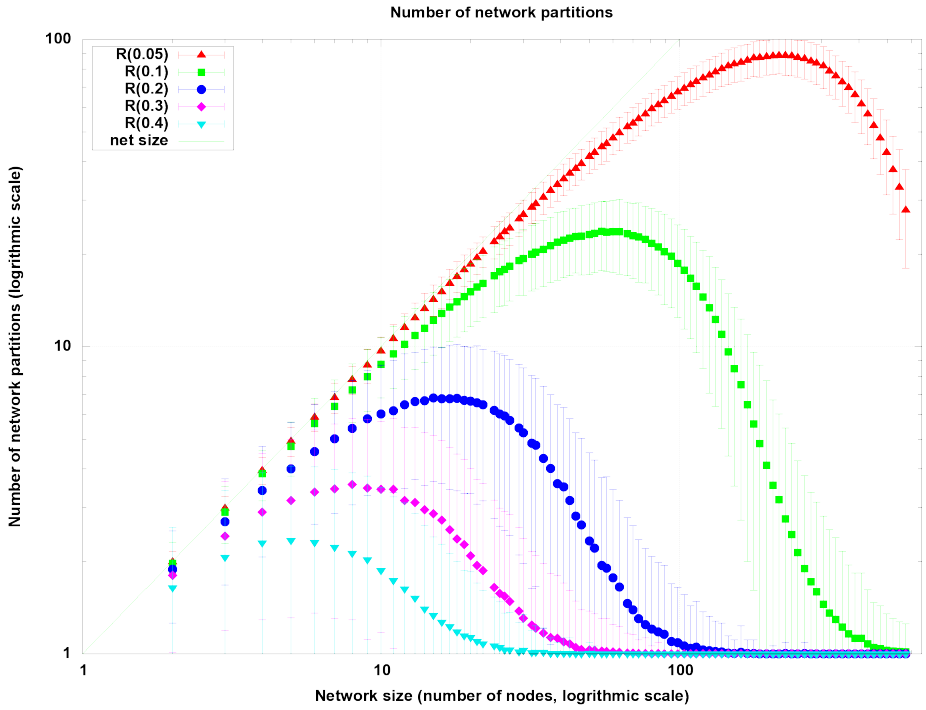

Figure 3.13: Number of network partitions for size and transmission radius

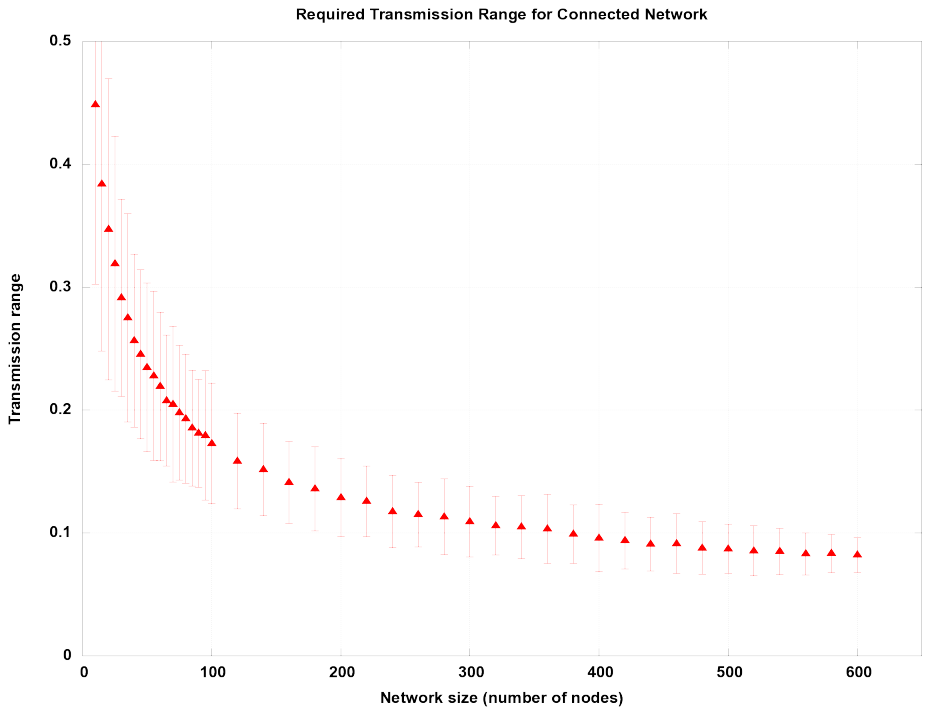

Figure 3.14: Required Transmission Range for Connected Network

share the same transmission radius, denoted $R(min)$, which is selected to be just large enough to guarantee the network is connected. Figure 3.14 shows the selected transmission ranges for various network sizes. By using these networks, we can elicit effects that are due simply to the rise in the number of nodes in the network.

3.6.2.2 Simulation of Route Discovery

In this section we present the trend analysis of high-level metrics to give an example of the type of information a developer would receive about his prototype implementation. to demonstrate this we use the route discovery

Version	Request	Reply	Target	Feature
Version 1	Flood	Flood	Node 2	
Version 2	Flood	Unicast	Node 2	
Version 3	Flood	Unicast	$Temp > 65$	
Version 4	Flood	Unicast	$Temp > 65$	Collects data en-route
Version 5	Flood	Unicast	$Temp > 65$	Flood limited to $Temp > 35$

Table 3.12: Summary of Route Discovery Tuple Versions

protocols described in Section 3.5. Each of the tuples derived in this section (summarized in Table 3.12) was injected into and used to find resources in networks of varying sizes. We will limit the results presented here to networks using the $R(min)$ and $R(0.2)$ transmission ranges which elicit interesting behaviors without the verbosity of all the collected data.

Experiment 1

In our first experiment, we examine the number of broadcast messages transmitted by example tuples. For this experiment we will simply chose two nodes in the network at random as the source and target of the route discovery. In connected networks we expect that any message that is broadcast will reach, and be re-transmitted by, every node in the network. More specifically, in Version 1 of our discovery tuple we expect to see $2 * N$ messages representing the broadcast of the request followed by the broadcast of the reply. In Version 2, we expect that the N broadcast messages used for the reply will be replaced by a multi-hop unicast message from the destination to the source. Figure 3.15 shows these trends for the $R(0.2)$ case, and Figure 3.16 shows these trends for

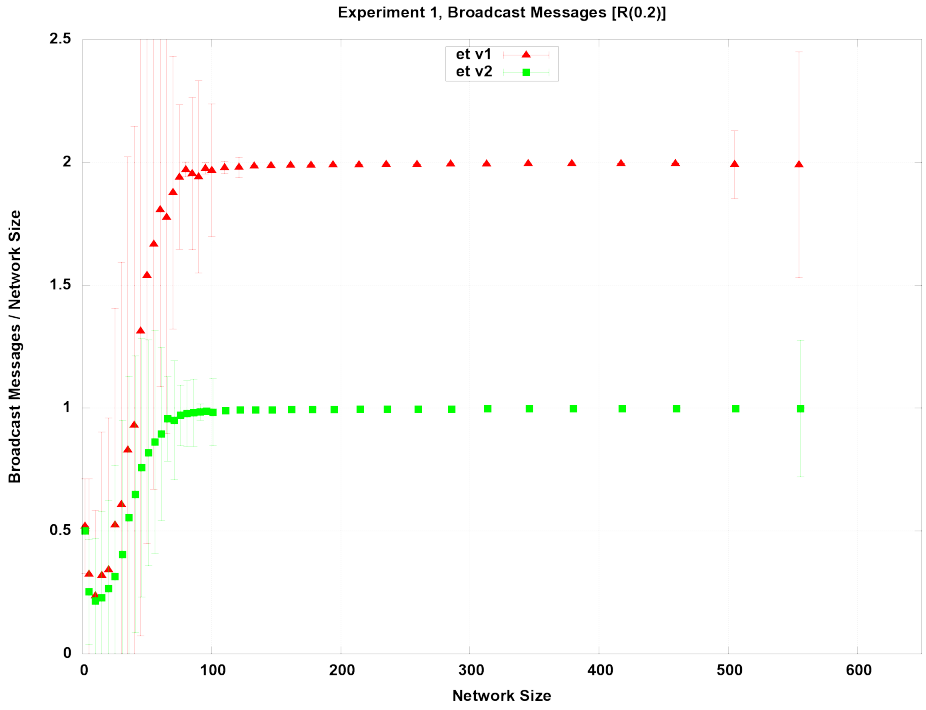

Figure 3.15: Experiment 1 - Broadcast Messages for Tx Range = $R(0.2)$

the $R(min)$ case. In both figures we can see a trend towards the expected values as the network size grows.

The unexpectedly large confidence intervals for et-v1 data points above ~500 nodes in Figure 3.15 are interesting outliers. The root cause is the load that large networks place on the simulator. This load causes a significant delay in the processing and distribution of messages. This can delay the route reply messages sufficiently that the simulated clients abandon routes as unreachable. Once abandoned, the simulation framework immediately tallies the sent messages and continues on to the next iteration. Since the message tallies are

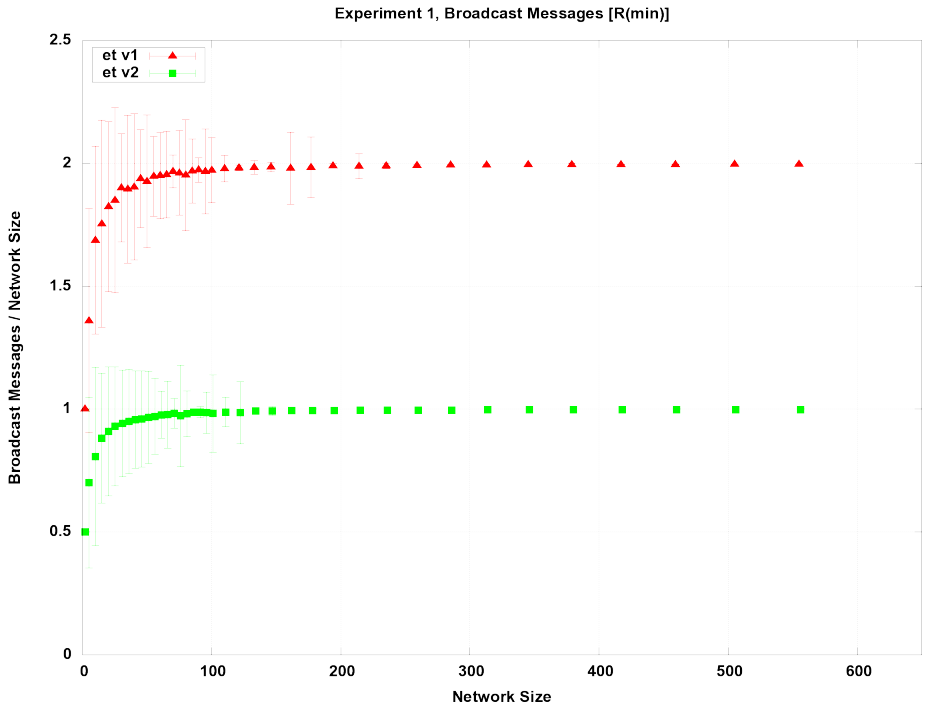

Figure 3.16: Experiment 1 - Broadcast Messages for Tx Range = $R(min)$

taken before the full route request/reply cycle has completed, it effectively *undercounts* the number of messages that *would have been sent* given more time. The undercount disproportionately effects the `v1` tuple since the uncounted messages are those sent later in the simulation during the reply phase.

The presence of this effect lends credibility to the simulations since the same delay would be encountered in a real deployment. However, the symptom (lower then expected message count) would be realized in a real deployment as a decreased number of returned routes. This is exactly the effect that we see in the simulation results shown in Figure 3.17; in networks larger then 500 nodes, the number of returned routes drops off.

Since the route reply messages in Version 2 traverse the discovered route using unicast messages, we expect the number of these messages to be equal to the length of the discovered routes.[13] Figures 3.18 and 3.19 show this trend. Here we also see the effect of message delay in the rising ratio for the `v2` protocol at $\tilde{5}00$ nodes in Figure 3.18. By giving up early, the client causes the simulator to count the only messages sent before the route reply can be sent back.

Experiment 2

In our second experiment, we examine how using context values instead of unique addresses effects the number of messages. In Versions 3, 4, and 5

[13]Since our tuples' routes contain the target address as well, we stipulate the route length to be one less then the number of nodes in the route.

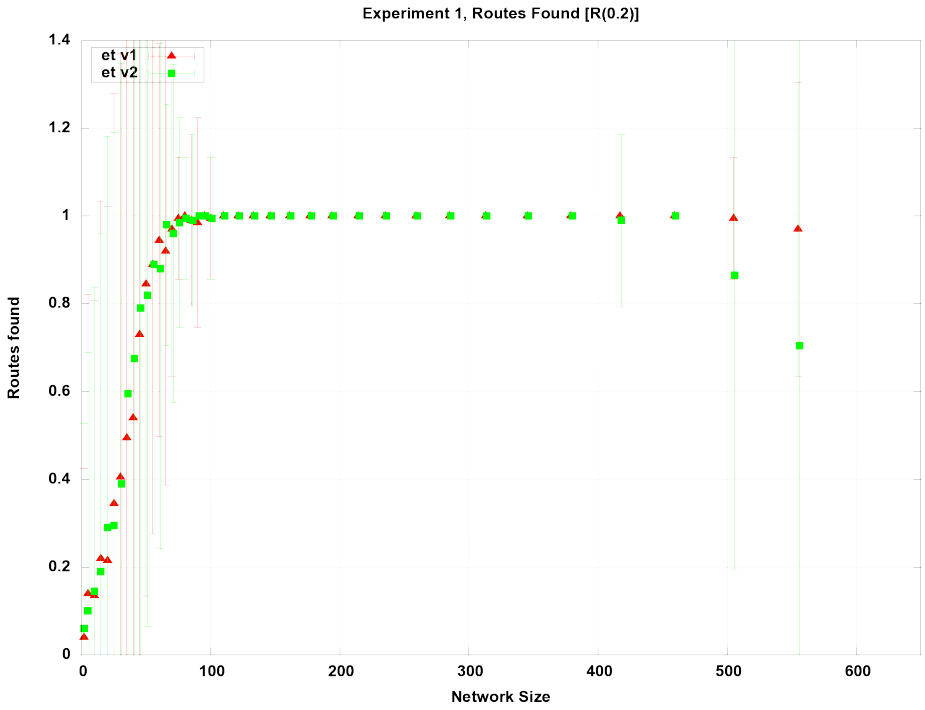

Figure 3.17: Experiment 1 - Routes found for Tx Range = $R(0.2)$

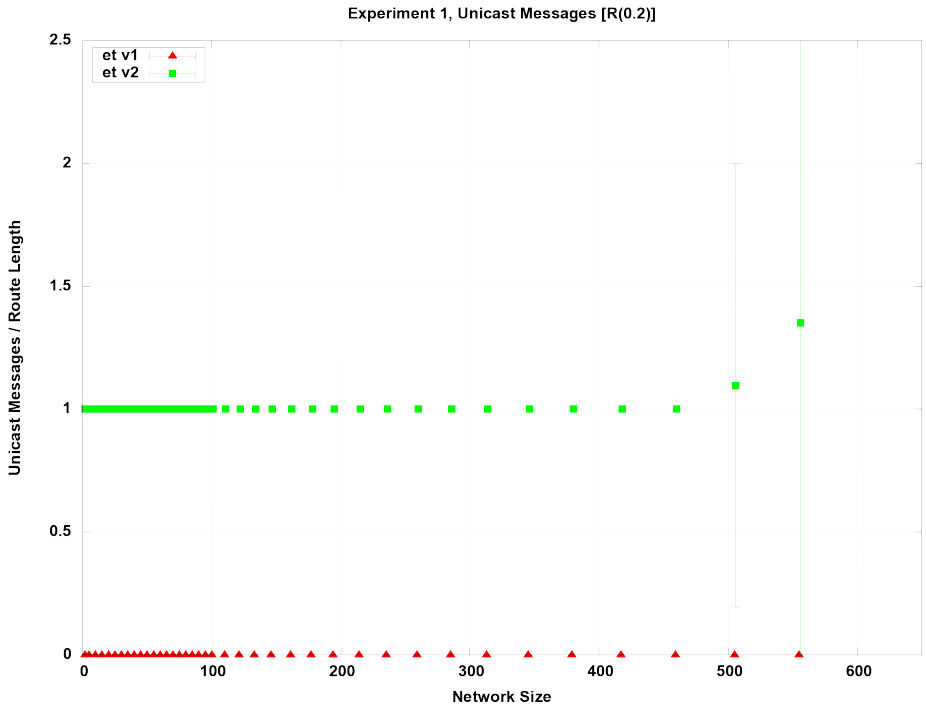

Figure 3.18: Experiment 1 - Unicast Messages for Tx Range = $R(0.2)$

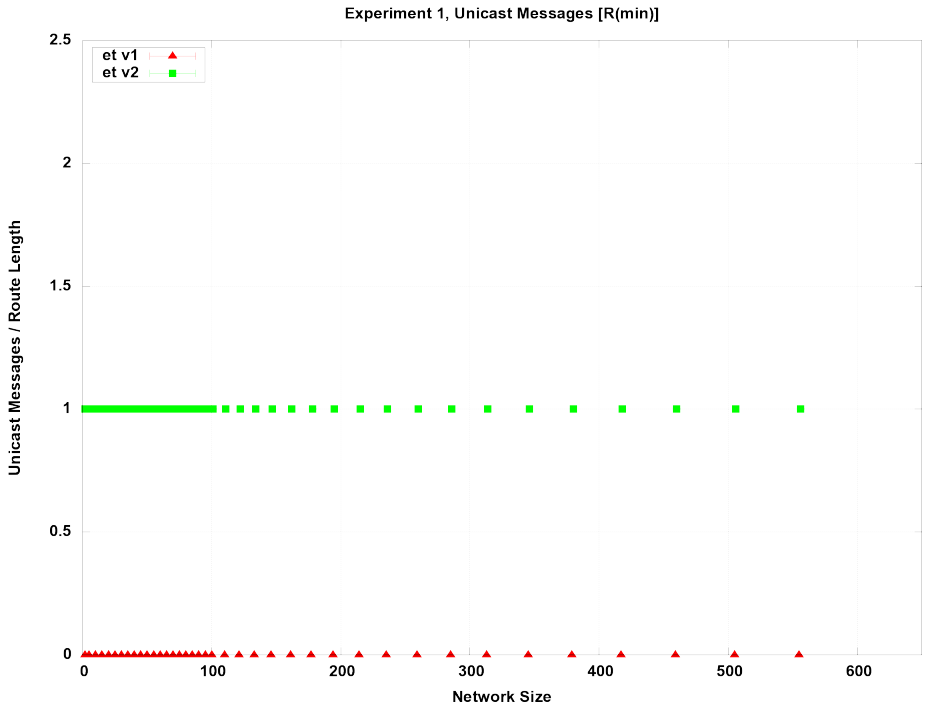

Figure 3.19: Experiment 1 - Unicast Messages for Tx Range = $R(min)$

of our discovery protocol, we expect that there will be more than one node replying to route requests sent from a randomly selected node. Since temperatures are assigned as a linear function of distance from the origin[14], our route discovery broadcast should reach many nodes where $T > 65$ that will immediately reply to the source. Since these nodes do not forward on the original request to neighbors, many of the nodes that are beyond the transmission radius of nodes where $T < 65$ will never receive a discovery message. This results in a drop in the number of broadcast messages as the replying nodes and the nodes that do not recieve the original broadcast never broadcast a route request message. This effect is even more dramatic in Version 5 of the protocol where nodes with $T < 35$ also limit the propagation of the discovery message. In Figure 3.20 and Figure 3.21, we see that Versions 3 and 4 of the protocol result in almost 50% drop in the number of broadcast messages, and Version 5 sees a 60% drop in networks with both connectivity ranges.

In Figure 3.22 we see another dramatic effect of using context-based route discovery. If we plot the same values as shown in Figures 3.18 and 3.19, we see a dramatic rise in the number of unicast messages in the network. This is due to the multiple routes that are now being returned to the source node. Indeed, the number of unicast messages sent when using context values to identify the target nodes is approximately equal to the average route length *multiplied by* the number of routes found.

[14]See Section 3.6.2.1

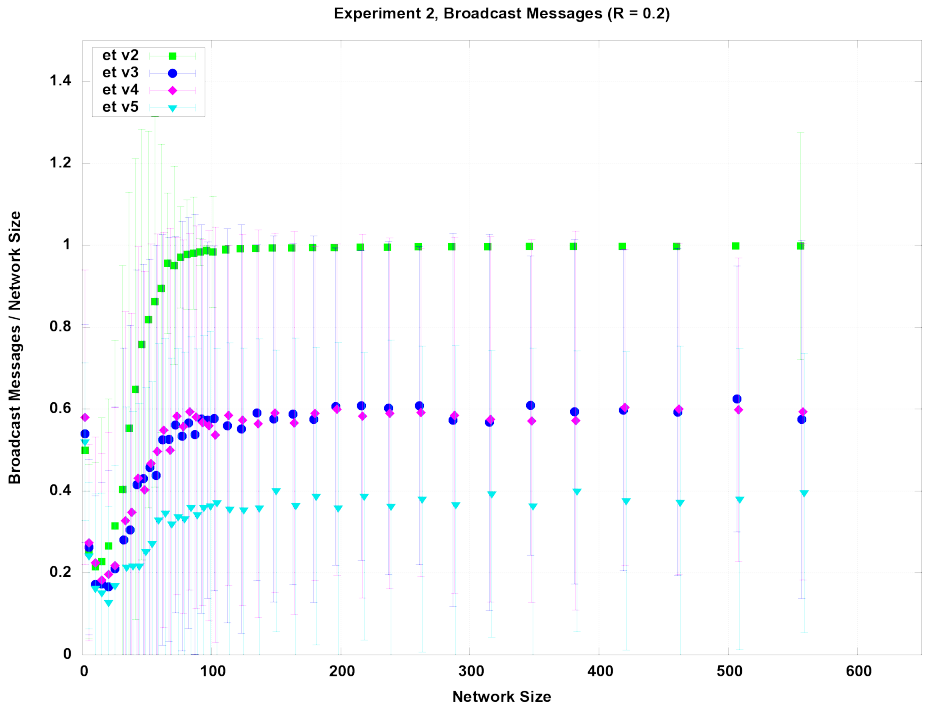

Figure 3.20: Experiment 2 - Broadcast Messages for Tx Range = $R(0.2)$

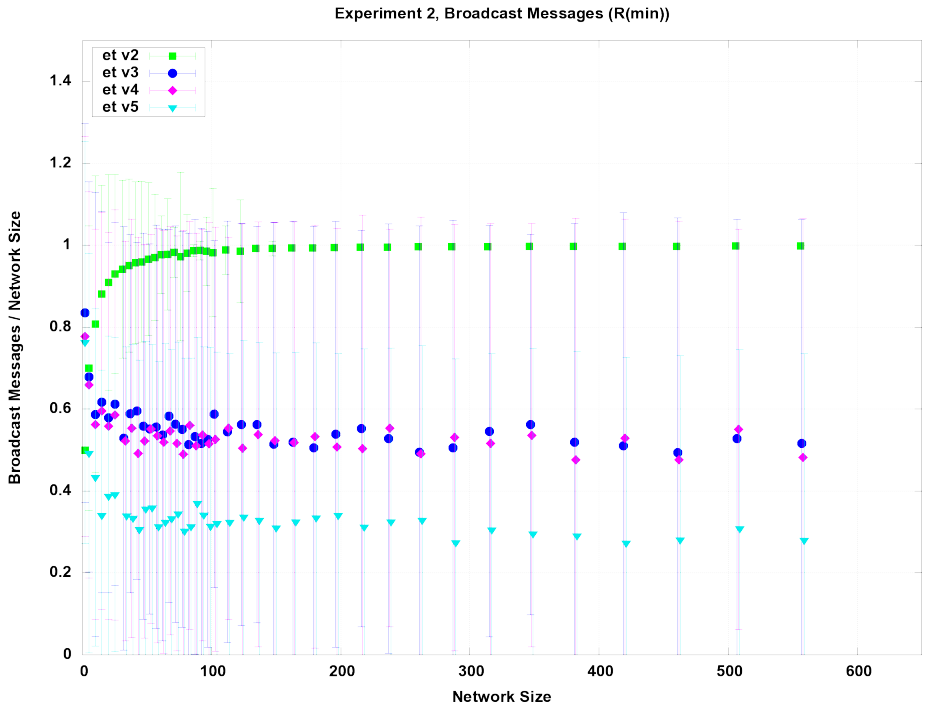

Figure 3.21: Experiment 2 - Broadcast Messages for Tx Range = $R(min)$

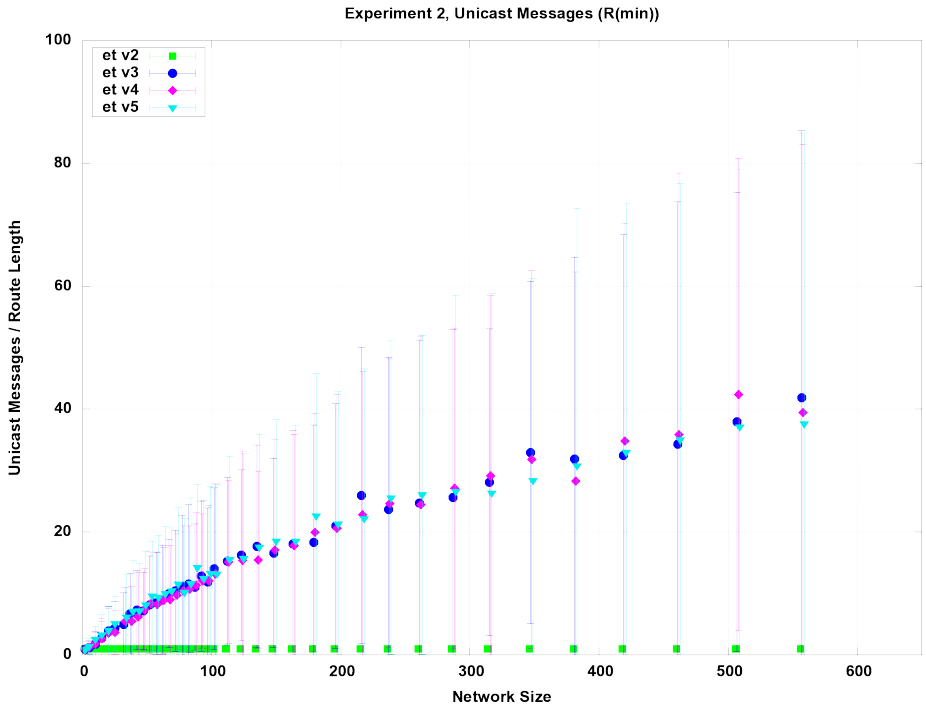

Figure 3.22: Experiment 2 - Unicast Messages for Tx Range = $R(min)$

3.7 Proposed Model Extensions

Over the course of developing the evolving tuples model, a number of possible extensions have been proposed without being fully investigated. In this section, we briefly discuss some of these extensions.

3.7.1 Single Field Routing

The current specification of the evolving tuples model uses two fields to control the propagation of the tuple from one node to the next, `destination` and `tupleId`. The primary purpose of the `tupleId` field is to support the dissemination of broadcasted tuples via a simple flooding based protocol. However, when flooding is being used, we know the destination field will be the broadcast address (typically -1). This hints that perhaps the inclusion of both the broadcast address and the tuple's id is redundant.

As an alternative, it has been proposed that the `destination` field could be replaced by a choice of one of `unicastAddress`, `multicastAddress`, and `broadcastId`. Using an intention-based name for the address of the tuple's intended target could provide advantages to both comprehension and implementation. A user can more readily understand the method of transmission along with the destination by looking at the field's name, and the `Send` process could be divided into 3 different processes, each concerned with a different method of delivery.

This alternative has some disadvantages though; most notably that the method of dissemination can not be changed after the tuple is created. Since the current model does not allow a field's name to be updated, a tuple that initially used broadcast addresses will not be able to "switch" to unicast or multicast addresses. It is our belief that this restriction is sufficiently disruptive to be unattractive to users of the model. However, if the model changes in the future in such a way that field names can be updated, this proposal may be reevaluated.

3.7.2 Single-Hop or Multi-Hop?

The evolving tuples model is intended for use in networks that may or may not support multi-hop routing internally. Since the model is specifically designed to allow users to collect and aggregate data from within a pervasive computing environment, the use of multi-hop routing may be detrimental to some applications. For example, these applications may wish to simply count the number of routing hops that a tuple transverses, or may with to control the details of routing themselves.

To support these applications, an extension to the evolving tuples model must be derived. The addition of another well-known field name (e.g., `allowMultiHopRouting`) could be an appropriate solution. However we feel that the model should be evaluated in its current incarnation before making this change.

3.8 Summary

In this chapter we have presented the Evolving Tuples Model and a prototype implementation. We then examined their characteristics as the model and prototype were applied to resource discovery and route discovery in pervasive computing networks. Working with this model has exposed several other avenues of potential research. In the next chapter we will discuss a few of these opportunities.

Chapter 4

Conclusions

This dissertation presents two models for supporting application development in pervasive computing environments. Current techniques for interacting with remote resources are often connection-oriented, lacking support for the very long-term conversations demanded by applications in pervasive computing environments. To provide this support, we have formulated the Application Sessions Model. Using this model, applications can declaratively specify target resources in the environment and thus delegate the details of setup and maintenance of short-term connections to the underlying system. This results in an intuitive interface for development that is adaptable to the variety of technologies and techniques available in pervasive computing environments.

To realize the full potential of the Application Sessions Model, new technologies for resource selection are required. The Evolving Tuples Model presented in this dissertation was derived to provide flexible and effective resource discovery. By embedding small amounts of behavior in network messages, values from across the network can be collected, combined, and evaluated *in situ*. The model's inherent flexibility also makes it suitable to other distributed

computing applications as it can be used to rapidly prototype coordination activities without requiring the deployment of new hardware or software to remote nodes. As an example of this, we also present how the model can be applied to context-sensitive route discovery.

The work described by this dissertation has the following impacts on the software engineering research community:

1. The Application Sessions Model is defined and lends formal semantics to the notion of conversations between peers in a pervasive computing environment. The model's formal components can be used by application developers to reason about the behavior of their applications and serves to precisely document the requirements of any implementation of the model.

2. The Application Sessions Middleware Design and Prototype serve as a practical realization of the Application Sessions Model. They are used to perform our feasibility study and to enable the development of additional pervasive computing applications.

3. The Application Sessions Feasibility Study demonstrates the use and usefulness of the Application Sessions Model and Middleware through case study applications from the Intelligent Construction Site domain.

4. The Evolving Tuples Model defines a method for calculating application-specific abstract values by embedding behavior in pervasive computing

network messages. This model can be used to reason about network messages and it documents, the expected behaviors for developers of devices participating in the network.

5. The Evolving Tuples Model Prototype forms the foundation for our feasibility study. It also serves as an example implementation and a testing environment. Furthermore, it lays groundwork for development and study of coordination protocols and pervasive computing applications under development.

6. The Evolving Tuples Model Feasibility Study describes the characteristics observed in the prototype implementation on sensor hardware and in software simulation. The study shows the Evolving Tuples Model is indeed a feasible solution to resource discovery. The study has also shown that the model can be successfully used for other purposes, specifically for route discovery.

As expected, both models have proven to be promising solutions for their problem domains. The Evolving Tuples Model has also proven to be an effective tool for related problem domains as well. With further research, these models may form the basis for practical solutions to the challenges presented by the Intelligent Construction Site and similar pervasive computing environments.

To guide this research, we believe the next step is to conduct *empirical studies* of software developers as they use the Application Sessions Model and Evolving Tuples Model. Measuring the interactions between developers and the model and measuring the reduction in application complexity will likely form the foundation of the study. As software developers are ultimately the consumers of the model, feedback from actual usage will be valuable in identifying potential weaknesses and motivating corrections to the models.

To truly study the value of the Application Sessions Model for field deployments we will require *integration with additional discovery protocols*. These integrations will also enable us to perform a more quantitative comparison of the Evolving Tuples Model's contribution to resource predicate evaluations. Likewise, implementations of the *Evolving Tuples Model for additional hardware* will also allow for quantitative comparisons of its applicability in heterogeneous environments. These Evolving Tuples implementations will also facilitate the study of Application Sessions Models in heterogeneous environments.

The promise that the Evolving Tuples Model has shown outside of resource discovery is also quite compelling. Some initial work has begun to *integrate the model with ongoing research* in the Mobile and Pervasive Computing Group [60]. The SEAP architectural pattern [38] under development is designed to allowing novice developers to create collaborative applications. This work has also shown long-term promise for enabling non-programmers to define impromptu collaborations amongst devices, the ultimate goal of much

of the research in our field.

The incorporation of sensing and controlling devices into our surroundings is an exciting and stimulating trend. To realize the potential value these environments offer, we must provide application developers the tools to manage coordination and collaboration of the devices. Providing easy, intuitive, and reliable mechanisms for interacting with them is a very real challenge, and one that can we have demonstrated can be undertaken through the proper application of software engineering.

Bibliography

[1] Serge Abiteboul. Querying semi-structured data. In *Proceedings of the 6th International Conference on Database Theory (ICDT '97)*, pages 1–18, London, UK, January 1997. Springer-Verlag.

[2] William Adjie-Winoto, Elliot Schwartz, Hari Balakrishnan, and Jeremy Lilley. The design and implementation of an intentional naming system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP '99)*, pages 186–201, New York, NY, USA, Dec 1999. ACM.

[3] Sanem Kabadayıand Christine Julien. A local data abstraction and communication paradigm for pervasive computing. In *Proceedings of the 5th Annual IEEE International Conference on Pervasive Computing and Communications (PerCom '07)*, 2007.

[4] Greg R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming.* Addison Wesley, 1999.

[5] R.J.R. Back and K. Sere. Stepwise refinement of parallel algorithms. *Science of Computer Programming*, 13(2-3):133–180, May 1990.

[6] R. Bagrodia, S. Bhattacharyya, F. Cheng, S. Gerding, G. Glazer, R. Guy, Z. Ji, J. Lin, T. Phan, E. Skow, M. Varshney, and G. Zorpas. iMASH:

Interactive mobile application session handoff. In *Proceedings of the 1st International Conference on Mobile Systems, Applications, and Services (MobiSys '03)*, pages 259–272, New York, NY, USA, May 2003. ACM Press.

[7] Magdalena Balazinska, Hari Balakrishnan, and David Karger. Ins/twine: A scalable peer-to-peer architecture for intentional resource discovery. In *Proceedings of the 1st International Conference on Pervasive Computing (Pervasive 2002)*, pages 149–153, Zrich, Switzerland, August 26-28 2002.

[8] Francisco J. Ballesteros, Enrique Soriano, Gorka Guardiola, and Katia Leal. Plan B: Using files instead of middleware abstractions. *IEEE Pervasive Computing*, 6(3):58–65, July-September 2007.

[9] Davide Balzarotti, Paolo Costa, and Gian Pietro Picco. The LighTS tuple space framework and its customization for context-aware applications. *International Journal on Web Intelligence and Agent Systems (WAIS)*, 5:2, 2007.

[10] P. Bellavista, A. Corradi, R. Montanari, and C. Stefanelli. Dynamic binding in mobile applications: A middleware approach. *IEEE Internet Computing*, 7(2):34–42, 2003.

[11] Jeffrey M. Bradshaw, editor. *Software Agents*. MIT Press, Cambridge, MA, USA, 1997.

[12] Frederick P. Brooks, Jr. No silver bulletessence and accidents of software engineering. In H.J. Kugler, editor, *Information Processing 86*, pages 1069–1076. Elsevier Science Publishers B.V. (North Holland), 1986.

[13] Paul Butcher. A behavioural semantics for Linda-2. *Software Engineering Journal*, 6(4):196–204, 1991.

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

[15] Nicholas Carriero and David Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, 1989.

[16] R. Chand and P.A. Felber. A scalable protocol for content-based routing in overlay networks. In *Proceedings of the 2nd IEEE International Symposium on Network Computing and Applications*, pages 123–130, April 2003.

[17] Guanling Chen and David Kotz. Context-sensitive resource discovery. In *Proceedings of the 1st IEEE International Conference on Pervasive Computing and Communications (PERCOM '03)*, page 243, Washington, DC, USA, 2003. IEEE Computer Society.

[18] Keith Cheverst, Nigel Davies, Keith Mitchell, and Adrian Friday. Experiences of developing and deploying a context-aware tourist guide: The

GUIDE project. In *Proceedings of the 6th Annual International Conference on Mobile Computing and Networking (MobiCom '00)*, pages 20–31, New York, NY, USA, 2000. ACM Press.

[19] J.-H. Choi and C. Yoo. CTP-aware source routing in mobile ad hoc networks. In *Proceedings of the 8th International Symposium on Computers and Communication*, pages 69–74, June-July 2003.

[20] Alan Cole, Sastry Duri, Jonathan Munson, Jay Murdock, and David Wood. Adaptive service binding middleware to support mobility. In *Proceedings of the 23rd International Conference on Distributed Computing Workshops (ICDCSW '03)*, pages 369–374, Washington, DC, USA, May 2003. IEEE Computer Society.

[21] Gianpaolo Cugola, Carlo Ghezzi, Gian Pietro Picco, and Giovanni Vigna. Analyzing mobile code languages. In J. Vitek and C. Tschudin, editors, *Mobile Object Systems: Towards the Programmable Internet*, volume 1222 of *Lecture Notes on Computer Science*, pages 93–111. Springer, April 1997.

[22] Matthias Dyer, Jan Beutel, Thomas Kalt, Patrice Oehen, Lothar Thiele, Kevin Martin, and Philipp Blum. Deployment support network - a toolkit for the development of WSNs. In *Procedings of the 4th European Conference on Wireless Sensor Networks (EWSN 2007)*, Lecture Notes in Computer Science, pages 195–211. Springer Berlin / Heidelberg, January 2007.

[23] Wolfgang Emmerich. Software engineering and middleware: A roadmap. In *Proceedings of the Conference on The Future of Software Engineering (ICSE '00)*, pages 117–129, New York, NY, USA, 2000. ACM.

[24] P. Engelstad, D.V. Thanh, and T.E. Jonvik. Name resolution in mobile ad hoc networks. In *Proc. of the 10th Int'l. Conf. on Telecommun.*, 2003.

[25] Paal Engelstad, Yan Zheng, Tore Jonvik, and Do Van Thanh. Service discovery and name resolution architectures for on-demand MANETs. In *Proceedings of the 23rd International Conference on Distributed Computing Systems Workshops (ICDCSW'03)*, pages 736–742, Los Alamitos, CA, USA, May 19-22 2003. IEEE Computer Society.

[26] Fiatech. Element 4: Intelligent & automated construction job site. `http://fiatech.org/tech-roadmap/roadmap-elements/element4.html`, 05 June 2009.

[27] Chien-Liang Fok, Gruia-Catalin Roman, and Gregory 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, February 2004.

[28] Chien-Liang Fok, Gruia-Catalin Roman, and Chenyang Lu. Rapid development and flexible deployment of adaptive wireless sensor network

applications. In *Proceedings of the 25th International Conference on Distributed Computing Systems (ICDCS'05)*, pages 653–662. IEEE, June 2005.

[29] Nathan Funk and Singular Systems. Jep-java math expression parser, version 2.4.1. `http://sourceforge.net/projects/jep/`, 1 Aug 2009.

[30] David Gelernter and Arthur J. Bernstein. Distributed communication via global buffer. In *Proceedings of the 1st ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 10–18, New York, NY, USA, 1982. ACM Press.

[31] Robert Grimm, Janet Davis, Eric Lemar, Adam Macbeth, Steven Swanson, Thomas Anderson, Brian Bershad, Gaetano Borriello, Steven Gribble, and David Wetherall. System support for pervasive applications. *ACM Transactions on Computer Systems*, 22(4):421–486, 2004.

[32] Erik Guttman. Service location protocol: Automatic discovery of IP network services. *IEEE Internet Computing*, 3(4):71–80, July-Aug. 1999.

[33] Joachim Hammer, Imran Hassan, Christine Julien, Sanem Kabadayı, William J. O'Brien, and Jason Trujillo. Dynamic decision support in direct-access sensor networks: A demonstration. In *Proceedings of the 3rd International Conference on Mobile Ad-hoc and Sensor Systems*, 2006.

[34] Radu Handorean, Jamie Payton, Christine Julien, and Gruia-Catalin Roman. Coordination middleware supporting rapid deployment of ad hoc

mobile systems. In *Proceedings of the 1st International Workshop on Mobile Computing Middleware, co-located with ICDCS 2003*, pages 362–368, May 2003.

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

[36] J. Hellerstein, W. Hong, S. Madden, and K. Stanek. Beyond average: Toward sophisticated sensing with queries. In *Procedings of the 2nd International Workshop on Information Processing in Sensor Networks*, pages 553–568, April 2003.

[37] Ophir Holder, Israel Ben-Shaul, and Hovav Gazit. Dynamic layout of distributed applications in FarGo. In *Proceedings of the 21st International Conference on Software Engineering (ICSE '99)*, pages 163–173, Los Alamitos, CA, USA, May 1999. IEEE Computer Society Press.

[38] Seth Holloway, Drew Stovall, Jorge Lara-Garduno, and Christine Julien. Opening pervasive computing to the masses using the seap middleware. In *Proceedings of Middleware Support for Pervasive Computing Workshop (PerWare '09 at PerCom '09)*, number TR-UTEDGE-2008-015, Galveston, Texas, 9–13 March 2009.

[39] Yih-Chun Hu, Adrian Perrig, and David B. Johnson. Ariadne: A secure

on-demand routing protocol for ad-hoc networks. *Wireless Networks*, 11(1–2):21–38, January 2005.

[40] IEEE. Wireless medium access control and physical layer specifications for low-rate wireless personal area networks. *IEEE Standard 802.15.4-2003*, pages 1–670, 2003.

[41] Chalermek Intanagonwiwat, Ramesh Govindan, and Deborah Estrin. Directed Diffusion: a scalable and robust communication paradigm for sensor networks. In *Proceedings of the* 6^{th} *annual international conference on Mobile computing and networking (MobiCom '00)*, pages 56–67, New York, NY, USA, 2000. ACM.

[42] Edward J. Jaselskis and Tarek Elmisalami. RFID's role in a fully integrated, automated project process. In *Proceedings of ASCE Construction Research Congress 7*, Honolulu, Hawaii, March 19–21 2003.

[43] David B. Johnson and David A. Maltz. Dynamic source routing in ad hoc wireless networks. In Imielinski and Korth, editors, *Mobile Computing*, volume 353, pages 158–163. Kluwer Academic Publishers, 8-9 December 1996.

[44] Christine Julien, Joachim Hammer, and William J. O'Brien. A dynamic programming framework for pervasive computing environments. In *Proceedings of the Workshop on Building Software for Pervasive Computing (co-located with OOPSLA 2005)*, October 2005.

[45] Christine Julien and Gruia-Catalin Roman. Egocentric context-aware programming in ad hoc mobile environments. In *Proceedings of the 10th International Symposium on the Foundations of Software Engineering*, pages 21–30, November 2002.

[46] Christine Julien and Gruia-Catalin Roman. Egospaces: Facilitating rapid development of context-aware mobile applications. *IEEE Transactions on Software Engineering*, 32(5):281–298, May 2006.

[47] Christine Julien and Meenakshi Venkataraman. Cross-layer discovery and routing in reconfigurable wireless networks. In *Proceedings of the 3rd International Conference on Mobile Ad-hoc and Sensor Systems*, 2006.

[48] Sanem Kabadayı, Christine Julien, William J. O'Brien, and Drew Stovall. Virtual sensors: A demonstration. In *The 26th International Conference on Computer Communications (INFOCOM): Demonstrations Track*, 2007.

[49] Cory D. Kidd, Robert Orr, Gregory D. Abowd, Christopher G. Atkeson, Irfan A. Essa, Blair MacIntyre, Elizabeth D. Mynatt, Thad Starner, and Wendy Newstetter. The aware home: A living laboratory for ubiquitous computing research. In *Proceedings of the 2^{n}d International Workshop on Cooperative Buildings, Integrating Information, Organization, and Architecture (CoBuild '99)*, pages 191–198, London, UK, 1999. Springer-Verlag.

[50] Michael Klein and Birgitta Konig-Ries. Combining query and preference: An approach to fully automize dynamic service binding. In *Proceedings of the IEEE International Conference on Web Services*, pages 788–791, San Diego, CA, USA, July 6–9 2004.

[51] David Kotz, Calvin Newport, and Chip Elliott. The mistaken axioms of wireless-network research. Technical Report TR2003-467, Department of Computer Science, Dartmouth College, July 2003.

[52] David Kotz, Calvin Newport, Robert S. Gray, Jason Liu, Yougu Yuan, and Chip Elliott. Experimental evaluation of wireless simulation assumptions. In *Proceedings of the ACM/IEEE International Symposium on Modeling, Analysis and Simulation of Wireless and Mobile Systems (MSWiM)*, number TR2004-507, pages 78–82. ACM Press, October 2004.

[53] Ulas C. Kozat and Leandros Tassiulas. Service discovery in mobile ad hoc networks: an overall perspective on architectural choices and network layer support issues. *Ad Hoc Networks*, 2(1):23–44, 2004.

[54] Li Li and Louise Lamont. A lightweight service discovery mechanism for mobile ad hoc pervasive environment using cross-layer design. In *Pervasive Computing and Communications Workshops, 2005. PerCom 2005 Workshops. Third IEEE International Conference on*, pages 55–59, March 2005.

[55] S. Madden, M.J. Franklin, J.M. Hellerstein, and W. Hong. TAG: A tiny aggregation service for ad-hoc sensor networks. In *Proceedings of the*

5th Symposium on Operating Systems Design and Implementation, pages 131–146, December 2002.

[56] David Malan, Thaddeus Fulford-Jones, Matt Welsh, and Steve Moulton. CodeBlue: An ad hoc sensor network infrastructure for emergency medical care. In *Proceedings of the International Workshop on Wearable and Implanted Body Sensor Networks*, April 2004.

[57] Marco Mamei and Franco Zambonelli. Self-maintained distributed tuples for field-based coordination in dynamic networks. In *Proceedings of the 19th Symposium on Applied Computing (SAC '04)*, pages 479–486, New York, NY, USA, 2004. ACM Press.

[58] Marco Mamei, Franco Zambonelli, and Letizia Leonardi. Tuples on the air: A middleware for context-aware computing in dynamic networks. In *Proceedings of the 23rd International Conference on Distributed Computing Systems Workshops*, pages 342–347, 19-22 May 2003.

[59] Cecilia Mascolo, Licia Capra, and Wolfgang Emmerich. *Middleware for Communications*. John Wiley, 2004.

[60] Mobile and Pervasive Computing Group. Mobile and pervasive computing group. `http://mpc.ece.utexas.edu/`, Aug 2009. Electrical and Computer Engineering, The University of Texas at Austin.

[61] Amy L. Murphy, Gian Pietro Picco, and Gruia-Catalin Roman. LIME: A middleware for physical and logical mobility. In *Proceedings of the*

21st International Conference on Distributed Computing Systems, pages 524–533, 2001.

[62] Amy L. Murphy, Gian Pietro Picco, and Gruia-Catalin Roman. LIME: A coordination middleware supporting mobility of hosts and agents. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 15(3):279–328, July 2006.

[63] R. K. Panta, I. Khalil, and S. Bagchi. Stream: Low overhead wireless reprogramming for sensor networks. In I. Khalil, editor, *Proceedings of the 26th IEEE International Conference on Computer Communications (INFOCOM 2007)*, pages 928–936, 2007.

[64] C.E. Perkins and P. Bhagwat. Highly dynamic destination-sequenced distance vector routing (DSDV) for mobile computers. In *Proceedings of the ACM SIGCOMM Conference on Communications Architecture, Protocols and Applications*, pages 234–244, August 1994.

[65] Charles E. Perkins and Elizabeth M. Royer. Ad hoc on-demand distance vector routing. In *Proceedings of the 2nd IEEE Workshop on Mobile Computer Systems and Applications*, pages 90–100, Los Alamitos, CA, USA, 1999. IEEE Computer Society.

[66] Gruia-Catalin Roman, Christine Julien, and Qingfeng Huang. Network abstractions for context-aware mobile computing. In *Proceedings of the 24th International Conference on Software Engineering (ICSE '02)*, pages 363–373, May 2002.

[67] Gruia-Catalin Roman, Christine Julien, and Amy Murphy. A declarative approach to agent-centered context-aware computing in ad hoc wireless environments. In A. Garcia, C. Lucena, F. Zambonelli, A. Omicini, and J. Castro, editors, *Proceedings of the 1st International Workshop on Software Engineering for Large-Scale Multi-Agent Systems*, May 2002.

[68] Gruia-Catalin Roman, Gian Pietro Picco, and Amy L. Murphy. Software engineering for mobility: A roadmap. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000)*, pages 241–258, New York, NY, USA, 2000. ACM Press.

[69] Caspar Ryan and Christopher Westhorpe. Application adaptation through transparent and portable object mobility in Java. In *Proceedings of OTM Federated Conferences*, pages 1262–1284. Springer Berlin / Heidelberg, 2004.

[70] Umar Saif and Justin Mazzola Paluska. Service-oriented network sockets. In *Proceedings of the 1st international conference on Mobile systems, applications and services (MobiSys '03)*, pages 159–172, New York, NY, USA, May 2003. ACM Press.

[71] Daniel Salber, Anind K. Dey, and Gregory D. Abowd. The context toolkit: Aiding the development of context-enabled applications. In *Proceedings of the SIGCHI conference on Human factors in computing systems (CHI '99)*, pages 434–441, New York, NY, USA, 1999. ACM.

[72] N. Shrivastava, C. Buragohain, D. Agrawal, and S. Suri. Medians and beyond: New aggregation techniques for sensor networks. In *Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 239–249, 2004.

[73] Doug Simon, Cristina Cifuentes, Dave Cleal, John Daniels, and Derek White. Java on the bare metal of wireless sensor devices. In *Proceedings of the 2nd International Conference on Virtual Execution Environments*, June 2006.

[74] Randall B. Smith, Cristina Cifuentes, and Doug Simon. Enabling java for small wireless devices with Squawk and SpotWorld. In *Proceedings of the Workshop on Building Software for Pervasive Computing (co-located with OOPSLA 2005)*, 2005.

[75] Drew Stovall. Application sessions website. `http://mpc.ece.utexas.edu/application-sessions`, 1 Aug 2009.

[76] Drew Stovall. Evolving tuples website. `http://mpc.ece.utexas.edu/evolving-tuples`, 1 Aug 2009.

[77] David L. Tennenhouse, Jonathan M. Smith, W. David Sincoskie, David J. Wetherall, and Gary J. Minden. A survey of active network research. *IEEE Communications Magazine*, 35(1):80–86, January 1997.

[78] C.N. Ververidis and G.C. Polyzos. Routing layer support for service discovery in mobile ad hoc networks. In *Proceedings of the 3rd IEEE*

International Conference on Pervasive Computing and Communications Workshops, 2005. (PerCom 2005 Workshops)., pages 258–262, March 2005.

[79] Torben Weis, Mirko Knoll, Andreas Ulbrich, Gero Muhl, and Alexander Brandle. Rapid prototyping for pervasive applications. *IEEE Pervasive Computing*, 6(2):76–84, April–June 2007.

[80] David Wetherall. Active network vision and reality: lessons from a capsule-based system. In *Proceedings of DARPA Active NEtworks Conference and Exposition*, volume 33, pages 25–40, San Francisco, CA, USA, May 2002.

[81] Ning Xu, Sumit Rangwala, Krishna Kant Chintalapudi, Deepak Ganesan, Alan Broad, Ramesh Govindan, and Deborah Estrin. A wireless sensor network for structural monitoring. In *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems (SenSys '04)*, pages 13–24, New York, NY, USA, November 2004. ACM Press.

[82] H. Zhou and S. Singh. Content based multicast (CBM) in ad hoc networks. In *Proceedings of the 1st Workshop on Mobile Ad Hoc Networking and Computing (Mobihoc at MobiCom 2000)*, pages 51–60, 2000.

[83] Peng Zhou, Tamer Nadeem, Porlin Kang, Cristian Borcea, and Liviu Iftode. EZCab: A cab booking application using short-range wireless

communication. In *Proceedings of the International Conference on Pervasive Computing and Communications*, pages 27–38, Los Alamitos, CA, USA, 8–12 March 2005. IEEE Computer Society.

[84] Gil Zussman and Adrian Segall. Energy efficient routing in ad hoc disaster recovery networks. In *Proceedings of the 22nd Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM '03)*, pages 682–691, March–April 2003.

Vita

Andrew Erich Stovall was born in Los Alamos, New Mexico on 17 June 1977, the son of James E. Stovall and Mary K. Stovall. He received the Bachelor of Science degree in Physics from Carnegie-Mellon University in 1999. He entered the private sector and worked for several software development companies including Trilogy, WhisperWire, and eBay. While still working, he earned his Masters of Science in Engineering in the field of Software Engineering from the University of Texas at Austin. In 2007, he left the private sector and enrolled as a full-time student at the University of Texas at Austin to pursue a Doctorate in Software Engineering.

Permanent address: 6522 Laird Drive
Austin, Texas 78757

This dissertation was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.