A Declarative Approach to Agent-Centered Context-Aware Computing in Ad Hoc Wireless Environments

Gruia-Catalin Roman[1], Christine Julien[1], and Amy L. Murphy[2]

[1] Department of Computer Science and Engineering
Washington University
Saint Louis, MO 63130
{roman, julien}@cse.wustl.edu
[2] Department of Computer Science
University of Rochester
Rochester, NY 14627
murphy@cs.rochester.edu

Abstract. Much of the current work on context-aware computing relies on information directly available to an application via context sensors on its local host, e.g., user profile, host location, time of day, resource availability, and quality of service measurements. We propose a new notion of context which includes in principle any information available in the ad hoc network infrastructure but is restricted in practice to specific views of the overall context. The context of each view is defined in terms of data, objects, or events exhibiting certain properties, associated with particular application agents, residing on particular hosts, and part of some restricted subnet. Location, distance, movement profiles, access rights, and much more can be used in view specifications. The underlying system infrastructure interprets the view specifications and continuously updates the contents of user-defined views despite dynamic changes in the specifications, state transitions at the application level, mobility of hosts in the physical space, and migration of code among hosts. In systems that are large-scale in terms of both space and numbers of agents, this local restriction will prove necessary for providing timely context information to application agents.

1 Introduction

The foundation of this work is the notion that context-aware computing holds the key to achieving rapid development of dependable mobile applications in ad hoc networks. *Context-aware computing* refers to the explicit ability of a software system to detect and respond to changes in its environment, e.g., a drop in the quality of service on a video transmission, a low battery level, or the sudden availability of much needed access to the Internet. Initial context-aware systems like Olivetti's Active Badge [1] and Xerox PARC's PARCTab [2] focused on user location to provide context-aware information in an office environment, while

more recent systems use location information for context-aware tour guides [3, 4]. Gradually, other aspects of context have been fed into applications, including time, weather, and user information, allowing, for example, researchers in the field to attach varied contextual information to their notes with FieldNotes [5]. With the increase in the variety and complexity of context information, much needed frameworks and systems for generalizing its treatment are being developed. The Context Toolkit [6] generalizes interaction among components through context widgets, while the Context Fabric [7] provides a service infrastructure. By and large, these systems limit the context to what a component can immediately sense, ignoring what other networked components can sense. While this need has been hinted at in discussions of context-aware software [8], no widespread system allows such access. When the needs of the application must reach beyond the basics (e.g., the application requires access to services available at a remote location), the programmer needs to contend with more complex processes that include discovery and communication. While these extra costs may be acceptable in wired networks where connections persist over extended periods of time, in ad hoc networks the complexity of managing frequent disconnections can significantly increase the programming effort. Yet, mobile systems do need access to a broad range of resources, maybe even more so than traditional distributed applications.

Of interest to us is the ease with which resources can be acquired and retained in the presence of mobility. Our specific environment consists of logically mobile agents that operate over a network of physically mobile hosts. These mobile agents coordinate with each other to accomplish their individual application needs. In many scenarios this network may include many agents and span a large physical space. Our work extends the notion of declarative specifications to a broad set of resources and provides the mechanisms needed to maintain access to the specified resources despite rapid changes in the environment caused by the mobility of hosts, migration of software components, and changes in connectivity. For instance, an application on a palmtop should be able to declare its need for printer access and, as the owner travels along, a printer should always appear on the desktop, as long as some printer exists within wireless communication range. Of course, building such an application with today's technology is feasible, but coding it cannot be reduced to the simple act of providing a declaration in the program. We contend that we can accomplish the latter (and more) by extending the notion of context-aware computing and by developing a software infrastructure that continuously secures the resources declared by the application program. In building these specialized contexts, we also recognize that different applications interact with available information in different ways. For these reasons, we introduce four distinct context-aware models that provide unique styles of interaction.

The remainder of this paper is organized as follows. Section 2 discusses the nature of declarative specifications. Section 3 provides details about four different models of context-awareness. Finally, conclusions appear in section 4.

2 Declarative Specification of Views

In our computing model, hosts can move in physical space, and the applications they support are structured as a community of software components called agents that can migrate from one host to another. Thus, an agent is the unit of modularity, execution, and mobility, while a host is a container characterized, among other things, by its location in physical space. Communication among agents and agent migration can take place whenever the hosts involved can physically communicate with each other, i.e., they are connected. Since the notion of context is always a relative one, we will use the term *reference agent* to denote the agent whose context we are about to consider, and we will refer to the host on which this agent is located as the *reference host*. An agent's location is always a host, while a host's location is always a point in some physical or logical space.

2.1 Informal View Definition.

A mobile ad hoc network is an opportunistically formed structure that changes rapidly in response to the movement of the mobile hosts involved. Initially, communication in such networks was point to point over a physical broadcast medium, the air waves. However, growth in performance and capabilities has allowed some mobile units to serve as mobile routers for others in the area. Through transitivity, routing in ad hoc networks has expanded the connectivity pattern beyond the limits of an immediately accessible region. In principle, the context associated with a given agent consists of all the information available in the ad hoc network. This includes all information stored by all hosts in the network as well as the context information (e.g., location, temperature, time) sensed by agents on those hosts. We refer to this as the *maximal context* of the reference agent. Of course, such broad access to information is generally costly to implement. In addition, various parts of the same application may need different resources at different times during the execution of the program. For this reason, we believe that it is important to structure the context in terms of fine-grained units which we call views. A *view* is a projection of the maximal context together with an interpretation that defines the rules of engagement between the agent and the particular view.

The concept of view is agent centric in the sense that every view is defined relative to a reference agent with respect to its needs for resources from and knowledge about its environment. An agent sees the world through a set of these individualized views. The set may be altered at will by defining, redefining, and deleting views as processing requirements demand. The software engineering gains derive, to a great extent, from the flexibility and simplicity we can offer the application programmer. Our strategy focuses on declarative specifications and employs a rich set of criteria. For instance, one ought to be able to describe the view contents in terms of phrases such as:

> *All specials (reference to objects) posted by family restaurants (reference to agents) within one mile (implicit reference to hosts) of my current location (property of the reference host).*

In general, constraints on the attributes of the desired resources (data or objects) and the agents that own them are an effective way to restrict a view's contents. They must be combined, however, with constraints on the attributes of the hosts on which the agents reside and with properties of the ad hoc network in the immediate vicinity. Security and network considerations emerge as important research issues in any effort to design a language for view specification. At the network level, for instance, an application may want to limit context to a connected subnet of the ad hoc network forming a region around the reference host. The network topology, geometry, physical distribution in space, and security enforcement procedures play a role in determining the shape of the region of interest. These considerations are new to context-aware computing and are injected by our focus on ad hoc mobility.

The next section provides a more formal treatment of this declarative view specification. The notation is only illustrative and assumes the underlying data representation to be that of a tuple space. Tuple space representations based on the Linda tuple space model [9] enjoy a great deal of popularity due to the content-based manner in which data is accessed. In mobile computing specifically, several systems have found success using shared tuple spaces. MARS [10] focuses on logical mobility, or the movement of agents over physically stationary hosts, using a tuple space to allow coordination among co-located mobile agents. LIME [11] combines support for logical mobility with support for physical mobility and relies on transient sharing of tuple spaces among agents and hosts within communication range. Each agent carries its own tuple spaces, and tuple spaces of connected agents logically merge to form a global tuple space as long as the agents are connected. This work reuses this notion of transient sharing of tuple spaces, combines it with a more flexible tuple representation, and allows more general access to the tuple space.

2.2 Formal View Definition.

We assume a tuple to be an unordered set of fields, each with a unique name. An individual agent owns tuples which it keeps in a local tuple space. Fundamentally, tuple access occurs by matching a provided pattern against the contents of the tuple. While adhering to the content-based nature of Linda pattern matching, we extend the traditional semantics to allow the provision of more flexible constraint functions over fields. The matching function, $\mathcal{M}$, described in detail in [12] requires that, for every constraint provided in a pattern, a field that satisfies the constraint exists in the tuple. While the matching function does require that each constraint be satisfied, it does not require that there be a constraint to match every field in the tuple.

In our model, the data, the agents owning the data, the hosts where the agents are located, and the paths to those hosts must all satisfy application-provided constraints. An agent can provide the view's data constraints through a pattern. The matching function, $\mathcal{M}$ is then used to filter the data tuples using this pattern. Hosts and agents in the system provide *profiles* containing personal information. Host profiles handle logical properties of the host, and may

relate to the user of the computer. Examples of such properties may include the host's id, the identity of the owner of the computer, or services provided by the computer. Agent profiles, on the other hand, are likely to contain properties related to the application on whose behalf the agent is running. The view's host and agent constraints then reduce to a pattern of constraints over a profile, and these constraints can also be evaluated using the matching function, $\mathcal{M}$. An agent provides network constraints by forming an abstraction of the network topology and its properties. This abstraction, detailed in [13], generates a subnet of the network around the specifying host. For example, the application can restrict its context with respect to the physical distance to other hosts. The subnet constructed includes hosts only within the application-specified distance; all other hosts are excluded.

In any shared data space, access control becomes a real problem. Our model addresses this issue by adding the notion of an access control function. Each agent specifies an individualized function that limits the ability of other agents to access its local data. From the opposite direction, an agent specifying a view attaches to the view a set of credentials that verify it to the other agents. Additionally, the specifying agent declares the operations it intends to perform over the view. These operations can include simple reading or removal of data or more complex operations such as reacting to the appearance of a particular piece of data. The provision of the operations can be viewed as a contract between the specifying agent and the system. Any attempt by the specifying agent to perform an operation that it didn't declare will result in an exception. When determining the contents of a view, the system evaluates each contributing agent's access control function over the view's credentials and potential operations. The fact that the access control function is evaluated on an individual basis for each tuple adhering to the view constraints provides a very fine level of granularity.

Figure 1 shows our computational model. A host, the outer rectangle in the figure has a physical location and a profile describing its properties. Each host contains mobile agents, the smaller rectangles in the figure. Each agent also stores its properties in a profile and has a logical location, the host on which it is located. Additionally, agents can define views which consist of the view specification, described in detail below, and the credentials provided for that view. For evaluating the view specifications of other agents, each agent also defines an access control function. Finally, every agent owns a local tuple space that contains its data items.

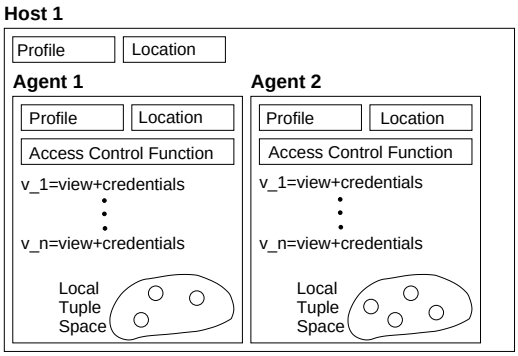

Fig. 1. The computational model.

Given a reference host r, we first define η, the subnet of the ad hoc network that satisfies the provided network constraints (n), to be a subset of the closure of r's network. η must be a tree, r must belong to η, and η must satisfy n. Given the host constraints (h), the agent constraints (a), the data constraints (d), the agent's credentials (κ), and the operations that will be performed on the view (ops), a view specified by a reference agent, r contains the tuples defined by:

$$
\begin{aligned}
&\mathsf{view}_{\mathsf{r}}(n, h, a, d, \kappa, ops) \triangleq \\
&\quad \langle \mathsf{set}\ \eta, \gamma, \alpha, \theta : \eta \subseteq \mathit{Closure}(\mathsf{r}) \wedge \mathit{tree}(\eta) \wedge \mathsf{r} \in \eta \wedge \eta\ \mathtt{sat}\ n \\
&\qquad\qquad \wedge \gamma \in \eta \wedge \mathcal{M}(\gamma.\mathsf{profile}, h) \wedge \alpha.\mathsf{loc} = \gamma \wedge \mathcal{M}(\alpha.\mathsf{profile}, a) \\
&\qquad\qquad \wedge \theta \in \alpha.T \wedge \mathcal{M}(\theta, d) \wedge \alpha.acf(\kappa, ops, \theta) \\
&\qquad\qquad :: \theta \rangle.\ ^{3}
\end{aligned}
$$

where γ is a host, α is an agent, and θ is a tuple. $\alpha.\mathsf{loc}$ refers to the host on which agent α is currently running, $\alpha.T$ refers to α's local tuple space, and $\alpha.acf$ to α's access control function. A tuple belongs to a view only if it satisfies the view constraints and the reference agent meets the requirements of the access control function of the agent owning the tuple.

As hosts and agents move and the available data changes, the view is updated to reflect the changing set of available tuples. From the application's perspective, all of these changes are transparent and manifest only in changes to the set of available data items. Therefore, the application agent can operate over a view without regard to the changes occurring in that view. The application also has the freedom to change the constraints associated with its view dynamically, and, when it does, the model adjusts the view to reflect the application's new needs.

The adoption of a declarative context specification is motivated by our belief that transparent context management will shift to the underlying middleware many of the burdens programmers face in the development of applications for use in ad hoc networks. Moreover, the programmer controls the scope of the view (a large or small neighborhood), the size of the view (the range of entities included), and the relative cost of executing a particular operation on that view (by defining the level of consistency, e.g., best effort versus transactional semantics). The presentation of the information in this view to the programmer can take a more abstract form than a simple tuple space. A variety of data structures in addition to a tuple space will prove useful to applications in different domains. Additionally, more sophisticated context-sensitive interactions can be provided through veneers that build on the basic model. The next section details some examples of such models in our system.

[3] The three-part notation $\langle \mathbf{op}\ \mathit{quantified\_variable} : \mathit{range} :: \mathit{expression} \rangle$ used here is defined as follows: variables from *quantified_variables* take on all possible values permitted by *range*. Each such instantiation of the variables is substituted in *expression*, producing a multiset of values to which **op** is applied, yielding the value of the expression. If no instantiation of the variables satisfies *range*, the value of the expression is the identity element for **op**, e.g., *true* when **op** is $\forall$; zero if **op** is "+".

3 Models of Context-Awareness

Because we see ad hoc mobility as a fundamental challenge to developing the next generation of consumer, industrial, and military applications, we seek to develop new models of context-awareness able to accommodate the complexities of mobile computing, to build middleware that embodies these models, and to evaluate both on interesting application test beds. This section offers a broad-brush discussion of the four types of context-awareness models we are currently developing. Our models reflect those popular in distributed computing, but we expect new technological advances to result from our special focus on their applicability to ad hoc networks, the introduction of declarative specifications of context, and automatic context maintenance.

To show how an agent's interaction with the view differs among the four models, we introduce an example that we will revisit throughout this section. Consider a team of robots exploring an uninhabited planet. The robots need to perform experiments that require precise relative locations and instrumentation that no single robot can carry. For example, some robots may be able to precisely sense their locations, some may be able to sense the ambient temperature, others may sense atmospheric pressure, and still others may collect data about the soil composition. All of these pieces of information have the potential to contribute to the operating context of any agent in the system. Now consider a specific reference agent that requires two pieces of location information from other robots (for determining relative locations) and a single piece of temperature information (for performing its particular experiment). To satisfy its needs, the agent defines two views. The first is defined to contain the location data items that are between some minimum and maximum distance from the agent's robot. The second view contains temperature data items within a specified number of network hops. The agent can dynamically adjust its view specifications as its needs change. The agent's style of interaction with these views depends upon the features of the context-awareness model in use by the system. As we describe our context-awareness models, we will revisit this example to elucidate how the agent interacts with its temperature view in each model.

3.1 Context-Sensitive Data Structures.

In many distributed systems, data access serves as the primary form of interaction among components. In mobile computing, several systems have used shared tuple spaces as a coordination medium. As discussed previously, MARS [10] employs a single tuple space per host to facilitate coordination among co-located mobile agents while LIME [11] relies on transient sharing of tuple spaces among agents on the same host and among hosts within communication range. Other systems have explored different data structures. PEERWARE [14], for example, stores documents in trees and adjusts the contents of the tree to account for mobility. All these systems assume a symmetric and transitive model of sharing. When a group of components is formed, they all share the same data, and they perceive it in the same manner. By contrast, our proposed model allows each

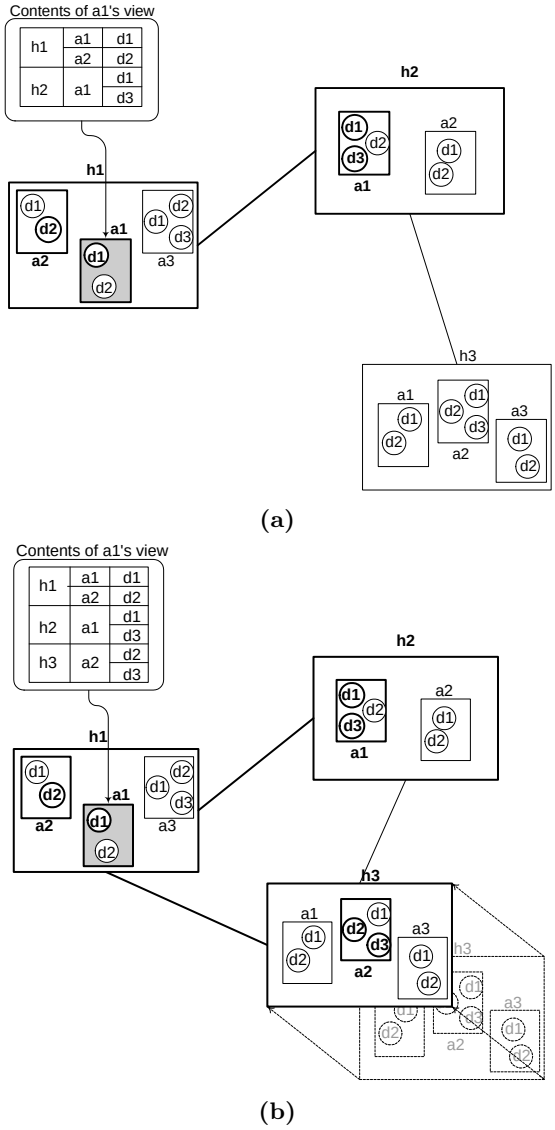

Fig. 2. View dynamics. Data items visible to reference agent `a1` located on host `h1` before and after `h3` moves into `h1`'s range. Hosts, agents, and data items with darkened borders contribute to the view, while ones with lighter borders do not satisfy the specification.

individual agent to define its own perspective of the data available in the world in terms of one or more views. This asymmetry, a distinguishing feature of our model, allows each agent to assume responsibility for and control over the size and scope of the data it accesses. For example, an agent associated with a managing robot that monitors the activities of other robots in its vicinity might define a view that includes the locations and activities of all other robots within a certain

distance, which may be continuously adjusted as the exploration progresses. One of the worker robots, however, may define a view containing only information it needs to accomplish its individual task; this view may have nothing to do with the monitoring agent.

In general, the agent's view contains a *representation* of a subset of the data available in the ad hoc network. The choice of representation is a defining feature of each specific instantiation of the general model. In the context-sensitive data structures model, the view's representation is a simple data structure (e.g., a tuple space). The three remaining models build on this foundation. The choice of data included in the view, i.e., its *contents*, is determined by the view specification. The latter is given in a declarative manner by stating constraints on the network, hosts, agents, and data that contribute to defining the view. One can impose restrictions on network properties (e.g., number of hops, distances, bandwidth, etc.) so as to define a connected subnet immediately surrounding the reference host. This kind of locality will help control the context maintenance costs while meeting the needs of most mobile applications. Within this contextual setting, an application can impose further restrictions on the properties of the physically mobile hosts (e.g., power availability, devices supported, etc.) in the subnet and of the mobile agents supported by the admissible hosts. Finally, data associated with the remaining eligible agents can be filtered to produce the actual contents for that view. As hosts and agents move and properties of the network components change over time, the contents of the view must be transparently updated for the reference agent.

The dynamic nature of the view definition is illustrated in Figure 2, where the depicted view of agent `a1` changes as the distance between hosts `h1` and `h3` decreases. Agent `a1` is grayed to indicate that it is the agent specifying the view. Hosts, agents, and data items that contribute to the view are shown with darkened borders. In part (a) of the figure, due to `a1`'s specification, only hosts `h1` and `h2` qualify to contribute agents to the view. Because of the restrictions on agent and data properties, only certain data items on certain agents on these hosts appear in the view. The balloon pointing to `a1` shows a table of the hosts, agents, and data items contributing to `a1`'s view. As part (b) shows, when host `h3` moves closer to `h1`, it satisfies the view's constraints. Again, only certain data items on certain agents appear in the view. Exactly which hosts, agents, and data items contribute is determined by the application-provided view specification.

In the context-sensitive data structures model, the view representation takes the form of a standard data structure. For the purposes of discussing our example, we assume this data structure is a tuple space with which the robot agent interacts by performing standard tuple space operations. Figure 3 shows this general pattern of interaction. This figure and all subsequent ones show a virtual picture of an agent's view where both remote and local tuples are included in a single "soup." The actual distribution of information in logical and physical space (as shown in Figure 2) is omitted. Tuple space operations, or requests, can include reading and removing data from the view. Additionally, the tuple space can provide reactive behaviors whereby a robot agent can react to the

appearance of new data items in the view. As discussed previously, tuples match operations or reactions through content-based pattern matching, i.e., an agent selects data by specifying constraints over the values of the tuples' fields. In the robot example, tuples from temperature sensors might contain fields including a unique id identifying the probe, the string temp indicating that the probe is a temperature probe, the value of the temperature at that probe, and other fields. The agent can provide constraints over all of the data item's fields or over a subset of them. A robot agent might gain an initial temperature reading by performing a read operation for a tuple corresponding to any probe (p), labeled as temperature data (by the string temp), with any temperature value (v):

$$\mathsf{read}(\langle probeId : \mathtt{p}, probeType = \mathsf{temp}, probeValue : \mathtt{v}\rangle)$$

This request constrains only the fields explicitly mentioned; it places no restrictions on other fields in the tuple. When the request completes, the probe id and value are stored in the local variables, p and v, respectively. (The reader is reminded that this notation and all similar notation is for illustration purposes only.) If the robot wants to receive later readings from the same temperature probe (p) that differ from the initial reading by more than 5 degrees, it might register a reaction:

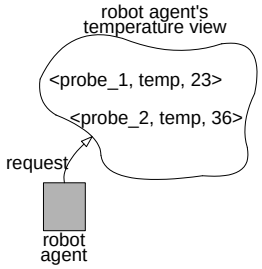

Fig. 3. Agent/view interaction in the context-sensitive data structures model.

$$\mathsf{react\ to}(\langle probeId = \mathtt{p}, probeType = \mathsf{temp}, probeValue : \mathtt{v}' :: (\mathtt{v} - 5) < \mathtt{v}' < (\mathtt{v} + 5)\rangle, \mathsf{A})$$

The action A will be performed whenever the temperature probe p outputs a new temperature reading that satisfies both the view specification and the value constraints provided in the react to operation.

3.2 Context-Sensitive References.

Traditional distributed systems, like CORBA-compliant systems [15] and Jini [16] hide many of the details of object distribution from the programmer. The general pattern of interaction requires a client to find an object using a lookup service and then bind to it, allowing the programmer to invoke methods on the remote object as if it were local. If the remote object fails, the client must revisit the lookup service to retrieve a new reference. This style of interaction, while common in traditional distributed systems has received only limited attention in ad hoc networks [17]. Our next model extends the context-sensitive data structures model so the view contains objects and object references instead of data items. An agent obtains an object reference and description from the view through a request similar to those used in the previous model. Because the object description contains information about the interface of the object, the application agent can use this information to interact with the remote object directly by invoking methods on the reference. The agent can continue to use the reference

but receives no guarantees regarding the stability of the remote object because the interaction occurs outside the view.

In using the context-sensitive references model in the robot environment, the temperature data is encapsulated in objects. Instead of reading data items directly from the view, the robot agent reads an object reference based on requirements it provides. The agent provides these requirements as a pattern that is matched (again, in a content-based fashion) against the object description stored in the tuple space. The reference returned indicates the remote object's location and information about how to interact with it (i.e., the object's interface). Figure 4 shows this style of interaction. A robot agent might request from the view a reference to a temperature object at a location (`loc`) within 2 meters of the agent's current location (`here`):

$$\mathsf{read}(\langle \mathit{objectReference} : \mathtt{r}, \mathit{probeType} = \mathsf{temp}, \mathit{location} : \mathtt{loc} :: |\mathtt{loc} - \mathtt{here}| < 2\mathsf{m} \rangle)$$

For a more complicated request, the agent could require that the object reference returned provide a particular method. Because the object reference is bound to `r` when the `read` operation returns, the robot agent can interact directly with the remote object by invoking methods on `r`. For example, a temperature object might have methods `getCelsius`() and `getFahrenheit`(), and the robot agent could call either method depending on its needs:

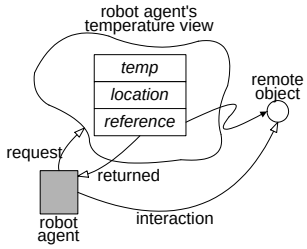

Fig. 4. Agent/view interaction in the context-sensitive references model.

```
r.getCelsius()
```

The agent can hold the reference as long as it desires, however, if the reference object disappears, an exception will be generated the next time the robot agent attempts to use the stale reference. In such a case, the agent must obtain a new reference from the view. Additionally, because both the robot agent and the agent holding the temperature object are mobile, the distance between them could grow to more than two meters without the robot agent's knowledge. The next model of context-awareness, while incurring additional overhead, helps the application programmer transparently cope with these deficiencies.

3.3 Context-Sensitive Bindings.

The need for load-balancing [18] and fault-tolerance [19] have been addressed in extensions to the CORBA specification. These additions accomplish their respective tasks by selecting from among a set of object replicas for each remote object call. In the case of load-balancing, consecutive remote method calls are not necessarily forwarded to the same object; instead calls are spread to multiple replicas. For fault-tolerance, consecutive calls can be forwarded to the same object instance until that object fails. In these cases, a different replica services later remote object calls. The DENO (Decentralized Network Objects) [20]

system also attempts to address these problems in the context of mobile and unreliable networks, adding object replication to increase efficiency, availability, and fault-tolerance. Our context-sensitive bindings model attempts to solve similar problems in the ad hoc environment. Instead of addressing the replication problem, however, our model concentrates on the binding aspect. The view abstraction allows our model to provide a more general and transparent solution.

In the mobile ad hoc environment, objects move, and bindings are even more likely to break. The middleware supporting the view concept transparently manages bindings, hiding both the lookup service and object mobility from the programmer. In general, the view contains a set of objects (and associated object descriptions) owned by connected agents. The set of available objects depends on the reference agent's view specification. However, the programmer does not access this set of objects directly. Instead he requests bindings to objects in the view, subject to certain policies. For example, if multiple objects available in the view match the binding request, the application might desire the nearest match. As agents and the objects associated with them move, the bindings are maintained and transparently updated to select new objects as needed. If an object matching the binding request in the view better satisfies the binding policy, the application's bound object is updated to reference the better match. Additionally, when bound objects move outside the view, a new object satisfying the binding request and located in the view replaces it. The change from one satisfying object to another is under the indirect control of the programmer through the binding policies he provides. Any effects of this rebinding are therefore the responsibility of the application itself.

As an example of a view, consider a reference agent responsible for printing documents. Its view might contain all printers available on the current floor in the current building. The agent might then request a binding to the highest quality printer. As the agent moves, the set of available printers changes, and therefore the binding automatically changes. This model may be added as a thin veneer over the context-sensitive references model. This veneer hides the view contents and services a binding request by locating an object in the view that matches the binding specification and policy and by creating the connection to it for the agent. The layer also responds to changes in the available set of objects in order to maintain, update, and break bindings when necessary.

Because the robot agent requires a single temperature reading, when using the context-sensitive bindings model, the agent requests a single binding to a temperature object. Because this model allows the agent to specify a binding policy which helps select the "best match" for the binding from among the objects in the view, the agent might request to bind to the temperature probe with the highest precision. Even though the object description might contain a wealth of information, the requesting agent can choose which fields of the description to provide constraints for. A binding request might look like:

$$\mathsf{bind}(\langle \mathit{objectReference} : \mathtt{r}, \mathit{probeType} = \mathsf{temp} \rangle)\ \mathsf{highest\_precision\_policy}$$

The agent interacts with the object by invoking methods on the binding:

$$\mathtt{r.getCelsius}()$$

Figure 5 shows these interactions. If the bound object disappears from the view, or a new object appears that better satisfies the binding policy, the middleware automatically updates the binding. The system generates an exception only when no object in the view satisfies the binding request. An agent can also request to receive a special notification that the bound object has changed to a new object.

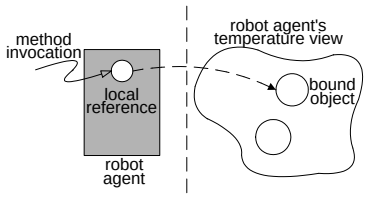

Fig. 5. Agent/view interaction in the context-sensitive bindings model.

3.4 Context-Sensitive Events.

The final model allows agents to interact through a language of events. In this case, the view contains events generated by components in the system. For example, an agent monitoring robot activity might define a view containing events generated when new robots (hosts) connect and are within a certain physical distance. Event-based interactions have become common in distributed systems. The JEDI system [21], for example, defines a distributed event dispatcher through which active entities communicate by generating events and registering to receive events. The SIENA event distribution service [22] addresses scalability issues by aggregating similar event subscriptions. Recent work [23] has targeted publish/subscribe systems for the ad hoc environment, specifically addressing reconfiguration algorithms much needed in the highly dynamic ad hoc environment. These systems address specific implementation concerns, while our goal is to apply the view's scope limiting concept to publish-subscribe models. Our generalized view concept provides allowances for ad hoc mobility and the capability to restrict the scope of visible events based on the network, hosts, agents, objects, and the events themselves.

In this case, objects themselves are not directly visible to agents, only the events they generate are visible. These events are filtered by an event specification. Agents operate on this resulting view of events by binding callback functions to events or prescribed sequences of events which pass through the filter. Any application-defined object can generate events, allowing agents to respond to both application specific events as well as generic events such as a change in an object data field. An agent must subscribe to receive particular event notifications, and an agent receives a notification only if it is subscribed for the event at the time it is generated. To ensure a unified treatment of all events and uniformity of the view contents, we introduce (for specification purposes) virtual objects so named as to refer to application agents, hosts, and network resources abstractly. These special objects pass on system generated events to the context, but their implementation is hard-coded in the middleware. The existence of these virtual objects allows an application agent to react to, for example, the

appearance in the view of a new contributing agent. The context-sensitive events model is provided as a veneer over the context-sensitive data structures model.

In this model, the example robot agent registers to receive temperature events from its view. As shown in Figure 6, this registration attaches a callback function provided by the agent to the generation of the relevant events. As the figure indicates, this style of interaction completely hides the view's contents from the robot agent. An example of this interaction using our illustrative notation is:

$$\mathsf{subscribe}(\langle \mathit{probeType} = \mathsf{temp} \rangle, \mathsf{C})$$

In this example, the callback function is called anytime a temperature probe generates a temperature event. Whether events are generated at a certain frequency or upon a temperature change is determined at the application level by the temperature probe's implementation. The callback function receives an instance of the event, which contains information about the event itself and about the temperature probe object generating the event. The agent will, however, receive all events generated by all temperature probes in the view. To handle this, the agent has several choices. One is simply to filter these events locally, at the application level. A second option would detect a single "first" event and remember the source, probe `p`. The callback for this event would deregister the initial registration and register the agent for only events originating at `p`. This second registration might look like:

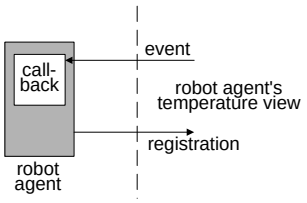

Fig. 6. Agent/view interaction in the context-sensitive events model.

$$\mathsf{subscribe}(\langle \mathit{probeId} = \mathtt{p}, \mathit{probeType} = \mathsf{temp} \rangle, \mathsf{C}')$$

Of course, this option requires the agent to explicitly handle the failure or disappearance of probe `p` by subscribing to events generated by the temperature probe's virtual object. As previously described, this virtual object and the events it generates are defined by the middleware, and an API for accessing this information is provided to the application programmer.

Even for this simple example, each model has advantages and disadvantages. The model chosen for use depends on factors as varied as the guarantees required by the system and the application developer's preferred programming paradigm.

4 Conclusions

Our experiences in the ongoing development of the LIME middleware provide us with a foundation for beginning this model's implementation. A prototype implementation of the basic context-sensitive data structures model builds directly on top of LIME and provides most of the capabilities outlined in this paper. This initial prototype allows us to begin the development of the applications that spurred this investigation. Further work on the middleware's development will provide the true asymmetric behavior and will allow for performance evaluation

studies to be carried out. We approach this development effort from a bottom-up perspective. The lowest level requires algorithms and protocols for gathering information from sensors and disseminating that information in a timely fashion. We have already developed an algorithm for consistent group membership [24] that uses location information to provide the appearance of announced disconnection in spite of host mobility. Other work on providing an abstraction of the network based on properties of network paths [13] establishes a foundation for implementing the view abstraction required in this model. Each layer of the implementation must address key issues related to the highly dynamic ad hoc environment. As mentioned in the introduction, one such issue concerns the application's ability to specify the level of consistency guarantees it requires for particular operations over particular views. As always, another key element of the final implementation involves tradeoffs between system expressiveness and the efficiency of its implementation. In particular, the view specification language should be as flexible as possible without losing the efficiency gains associated with the provision of the asymmetric model. The prototype will be useful in evaluating possible specification mechanics, but conclusive evaluation results will only be available once the implementation of the asymmetric model is fully operational.

As software must function in settings that are increasingly open and highly dynamic, software development is becoming more complex. While we cannot eliminate intrinsic complexities of software artifacts operating under such demanding circumstances, we can reduce the complexity of application development by shifting much of the burden onto the system support infrastructure. Programming power can be amplified by allowing the developer to think at a new and high level of abstraction. Effective use of the limited resources often associated with mobile systems can be achieved by having the system infrastructure explicitly know what the application needs are at any given point in time.

Acknowledgments
This research was supported in part by the National Science Foundation under Grant No. CCR-9970939 and by the Office of Naval Research MURI Research Contract No. N00014-02-1-0715. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the National Science Foundation or the Office of Naval Research.

References

1. Harter, A., Hopper, A.: A distributed location system for the active office. IEEE Networks **8** (1994) 62–70
2. Want, R., et al.: An overview of the PARCTab ubiquitous computing experiment. IEEE Personal Communications **2** (1995) 28–43
3. Abowd, G., Atkeson, C., Hong, J., Long, S., Kooper, R., Pinkerton, M.: Cyberguide: A mobile context-aware tour guide. ACM Wireless Networks **3** (1997) 421–433

4. Cheverst, K., Davies, N., Mitchell, K., Friday, A., Efstratiou, C.: Experiences of developing and deploying a context-aware tourist guide: The GUIDE project. In: Proceedings of MobiCom, ACM Press (2000) 20–31
5. Ryan, N., Pascoe, J., Morse, D.: Fieldnote: A handheld information system for the field. In: First International Workshop on TeloGeoProcessing. (1999) 156–163
6. Salber, D., Dey, A., Abowd, G.: The Context Toolkit: Aiding the development of context-enabled applications. In: Proceedings of CHI'99. (1999) 434–441
7. Hong, J., Landay, J.: An infrastructure approach to context-aware computing. Human Computer Interaction **16** (2001)
8. Schilit, B., Adams, N., Want, R.: Context-aware computing applications. In: IEEE Workshop on Mobile Computing Systems and Applications. (1994) 85–90
9. Gelernter, D.: Generative communication in Linda. ACM Transactions on Programming Languages and Systems **7** (1985) 80–112
10. Cabri, G., Leonardi, L., Zambonelli, F.: MARS: A programmable coordination architecture for mobile agents. Internet Computing **4** (2000) 26–35
11. Murphy, A., Picco, G., Roman, G.: LIME: A middleware for physical and logical mobility. In: Proceedings of the 21^{st} International Conference on Distributed Computing Systems. (2001) 524–533
12. Julien, C., Roman, G.C.: Egocentric context-aware programming in ad hoc mobile networks. In: Proceedings of the 10^th International Symposium on the Foundations of Software Engineering (FSE-10). (2002) 23–30
13. Roman, G., Julien, C., Huang, Q.: Network abstractions for context-aware mobile computing. In: Proceedings of the 24^{th} International Conference on Software Engineering. (2002) 363–373
14. Cugola, G., Picco, G.: PEERWARE: Core middleware support for Peer to Peer and mobile systems. Technical report, Politecnico di Milano (2001)
15. Emmerich, W.: Engineering Distributed Objects. John Wiley and Sons, Ltd. (2000)
16. Edwards, K.: Core JINI. Prentice Hall (1999)
17. Handorean, R., Roman, G.: Service provision in ad hoc networks. In: Proceedings of the 5^{th} International Conference on Coordination Models and Languages. (2002) 207–219
18. Othman, O., O'Ryan, C., Schmidt, D.: Strategies for CORBA middleware-based load balancing. IEEE Distributed Systems Online **2** (2001)
19. Object Management Group: Fault tolerant CORBA specification. OMG Document ptc/2000-04-04 (2000)
20. Keleher, P., Cetintemel, U.: Consistency management in Deno. Mobile Networks and Applications **5** (2000) 299–309
21. Cugola, G., Nitto, E.D., Fuggetta, A.: The JEDI event-based infrastructure and its application to the development of the OPSS WFMS. IEEE Transactions on Software Engineering **27** (2001) 827–850
22. Carzaniga, A., Rosenblum, D., Wolf, A.: Design and evaluation of a wide-area event notification service. ACM Transactions on Computer Systems **19** (2001) 332–383
23. Cugola, G., Picco, G., Murphy, A.: Towards dynamic reconfiguration of distributed publish-subscribe middleware. In: Third International Workshop on Software Engineering and Middleware. (2002)
24. Roman, G., Huang, Q., Hazemi, A.: Consistent group membership in ad hoc networks. In: Proceedings of the 23^{rd} International Conference on Software Engineering. (2001) 381–388