Application Session Semantics for Mobile Ubiquitous Computing

Christine Julien

TR-UTEDGE-2005-002
ABSTRACT

Meeting the demand for dynamic ubiquitous computing applications requires programming abstractions and development tools tailored to this unique environment. This paper provides a programming framework tailored for ubiquitous computing applications that allows developers to express their applications’ needs in terms of interactive application sessions. These sessions are defined by a set of operations performed with remote resources. Our framework allows the developer to delegate responsibility for the construction and maintenance of the communication links necessary to support the application’s sessions to an underlying middleware. In this paper, we formalize a suite of session definitions necessary for general classes of dynamic ubiquitous computing applications. We also present a programming framework and middleware implementation that directly support the software development task. Finally, we demonstrate the simplicity and flexibility of our framework using a real-world application. Our approach promises to greatly simplify the programming of applications, enabling novice programmers to create expressive and adaptive ubiquitous computing applications.

1. INTRODUCTION

The increasing pervasiveness of computing capabilities in our environments has enabled new classes of ubiquitous computing applications. Such applications do not operate autonomously but instead interact with dynamically available resources to provide the user an adaptive, responsive, and intuitive computing experience. Many applications have been built, for example EZCab [34], the Aware Home [26], and a navigation system for the blind [27]. However, existing development tools are not flexible enough to meet the demands of interactive general-purpose ubiquitous computing applications. In this paper, we describe a framework that serves as such a development tool by allowing applications to specify their interactions in terms of the resources they require, removing the need for an application programmer to be intimately familiar with the inner-workings of communication in pervasive computing environments. Such an approach promises to enable rapid development of applications by promoting abstraction, reuse, and transparency.

Within this paper, we use two application domains that exemplify the unique challenges of building ubiquitous computing applications. In first responder applications, a dynamic set of participants is deployed in an emergency situation. People with differing tasks (e.g., paramedics, firemen, policemen, search and rescue personnel, etc.) converge on a geographic area, bringing with them computing, communicating, and sensing devices. Their applications benefit significantly from heightened degrees of cooperation involving pairs of participants or large dynamic groups of people. As a second example, construction sites are becoming increasingly loaded with sensing and computing capabilities. Supervisors and workers on the site desire to connect to local resources in real time to monitor and maintain safety or to track materials for planning.

This work supports applications like those above by providing abstract programming constructs that allow developers to create application sessions that dynamically connect users to resources. We define an application session to be a temporary logical connection among two or more networked devices over which application data is exchanged. The session is further defined by the set of operations the application intends to perform over the logical connection, which is provided by an underlying physical connection with two (or more) distinct endpoints. The application expects the session infrastructure to maintain at least the appearance of a “live” connection, since at any time the client (i.e., the device initiating the session) may dispatch requests, jobs, etc. to the remote resource. From the other direction, the server(s) may be obligated to notify the client of changes or to send periodic information.

Our application session is not to be confused with the Session Layer of the OSI Reference Model [10, 11] of networks, which has specific duties with respect to other layers in the model and uses designated protocols to perform those duties. However, the abstract goal of the OSI Session Layer (i.e., to manage dialogues between two processes) is similar in spirit to our goal as both approaches aim to make the network and its low-level protocols transparent to application processes. Our framework provides a richer, more expressive set of session definitions and, by not rigidly adhering to a particular layered architecture, is able to leverage application information across traditional architectural layers to increase applications’ performance.
This paper’s contributions are on two fronts, both focused on enabling development of dynamic ubiquitous computing applications. First, we clearly identify a set of useful application session semantics and provide formal characterizations of them that communicate the constructs’ behavior to application developers. Second, we provide a middleware infrastructure that allows application developers to use these session constructs to create flexible and adaptive applications. Our framework is the first such programming environment to recognize applications’ needs for diverse session semantics and to provide them in a unified manner.

This paper is organized as follows. Section 2 identifies specific challenges presented by ubiquitous computing environments. Section 3 describes related projects, their benefits, and their shortcomings with respect to providing flexible application sessions. Section 4 presents our framework’s key components, carefully formalizing each session type’s operational semantics. Sections 5 and 6 detail the framework’s programming interface and support infrastructure, respectively. Section 7 demonstrates the use of the framework in a real-world scenario, and Section 8 concludes.

2. CHALLENGES OF THE ENVIRONMENT

This work is motivated by an increased demand for ubiquitous computing applications and the ensuing need to enable novice programmers to create these applications. Several challenges arise in ubiquitous computing environments that must be directly addressed by both the abstractions we provide and their underlying implementation:

- **heterogeneous devices**: the framework must support a variety of devices including laptops, PDAs, sensors, and lightweight domain specific devices (e.g., small medical tags [24] or specialized construction sensors [33]).
- **unpredictable connectivity**: due to mobility and other outside forces (e.g., unreliable wireless communication), the framework must help applications handle intermittent connectivity.
- **unpredictable and dynamic resource availability**: the unpredictable connectivity results in variable resource availability, requiring the framework to opportunistically leverage resources when they are available.
- **large scale and distribution of the network**: pervasive computing networks can grow very large, requiring coordination through multi-hop connections and scalable communication protocols.
- **need for application coordination**: ubiquitous computing applications do not operate autonomously and must coordinate to gather data and access digital resources.
- **locality of information and interactions**: finally, applications’ interactions are commonly defined by some abstract characterization of locality.

The operating environment’s characteristics overburden application developers as they must be concerned with handling the environment’s extreme scale, dynamics, and unpredictability while still meeting applications’ demands for richly expressive coordination and resource access. The work we describe in this paper eases some of this development burden by abstracting many of these facets into a framework that allows applications to specify dynamic and long-lived resource interactions through an intuitive interface.

3. RELATED WORK

As ubiquitous and mobile computing have come to the forefront, research projects have increasingly focused on providing applications dynamic access to a changing set of resources. We do not attempt to provide a comprehensive review but instead highlight the most relevant projects. Many projects have focused on mediating quality of service requirements by leveraging object mobility. For example, FarGo [17], and more recently MobileX [30] and One.World [14] use object mobility to enhance application responsiveness and network-wide performance metrics. These approaches focus on bringing objects closer to clients instead of on the notion that clients themselves are mobile and resources may be inherently location-dependent.

Projects more closely aligned with our goals address exactly this requirement by updating bindings between clients and services as processing or environment dictates [5, 23]. A follow-me session [16] provides constant connectivity to services by transferring a connection from one service provider to another. Context-Sensitive Bindings [16, 29] implement the follow-me session by defining a context surrounding the application of interest and selecting resources from that context that match an application’s specification. However, the implementation relies on a coordination infrastructure (LIME [25]) that does not scale well to large networks. Service Oriented Network Sockets [31] provide a similar abstraction but use well-accepted service discovery mechanisms to gather all matching services locally, then decide 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 [4] presents a dynamic session handoff scheme specific to medical informatics. This solution relies on knowledgeable intermediaries that handle service switches on behalf of clients and resources. Similarly, Atlas [9] uses a central server to mediate the transfer of a service binding from one provider to another.

In addition to the distinctions highlighted above, our framework differs from these existing projects on several dimensions. First, we seek not to limit an application’s session definitions to a single type but to create a framework that flexibly adapts to an application’s needs, whether it simply needs to query a resource, to connect to a single other device, transparently migrate among satisfactory service providers, etc. We provide a framework for application session semantics in which an application programmer can define and use different types of sessions through the same intuitive interface. Second, while we aim to decouple the semantics of application sessions from the implementation of the communication supporting the session, we also recognize that the extreme scale and device constraints of ubiquitous computing applications necessitate communication protocols tailored to particular session requirements. Instead of requiring all session types to use the same communication style, our framework incorporates a suite of novel protocols that efficiently support a variety of session semantics.

4. A FRAMEWORK FOR SESSION SEMANTICS

Our framework provides a set of application session definitions that reduce the complexity of programming long-lived distributed interactions in dynamic ubiquitous com-
puting applications. Our framework is carefully designed to explicitly separate a user program (i.e., the application) from the session management infrastructure that interacts directly with available communication protocols to provide applications the semantics they request.

Figure 1 shows this separation pictorially. In general, the only knowledge shared between the session management capabilities and the program the user creates are two components (labeled shared variables in the figure): a specification (spec) that describes the resource(s) the application is looking for and an object handle (o) that allows the application to access the resource(s) that the infrastructure connects it to. The application completely delegates responsibility for maintaining connections to resources to the infrastructure. This allows the developer to instead focus his attention on the application’s behavior.

The behavior of the different styles of sessions are described below. Each requires the application to provide the resource specification, and the infrastructure fills in and maintains the object handle on behalf of the application. In general, an application invokes a session using code with semantics similar to:

\[
spec = \text{specification}
\]

\[
\text{[request session]}
\]

\[
\text{[use o]}
\]

The uninitialized value (\(\perp\)) indicates that a resource \(o\) declared by an application has not yet been modified by the session management scheme. A null value (\(\epsilon\)) indicates that a matching resource does not exist (or no longer exists). The use of this null value allows the infrastructure to communicate the lack of a match to the application.

Substantial work focuses on allowing applications to abstractly define their resource needs through a variety of specification mechanisms. While the elucidation of expressive specification machinery is outside the scope of this paper, we assume resource requirements are described using semi-structured data \([1]\), an approach common among description languages \([6, 8, 12, 15, 32]\) and tuple based systems \([7, 13, 20, 25]\). The selection of a particular description scheme

4.1 Basic Session Types

We next overview our four basic types of sessions. We will then build on these four types with a few generic extensions.

4.1.1 Query Session

Some application requests are simple data queries. For example, a first responder might request a copy of a nearby building’s blueprints. After downloading the blueprints, the application may have no further need for interactions with the device providing the data. Using the constraints provided in the specification, the application should be connected to a single resource for the duration of the operation. Our first session type provides no long-lived interaction with the selected resource. This can be both beneficial (in terms of reduced network overhead) and limiting (in terms of capturing the environment’s dynamics). Because the session provides merely the capability of copying a single object across the network, it should be used only in cases when the application’s needs for a resource will not require further interactions. We write the semantics of a query session as:

\[
\sigma = \text{specification}
\]

\[
\perp \Rightarrow o = o'.(o' \models spec \land o'.\text{connected})\]

We will use notation similar to the above to express the operational semantics of an application’s sessions. In such a definition, the expression in the box denotes the particular session semantic; in this case, the query semantic is expressed by assigning the specification to the shared object handle, \(o\). The remainder of the expression defines the session’s semantics. In a query session, the value assigned to \(o\) is nondeterministically selected from all objects that satisfy the specification \(spec\) and are connected. The connected relationship models the requirement that the application’s physical device must be able to communicate with the selected resource’s physical device. This abstraction allows the application developer to delegate management of communication links and their dynamics to the middleware that implements the session constructs.

The nondeterministic selection of a resource matching a specification uses the nondeterministic assignment statement \([3]\). A statement \(x := x'.Q\) assigns to \(x\) a value \(x'\) nondeterministically selected from among the values satisfying the predicate \(Q\). If such an assignment is not possible, the statement aborts, and we assume this results in the statement assigning \(\epsilon\) (a null value) to \(x\).

We use entails (\(\models\)) to express the fact that a resource (e.g., \(o'\)) satisfies a provided specification (e.g., \(spec\)).
Figure 2: Using a Provider Session in a first responder application. As the paramedic treats a patient, his interactions should be guaranteed to be with the same device (the particular patient’s medical tag).

4.1.2 Provider Session

In many cases, once an application connects to a resource, it needs to perform several operations with that specific resource. For example, a paramedic may request a connection to a critical patient designated by a medical tag [24] placed by a triage worker. Once a patient is discovered, the paramedic may further query the patient’s tag for injury information, vital signs, etc., and may wish to change and/or add information (e.g., drugs administered). As depicted in Figure 2, to ensure data consistency, the paramedic must interact with the same tag that satisfied the initial request. The operational semantics for this session are:

\[ \odot \leftarrow \text{spec} \]

\[ \Delta \ o = o', (o' \models \text{spec} \land o'.\text{connected}) \]

if \( o \neq \epsilon \) then

\( \langle \text{await} \neg o.\text{connected} \rightarrow o = \epsilon \rangle \)

fi

In a provider session, an application connects to a resource and requests that the communication infrastructure maintains the connection to this particular resource given dynamics in the network topology. In the above expression, the application attaches the specification of the desired resource (\text{spec}) to the object handle \( o \). This is equivalent to nondeterministically selecting a satisfactory connected object. If an object is found, the connection to it is monitored, and as long as the middleware infrastructure can maintain communication between the application and the resource, it does so. This session is a two way connection, so not only can the client application make requests of the resource at any time, but if the resource changes, the client is also updated. For example, if two paramedics are treating the same patient, they may each maintain a provider session with the patient’s tag. If one paramedic changes the resource (e.g., updates the patient’s record), this change is propagated to the second paramedic. From this perspective, the application’s resource handle \( o \) is a local reflection of the remote resource. When the connection to the resource fails (i.e., when \( o.\text{connected} \) becomes false), the handle is assigned \( \epsilon \) (a null value), which effectively notifies the application that the requested resource is no longer available.

4.1.3 Type Session

In other scenarios, an application may need a persistent connection to any resource that matches a request. On the construction site, safety applications may require that a device always knows its location (or at least has an estimate of its location). Due to cost and power limitations, every device may not be equipped with a GPS receiver. Instead, location servers around the site may periodically publish a region identifier to locally connected clients. As a vehicle moves through the site, its application may request continuous connection to a location server within a specified distance. As shown in Figure 3, as the vehicle moves, the particular server offering the location data may change, but the application receives a steady stream of location updates. We express the operational semantics of a type session as:

\[ o \leftarrow \text{spec} \]

\[ \Delta \ o = o', (o' \models \text{spec} \land o'.\text{connected}) \]

while \( o \neq \epsilon \) do

\( \langle \text{await} \neg o.\text{connected} \rightarrow o = \epsilon \rangle \)

od

This expression uses an open arrow (\( \leftarrow \)) to represent the dynamic nature of a type session. As shown, when a resource attached to an application through a type session becomes unavailable, the infrastructure attempts to locate a new resource that is connected and at the same time matches the application’s specification. As long as such a resource is available, the application is connected to one, nondeterministically chosen from all those that meet the requirements. If at any time, a match is not possible, the application’s reference handle is assigned \( \epsilon \), which communicates to the application the fact that no matching resource is available. The above definition for a type session is a bit restrictive in that a satisfactory resource is not available, the application must continuously poll the system until one becomes available. This limitation will be addressed in Section 4.2.

4.1.4 Group Session

Some applications require a session with a group of resources or other devices. For example, an application may monitor the movement of a crane arm and the movement of workers and vehicles within the arc of the crane’s movement [19]. A device in the crane needs a session that includes the devices of all workers and vehicles in this region, as shown in Figure 4. This session can be expressed as:
support for certain groups have acceptable performance under reasonable guarantees. The differences between these groups and the mechanisms our infrastructure uses to communicate the differences to the application programmer are discussed in Sections 5 and 6.

4.2 Session Extensions

We next describe generic extensions to the above basic types that add increased flexibility and expressiveness.

4.2.1 Specifications of Preference

In many instances, an application would like to express preferences that determine a partial ordering of matching resources. In these cases, we allow programmers to specify a metric (referred to as $f(x)$) that selects an application-preferred resource over others. This preference may be specified for any of the first three types of sessions: query sessions, provider sessions, or type sessions. In the first case, the semantics of the augmented query session are:

$$o = \{1\} \text{spec}$$

$$= (\text{set } o' : o' \models \text{spec} \land o'.\text{connected} :: o')^4$$

while $o \neq \emptyset$

\[\text{await } \text{group-change} \rightarrow\]

$$o = (\text{set } o' : o' \models \text{spec} \land o'.\text{connected} :: o')$$

od

where group-change is defined by the following expression:

$$\text{group-change}$$

$$= (\forall o' : o' \in o \land o'.\text{connected})$$

$$\lor (\exists o' : o' \in o \land o' \models \text{spec})$$

$$\lor (\exists o' : o' \notin o \land o'.\text{connected} \land o' \models \text{spec})$$

In a group session, the application is connected to every resource that matches its specification, and the connections to matching resources are maintained as long as some resource matches. That is, the application’s object handle $o$ is connected to a set of objects that match the specification, and the application can subsequently use set operations to interact with the resources. As the set of matching resources changes (either because a matching resource disconnected, dynamics caused a matching resource to no longer satisfy spec, or because a new matching resource connected), the set reflects all of the connected matching resources. As with the type session, as soon as no matching resource exists, the handle is no longer updated to reflect changing state. This issue is discussed in more detail in Section 4.2. Some group definitions are easier to maintain than others, i.e., the communication constructs required to provide implementation

The three-part notation $(\text{op quantified_variables : range :: expression})$ is defined as follows: the variables from quantified_variables take on all possible values permitted by range. Each instantiation of the variables is substituted in expression, producing a multisets 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 when op is $\lor$ or $\emptyset$ when op is set.

4.2.2 More Persistent Connections

Another disadvantage of the basic session types described above is that, if, at any time an applications’ request for a resource cannot be satisfied, the infrastructure ceases looking for matches. This has the benefit of reducing communication overhead, but an application that cannot continue without a matching resource must poll the infrastructure on its own. For this reason we augment our type and group sessions with the ability to request that a session remain “active” even in the absence of a matching resource. Thus, when no resource

Figure 4: Using a Group Session on a construction site. The crane maintains connections to all workers and vehicles within its “danger zone.” As a worker moves into the zone, it is added to the group.
matches the specification, the session remains “active,” and as soon as a satisfactory resource appears, it is connected to the object handle \( o \). In these cases, the only way a session ends is when the application explicitly shuts it down (as described in Section 5). As an example, the new operational semantics for an active type session are:

\[
\begin{array}{l}
\text{let } \mathbf{spec} = \text{spec} \\
\mathbf{spec'} = \mathbf{spec} \land \mathbf{o'.connected} \\
\text{while } \neg \mathbf{stop} \\
(\text{await } \mathbf{a} = \epsilon \lor \neg \mathbf{o.connected} \\
\mathbf{a} = \mathbf{o'}(\mathbf{a'} = \mathbf{spec} \land \mathbf{o'.connected})) \\
\text{od}
\end{array}
\]

This differs from the regular type session in only a few subtle ways. First, the guard on the await statement now also attempts to reassign a resource to the reference handle \( o \) when it is already \( \epsilon \). Second, the condition on the \textbf{while} loop is \( \neg \mathbf{stop} \), which references a third shared variable that is true when the session begins and set to false when the application decides to quit the session. In truth, as described in Section 5, our implementation uses this shared variable in all cases. In the model above, without the \textbf{stop} variable, an application simply stops using the object handle \( o \), which implicitly signals the end to the session. In the implementation, however, the underlying communication protocols should stop maintaining the session as soon as possible to ensure the best overall network performance (i.e., to ensure that network communication is not wasted maintaining connections an application no longer needs).

4.2.3 Maintenance and Migration of State

One aspect of sessions we have ignored within this model is the migration of the state associated with sessions that transfer from one resource provider to another. This is significant in the case of the type session (as it directly involves moving an ongoing session from one provider to another) and may also affect group sessions (if a newcomer needs the history of an ongoing session). For now, our framework does not support the transfer of such session state and instead leaves its maintenance up to the application. Many systems use some form of check-pointing for this task [4, 16], and integrating such a scheme into our model and implementation should be a straightforward task.

5. APPLICATION SPECIFICATIONS: A PROGRAMMING INTERFACE

We provide our session constructs in a programming framework that enables rapid development of ubiquitous computing applications. We briefly detail the abstractions available to developers. In the next section, we describe how our middleware supports this programming interface.

5.1 Data Types

We first present the data types required to interact with the session management framework. While our model does not restrict the format of descriptions and specifications, our implementation uses the ELIGHTS tuple space implementation [20]. Resources are provided as tuples, and the fields of the tuples provide descriptive elements that contain not only the resource (or the proxy information) but also describe its properties. Our framework provides a Resource class that serves as a wrapper for the ETemplate in ELIGHTS. In the same vein, our Specification class is a wrapper of the ETemplate from ELIGHTS and guides developers in defining resource specifications. We show examples of the use of both in Section 7.

The Metric interface allows applications to provide preferences of resources as described in Section 4.2.1. The interface requires an implementing class to provide an evaluate method, which takes as a parameter a Resource and returns the metric’s value for this resource. As described in Section 4.2.1, the application is connected to the resource with the largest value for the Metric.

Finally, the Region class is used for defining group sessions. We explicitly separate the properties of an application’s group specification into two categories. The Region contains all those properties that can be used to restrict the communication region (e.g., distance, latency of communication, bandwidth, etc.). The remainder of the properties are placed in a regular Specification. Our implementation provides specific Region classes that can be used by applications, and an example will be presented in Section 7. This approach allows us to use the properties of the Region to parameterize the communication protocols, thereby maximizing the application’s performance.

5.2 The Session Manager

The major point of interaction between an application’s code and the framework is through the SessionManager. This class offers the nine public methods shown below for constructing and using application sessions:

```java
public class SessionManager {
    public Resource createQuerySession(Specification spec);
    public Resource createProviderSession(Specification spec);
    public Resource createTypeSession(Specification spec);
    public Resource createGroupSession(Specification spec, boolean active);
    public Resource createGroupSession(Specification spec, Metric m);
    public Resource createProviderSession(Specification spec, Metric m);
    public Resource createTypeSession(Specification spec, Metric m, boolean active);
    public void endSession(Specification spec);
    public void postResource(Resource r);
}
```

The first three methods create simple query, type, and provider sessions using a provided specification. The active boolean in the type session designates whether the infrastructure should actively poll for a new match (as described in Section 4.2.2). The fourth method, createGroupSession, allows the application to provide information about the Region of communication. The application can omit either of this method’s first two parameters (by providing a null value). If the Specification is omitted, the query will connect the application to all resources within the region. If the Region is omitted, the query will connect the application to all resources matching the specification in the entire network. In many cases, the latter instance requires flooding the entire pervasive computing network with the request, which can incur extreme communication overhead.

The next three methods in the SessionManager interface are used to specify a metric for preference in addition to
the resource specification. All metrics must implement the Metric interface. Section 7 includes an example.

The method `endSession` allows the application to determine when a session ends (instead of simply waiting until a resource is no longer available). This method uses the provided `Specification` to determine which session to close.

Participants in a distributed application coordinate by sharing resources. The final method allows application components to make their data or resources available to other components. In the next section, we will show how the applications’ postings eventually become available resources for coordination with other components.

6. COMMUNICATION AND PROTOCOL SUPPORT

Due to the challenges imposed in ubiquitous computing environments, an important aspect for any middleware is the efficiency of the underlying network communication. Our implementation pays careful attention to selecting the best communication paradigm for a particular situation.

6.1 Overview of our Approach

In Section 3, we described existing approaches to providing application sessions in mobile environments. In addition to lacking the necessary expressiveness for a wide range of ubiquitous computing applications, these approaches use communication models that do not fit the unique challenges of ubiquitous computing. First, these approaches do not adapt the communication paradigm to network conditions or changing application requirements. Some of the approaches [4, 9] require persistent connectivity to an infrastructure which introduces a communication bottleneck and may not even be feasible in many instances. Those approaches that do not require centralized communication [16, 31] use communication models that do not scale well to large ubiquitous computing networks. Our approach instead uses a suite of context- and content-sensitive communication protocols that can adapt to a particular type of session, the requirements encapsulated in an application’s request, and even to the conditions of the operating environment.

6.2 Network Support Implementation

Figure 5 overviews our architecture. We describe this implementation by looking at how it supports each type of session in more detail.

6.2.1 Sharing Resources

When an application posts a resource through the `postResource` method in the `SessionManager`, the resource is stored in a local repository (shown within the Session Manager in Figure 5). If the resource discovery mechanism is also using a registry (as described below), the description is also posted to that registry. Many of the underlying protocols use peer-to-peer communication, which requires each session manager to respond to the requests of remote application components. When these requests arrive, they propagate up the architecture stack shown in Figure 5 until they reach the session manager which can determine whether a matching resource exists at this location by looking within the local repository. Because our implementation represents resources and requests as tuples and templates, this matching is performed within `ELIGHTS` [20].

![Figure 5: The Architecture of the Application Session Support Framework](image_url)

6.2.2 Supporting Query Sessions

Query sessions are the simplest of the session constructs. However, efficiently discovering a resource in a highly dynamic pervasive computing environment with little a priori knowledge can be very difficult. As Figure 5 shows, we use a package of discovery protocols to accomplish this task. In relatively static environments, where the set of devices and the set of provided resources does not change very often, we use a registry-based method. Our solution provides functionality similar to Jini [12], and is based on Dynamic Source Routing (DSR) [18]. The application’s session manager queries a well-known lookup service for the resource it desires. The response from the lookup service contains the identity of the device(s) that can offer the requested resource, and the discovery package subsequently contacts one of these devices and downloads the requested data.

While such an approach accomplishes the desired coordination and is straightforward to implement, we have shown that a more application-aware protocol can function with higher efficiency in dynamic environments [22]. For these situations, we have created Cross-Layer Discovery and Routing (CDR) that uses information encapsulated in application requests to perform a distributed query without the assistance of a lookup service. Details of this protocol are outside the scope of this paper and can be found in [22].

Our evaluations have shown that an ideal protocol for supporting this style of resource directed discovery may lie between the above two implementations (the “Hybrid” in Figure 5). A protocol that fits this classification is under development and will include a mix of the proactive style of the registry approach and the reactive style of CDR.

This approach also accommodates applications’ preference specifications. Because the tuple based approach uses a semi-structured data representation, descriptions contain “advertised” properties of the resource. Based on these properties and the conditions of the network (e.g., latency, bandwidth, and mobility conditions), the discovery package on the application’s device can determine which discovered resource best satisfies the complete application request. In our current implementation, the protocol waits for a pre-determined time (based on the double of an estimate of the network’s worst case round trip time) to ensure that it has
received a response from the "best" resource.

Finally, some of the information carried within the application's resource specification and its preferences may define quality of service (QoS) requirements. In our current implementation, we do not use this information to augment the discovery protocol(s). Instead, QoS requirements and preferences are sorted out as part of the resource matching process. In the future, using this information within the communication protocol to ensure that no connections are created to resources that lack certain qualities of service may boost performance. Ongoing work with our CDR protocol is investigating the use of such QoS information in routing. This approach has already shown some success within our group communication protocol, described below.

### 6.2.3 Supporting Provider Sessions

A provider session requires discovering a satisfactory resource (as in the query session) and maintaining a connection to the device that offers the resource. To provide the former, we use the schemes described above. The long-lived connection to the discovered resource is maintained using existing mobile ad hoc routing schemes. Our implementation uses DSR because it is already in use within our framework, but other similar schemes would work equally well.

### 6.2.4 Supporting Type Sessions

Figure 5 shows the type session on top of the provider session. This is because, in our current implementation, we provide type sessions as a series of provider sessions. This approach has the benefit of being simple to implement but can potentially introduce a lag between the time when a new, better resource is available, and the time when the application switches to the new resource. This issue arises only when an application specifies preferences.

In the simple case, the initiation of a type session occurs in the same manner as the provider session. The connection to the first discovered resource is maintained as long as possible. When the connection to the resource breaks, the application is not immediately faced with a null value for its handle. Instead, the implementation attempts to launch another provider session. As long as this is successful, the application remains connected to a satisfactory resource.

When applications specify preferences, the implementation must constantly monitor the network for new resources that better satisfy the request. The initial discovery occurs as above. In addition, the type session implementation periodically reissues this initial request to determine whether a better resource exists. In our implementation, we reissue this query once a second. Preliminary tests of CDR [22] show that this data rate provides acceptable network performance when a limited number of devices are performing requests. Further experimentation will help us select the ideal period and may result in an adaptive approach that changes the period in response to network conditions.

### 6.2.5 Supporting Group Sessions

Due to the different nature of group sessions, we use an entirely different communication approach to ensure the most efficient communication. Our approach is based on our context-aware Source-Initiated Context Construction (SICC) protocol [21, 28]. SICC creates and maintains connections to a set of devices that satisfy the application's region specification. Effectively, SICC creates a reverse multicast tree that allows information to funnel back to the requesting device from other devices within the region. The specific details of the protocol can be found in [21]. One important aspect with respect to our framework's implementation is that the metrics for defining regions must adhere to certain properties. By providing the region abstraction to the developer, our framework ensures that the regions a programmer defines satisfy the underlying protocol's requirements.

Once SICC has created connections to the group's devices, the group session implementation issues queries according to SICC's interface to discover resources that satisfy the application's specification. By issuing long-lived queries over SICC's network structure, the group session implementation can be assured that it receives notification of new resources and removes old resources as mobility and other dynamic conditions change the group membership.

### 7. AN APPLICATION SCENARIO

To demonstrate how a developer uses our framework to build interactive pervasive applications, we consider a team of first responders deployed in an urban environment. The applications used by responders with different tasks (e.g., policemen, firemen, search and rescue personnel, paramedics, etc.) may have significantly different functionality even though they coordinate to perform their tasks. This scenario is highly analogous to many others including the deployment of scout teams in military applications.

We consider a responder tasked with search and rescue. She moves from building to building, looks for survivors, and calls for help. In our current implementation, we do not use this information to augment the framework's discovery protocol(s). Instead, QoS requirements and preferences are sorted out as part of the resource matching process. In the future, using this information within the communication protocol to ensure that no connections are created to resources that lack certain qualities of service may boost performance. Ongoing work with our CDR protocol is investigating the use of such QoS information in routing. This approach has already shown some success within our group communication protocol, described below.

#### Finding a local map:
When the responder is first deployed, she may download a street map of the region. Depending on the situation, this map may be available locally (e.g., on the device of another responder who has already downloaded it) or it may need to be downloaded from a centralized server in the infrastructure. The implementation of the framework makes this determination transparently on behalf of the application, using the CDR protocol for local interactions and the repository service for a centralized lookup. The application code that performs this action is:

```java
Specification spec = new Specification();
spec.addConstraint(type, Specification.EQUALS, "Map");
Map localMap = (Map) sessionManager.createQuerySession(spec);
if (localMap != null) {
    [display map]
}
```

In the code, `sessionManager` refers to the application's handle to the local `SessionFactory`, and the `Map` class is defined within the first responder application. The first two lines define the simple specification of the resource the application requests. The constraint the third line of code adds to the specification requires that the type field of the returned resource must have exactly the value "Map." The
third line calls the `createQuerySession` method to retrieve the specified resource. When a map has been discovered, the value of localMap will be some value other than null, and it can be displayed to the user.

**Staying connected to local blueprints:** As the responder moves from one building to the next, she will likely want a copy of the blueprints of the local building if they are available. These blueprints could be stored in a device in the building itself (e.g., as part of the building or home’s security system which is likely connected to an emergency power supply) or stored on or constructed by the device of another nearby responder. In this case, the application creates a type session:

```java
Specification spec = new Specification();
spec.addConstraint(type, Specification.EQUALS, "Blueprint");
Metric local = new MyBuildingMetric();
Blueprint building = (Blueprint)sessionManager.createTypeSession(spec, local, true);
[display blueprints when available]
```

One significant benefit of our framework is that, even though they provide significantly different functionalities, the two code fragments above are very similar. The type session `prefers` blueprints for the current building over any others. This preference is encapsulated in `MyBuildingMetric`, whose evaluate method assigns the value “1” to resources in my building and the value “0” to any other resource. When the responder moves to a new building, a different set of blueprints are automatically attached to the `building` handle and can be displayed by the application.

One drawback of our approach is that it is possible for the application’s session to connect to a blueprint for a building other than the current one if a blueprint for the current building is unavailable. This disadvantage may be overcome by an extension of our approach that allows specifications to be based on contextual properties. In the above example, this would allow the specification to require that a matching resource is within the current building. Future work will investigate both the proper abstractions for this approach and the communication protocols necessary to support it.

**Learning about nearby workers’ movements:** Once the responder has a good picture of her environment, she wants to coordinate with fellow searchers to ensure that their tasks are performed efficiently. In our application, each responder keeps track of the buildings (and the rooms within the buildings) he or she has recently visited. Then the map (or the blueprint) can be overlaid with this information to ensure that our responder does not cover the same territory that has been searched by one of her colleagues. The code to discover and monitor these trajectories is:

```java
Specification spec = new Specification();
spec.addConstraint(type, Specification.EQUALS, "Trajectory");
Region r = DistanceRegion(100);
Trajectory[] trajectories = (Trajectory[])sessionManager.createGroupSession(r, spec, true);
[display trajectories on map]
```

This code fragment defines a `DistanceRegion` that restricts the returned trajectories to those belonging to other first responders within 100 meters. This `DistanceRegion` class is provided within our framework and restricts a group to only those devices within the number of meters specified. The communication protocol that ensures this is a bit more complicated than we represent it here; details can be found in [21]. Once this session is created, our responder’s application will be constantly updated with respect to changes to the trajectories of other responders within 100 meters.

**Summoning evacuation transportation:** Once our responder has located a survivor, she tags him [24] and loads information about the survivor’s condition and location into the tag. She then needs to contact some form of evacuation vehicle to retrieve the survivor and transport him to safety. Depending on the survivor’s condition, this could be an ambulance or some other vehicle (e.g., a bus). However, the responder would like to contact a particular vehicle, transfer the information about the survivor (including his location), and receive a confirmation that a particular vehicle will be retrieving the survivor. To ensure data consistency, the responder’s device should connect to a proper vehicle and remain connected for the duration of the exchange:

```java
Specification spec = new Specification();
spec.addConstraint(type, Specification.EQUALS, "Ambulance");
Vehicle ambulance = (Vehicle)sessionManager.createProviderSession(spec);
[transfer information about survivor]
[receive confirmation]
```

Again, this code has many similarities to the other examples, a significant benefit in simplifying programming. In addition, because this session is defined by a discrete number of well-known tasks, when the session completes, the application invokes the `endSession` method to tear down the communication lines that were created for the session.

**Sharing resources:** All of the previous discussions have assumed that some other application component in the ubiquitous computing environment has made the requested resource available. As described in Section 6, when an application shares a resource, the resource and its description are placed in a local repository within the session manager. For example, when a first responder creates its trajectory to share with other nearby responders, it creates an instance of the `Trajectory` class (which extends the `Resource` class). As the responder moves, he updates his trajectory, actually changing the resource stored in the repository within the local session manager. This change then propagates to, for example, our first responder who has requested a group session that monitors other nearby responders.

8. **CONCLUSIONS**

In this paper, we presented a comprehensive development framework and programming interface that allows novice programmers to write ubiquitous computing applications that entail sophisticated interactions with resources in the environment. We introduced the notion of an application session with respect to ubiquitous computing applications and demonstrated a novel set of such sessions that prove useful to a wide range of dynamic interactive applications. By subsequently capturing our rigorously defined sessions in a programming infrastructure, we present application developers with abstractions that ease their programming burdens and enable novice programmers to write complex, adaptive applications. The middleware infrastructure that sup-
ports these varying session definitions pays careful attention to essential requirements of pervasive computing environments. By incorporating a suite of dynamic and adaptive communication protocols, the middleware ensures that it provides the most appropriate, efficient, and scalable form of communication for different session types in varying environments. Such an integrative approach to abstraction and communication is imperative to meeting the rapidly growing demand for ubiquitous computing applications.

9. REFERENCES


