Enabling Ubiquitous Coordination Using Application Sessions

Christine Julien and Drew Stovall

The Center for Excellence in Distributed Global Environments
The Department of Electrical and Computer Engineering
The University of Texas at Austin
{c.julien, dstovall}@mail.utexas.edu

Abstract. Enabling coordination among ubiquitous computing applications and resources requires programming abstractions and development tools tailored to this unique environment. This paper introduces a suite of coordination abstractions that enables expressive interaction between ubiquitous computing applications and dynamically available resources. In our model, applications express their coordination needs in terms of *application sessions* that are loosely defined by a set of interactions with remote resources. Our approach allows developers 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 for coordination in general classes of ubiquitous computing applications. We also present a middleware based on this coordination model that directly supports the software development task. Finally, we demonstrate the simplicity and flexibility of our approach using a real-world application.

1 Introduction

The increasing pervasiveness of computing capabilities has enabled new classes of ubiquitous applications that rely on interactions with dynamically available resources to provide an adaptive, responsive, and intuitive computing experience. Many applications have been built, but existing development tools are not flexible enough to meet the demands of interactive general-purpose applications. This paper undertakes a coordination approach to specifying and managing the interactions between application and resources. We leverage the benefits of this coordination to realize a programming framework that removes the need for an application programmer to be intimately familiar with the details of communication in pervasive computing. Our approach promises to simplify application development 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 defines new coordination mechanisms specifically tailored to pervasive computing applications. We define an application session to be *a temporary logical connection among two or more networked devices over which application data is exchanged*. We differentiate this application session from other connection mechanisms in that the state maintained involves an application-level dialog between the communicating entities and depends significantly on the application. As such, a 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 between two (or more) distinct endpoints.

This paper's contributions are on two fronts, both focused on using coordination to simplify application development for ubiquitous computing. First, we define a coordination framework around the concept of application sessions and provide formal characterizations of a useful set of such sessions that clearly communicate the constructs' behavior to application developers. Second, we provide a middleware infrastructure that allows application developers to use these coordination 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 describes related projects. Section 3 introduces the new coordination constructs. In Section 4 we describe the programming interface and middleware implementation. Section 5 demonstrates the use of the framework in a real-world scenario, and Section 6 concludes.

2 Related Work

It has been shown that adopting a coordination approach to handling the unpredictability inherent in mobile computing can lead to solutions that simplify programming [1]. Several middleware solutions have taken this approach [2–4] but focus on exchanging data items in dynamic conditions and not on generic resource usage in pervasive computing situations. As ubiquitous computing has come to the forefront, projects have increasingly focused on providing dynamic access to a changing set of resources. Many efforts mediate quality of service requirements by leveraging object mobility to enhance application responsiveness and network-wide performance metrics [5–7]. These approaches focus on bringing objects closer to clients instead of on mobile clients that require inherently location-dependent resources.

Projects closer to our goals update bindings between clients and services as processing or environment dictates [8, 9]. A *follow-me session* [10] provides constant connectivity to services by transferring a connection from one provider to another. Context-Sensitive Bindings [10, 11] implement the follow-me session by

defining a *context* and selecting resources from that context that match an application's specification. The approach favors complete transparency, and assumes that a resource binding should always be transferred, subject to an application's specified policies. *Service Oriented Network Sockets* [12] 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 [13] presents a dynamic session hand-off scheme but relies on knowledgeable intermediaries that handle service switches on behalf of clients and resources. Similarly, Atlas [14] uses a central server to mediate the transfer of a service binding from one provider to another.

Our framework differs from these projects in several ways. First, we seek not to limit an application's sessions to a single type but to adapt to an application's needs, including simple queries, lasting connections, transparent resource migration, etc. Second, while we aim to decouple the semantics of application sessions from the implementation supporting the session, we recognize that the extreme scale and device constraints necessitate communication protocols tailored to particular session requirements. 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 coordination semantics.

3 Defining Application Sessions

Our model introduces a set of application session definitions that coordinate interactions between ubiquitous computing applications and dynamic resources.

As shown in Fig. 1, we explicitly separate a user program (i.e., the application) from the session management infrastructure that manages coordination with available resources. The only knowledge shared between the session management and the user program are 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. Through the coordination primitives this framework provides, the application completely delegates responsibility for maintaining resource connections to the infrastructure.

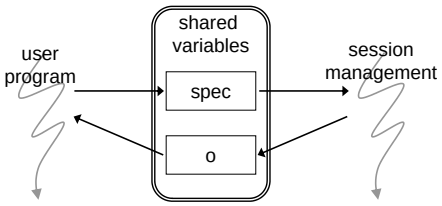

Fig. 1. Separation of Session Management from User Program

Substantial work has focused on allowing applications to abstractly define their resource needs through a variety of specification mechanisms. We assume resource requirements are described using semi-structured data [15], an approach common among description languages [16–18] and tuple based systems [2, 3, 19, 20]. Our approach can incorporate any of these schemes, so application developers can utilize specification languages with which they are familiar.

3.1 A Notation

Section 3.2 will introduce the sessions that provide varying coordination semantics between applications and resources. Each requires the application to provide a resource specification, and the session mechanics fill in and maintain the object handle on behalf of the application. In general, an application will invoke a session using code with semantics similar to those shown in Fig. 2.

The uninitialized value ($\perp$) indicates that a resource o declared by an application has not yet been modified by the session management scheme (i.e., a search is in progress). A null value (ϵ) indicates that a matching resource does not exist (or no longer exists). The $\langle$**await** $B \rightarrow S\rangle$ construct [21] allows a program to delay execution until the condition B holds. When B is true, the statements in S are executed in order. The angle brackets enclosing the construct indicate that the statement is executed atomically, i.e., no state internal to S is visible outside the execution of S. If S is omitted (as in Fig. 2), then the entire expression signifies a point of conditional synchronization.

```
spec = specification
[request session]
⟨await o ≠⊥⟩
if o ≠ ϵ then
    [use o]
fi
```

Fig. 2. Application Session Interaction

Throughout the next section, we will use some additional notational conventions. First, the *entails* ($\models$) relation expresses the fact that a resource satisfies a specification, i.e., $o \models spec$ indicates that the resource o satisfies the specification *spec*. The selection of a resource matching a specification will use the *non-deterministic assignment statement* [22]. A statement $x := x'.Q$ assigns to x a value x' nondeterministically selected from among the values satisfying the predicate Q. If an assignment is not possible, the statement aborts; we assume this results in assigning ϵ (a null value) to x. Within our model's semantics, we will use this notation to indicate that a resource is selected nondeterministically from any that satisfy the application's provided specification. Finally, we will also use a *three-part notation*: $\langle$**op** *quantified_variables* : *range* :: *expression*$\rangle$, in which the variables from *quantified_variables* take on all possible values permitted by *range*. Each instantiation of the variables is substituted in *expression*, producing a multiset of values to which **op** is applied, yielding the value of the three-part expression. If no instantiation of the variables satisfies *range*, then the value of the three part expression is the identity element for **op**, e.g., *true* if **op** is $\forall$ or $\emptyset$ when **op** is set.

3.2 Basic Session Types

We next detail four basic sessions that form the foundation of our coordination framework. In Section 3.3, we describe a few generic extensions.

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). We write the semantics of a query session as:

$$\boxed{o = spec} \triangleq o = o'.(o' \models spec \wedge o'.\texttt{connected})$$

In these definitions, 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 device must be able to communicate with the selected resource's device. This abstraction allows the developer to delegate communication management to the middleware that implements the session constructs. In some cases, connectedness alone may not be enough to model usefulness of a resource; other characteristics can be handled as discussed in Section 3.3.

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 [23] 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. As depicted in Fig. 3, to ensure data consistency, the paramedic must interact with the *same* tag that satisfied the initial request. The operational semantics for this session are:

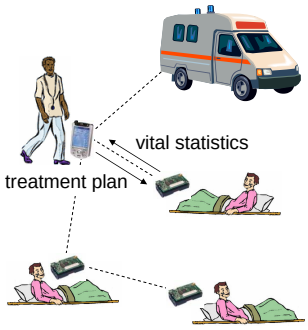

Fig. 3. Using a *Provider Session* in a first responder application.

$$\boxed{o \blacktriangleleft spec} \triangleq o = o'.(o' \models spec \wedge o'.\texttt{connected})$$
$$\textbf{if } o \neq \epsilon \textbf{ then}$$
$$\quad \langle \textbf{await } \neg o.\texttt{connected} \rightarrow o = \epsilon \rangle$$
$$\textbf{fi}$$

In a *provider session*, an application requests that the infrastructure maintains the connection to a particular resource given dynamics in the network topology. The application attaches the specification (*spec*) to the object handle o. If an

object is found, the connection to it is monitored, and as long as the middleware can maintain communication between the application and the resource, it does so. This session is a two-way connection, so not only can the application make requests of the resource, but, if the resource changes, the client is also updated. If two paramedics are treating the same patient, and one changes the resource (e.g., updates the patient's record), this change is propagated to the second paramedic. The application's resource handle o is a local reflection of the remote resource. When the connection to the resource fails (i.e., when $o.\texttt{connected}$ becomes false), the handle is assigned ϵ, which effectively notifies the application that the requested resource is no longer available.

Type Session. In other scenarios, an application may need persistent connection to *any* matching resource. On the construction site, safety applications may require that a device always knows its location (or an estimate of its location). Location servers around the site may periodically publish a region identifier, and a vehicle moving through the site can maintain a connection to a nearby location server. As Fig. 4 shows, 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 a *type session* as:

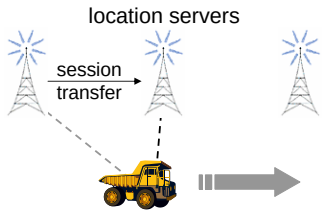

Fig. 4. Using a *Type Session* on a construction site.

$$\boxed{o \Leftarrow \mathit{spec}}$$
$$\triangleq\ o = o'.(o' \models \mathit{spec} \wedge o'.\texttt{connected})$$
$$\textbf{while}\ o \neq \epsilon\ \textbf{do}$$
$$\quad \langle \textbf{await}\ \neg o.\texttt{connected} \rightarrow o = o'.(o' \models \mathit{spec} \wedge o'.\texttt{connected}) \rangle$$
$$\textbf{od}$$

This expression uses an open arrow ($\Leftarrow$) to represent the dynamic nature of a *type session*. When an attached resource becomes unavailable, the infrastructure attempts to locate a new resource that *is* connected and matches the specification. As long as such a resource is available, the application is connected to one, nondeterministically chosen from those that meet the requirements. If a match is not possible, the application's reference handle is assigned ϵ, which indicates that no matching resource is available. The above definition is a bit restrictive in that if a satisfactory resource is not available, the application must poll until one becomes available. This limitation will be addressed in Section 3.3.

Group Session. Some applications require a session with a *group* of resources. For example, an application may monitor the movement of workers and vehicles within the arc of a crane's movement. A device in the crane needs a session that includes the devices of workers and vehicles in this region, as shown in Fig. 5. 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. This session can be expressed as:

$$\boxed{o \Leftarrow_{\{\}} \mathit{spec}}$$
$$\triangleq \quad o = \langle \text{set } o' : o' \models \mathit{spec} \wedge o'.\texttt{connected} \wedge :: o' \rangle$$
while $o \neq \emptyset$ **do**
$\quad \langle$**await** *group-change* $\rightarrow o = \langle \text{set } o' : o' \models \mathit{spec} \wedge o'.\texttt{connected} :: o' \rangle \rangle$
od

where *group-change* is defined by the following expression:

$$\boxed{\mathit{group\text{-}change}}$$
$$\equiv \quad \langle \exists o' : o' \in o \wedge \neg o'.\texttt{connected} \rangle$$
$$\vee \langle \exists o' : o' \in o \wedge o \not\models \mathit{spec} \rangle$$
$$\vee \langle \exists o' : o' \notin o \wedge o'.\texttt{connected} \wedge o' \models \mathit{spec} \rangle$$

The 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 this set 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. Some group definitions are easier to maintain than others, i.e., the communication constructs required for certain group definitions have acceptable performance under reasonable guarantees. The mechanisms our infrastructure uses to provide group communications are discussed in Section 4.

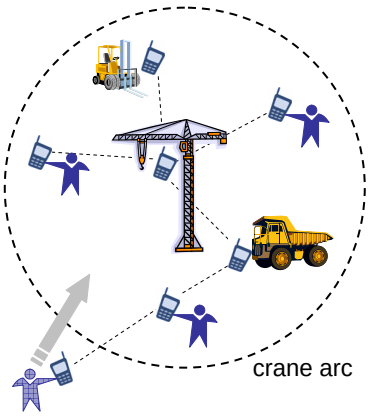

Fig. 5. Using a *Group Session* on a construction site.

3.3 Session Extensions

We next describe generic extensions that add flexibility and expressiveness.

Specifications of Preference. In many instances, an application would like to express preferences that determine a partial ordering of matching resources. We allow programmers to specify a metric ($f(R)$) that selects a preferred resource over others. Generically, a metric accepts a resource's description (which can include information about the device where the resource is located) and generates an integer. Preferences may be specified for query sessions, provider sessions, or type sessions. The semantics of the augmented query session are:

$$\boxed{o = \mathit{spec}/f(R)}$$
$$\triangleq \quad o = \langle \textbf{max } o' : o' \models \mathit{spec} \wedge o'.\texttt{connected} :: f(o') \rangle$$

This statement selects the resource with the largest metric value. If multiple resources have the same value, one is selected nondeterministically. For a provider session, the selection statement is very similar. In a type session, an additional change ensures that the connection is maintained to the most preferred resource:

$\boxed{o \Leftarrow spec/f(R)}$

$\triangleq$ $o = \langle \mathbf{max}\, o' : o' \models spec \wedge o'.\texttt{connected} :: f(o') \rangle$
while $o \neq \epsilon$ **do**
$\langle \mathbf{await}\, \neg o.\texttt{connected} \vee \langle \exists o' : o'.\texttt{connected} \wedge o' \models spec \wedge f(o') > f(o) \rangle \rightarrow$
$o = o'.(o' \models spec \wedge o'.\texttt{connected}) \rangle$
od

For brevity, the mechanics behind metric definition are omitted from this paper; an example is provided in Section 5. Useful metrics include:

- *relative mobility*: more stationary (i.e., less mobile) resources may be preferable due to their increased stability.
- *proximity*: closer resources (or resources in the same building) may often be preferable to more distant ones.
- *reliability*: resources with more consistent up-times are likely to be preferable.
- *error rate*: resources with smaller potential for error are more desirable.

These metrics can also be used to account for the cost or quality of service associated with using a particular resource, based on application-level definitions.

More Persistent Connections. In the basic session types, if an application's request cannot be satisfied, the infrastructure ceases looking for matches. This reduces communication overhead, but an application that cannot continue without a matching resource must poll 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. As soon as a satisfactory resource does appear, it is connected. An active session ends only when the application explicitly shuts it down. The semantics for an *active type session* are:

$\boxed{o \Leftarrow spec}$

$\triangleq$ $o = o'.(o' \models spec \wedge o'.\texttt{connected})$
while $\neg stop$ **do**
$\langle \mathbf{await}\, o = \epsilon \vee \neg o.\texttt{connected} \rightarrow o = o'.(o' \models spec \wedge o'.\texttt{connected}) \rangle$
od

This differs from the regular type session in a few subtle ways. First, the guard on the await statement now also attempts to reassign a resource when o is already ϵ. Second, the condition on the **while** loop is $\neg stop$, which references a third shared variable that is true when the session begins and set to false when the application quits the session. Without the *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, so our implementation uses the stop variable in all cases.

Maintenance and Migration of State. One aspect of sessions we have ignored so far is the migration of session state 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. Future work will include the formalization of such state transfers and their integration into our middleware.

4 Application Sessions: A Middleware

We provide our session constructs in a programming framework that enables rapid development of ubiquitous computing applications. We briefly detail the programming interface and our prototype implementation. Where appropriate, we also describe intended enhancements to the existing prototype.

4.1 Data Types

While our model does not restrict the format of descriptions and specifications, our implementation uses the eLIGHTS tuple space implementation [3]. Resources are provided as tuples that contain not only the resource (or its proxy) but also describe its properties. The `Resource` class serves as a wrapper for the `ETuple`; the `Specification` class is a wrapper of the `ETemplate` and provides restrictions over `Resource`s. The `Metric` interface allows applications to provide resource preferences and requires an implementing class to provide an `evaluate` method, which returns the metric's value for a provided `Resource`. Finally, 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 applications can use. We can use the `Region` to parameterize the communication protocols, thereby maximizing the application's performance.

4.2 The Session Factory

The major point of interaction between an application and the framework is the `SessionFactory`. A version of its interface (slightly simplified for presentation purposes) is shown in Fig. 6. The first three methods create basic sessions using a provided specification. The `active` boolean in the *type session* designates whether the middleware should monitor the available resources for a new match. The fourth method, `createGroupSession` uses information about the `Region` of communication. The next three methods allow a metric for preference in addition to the resource specification. The method `endSession` allows the application to determine when a session for a given `Specification` ends (instead of waiting

```
public class SessionFactory {
  public Resource createQuerySession(Specification spec);
  public Resource createProviderSession(Specification spec);
  public Resource createTypeSession(Specification spec, boolean active);
  public Resource[] createGroupSession(Region r, Specification spec,
                                       boolean active);
  public Resource createQuerySession(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 addResource(Resource r);
}
```

Fig. 6. Application Sessions Programming Interface

until a resource is no longer available). The final method allows applications to make resources available to other components.

4.3 Middleware Support

Fig. 7 overviews our middleware's architecture. Many of the underlying protocols use peer-to-peer communication, which requires each session factory to respond to remote applications' requests. When requests arrive, the session factory determines whether a matching resource exists at this location (by looking in the local repository). Because our implementation represents resources and requests as tuples and templates, this matching is performed within eLights. While matching tuples against templates is straightforward, the complexity of checking $o \models spec$ depends on both the specification language used (e.g., eLights vs. another service description language) and the application. Future work will evaluate the difficulty associated with this aspect of the framework.

Efficiently discovering a resource in a dynamic pervasive computing environment can be very difficult. As Figure 7 shows, we use a package of discovery protocols. In relatively static environments, where the devices and resources change rarely, we use a registry method similar to Jini [18]. While such an approach is straightforward to implement, we have shown that a more application-aware protocol is more efficient in dynamic environments [24]. We have created Cross-Layer Discovery and Routing (CDR) [24] that uses information encapsulated in application requests to perform distributed resource discovery without a lookup service. Our evaluations have further shown that an ideal discovery protocol may lie between the above two implementations. A hybrid protocol that combines the proactive style with the reactive style is under development. Currently, the selection of protocols associated with static or dynamic environments is performed off-line; future work will integrate context-awareness and adaptation into the middleware to allow it to switch between protocols as the environment dictates.

In our tuple based approach, descriptions contain "advertised" resource properties. Based on these properties and network conditions (e.g., latency, bandwidth, and mobility conditions), a session can use the application's preferences

to determine which discovered resource best satisfies a request. In our prototype, the protocol waits for a predetermined 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. Currently, QoS requirements and preferences are sorted out as part of the resource matching process. In the future, using this information as part of the communication protocol may boost performance; we have seen promising results with the protocol for group communication (below) and are incorporating similar mechanisms into our CDR protocol.

To provide the long-lived connection required by a provider session, we use a mobile ad hoc routing scheme (DSR [25]) to maintain a route and discover when the route fails. In our current implementation, we provide a type session as a series of provider sessions. The connection to the first discovered resource is maintained as long as possible. When the connection to the resource breaks, 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 monitor the network for new resources that better satisfy the request. In this case, the middleware periodically reissues this initial request to determine whether a better resource exists. This polling implementation does not exactly match the semantics of the type session given in Section 3, and future work will develop reactive protocols for updating type session bindings that are not cost or performance prohibitive.

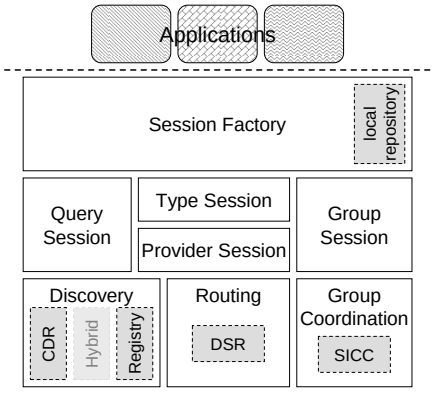

Fig. 7. Application Session Middleware

We use an entirely different communication approach to provide efficient communication in group sessions. Our approach is based on our Source-Initiated Context Construction (SICC) protocol [26] that 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. By providing the region abstraction to the developer, our framework ensures that the regions a programmer defines satisfy the underlying protocol's requirements. By issuing persistent 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 conditions change the group membership. Our current implementation allows iteration over the group of resources; future work will formalize the semantics of varying forms of iteration.

A prototype implementation of this coordination middleware and its associated documentation are available at `http://dstovall.org/servicesessions/`.

5 An Application Scenario

To demonstrate how a developer uses our framework to build pervasive applications, we consider a team of first responders in an urban environment, tasked with search and rescue. A responder moves from building to building, looks for survivors, tags them with small sensors that emit information about their conditions and locations, and summons transportation. We take a few of the tasks that the responder's application supports and examine how these operations use our framework to find and coordinate with resources in the environment. To simplify the example code fragments, we use very simple resource specifications that search for resources based only on their types; most applications (including the ones we describe) will use more sophisticated requests.

Finding a local map: When the responder is first deployed, she may download a street map of the region. This map may be available on the device of a nearby responder who has already downloaded it or it may need to be downloaded from a central server. The application code that performs this action is:

```
Specification spec = new Specification();
spec.addConstraint(type, Specification.EQUALS, ‘‘Map’’);
Map localMap = (Map)sessionFactory.createQuerySession(spec);
if(localMap != null)
   [display map]
```

The `Map` class extends `Resource` and is defined within the application. The first two lines define the simple resource specification. The third line requires that the returned resource must be of type "Map" and calls the `createQuerySession` method to retrieve the specified resource. When a map has been discovered, `localMap` will reflect the map, 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's security system) or constructed by the device of a nearby responder. The application creates a type session:

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

The type session *prefers* blueprints for the current building over any others. This preference is encapsulated in `MyBuildingMetric`, whose `evaluate` method

assigns "1" to resources in my building and "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.

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 this approach.

Learning about nearby workers' movements: Once the responder has a good picture of her environment, she wants to coordinate with other responders. 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:

```
Specification spec = new Specification();
spec.addConstraint(type, Specification.EQUALS,‘‘Trajectory’’);
Region r = DistanceRegion(100);
Trajectory[] trajectories =
      (Trajectory[])sessionFactory.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. 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 and loads information about the survivor's condition and location into the tag. She then needs to contact some form of evacuation vehicle to transport the survivor to safety. 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:

```
Specification spec = new Specification();
spec.addConstraint(type, Specification.EQUALS, ‘‘Ambulance’’);
Vehicle ambulance = (Vehicle)sessionFactory.createProviderSession(spec);
[transfer information about survivor]
[receive confirmation]
sessionFactory.endSession(spec)
```

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: The previous discussions assume that another application component has made the requested resource available. When an application shares a resource, the resource and its description are placed in a local repository. For example, a first responder creates an instance of the `Trajectory` class (which extends the `Resource` class). As the responder moves, he updates his trajectory, changing the resource stored in the local repository. This change then propagates to a first responder who has requested a group session that monitors other nearby responders.

6 Conclusions

Simplifying the development of pervasive computing applications requires coordination abstractions that succinctly represent the interactions among applications and ubiquitous resources. In this paper, we have defined such a coordination model based on *application sessions* 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 programmers to create complex, adaptive applications. By incorporating a suite of dynamic and adaptive communication protocols, the middleware that supports these session definitions provides 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.

References

1. Roman, G.C., Murphy, A., Picco, G.: Coordination and mobility. In Omicini, A., Zambonelli, F., Klusch, M., Tolksdorf, R., eds.: Coordination of Internet Agents: Models, Technologies and Applications. (2000) 254–273
2. Murphy, A., Picco, G., Roman, G.C.: LIME: A middleware for physical and logical mobility. In: Proc. of ICDCS. (2001) 524–533
3. Julien, C., Roman, G.C.: Egocentric context-aware programming in ad hoc mobile environments. In: Proc. of FSE-10. (2002) 21–30
4. Fok, C.L., Roman, G.C., Hackmann, G.: A lightweight coordination middleware for mobile computing. In: Proc. of Coordination. (2004) 135–151
5. Grimm, R., Davis, J., Lemar, E., MacBeth, A., Swanson, S., Anderson, T., Bershad, B., Borriello, G., Gribble, S., Wetherall, D.: System support for pervasive applications. ACM Trans. on Computer Sys. **22**(4) (2004) 421–486
6. Holder, O., Ben-Shaul, I., Gazit, H.: Dynamic layout of distributed applications in FarGo. In: Proc. of ICSE. (1999) 163–173
7. Ryan, C., Westhorpe, C.: Application adaptation through transparent and portable object mobility in java. In: Proc. of OTM Federated Confs. (2004) 1262–1284
8. Bellavista, P., Corradi, A., Montanari, R., Stefanelli, C.: Dynamic binding in mobile applications. IEEE Internet Comp. **7**(3) (2003) 34–42

9. Klein, M., Konig-Ries, B.: Combining query and preference—an approach to fully automize dynamic service binding. In: Proc. of the Int'l. Conf. on Web Services. (2004) 788–791
10. Handorean, R., Sen, R., Hackmann, G., Roman, G.C.: Context aware session management for services in ad hoc networks. In: Proc. of the Int'l. Conf. on Services Comp. (2005) 113–120
11. Roman, G.C., Julien, C., Murphy, A.: A declarative approach to agent-centered context-aware computing in ad hoc wireless environments. In: Soft. Eng. for Large-Scale Multi-Agent Sys. Volume 2603 of LNCS. (2003) 94–109
12. Saif, U., Paluska, J.: Service-oriented network sockets. In: Proc. of MobiSys. (2003) 159–172
13. Bagrodia, R., Bhattacharyya, S., Cheng, F., Gerding, S., Glazer, G., Guy, R., Ji, Z., Lin, J., Phan, T., Skow, E., Varshney, M., Zorpas, G.: iMASH: Interactive mobile application session handoff. In: Proc. of MobiSys. (2003) 259–272
14. Cole, A., Duri, S., Munson, J., Murdock, J., Wood, D.: Adaptive service binding middleware to support mobility. In: Proc. of ICDCS Wkshps. (2003) 396–374
15. Abiteboul, S.: Querying semi-structured data. In: Proc. of the 6^{th} Int'l. Conf. on Database Theory. (1997) 1–18
16. Bremers-Lee, T., Hendler, J., Lassila, O.: The semantic web. Scientific American **284**(5) (2001) 34–43
17. Christensen, E., Gubera, F., Meredith, G., Weerawarana, S.: Web services description language (WSDL) 1.1 (2001) Current as of 2005.
18. Edwards, K.: Core Jini. Prentice Hall (1999)
19. Cabri, G., Leonardi, L., Zambonelli, F.: MARS: A programmable coordination architecture for mobile agents. IEEE Internet Comp. **4**(4) (2000) 26–35
20. Gelernter, D.: Generative communication in Linda. ACM Trans. on Prog. Langs. and Sys. **7**(1) (1985) 80–112
21. Andrews, G.: Foundations of Multithreaded, Parallel, and Distributed Programming. Addison Wesley (1999)
22. Back, R., Sere, K.: Stepwise refinement of parallel algorithms. Science of Computer Prog. **13**(2-3) (1990) 133–180
23. Malan, D., Fulford-Jones, T., Welsh, M., Moulton, S.: CodeBlue: An ad hoc sensor network infrastructure for emergency medical care. In: Proc. of the Int'l. Wkshp. on Wearable and Implanted Body Sensor Networks. (2004)
24. Julien, C., Venkataraman, M.: Resource-directed discovery and routing in mobile ad hoc networks. Technical Report TR-UTEDGE-2005-01, Univ. of Texas (2005)
25. Johnson, D., Maltz, D., Broch, J.: DSR: The dynamic source routing protocol for multi-hop wireless ad hoc networks. Ad Hoc Networking (2001) 139–172
26. Julien, C., Roman, G.C.: Supporting context-aware interaction in dynamic multi-agent systems (invited paper). In: Environments for Multiagent Sys. Volume 3374 of LNCS. (2005) 168–189