Application Sessions: Conversation Abstractions for Pervasive Computing

Drew Stovall and Christine Julien

The Mobile and Pervasive Computing Group, The Center for Excellence in Distributed Global Environments, The University of Texas at Austin

TR-UTEDGE-2010-014
Application Sessions: Conversation Abstractions for Pervasive Computing

Drew Stovall and Christine Julien
Mobile and Pervasive Computing Group
University of Texas at Austin
{dstovall, c.julien}@mail.utexas.edu

ABSTRACT

Pervasive computing application development demands abstractions that reify the notion of a conversation among distributed entities in dynamic and unpredictable environments. We define the Application Sessions model for representing and managing long-term conversations on behalf of applications. This model has been designed to remove the application developer’s need for intimate familiarity with low-level communication constructs and their behaviors. Instead the developer delegates these concerns to a middleware that embodies the Application Sessions model. This paper derives the model, describes a middleware that implements the model, and demonstrates its use in real-world applications. Ultimately, the Application Sessions model and middleware provide a suite of application-centered abstractions for creating meaningful pervasive computing applications in a simple, intuitive, and expressive manner.

1. INTRODUCTION

Enabling pervasive computing application development requires programming abstractions that elevate the notion of conversations among distributed components. These conversations must satisfy pervasive computing requirements that include the ability to seamlessly respond to dynamic environmental and network conditions and to operate under extreme unpredictability. This paper introduces the Application Sessions model and middleware that formally define conversational semantics for these dynamic environments and create tangible programming constructs that put these abstractions firmly in application developers’ hands.

Consider an intelligent construction site outfitted with a variety of sensing, computing, and communication capabilities (see Section 2). Workers may carry handheld devices that communicate with sensors attached to heavy equipment to warn of dangerous situations. Distributed sensors embedded in structures under construction monitor their stability and the progress of the project. A supervisor monitors the locations of teams for safety or progress. Each of these scenarios, while utilizing the same underlying network of devices, entails a different style of conversation.

Previous work, described in Section 3, has begun to address the need for coordination in pervasive computing. Our Application Sessions model furthers these abstractions by directly and formally defining a conversation using intuitive metaphors that conceal the complexities of establishing and maintaining connections. These metaphors fully embody the application developer’s view of the environment and the application’s coordinating partners in that environment. This shifts the developer’s focus away from the network details and back to problems in its domain of expertise (in the examples above, the requirements of applications in the construction domain). Application developers no longer need to learn the details of discovery and routing protocols, nor how to deal with the arrival and departure of network peers, nor the libraries that implement these capabilities.

Our primary metaphor is the application session, which we define to be a collection of robust logical connections to one or more networked devices over which application data is exchanged. To create the connections that are members of the application session, a set of remote applications, devices, sensors, and actuators (collectively referred to as “resources”) must be selected. We define three components to manage this selection, the resource predicate, resource preference, and session limit. The resource predicate describes how an application defines interesting for a resource. The resource preference describes how an application defines more interesting. The session limit caps the number of resources included an application session. These elements are typically based on the application’s non-functional requirements or dynamic environment characteristics. A fourth component, the application’s connection maintenance strategy, determines how the system manages the potentially dynamic connections to resources. For example, we may or may not wish to reconnect to resources that become unavailable. These components of the Applications Sessions model are formally defined in Section 4. To demonstrate the effectiveness and usefulness of the model, we also describe its realization in middleware (Section 5) and evaluate its success using our example applications (Section 6).

The use of an expressive and abstract model such as the Application Sessions model will make the developing pervasive computing applications a more approachable task. The Application Session model’s explicit separation of concerns both architecturally and organizationally will lead to more robust, more successful, and ultimately more valuable software for pervasive computing environments.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright 200X ACM X-XXXXXX-XX-XX/XX ...$10.00.
2. EXEMPLAR APPLICATIONS

On an intelligent construction site, people and equipment are assembled for a variety of tasks. Both people and equipment are increasingly associated with devices such as handheld computers or embedded sensors. Applications in such environments generally monitor and mediate interactions between devices to improve safety, efficiency, or economy. Some of these tasks involve short duration interactions with a simple device, such as downloading a file or punching a time clock. Other tasks, such as environmental monitoring, require long-lived interactions with multiple resources. We detail three specific applications to serve as examples of the larger body of scenarios of pervasive computing applications.

2.1 Example 1: Monitoring Curing Concrete

As concrete cures, moisture levels must be held within a prescribed range. On intelligent construction sites, humidity sensors are placed in and around the concrete to measure moisture levels, and temperature sensors monitor the heat released. A pervasive computing application for achieving this in a distributed manner monitors curing conditions by periodically collecting values from these sensors and synthesizing the results on a handheld device belonging to a worker or supervisor. It may be necessary to select sensors subject to constraints on a variety of attributes, e.g., the type of data, the location of the sensor, and the proximity of the sensors to each other (for correlating readings).

The results may be passed to software to adjust water valves or notify staff. The readings could also be used by inspectors or insurance companies to ensure compliance, or by architects and engineers to analyze failures. Readings need only be taken every ten minutes or so—a very low sampling rate by networking standards. Given this low rate of acquisition, a constant data stream from the sensors would needlessly consume network resources. Even the maintenance of network routing information between samples may be unnecessary. Instead, a state-less query-and-respond approach periodically contacts each sensor to retrieve its current reading. There is no need to maintain any information about the query after the sensors have responded.

2.2 Example 2: Monitoring Crane Movement

Our second application records usage information from sensors attached to cranes on the site. Once the data is recorded, it can be analyzed off-line to schedule routine servicing and other maintenance activities. This application requires much finer-gained information from the sensors than the previous one. Specifically, this monitor requires a constant connection to each sensor attached to the crane.

The application first selects the sensors attached to a specific crane. It constantly monitors reported values as long as the crane is on the site. In effect, we are carrying on a long-term conversation with these sensors. The length of this conversation may range from hours to days, and even months. Over this time, the connections supporting the conversation are subject to various volatilities. Individual resources may be unavailable for milliseconds, hours, days, or longer. The application must be reconnected to these resources either by short-term route repair or long-term rediscovery.

2.3 Example 3: Monitoring Danger

Safety is an ongoing concern on a construction site. Arrival of new equipment, the progression of the project, changes in codes, and simple changes in perception all contribute to the dynamic definition of “danger”. Some dangerous conditions are explicitly sensed by purpose-built hardware (e.g., volatile organic compound sensors, fire detectors), while other conditions must be inferred through calculations using readings from multiple cooperating sensors (e.g., trucks in each other’s path). Formulas can be set up by the site management or by individuals who wish to define their own “dangerous conditions”. No matter how the dangerous condition is defined, the application must evaluate conditions and prioritize alerts as they relate to the current user.

An application on a user’s device will monitor the status of nearby devices at all times. More specifically, it will monitor devices reporting danger in geographical proximity to the user. Over time, dangerous conditions and the sensors reporting them will come and go. Therefore it is the job of the supporting components to provide mechanisms for retrieving the data that is most relevant to the user at any given time, even as the set of relevant devices changes.

3. RELATED WORK

It has previously been shown that a coordination approach for handling the unpredictability inherent in mobile computing can simplify programming [21]. Several middleware solutions have taken this approach [7, 13, 18] but focus on exchanging data in dynamic conditions and not on generic resource usage in pervasive computing. As pervasive computing has come to the forefront, projects have increasingly provided 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 [8, 10, 22]. These approaches focus on bringing objects closer to clients instead of on mobile clients that require inherently location-dependent resources.

Projects that update bindings between clients and services as processing or environment dictates [4, 16] are closer to our goals. A follow-me session [9] provides constant connectivity to a service by transferring a connection from one provider to another. Context-Sensitive Bindings [9, 20] implement the follow-me session by dynamically selecting resources that match an application’s specification. This decouples implementations from their realizations, and handles many of the migration concerns that apply in applications like our danger monitor. This approach also favors complete transparency and assumes that a binding should always be transferred. It does not support the wide variety of conversations needed for the previous scenarios, nor does it enable conversations between groups of entities simultaneously.

Service Oriented Network Sockets [23] addresses client concerns at the lower level of network sockets as opposed to application objects. The approach assumes the availability of well-accepted service discovery mechanisms that gather all matching services before deciding which to connect to. This can incur significant overhead in networks that are dynamic, large in size, or contain numerous satisfactory services. iMash [3] addresses migration of user connections as services are discovered but does not provide a convenient interface for specifying resources or enforcing conversation semantics. In addition, iMash relies on knowledgeable intermediaries that handle service migrations. Similarly, Atlas [5] uses a central server to mediate transfer of service bindings. Scenes [15] and Network Abstractions [19] provide alternate mechanisms for selecting resources, but they do not
allow clients to specify limits or orderings on the returned resources. In the case that networks implement these constructs to provide discovery, it may be possible to translate our resource predicate into the proper input for these mechanisms. This would be one example of leveraging existing functionality to optimize our middleware’s operations.

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

4. THE APPLICATION SESSIONS MODEL

From an architectural perspective, our model sits directly below the domain-specific application and just above the discovery and routing protocols used to find and access remote resources. In implementation terms, we replace the use of sockets, RMI, and other solutions with one that directly addresses the challenges of pervasive computing. In this section, we define the operational semantics of our model, which demonstrate the model’s ability to transparently adapt its functionality to a mix of application requirements and operational environments.

4.1 Resource Model and Specifications

We provide a model of resources that stipulates how our components interact; it should not be construed to imply a complete model for resource implementations. This model seeks to declare a minimal interface that may be expanded by individual providers to support other technologies.

A resource is a set of (potentially nested) values. Each value as an associated key, or name, that describes the nature of the value and a type that describes the representation of the value. The name of a value is a character string, while the type is an element of some large set of enumerated available data types. Like a program variable, values are shared between processes and may change at any time. This results in a semi-structured data model [1], leaving the definition of value dependence and interaction to the applications.

Obviously there is a need for standardized names, types, and units to ensure unambiguous definitions; it is necessary for users to share an understanding of how data will be named. Distributing shared naming schemes in dynamic environments is a well-studied problem and not one we intend to undertake here. Instead, we simply assume the presence of this shared knowledge and use it to semantically tag data.

Resource Predicate Function. An application provides a resource predicate that filters available resources to meet the client’s requirements. This predicate, \( \text{pred}(x) \), is a conditional statement that evaluates to true or false for a given resource \( x \). The \( \text{pred} \) definition prescribes the minimal acceptable characteristics of the target resources. A predicate typically includes the type of resource (e.g., thermometer for the Curving Concrete task) but may also filter on attributes such as the resource’s owner or location.

When supported by discovery mechanisms, \( \text{pred}(x) \) can also specify connection or environmental attributes [11, 12, 14]. The client could specify a “maximum hop-count” to prevent protocols from searching distant areas of the network. Even though this is associated with the connection between two hosts, it can be modeled as attributes of each resource as viewed by the client. Metrics such as latency and bandwidth can also be measured for remote resources.

Resource Limit. Since we are designing a model for pervasive computing, the scope of a client’s request must be limited to avoid overloading the network, the hardware, or the client application. We include a second parameter, the resource limit to explicitly limit the number of resources that should be provided by the infrastructure for a particular session. While the primary motivation for specifying a limit is to restrict the load imposed on the network and remote resources, it can also be used to intimate application limitations. For example, the small screen size of handheld devices may limit the amount of data that can be presented.

Resource Preference Function. The predicate function determines which available resources are acceptable. However, we would like to return the best possible results. This is particularly important when the client has specified a resource limit smaller than the number of available acceptable resources. We must make some decision as to which connections are established and which are omitted. Simple algorithms would include non-deterministically choosing resources or choosing the first \( n \) resources returned by the discovery protocols. These methods serve as the default approach for choosing between otherwise equal resources. However, we allow the client to explicitly dictate a selection algorithm through the resource preference function.

This function allows the model (and compatible discovery protocols) to reduce and sort the set of resources returned. Specifically, the model can return an ordered list of resources that are “better” than all other available resources, where “better” is defined by the client. For example, the client application in the Danger Monitor views resources that are geographically closer to the user as “better”. We define the resource preference function, \( \text{pref}(x, y) \), to provide this functionality: (1) \( \text{pref}(x, y) > 0 \iff x \) is “better” than \( y \); (2) \( \text{pref}(x, y) < 0 \iff y \) is “better” than \( x \); and (3) \( \text{pref}(x, y) = 0 \iff \) no preference.

As we discussed above, discovery and routing protocols may be able to calculate connection-specific information. Since we model these values as attributes of the resources, they can also be used for preference functions. For example, a client could use hop-count to limit the scope of discovery to nearby resources and then use latency in a preference function to select the “quickest” resources.

Resource Selection. Using the predicate \( \text{pred}(x) \), preference \( \text{pref}(x, y) \), and a resource limit \( (n) \), we can create a function \( \text{findBest} \) that returns a list of resources ordered by preference. To support the connection maintenance strategies discussed below, the list of resources returned is padded with null values \( \emptyset \) to the size specified by \( n \).

\[
\text{findBest}(\text{pred}, \text{pref}, n) = R[0..(n-1)] \in \{
\forall r \in R :: r\text{reachable} \land \forall r' \in R :: \text{pred}(r') \land
\forall r \in R, r' \notin R, r'\text{reachable} :: \text{pref}(r) \geq \text{pref}(r') \land
\forall i \in 0..(n-2) :: \left(R[i + 1] = \emptyset \right) \lor \left(\text{pref}(R[i], R[i + 1]) \geq 0\right)
\}
\]

\[1\]The construct \( x.(\text{condition}) \) non-deterministically returns any value that matches the \( \text{condition} \) and will return \( \emptyset \) (null) immediately if no value can be found.
This function returns a list of reachable resources $R_\text{...}$ of size $n$, all of whose members satisfy the predicate; the members are ordered according to the preference function. The exact order may be non-deterministic since resources the preference function considers equal can be in any order.

### 4.2 Connection Maintenance Strategies

Using the specifications above (predicate, limit, and preference) and the $\text{findBest}(\text{...})$ function, the model can now select an initial set of resources to satisfy a client's request. However, in a pervasive computing environment, the client's conversations may last longer than the underlying best-effort connections can support, especially in physically dynamic environments. Rather than using the results of $\text{findBest}(\text{...})$ directly, applications create a session (via a function $\text{newSession}(\text{...})$) and interact with the returned session in much the same way as a list. We adopt the notation $\text{findBest}(\text{...})$ to denote the $i^{th}$ element of the session $S$. Once an application has created a session, each of the session’s elements provides a connection to a remote resource.

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

Our model allows the client to select one of three connection maintenance strategies. Below we describe each strategy, define its invariant, and give an operational definition that satisfies the invariant. Since the invariants and definitions below describe the state of the session over time, we must introduce a bit of notation: a subscript attached to $S$ denotes the logical time of the session. Thus $S_0$ refers to the session at time 0 (the instant it is created) and $S_τ$ refers to the same session at time $τ$. Therefore $S_τ[5]$ designates the fifth element of session $S$ at time $τ$.

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

To make the strategies concrete, we provide a graphical representation of the short-term network connections with each description. We depict each session at three different (logical) times with the three different network configurations shown in Figure 1. At time 0, all nodes are reachable. At time 1, six of the nodes (in the shaded region) have become disconnected, and the left-most node has changed values from 42 to 200. At time 2, two of the disconnected nodes have rejoined, the left-most node has reverted its value to 42, and another node has changed values from 81 to 72.

**Figure 1: Example network configurations**

To emphasize the distinctions between the strategies, the examples below pass the same parameters (excepting the strategy) to $\text{newSession}(\text{...})$: the predicate selects only circle-shaped resources, the preference function favors larger values, and the session is limited to six resources. Using these values, the initial resources selected for each session are the same; this initial selection is shown in Figure 2.

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

Figure 3 depicts this interaction over the network configurations discussed above. This figure shows the connections to $S[3]$, $S[4]$, and $S[5]$ are released when the associated resources leave the network at time 1. Even when one of those resources rejoins, the connection is not reestablished. The same connection release can be seen as the client uses the remote resources $S[0]$ and $S[1]$ at times 0 and 1, respectively.

**Figure 2: Initial resource selections for all sessions**

**Figure 3: Resource assignments for query session**

The formal definition of the fundamental invariant of the Query Strategy is shown in Figure 4. $Φ(t)$ serves only to simplify the rest of the definition; it returns the indices of $S$ that hold resources that were either used or unreachable at some time $τ$ between 0 and $t$. In other words, $Φ(t)$ returns indices of resources that should be disconnected. The Query Strategy Invariant says that at time 0, the session $S_0$ is equal to the results returned by $\text{findBest}(\text{...})$. At time $τ$, the session elements that were assigned to used and unreachable resources (indicated by $Φ(t)$) should contain the null value ($\emptyset$). The elements representing unused and still reachable resources still contain the original connections.

To show how this invariant can be satisfied in practice, we present an operational model of the Query Session Strategy in Figure 5. Here $S[...]$ is initially (atomically) assigned the values

**Figure 4: Query session invariant**

0 1 2 3 4 5
42 48 56 64 72
90 96 84 82 80

0 1 2 3 4 5
90 96 84 82 80
48 42 56 64 72
90 96 84 82 80
48 42 56 64 72

We provide an operational model in Figure 5 to show how the application of this strategy effects resource connections. The initial connections are shown in Figure 2. At time 1, the session’s connections are broken. However, the session reconnects to one of the original resources (node with value 77) when it becomes available again at time 2.

We provide an operational model in Figure 8 to show how the application of this strategy effects resource connections. The initial connections are shown in Figure 2. At time 1, the session’s connections are broken. However, the session reconnects to one of the original resources (node with value 77) when it becomes available again at time 2.

Figure 6 continues our running example by showing how the application of this strategy effects resource connections. The initial connections are shown in Figure 2. At time 1, when three of the originally included resources are no longer available, the session’s connections are broken. However, the session reconnects to one of the original resources (node with value 77) when it becomes available again at time 2.

Figure 7 states the Provider Session Invariant. At time 0 ($S_0$) the session is equal to the results of $\text{findBest}(..., \Gamma)$. At time 1, the session’s connections are broken. However, the session reconnects to one of the original resources (node with value 77) when it becomes available again at time 2.

We provide an operational model in Figure 8 to show how the application of this strategy effects resource connections. The initial connections are shown in Figure 2. At time 1, when three of the originally included resources are no longer available, the session’s connections are broken. However, the session reconnects to one of the original resources (node with value 77) when it becomes available again at time 2.

We provide an operational model in Figure 8 to show how the application of this strategy effects resource connections. The initial connections are shown in Figure 2. At time 1, when three of the originally included resources are no longer available, the session’s connections are broken. However, the session reconnects to one of the original resources (node with value 77) when it becomes available again at time 2.

Figure 9: Resource assignments for Type session

These updates ensure that references are connected to the
session elements in order of decreasing value. We express this formally as the Type Strategy Invariant (Figure 10), which allows the client to iterate through the session elements to find the best resources at any given time.

**Type Strategy Invariant**

\[
\forall i, j, t : 0 \leq t : S_t = \text{findBest}(\text{pred}, \text{pref}, n)\]

Figure 10: Type session strategy invariant

Figure 11 gives an operational definition for the Type session that satisfies the invariant. The simplest way to model a Type strategy is to periodically update the entire list with the results of a full reevaluation of available resources (i.e., findBest(...)). We avoid needlessly reevaluating findBest(...) by waiting until there is a change that could affect the session. Specifically, we wait for any of three conditions: a change in the value of a resource in the session, the departure of a resource in the session, or the arrival of a resource not in the session.

\[
\langle S[i] \rangle \leftarrow \text{findBest}(\text{pred}, \text{pref}, n)\]

co

\[
\langle \text{wait}(S[i].\text{valueChange}) \Rightarrow (S[i] = \text{findBest}(\text{pred}, \text{pref}, n))\rangle \]

\[
\langle \text{wait}(\neg S[i].\text{reachable}) \Rightarrow (S[i] = \text{findBest}(\text{pred}, \text{pref}, n))\rangle \]

\[
\langle \text{wait}(r \notin S[i].\text{reachable}) \Rightarrow (S[i] = \text{findBest}(\text{pred}, \text{pref}, n))\rangle \]

Figure 11: Type session operational model

The Type Strategy is a natural implementation path for the danger monitor example. The original sessions would choose the closest resources that represent dangerous conditions. As network conditions fluctuate and sensor values change, the list of resources is constantly updated to present the application with the most current readings.

## 5. A MIDDLEWARE DESIGN

We now describe a middleware based design to support the Application Sessions model. The middleware architectural pattern simplifies the details of distributed communication and coordination using high-level primitives presented to the application developer [6]; in this case these primitives are the primary components of the Application Sessions model described in the previous section. A common goal of distributed computing middleware in general, and mobile computing middleware specifically, is to hide the distributed nature of the environment as much as possible and create what appears to be a “single integrated computing facility”[17]. However to enable expressive use of resources in pervasive computing environments, we hide only the mechanics of the distribution and present the potentially interesting properties of the environment, including its many independent nodes, to the application programmer as cleanly and clearly as possible. We feel that this enables more control and flexibility in developing applications. Since our middleware design directly reflects the model, our discussion mirrors the organization of Section 4.

### 5.1 Resource Model and Specifications

In a middleware design, resources must be modeled with a uniform interface that balances application needs and the discovery and routing protocols’ typical capabilities. We model resources as sets of (key, type, value) triples. However, our design is targeted for a strongly typed language, allowing us to infer the type at runtime from the value of each tuple. This reduces resources to a set of (key, value) pairs, a very common structure referred to as an “associative container” or (more commonly) a “Map”. A resource’s value is typically accessed by resource.get(key), and the type can be inferred by resource.get(key).getType().

This brings us to the problem of acquiring or discovering the resource. Since most resources are accessed through a network, an application generally interacts with local proxies that forward messages to remote resources. Each proxy’s implementation is specific to the particular combination of network and discovery protocols used to deploy the resource. To unify this diversity, we define a Discovery interface that separates the deployment-time specifics of protocol interactions from the development-time concerns addressed by our middleware. Implementations of the Discovery interface provide a mechanism to supply resources (local, proxied, or otherwise) through the method Discovery.find(...).

In addition to providing resources on-demand, many discovery and routing protocols also provide support for asynchronous notification of resource arrivals, departures, and updates to remote values. A client typically registers a listener with the discovery mechanism or with the resource itself. Our design provides a DiscoveryListener interface used by Discovery implementations to make these notifications. Like the Discovery interface, this interface provides a single coordination mechanism between the middleware and the discovery protocols available at a specific deployment. The notification events passed to the middleware are used to maintain the session invariants. When direct support for listeners is not provided by a discovery protocol, it can be approximated by a Discovery component by periodically polling of the network’s membership and values.

**Resource Predicate Specification.** The resource predicate acts as a filter to limit resources considered for inclusion in an application session. Traditional network applications use unique identifiers to limit the potential resources. In pervasive computing networks, these ids are not always well known, and often we rely on attribute-based descriptions of resources. Some discovery and routing protocols support this type of selection, but ultimately it is the responsibility of the middleware to ensure the resources returned to the client satisfy the predicate. The simple language defined here is designed for use by our middleware, but it may be directly translatable to some discovery protocols. Furthermore, we expect the development of more elaborate structures to support more elaborate formulas as the Application Sessions model is field tested.

Figure 12 shows our resource predicate language’s grammar. We define two comparison methods (within U), one for each of the allowed value types in our simple tuple-based model. To limit a resource based on a numerical value, the programmer needs only to specify an attribute name (A), a comparison operator (O), and the target value (V). For example, the specification “\( \text{altitude} > 2000 \)” will restrict resources to those that have a numerical value for altitude that is greater then 2000. To limit a resource

\[
A \rightarrow \text{attribute name} \quad V \rightarrow \text{value} \\
O \rightarrow \text{== | != | > | >= | < | <=} \\
T \rightarrow \emptyset | T \cdot T | V \\
U \rightarrow (A \cdot O \cdot V) | (A \in (T)) \\
S \rightarrow \emptyset | U \cdot (S \text{ and } U) \mid (S \text{ or } U)
\]

Figure 12: Predicate specification grammar
by a character string value, the in operator is used with a set of possible values. For our concrete curing example, the programmer would use a predicate of "type in {"humidity"}" to limit the search to resources whose type attribute has the string value "humidity". We anticipate the need for other operators and other value types in the future (for example the not-in operator and regular expressions), but limit ourselves to keep the prototype simple.

Value restrictions can be combined using logical and and or operators, which are subject to standard rules for order of operations, allowing a full range of predicates to be formed. Nested expressions are not yet supported ("(A or B) and C") would have to be expanded to "A and C or B and C"), but this will be incorporated into future versions.

Resource Preference Specification. The resource preference language is very similar to the predicate language. It too is specified with limited capabilities while the design is developed and will be extended as experience guides us.

As with the predicate language, the preference specification may be passed along to underlying technologies if they support discovery based on application-level concerns. The language in Figure 13 is made up of a series of attributes, which are compared, in order, until a preference for one resource is established. Since some attributes are “better” when they are larger (e.g., bandwidth) and some attributes are “better” when they are smaller (e.g., latency), each attribute is annotated with the sorting parity to be used (indicated using a plus (+) or minus (−) sign after the attribute name). The specification (latency+, bandwidth−) would sort resources that are quicker to respond (smaller latency) to the first positions and, in the case of equal latency, resources with larger bandwidth will be chosen first.

Resource Selection Algorithm. The application can use predicates and preferences to describe its definition of “best” to the middleware. To describe findBest(...), we begin with a basic case and build on it to incorporate more complex configurations. Let us first assume that all regeneration with a basic case and build on it to incorporate more

resources with larger bandwidth will be chosen first. Let us first assume that all resources are local with respect to the middleware. That is, all resources are components in our memory space, and we can retrieve references by calling Discovery.find(...). Given this, the “best” resources are selected by applying the predicate to filter the resources, applying the preference specification to order the selected resources, and then pruning the remaining resources to the right number. This process is outlined in Figure 14. In this figure, the parameters predicate and preference are executable versions of the predicate and preference specifications provided in the languages described above, and maxSize is the resource limit.

To remove our assumption of local resources and use resources provided over a network, we must consider the performance implications of this algorithm. Many discovery protocols support some of the selection features required by the model. For any discovery mechanism used by the middleware, there are three basic cases to consider:

- Full Support: When Discovery directly supports the use of predicate, preference, and limits, findBest(...) delegates entire requests to Discovery’s find(...) method. Returned resources are simply wrapped by proxies to provide the Resource interface.

- Partial Support: If Discovery supports only some of the selection behavior (e.g., predicates but not preferences), the remaining functions must be implemented locally. By wrapping the resources returned by Discovery after applying the selections that are supported, the ResourcePredicate and/or ResourcePreference can be applied to complete the process. In this case, information about resources that the client may not be interested in is returned to the client’s device, resulting in an increase in network overhead.

- No Support: When Discovery does not support any selection of resources, all remote resources must be wrapped with a proxy providing the Resource interface and passed through the basic algorithm shown in Figure 14. A naive proxy that forwards individual requests for resource values could easily lead to large amounts of network traffic as sessions are created.

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

Regardless of the Discovery implementation, the middleware may also supply a DiscoveryListener to monitor changes in the network. The middleware can be notified when there are changes in the network that potentially invalidate the “best” property of the returned list.

5.2 Session Creation and Maintenance

In this section we present the analog to the model’s session S[...]. the Session interface. The Session’s get(...) method returns an object supporting the Resource interface described in Section 5.1. The session’s size() method returns the size limit defined when the session was created. If there are fewer resources then the session’s limit, the session is padded with null values.
Each of the middleware’s strategy components calls on `findBest(...)` to provide the initial resources. Continuing with the operational model, the middleware then updates these assignments to ensure the strategy’s invariant. The component maintaining the assignments must be able to intercept both the application’s use of resources and any connectivity or value changes in the networks. By implementing both the `Session` and `DiscoveryListener` interfaces, a single component can receive all the events that trigger the `await` conditions of the operational models. The remainder of this section is devoted to summarizing how we implement each session in our middleware. For brevity, synchronization and mutual exclusion concerns are omitted from this discussion. In Section 6 we will present some details of an actual implementation of this design. The source code for the evaluation implementation [24] does provide the necessary guarantees and can be consulted for more details.

**Query Session.** In this section we describe the `QuerySession` component, which implements the query session operational model and guarantees the query session invariant. A pseudo-code version of the `QuerySession` is shown in Figure 15. When a new instance is created, the results of `findBest(...)` are stored in an internal list, `resources`. The client’s access to these resources is mediated by the `Session.get(i)` method. When the client invokes the `get(i)` method, the content of the `resources` list at index `i` is returned and replaced with the `null` value. This behavior implements the first `await` statement in the operational model. The second `await` statement is triggered by calls to the `onDisconnect()` method inherited from the `DiscoveryListener` interface. When this event is fired, the appropriate element of the `resources` array is set to `null`.

**QuerySession**

```
List resources;
init(...) {
    resources.addAll(findBest(...));
}
get(int i) {
    tmp = resources.get(i);
    resources.set(i,null);
    return tmp;
}
onDisconnect(resource) {
    i = resources.indexOf(resource);
    connected.set(i,true);
}
```

**Figure 15: Pseudo-code for query session**

**Provider Session.** The `ProviderSession` component’s implementation is drawn directly from the operational model for the session strategy of the same name. An outline of its implementation is shown in pseudo-code in Figure 16. As with the `QuerySession`, we initialize an internal list of resources with the results returned by `findBest(...)`. At the same time, we also create a list of boolean variables `connected` to represent the “connected-ness” of the resource with the same index in `resources`. When clients access the session’s elements via `Session.get(i)`, the status in `connected` is checked. If the resource is connected, it is returned from the `resources` list. Otherwise, `null` is returned. The values in the `connected` list are maintained by implementing the `DiscoveryListener` interface. When resources are connected to, or disconnected from the network, the matching value in `connected` is updated to reflect the new status.

**ProviderSession**

```
List resources;
List connected; // list of booleans
init(...) {
    resources.addAll(findBest(...));
    connected = new List(resources.size());
    connected.setAll(true);
}
get(int i) {
    if (connected.get(i)) {
        return resources.get(i);
    } else {
        return null;
    }
}
onConnect(resource) {
    i = resources.indexOf(resource);
    connected.set(i,true);
}
onDisconnect(resource) {
    i = resources.indexOf(resource);
    connected.set(i,false);
}
```

**Figure 16: Pseudo-code for Provider session**

**Type Session.** As with the other `Session` components, we begin our pseudo-code implementation of the `TypeSession` in Figure 17 with a call to `findBest(...)`. However, in this invocation, we ignore the session limit specified by the client and greedily select as many resources as are available. Then we use the implementation of `get(i)` to restrict the client from accessing elements of the `resources` list with indices larger than the size limit. When a resource is removed from `resources`, the remaining elements are still in order, and the `get(i)` method still returns the correct resource for each index. The implementation of the third `await` statement in the operational model, implemented via the `onDisconnect` method, relies on this property.

The second `await` statement of the operational model (Figure 11) is implemented by `onConnect`. Here the resource is checked against the session’s predicate and added to the internal list, which is then sorted. The final `await` statement is implemented in the `onValueChanged` method and blends the techniques used by the other methods. If a value change causes a remote resource to be included, then it is added to the internal list, `resources`. If it is already there, this addition is skipped; either way the list is re-sorted to maintain the invariant. If the resource fails the predicate, it is simply removed from the internal list.

### 6. EVALUATION

To evaluate the Application Sessions model’s usefulness, expressiveness, and applicability to real-world pervasive computing applications, we implemented the model using the middleware design described in the previous section. We then created the example applications described previously.

**Scenario Setup.** We created a suite of simulated sensors on a hypothetical construction site. We defined five sensors that provide varying types of data for the applications: temperature and humidity sensors that return real valued environmental readings, a sensor to detect the presence of volatile organic compounds, and two sensors attached to a crane to report “in motion” and “carrying a load.” Each
TypeSession implements Session, Listener

List resources;
init(...) {
    resources.addAll(findBest(...));
}
get(int i) {
    if (i < min(resources.size(), maxNum)) {
        return resources.get(i);
    } else {
        return null;
    }
}
onValueChange(resource) {
    if (predicate.evaluate(resource)) {
        if (! resources.contains(resource)) {
            resources.add(resource);
        }
        sort(resources);
    } else {
        resources.remove(resource);
    }
}
onConnect(resource) {
    if (predicate.evaluate(resource)) {
        resources.add(resource);
        sort(resources);
    }
}
onDisconnect(resource) {
    resources.remove(resource);
}

Figure 17: Pseudo-code for Type session

sensor also reports its longitude, latitude, and altitude. To roughly model an intelligent construction site, we created a grid with 25 of each sensor as shown in Figure 18.

Figure 18: Sensor locations on hypothetical site

The locations for each sensor remained constant while the other data provided by each sensor is generated using time-varying periodic functions that were chosen to demonstrate the model and were not meant to precisely model events on a construction site. The sensors also experience variations in connectivity. In our hypothetical scenario, each sensor is disconnected from the network 20% of the time.

Implementation details for each scenario and session, screen shots of the applications running on our test network, and detailed descriptions of how various factors impact the sessions and applications are available online at [24].

Experiment 1: Query Session. The concrete curing monitoring application periodically requests temperature and humidity values from a specific region in space. Using the Application Sessions model to perform each request, we repeatedly establish a session using the code in Figure 19 and pass the results to a user interface. The predicates specify a restriction for sensors with longitude > 8. The preference function for the humiditySession (humidity+) prefers sensors with lower humidity, while the preference function for the temperatureSession (celcius-) prefers sensors with higher temperature. In both cases, if there is a problem, these readings will be the indicators.

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

Figure 19: Establishing a Query session

Experiment 2: Provider Session. In our second experiment, the crane monitor application retrieves values from sensors periodically. In comparison to the concrete curing application, the interval between samples is much shorter, and the application’s time frame much longer. Therefore, we implement our crane monitor using the Provider session; the pseudocode is given in Figure 20. By using the exists(...) construct, the predicates in Figure 20 select resources explicitly by an attribute they provide. The preference function sorts the sensors from north to south and then east to west.

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

Figure 20: Establishing a Provider session

Experiment 3: Type Session. Our final experiment implements the danger monitor application using a pair of Type sessions—one that monitors VOC detectors and another that monitors the movement of nearby cranes. We establish both sessions with a predicate to select sensors in the general vicinity of the user, as depicted in Figure 21. Here, the first four terms of the predicates select sensors within five units of latitude and longitude from the user (who, in this example, is located at 3 units of latitude and -3 units of longitude). The last term in each predicate selects only sensors reporting conditions of which the user needs to be aware (where danger and moving are the types of VOC sensors and crane motion sensors, respectively). In this case,
the preference function is left blank.

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

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

Figure 21: Establishing a Type session

7. CONCLUSIONS

The correct programming abstractions will ease the software development transition from traditional networks to pervasive environments. This will in turn drive more adoption of pervasive networking technologies and applications, leading to a better understanding of user needs and technology utility and ultimately add more value to the field. Presented here, the Application Sessions model has been developed to provide this support. We have also presented a design for implementation of this model as a middleware. Both the design and the model have been evaluated through their application to scenarios found on the Intelligent Construction Site. This evaluation has shown that the model and design are both feasible and appropriate abstractions for pervasive computing environments.

Acknowledgments

The authors would like to thank the Center for Excellence in Distributed Global Environments for providing research facilities and the collaborative environment. This research was funded in part by NSF Grant #CNS-0626777 and AFOSR Grant #FA9550-07-1-0157. The views and conclusions herein are those of the authors and do not necessarily reflect the views of the sponsoring agencies.

8. REFERENCES