Providing Contextual Information to Ubiquitous Computing Applications

Glenn Judd Peter Steenkiste

July 2002
CMU-CS-02-154

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Ubiquitous computing applications are increasingly leveraging contextual information from several sources to provide users with behavior appropriate to the environment in which they reside. If these sources of contextual information are used and deployed in an ad hoc manner, however, they may provide overlapping functionality, fail to provide needed functionality, and require the use of inconsistent interfaces by applications. To overcome these problems, we introduce a concise organization of services and a single service interface that provide applications with contextual information in a unified manner. We show, via example applications and services that we have implemented, how our service organization and interface can be used to allow proactive applications to adapt their behavior to match a user's current environment.

This research was sponsored in part by the Defense Advanced Research Project Agency and monitored by AFRL/IFGA, Rome NY 13441-4505, under contract F30602-99-1-0518. Additional support was provided by Intel. Glenn Judd is supported by a DoD National Defense Science and Engineering Graduate (NDSEG) Fellowship.

Keywords: pervasive computing, context-aware

1 Introduction

Fueled by advances in processing capability, storage capacity, and battery life, the proliferation of mobile computing devices is rapidly turning the focus of computing away from desktop personal computers and towards a collaboration between several mobile devices, traditional personal computers, and servers. Unfortunately, as the usage of mobile computing devices has increased, the amount of user effort required to operate these devices has increased as well.

As part of the Aura Project [1] at Carnegie Mellon University, we are investigating how applications can proactively adapt to the environment in which they operate to provide users with more intelligent application behavior, thus allowing users to focus on higher level tasks.

To provide adaptive applications with environmental information, we have developed a set of basic "contextual services". These contextual services provide applications with properties of both physical entities and available resources such as: the location of people, the location and properties of printers, the amount of network bandwidth available, the CPU load on various servers etc. Applications communicate with these contextual services via an SQL-like query interface that allows applications to easily synthesize information from several different sources. Synthesizing this information allows applications to adapt to environmental and resource changes without user intervention.

To show how contextual services can enable proactive and adaptive applications, we will consider two simple motivating examples. In the first example we will consider a user, George, who is giving a presentation at a meeting with a remote participant. Contextual services will allow George to perform tasks such as selecting a conference room with both a video projector and enough wireless bandwidth for videoconferencing. They will also allow him to discover the whereabouts of late participants to the meeting. In the second example, we will show how contextual services can assist a user, Jane, who has demanding network bandwidth requirements in a bandwidth scarce environment. In this example, contextual services will allow Jane to move to a location where her bandwidth demands can be met.

Clearly, contextual services can provide useful information. However, designing a single interface for accessing contextual information is a difficult task. Consider some of the requests that might be used to implement the scenarios described above:

- What devices are in room 160?
- What is the expected bandwidth in room 160 between 2pm-3pm tomorrow?
- Where, within 300 meters of my current location, am I likely to find the best wireless network bandwidth in the next 5 minutes?

In addition, since we desire to support a wide variety of applications beyond the given scenarios, we consider a wide variety of contextual information requests such as:

- What devices does John currently have with him?
- When will network bandwidth be best, within the next hour, to flush my distributed file system's cache?
- Where has John been today?

- What is the compute server's load likely to be in the next minute?
- Inform me whenever John moves more than 50 meters.
- Where is the closest color printer with an empty print queue?

A straightforward approach to providing the information desired in the requests listed above is to write custom services, as needed, to provide the requested information. Unfortunately, using such an ad hoc approach will result in multiple services with multiple interfaces and inefficient design. Even services that function well individually could become fractured and inefficient when deployed and used in conjunction with other contextual services. Without a rationale for designing services, functionality may overlap and compete with other services. Without a common service interface, developers must learn multiple interfaces, and clients must include runtime support for multiple interfaces.

To avoid such problems, contextual services must be developed with coherent and focused functionality. In addition, there must be a single interface to all services to provide a consistent mechanism for client access, eliminate redundant client code, and aid deployment of new services.

We propose an organization of contextual services that allows them to be developed cleanly, and we propose an interface that allows a wide variety of contextual services to be accessed. Using our service organization and Contextual Service Interface (CSI), we have developed a set of contextual services that support proactive and adaptive applications such as those mentioned in the two examples mentioned earlier.

Our discussion will proceed as follows: Section 2 will outline our service and service interface architecture. Section 3 will outline service interface requirements. Section 4 will introduce major CSI functions. Section 5 will discuss alternative methods of CSI implementation. Section 6 will discuss our CSI implementations. Section 7 will give an overview of services that we have deployed. Sections 8 and 9 will discuss example application implementations. Section 10 will give an overview of related work, and Section 11 will conclude our discussion.

2 System Architecture

2.1 Service Architecture

A key difference between traditional computing applications and ubiquitous computing applications is the need for ubiquitous computing applications to understand the physical environment in which they reside. We argue that providing information on the aspects of the physical environment that are most relavant to mobile applications can be accomplished by providing information on entities that can be grouped into a small number of classes.

In particular, our architecture provides information on four classes of entities: people, devices, physical spaces, and networks. While we hold open the possiblity of adding new classes of entities, we intentionally construct our architecture to contain as succinct a representation of the world as possible. We discuss each entity class briefly, noting alternative classes that could be considered.

Clearly ubiquitous applications will need information on people, devices (e.g. printers), and physical spaces; hence, we define a service class for each of these. Arguably, we could have defined classes for generic physical objects (e.g. tables) and a class for vehicles. For now, we choose to

reduce the number of entity classes in our model by treating physical objects as ”dumb” devices, and vehicles as spaces without fixed location in the world. We could also have defined a class for power sources; however, as a power source essentially amounts to either an electrical outlet in a physical space or a battery on a device, we treat these sources as attributes of physical spaces and devices respectively.

As the behavior of many applications is tightly coupled to the ability or inability to communicate, and available communication varies greatly with location, we introduce a networks class to provide communication information to applications that require it. This is needed, for instance, if Jane desires to know a nearby location where she can download a large multimedia file quickly, or where on her business trip she will be entirely out of network range.

In addition to defining service classes for each of the entity classes listed above, we define a service class that tracks the relationship between each pair of service classes as shown in Figure 1 (with the sole exception of people and networks since people are not usually directly related to networks, but use them indirectly).

There are several other potential classes of entities that we could have considered. For instance, we could have defined classes for weather or living things. As the need for additional classes is not clear, they are currently omitted from our architecture.

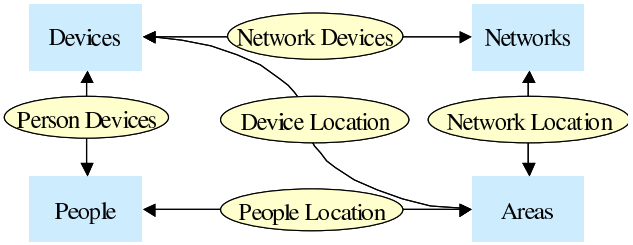

Figure 1: Entity and Entity Relation Classes

2.2 Service Interface Architecture

Ideally, from an application’s point of view, contextual services would accept arbitrary queries for contextual information without burdening the application with the details of how the information was obtained. Though implementing this idealized model is not feasible, we use it as a starting point in developing an architecture that enables applications to easily synthesize environmental information.

Our system provides applications with a database-like abstraction of contextual services. Under this scheme, clients issue queries, the queries are decomposed to operate on individual services, and results are synthesized and returned to the application. For services that contain relatively static data, this scheme allows for straightforward implementation using a database. For services that provide dynamic information, a database may not be an appropriate implementation. Using a database-like interface, however, provides applications with a consistent interface to these dynamic services.

To illustrate how our interface simplifies communication with both services that statically store data and those that actively compute the results to queries, consider figures 2, 3, and 4. Under

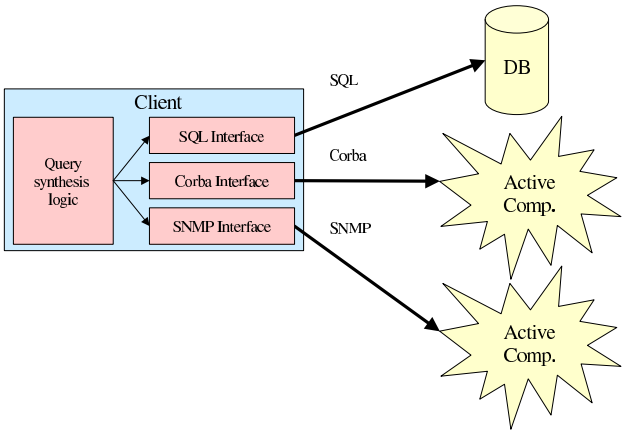

Figure 2: Traditional Client-Service Interaction

the traditional model (shown in Figure 2) synthesizing information from several services requires individual communication with each service using multiple incompatible interfaces. Applications must include support for these interfaces and know how to synthesize the information returned.

As show in Figure 3 our interface allows applications to use a single interface to retrieve information from both services that are implemented as a database as well as services that actively compute the results of queries. (A simple wrapper allows our interface to query SQL databases without any additional coding.)

Finally, to remove the need for clients synthesize query results from multiple services, we introduce a proxy (Figure 4) that can decompose queries sent from clients and synthesize the results before returning them to clients. This enables client applications to focus on the information that they desire, and reduces the need to worry about how contextual information is retrieved or how expensive it is to compute. Thus extremely thin clients can issue rich queries without incurring the large communcation and computational expenses of such queries.

3 Requirements of the Contextual Service Interface

Using a single interface for all contextual services as we propose imposes several demands on the interface. In this section, we discuss several requirements and design guidelines of the Contextual Service Interface. Once we have established the requirements of the Contextual Service Interface, we will evaluate several possible implementations of the interface.

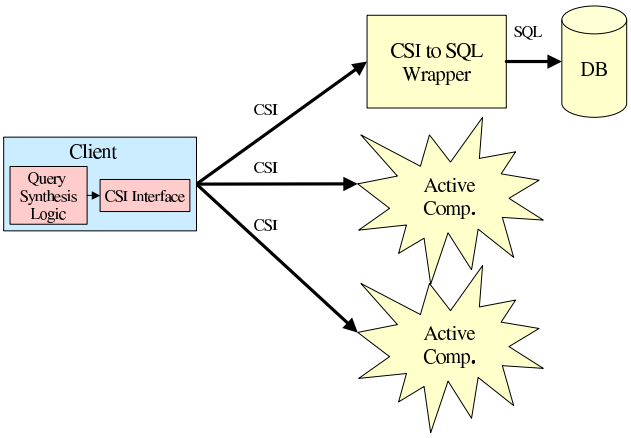

Figure 3: CSI Client-Service Interaction

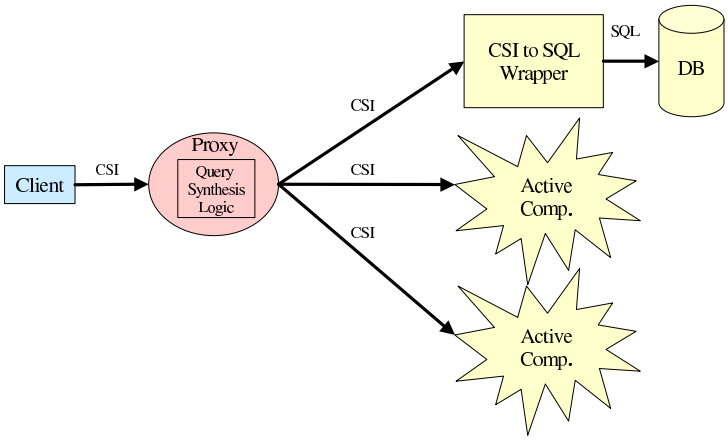

Figure 4: Proxy-Based CSI Client-Service Interaction

3.1 Support for Multiple Implementations

In many instances, the data supplied by contextual services will be stored statically by services. In these cases the most convenient implementation is likely to be a database. For instance, information regarding building layout would likely be stored in a database. Since many services will store static data, the service interface should cleanly support service implementation via a database.

In other cases, services may desire to always actively compute the answer to a query. For instance, a person location service may desire to only compute the location of a person when a query is received. In cases such as these, we should be careful not to force these services to use a database since this could severely bloat services that would otherwise be very light.

In addition, services that actively gather data may desire to cache some amount of information. Client-side libraries used to communicate with contextual services and the aforementioned proxies may also desire to cache query answers. Hence, we should design the interface in a way that allows caching at any stage shown in Figure 4. (This is particularly important when considering support for dynamic attribute requirements as discussed in the next section.)

Finally, the service interface must also allow clients to issue queries that span multiple services. This is critical in order to allow either a proxy or services themselves to perform the work of synthesizing the result from multiple logical queries.

3.2 Support for Dynamic Attributes

In many cases the service attributes that service clients are interested in obtaining have relatively static values and little uncertainty associated with that value. For instance, "Jane's user ID" is a relatively static attribute with little uncertainty. In other cases, service clients may be interested in attributes that have dynamically changing values as well as uncertainty associated with them. For these types of attributes, clients may require services to provide various meta-attributes. We desire to support the following meta-attributes of dynamic attributes:

- **Accuracy.** Specifies to what degree of accuracy the value of this dynamic attribute is known. For instance, if John is looking for Dave, a person location service could inform John that Dave is at a particular location plus or minus some range.
- **Confidence.** Specifies to what degree of confidence the value and accuracy are known. For instance, the bandwidth tracking service could tell Jane that bandwidth will be poor with a high degree of confidence.
- **Update time.** Specifies at what time this attribute value was last measured or modified.
- **Sample interval.** Specifies over what interval of time the attribute value was gathered. This information is necessary when sampling attributes that change continuously with time.

In many instances when querying dynamic attributes, applications may need to place constraints on the meta-attributes of the dynamic attribute that they are looking for. For instance, an application may desire to know a person's location with a particular granularity: "Is Dave home or at school?" vs. "where exactly is Dave within the room?". Also, applications may need to know information that is fresh to a certain degree: "What is Dave's location (updated within the last

minute)?". To support this type of functionality, for each of the meta-attributes listed previously, we desire to allow clients to specify desired constraints in the form of a minimum and maximum acceptable bound.

The update time constraint is special in that it allows applications to specify either a relative or an absolute time. This gives applications the ability to require that results be fresh enough to be useful. In addition, this constraint can be used to specify that a future or historical value of an attribute is desired.

Again, it is important to stress that not all services need to allow clients to specify attribute requirements. However, for some services it is critical to support this functionality.

3.3 Query execution time

In some instances, the timeliness of a query is important. For instance, a driver of a car desiring to know the location of the closest restaurant cannot likely afford to wait several tens of seconds for an answer.

In situations such as this, a low delay waiting for rough answer from a service may be better than a long delay waiting for a precise answer. Clients should be able to indicate to services how long they expect to wait for an answer to a query. This can also give a service an indication of how much effort it should expend in processing the query. We anticipate that this mechanism will usually be used in a very coarse grained manner such as: if time limit is less than some threshold return a cached answer; otherwise obtain fresh data.

While it would be possible to allow clients to specify further query constraints, such as a limit on power consumed, we do not support any additional constraints since it is not clear that such constraints would be worth the additional complexity.

3.4 Support for multiple platforms

The Contextual Service Interface must support clients and services that reside on a variety of platforms, and are written in a variety of languages. The interface must be lightweight enough to be practical on clients, and potentially services, running on small portable devices. A fundamental way in which this can be accomplished is by not requiring all services, and clients, to implement every aspect of the interface. Rather, services need only support the subset of the features described in the interface definition that they need. Multi-platform support can also be facilitated if the service interface allows clients and services to use existing technology as much as possible.

4 Contextual Service Interface Functions

We now discuss the functions defined by the Contextual Service Interface. We focus our discussion on the fundamental interface function Query, and then briefly cover more advanced functions that are essentially extensions of this function. We omit coverage of minor support functions.

4.1 Query

The primary function defined by the Contextual Service Interface is the Query function, and it is likely that this is the only function that many services will support. As we will discuss in Section 5, an SQL database makes a convenient service implementation in some circumstances; hence, we make an effort to make using an SQL database simple while still retaining support for the dynamic attribute requirements mentioned previously. As a result, the Query function can be viewed as a simplified SQL query with added provisions for attribute requirements, timely execution, and support for meta-attributes in the result of the query. We now define the Query function and its arguments:

```
QueryResult Query(selectedAttributes, serviceNames,
                  selectionExpression,  attributeReqs,
                  timeLimit)
```

- **selectedAttributes.** This is a list of attributes to be returned by the query. This corresponds to the "select" clause in an SQL query.
- **serviceNames.** A list of service(s) that should handle the query. This corresponds to the "from" clause in an SQL query. Many services will only support a single entry in this list (the name of the single service supported). However, allowing more than one name is critical in allowing clients to express synthesis of information from multiple services. These multi-service queries can then be used by either proxies that break this query into multiple single-service queries or by services (such as those implemented via a database) that can support multi-service queries).
- **selectionExpression.** Expression that selects which entity or entities the query refers to. This corresponds to the "where" clause in an SQL query though our expressions are more restricted than SQL. Again, an essential element in attaining our goal of service simplicity is that we do not require all services to accept all expressions. So a person location service might only accept expressions of the form "personID=x".
- **attributeReqs.** For each of the meta-attributes of dynamic attributes listed in Section 3.2, we may specify a minimum and maximum required bound. Again, the precise attribute requirements supported and exactly which attributes these requirements can be applied to is service specific.
- **timeLimit.** The time in which the client needs a reply. This argument can also be viewed as a hint to the service on how much effort to expend in answering the query. High time limits imply the client desires a precise answer while low time limits imply that the client prefers a timely answer.

The result of a query is contained in a QueryResult structure which contains one or more lists of attributes. Each attribute list corresponds to an entity selected by the selectionExpression, and each list contains the attributes requested by the selectedAttributes parameter. Each entry in an attribute list is either a StaticAttribute structure or a DynamicAttribute structure. Static attribute structures simply contain the name of the attribute and its value. In addition to name and value,

dynamic attributes may contain the additional meta-attributes discussed in Section 3.2 (the service decides exactly which meta-attributes to include).

In addition, the QueryResult contains a completion flag that indicates whether or not the service was able to completely satisfy the constraints of the query. In some circumstances, for instance, a low time limit and stringent attribute requirements will preclude the service from satisfying both. In these cases, the service may set this flag to indicate that the answer provided does not satisfy the attribute requirements specified. Finally, the QueryResult also contains a timestamp of the time (local to the service) at which the service executed the query. This is for convenience in interpreting times reported in results of the query.

While the Query call suffices in many instances, there are situations in which it is insufficient, inefficient, or inconvenient to rely solely on the Query call. We now introduce a small number of extensions to the query call.

4.2 PostQuery

In many instances, clients may need to repeatedly obtain the same result from a service. While in some instances this can be accomplished via repeated Query calls, this can be inconvenient; moreover, repeated Query calls may fail to provide needed functionality. For example, consider a client that desires to hourly receive network bandwidth measurements sampled over 1 hour intervals. Implementing this via repeated Query calls would require the client to suspend execution waiting for the completion of each call. Further, processing overhead between each call would cause the 1 hour sample intervals to drift.

To relieve clients of the burden of repeated Query calls, reduce communication overhead, and remedy the problems discussed above, we introduce a function we call PostQuery. Services that support this call simply execute the specified query repeatedly at a given interval and use a callback to send results to the client at a host and port specified by the client. PostQuery is defined as follows:

```
QueryID PostQuery(selectedAttributes, serviceNames,
                  selectionExpression,  attributeReqs,
                  timeLimit, execInterval, queryClient)
```

Arguments to this function are the same as the Query call with the exception of two new arguments defined as follows:

- **execInterval.** Period at which to execute this query.
- **queryClient.** The client (hostname and port) to report the results to.

The QueryID returned is a handle that is used to tell which posted query a particular callback is generated from and to stop execution of the posted query.

4.3 PostCondTrigger & PostModTrigger

While using PostQuery instead of repeated Query calls can be useful, the amount of work performed by the client is essentially unchanged. In some circumstances, we may wish to push some

of this work onto the service. This can increase simplicity of client code and reduce power consumption at the client.

Consider, for instance, the airport bandwidth example mentioned previously (and described in detail in Section 9). Jane's client needs a way for bandwidth sampling to take place, but it only wants to be informed when the available bandwidth drops below a certain threshold. For this type of functionality we introduce the PostCondTrigger function where clients are informed of query results only when a specified condition holds. PostCondTrigger is defined as follows:

```
QueryID PostCondTrigger(selectedAttributes, serviceNames,
                        selectionExpression,  attributeReqs,
                        timeLimit, execInterval,
                        triggerExpression, queryClient)
```

Arguments to this function are the same as to PostQuery with the exception of one additional argument defined as follows:

- **triggerExpression.** Predicate that controls when the callback will be triggered.

Note that PostCondTrigger can fundamentally be considered a PostQuery call that only sends information when the given triggerExpression holds. It is important to take into consideration the fact that while some services may be able to track each and every value change of its attributes, others may require active work to measure attributes (or may have attributes that are constantly in flux). For this reason we retain the execInterval parameter which tells these types of services how often to check for changes (clients may also request notification of any and all changes from services that can support this functionality).

Finally, while PostCondTrigger allows clients to receive information when some absolute condition holds, there are situations where applications may need notification of a relative change in attribute value. For these situations we introduce the function PostModTrigger which is defined below:

```
QueryID PostModTrigger(selectedAttributes, serviceNames,
                selectionExpression, attributeReqs,
                timeLimit, execInterval, triggerAttributes,
                triggerDeltas, queryClient)
```

Arguments to this function are the same as PostCondTrigger where the triggerExpression is replaced by triggerAttributes and triggerDeltas:

- **triggerAttributes.** List of attributes to watch for changes.
- **triggerDeltas.** List of values (corresponding to this triggerAttributes) which will cause the callback to be executed when a given attribute changes by more than its corresponding triggerDelta entry.

5 Implementation Alternatives

Given the requirements and API we have discussed, we now evaluate several alternative implementations of the interface. As we desire to leverage existing technology as much as possible, all of our considerations will be based, to varying degrees, on existing standards. In the interest of brevity, we will not address all of our requirements point-by-point; rather, we will discuss the major benefits and drawbacks of each approach.

5.1 SQL Database

A natural candidate for service interface implementation is to use an SQL [3] database coupled with ODBC or a similar communication scheme. A major advantage of using SQL is that services that resolve queries from data stored statically can be implemented directly as an SQL database; in addition, a large number of existing SQL implementations are available. Lacking in SQL, however, is support for requirements on dynamic attributes.

A more serious drawback with SQL arises in the case of lightweight services that dynamically compute results to queries. For these services, SQL is an exceptionally poor choice as using an existing SQL implementation would result in running a complete database on the server whether or not this database is needed. For these simple services that compute query results on demand, a database would merely add unnecessary overhead.

5.2 LDAP

As the primary function required of the Contextual Service Interface is attribute value lookup, LDAP [2] systems are another natural consideration for interface implementation since they provide the capability to retrieve values that are organized in a hierarchical namespace. The benefits and drawbacks of using LDAP are somewhat similar to those of using SQL. On the positive side implementations are readily available. However, these implementations use a database to store attribute values which is not desirable for lightweight services. Hence using LDAP would either require a fresh implementation of the communication protocol, or extensive modification of existing implementations. Like SQL, LDAP also does not provide direct support for attribute requirements while providing a richer query language than some services may desire to support.

5.3 Simple Queries Encoded in XML Sent via HTTP

The solution we chose for implementation of the Contextual Service Interface was to use an SQL-like query language encoded in XML [9] and transported over HTTP. Both XML and HTTP are well established and widely deployed standards. Over this communication substrate, we define a small set of query functions that clients and services use for communication. By encoding queries in this manner, we can retain the benefits of SQL (ease of service implementation for static services and powerful queries) while avoiding the overhead requiring serviced implementation via a database. In addition, by allowing services to decide the types of queries they support, we allow for both simple clients and services.

5.3.1 Alternative Transport Mechanisms: CORBA, SOAP, RMI, RPC-2

Our choice of XML and HTTP for contextual service communication is based on the ubiquity of XML and HTTP deployment and simplicity of their use. We also considered several variants of remote procedure call for this purpose, but we chose not to use them for a variety of reasons. In the case of CORBA [4], much of the functionality provided was seen to be extraneous and rather heavy for extremely thin clients and services. SOAP [5] was viewed as too immature since the standardization effort was ongoing at the time we commenced this work. RMI [6] is unattractive for non-Java applications. RPC-2 [7] is unattractive for Java implementations, and, in addition, it is increasingly passed over in many Internet settings in favor of HTTP [8] based protocols (such as SOAP). Should, however, one of these transport mechanisms or another transport mechanism become extremely attractive in the future, we could replace the current XML/HTTP mechanism.

6 Service Interface Implementation

Currently we have two implementations of the Contextual Service Interface: one in C and one in Java. Both of these provide basic support for both client and service functionality, and are capable of interoperating with each other.

6.1 CSI Implementation

Our C-based CSI implementation provides both client-side and service-side support for direct access to the CSI functions defined previously. XML processing is implemented using the implementation of the DOM API provided by Apache's Xerces project [19].

In addition to basic support, our Java CSI implementation provides functionality to make developing clients and services easier. For clients, the Java implementation provides three different sets of methods for calling interface functions:

- **Direct methods.** These methods provide direct access to the Contextual Service Interface. That is, clients supply all arguments to these functions as defined in Section 4. This gives clients complete control, but can be somewhat verbose.
- **Parser based methods.** These methods simplify clients by allowing them to use SQL-like expressions to construct queries.
- **Attribute value retrieval methods.** These methods provide simple means for clients to retrieve a single value using a simple key (e.g. "What is Jane's phone number?") without worrying about more advanced features such as attribute requirements.

For services, the Java implementation provides support for writing services from the ground up, and convenience methods to make tasks such as error checking queries easier. Support is also provided for exporting any SQL database (with JDBC support) as a contextual service; this allows basic contextual services to be developed without any coding. This useful capability is a direct result of the support for multiple implementations designed into the Contextual Service Interface.

Our Java implementation uses the DOM API implementation present in JDK 1.4 for XML processing.

6.2 Interface Implementation Performance

While our focus in our implementation thus far has not been on performance, we have run some initial tests to determine how our implementation performs. All of our tests were conducted on services and clients implemented in Java and running under JDK 1.4 on Windows 2000 machines.

We first measured the performance of the Contextual Service Interface compared to a mature communication protocol: RMI. The services resided on a Pentium II 500 MHz machine with 128 MB of RAM attached to a wired 100 Mbps Ethernet while the clients resided on a Pentium III 600 MHz machine with 128 MB of RAM attached to a 11 Mbps wireless LAN (with a maximum actual throughput of around 4.4 Mbps). The different subnets were connected through a routed backbone.

To compare the two communication methods, we performed a simple "ping-pong" test where each client called a method on the service that does nothing except return to the client. We timed runs of 1,000 consecutive calls and averaged the measured time over the 1,000 calls. We then repeated this test 20 times, computed a global average, and computed a confidence interval for that global average. Table 1 shows our results.

	Average	95% Confidence Interval
CSI	7.76 ms	+/- 0.29 ms
RMI	3.85 ms	+/- 0.23 ms

Table 1: Contextual Service Interface Performance vs. RMI

Our results show that the Contextual Service Interface (CSI in the table) executes an empty call in roughly twice the time required for an empty RMI call. While we do not expect to entirely match the performance of RMI for this test, we do expect that future implementations can significantly narrow this performance gap. Currently we are using computationally expensive serialization and deserialization methods such as the XML DOM API which unnecessarily creates a tree representation of the serialized call. Switching to the SAX API should provide a significant speed increase.

7 Deployed Contextual Services

Using the Java Contextual Services package we have deployed a basic set of contextual services with one service per class of service depicted in Figure 1 (in general there may be several service instances per service class). These implemented services are depicted in Figure 5.

Services in normal typeface were implemented using a simple database. For instance the Devices service is implemented as a SQL relation with attributes of deviceID, deviceType while the PersonDevices service is implemented as a SQL relation with attributes personID and deviceID.

Services in italicized type required custom code to implement. The Access Point and Access Point Devices services use SNMP [10] to obtain bandwidth and population information respectively from 802.11 access points; these services make extensive use of dynamic attributes and attribute requirements to provide users with a variety of information while only retrieving SNMP

information as needed. The space service uses both custom code and a database to provide information regarding the structure of physical spaces; for more information see [18].

The Java Contextual Service packages support for multiple implementations has allowed us to quickly deploy simple attribute retrieval based services by simply using a database while allowing us to write custom code for services where database implementation was not appropriate. In addition, implementing multiple services with a single database reduces the amount of communication required since clients can issue a single multi-service as discussed in Section 3.1 query rather than multiple simple queries.

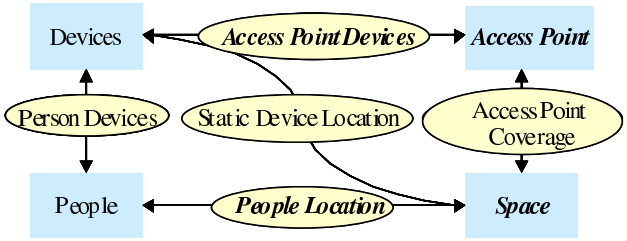

Figure 5: Implemented Services

We now illustrate how these contextual services we have deployed can be used to implement the scenarios mentioned earlier. (In these scenarios users interact with "Aura clients" that contact contextual services on behalf of users.)

8 Presentation Scenario

To illustrate how contextual services can assist applications and users, we consider in detail the presentation scenario briefly mentioned previously:

George works on a campus equipped with contextual services and a wireless LAN. He is planning on giving a presentation at a meeting where there will be a remote participant; George will bring a laptop equipped with a video camera and wireless network card which will allow the remote user to participate in the meeting.

Contextual services allow George's Aura client to select a video projector equipped conference room that is covered by two independent wireless access points. His Aura client is able to determine that both of these access points will be highly likely to have bandwidth adequate for the videoconference for the duration of the meeting. As the time of the meeting arrives, a key participant is absent. Contextual services allow George to determine that this participant is on the way.

This scenario requires fairly simple information from a variety of services. We divide the presentation scenario into the following steps used by George's client to find the information he requires (in parenthesis we list services contacted to acquire information at each step).

1. Find all conference rooms in the desired building (Space).
2. For all conference rooms discovered in step 1, find all conference rooms with projectors (DeviceLocation, Device).

3. For each conference room with a projector find all access points that cover the room (AccessPointCoverage).

4. For each of these access points predict the network bandwidth during the time of the meeting (AccessPoint).

5. Find the remote participant (People, PersonLocation).

Steps 1-3 require relatively straightforward queries on static attributes (attribute requirements and meta-attributes are not needed). As mentioned previously, the contextual service interface allows a database wrapper to deploy the services contacted in steps 2 and 3 using a single database; hence, for steps 2 and 3 a single multi-service query can be used to increase efficiency. The Space service contacted in step 1 is also implemented using a database as well, but the version of the space service used here was more convenient to implement using a dedicated database since it required a small amount of code in addition to the database.

Both the bandwidth and location information required in steps 4 and 5 are dynamic attributes. For these steps the ability to specify requirements on attributes is essential. Likewise, the meta-attributes discussed previously are important in interpreting the results returned by the services.

In step 4, providing requirements for the updateTime and sampleInterval meta-attributes of bandwidth allows us to specify that we desire a prediction of future bandwidth over a specified interval. The AccessPoint service can then return a prediction and also give an indication of how confident it is in this prediction via the confidence meta-attribute.

For step 5, we use two pseudocode excerpts to illustrate in more detail how the Contextual Service Interface is used. The first part of this step is obtaining a unique id for the user to be located (John Doe). An attribute value retrieval method allows this to be accomplished easily:

```
remoteUID = PersonService.get(personID, name, "John Doe")
```

Now we may query the person location service for the remote user's location. We use attribute requirements in this query to specify that we desire fresh location information, but that we only need a very rough accuracy (is the user on campus or not).

```
query(location,                    // selected attrib
      PersonLocationService,       // from service
      PersonID = remoteUID         // where clause
      location.updateTime =        // requirements
           within 2 minutes of present time
      location.accuracy
           within 500 meters of actual location
      10 seconds)                  // time limit
```

Now in resolving this query, our prototype PersonLocationService can potentially use location information from a variety of sources. For instance we gather information from user calendars, user login information, as well as the location of a user's wireless device. The wireless device location is the most accurate of these. Unfortunately, obtaining a list of devices in a single wireless cell takes several seconds. As there are over 600 wireless cells in our wireless network, access

points must be queried infrequently. Hence, despite the relatively fine spatial resolution obtained from wireless device location, the updateTime requirement of 2 minutes precludes the use of this information as it is likely to be too stale.

Now checking for a single device (instead of all devices) in a wireless cell takes approximately one second so we might try gather fresh data to determine John's location. However, the time limit of 10 seconds indicates both that George cannot afford to wait several minutes for this response, and that George is not interested in placing large demands upon the service.

Fortunately, the course grained location accuracy requirement allows a coarse-grained location method such as login information or calendar information to be used. Both these types of queries take under 1 second to gather the requested information so the people location service can check both John's login status as well as his appointment calendar to determine whether he is on campus.

As can be seen in this example, the ability of clients to communicate attribute requirements to services is critical for allowing services to provide clients with the information they desire in an efficient manner. We have further shown how CSI's support for multiple implementations allows simple services to be efficiently implemented while still supporting more complex services.

9 Bandwidth Advisor Scenario

9.1 Implementation of Bandwidth Advisor Scenario

Now consider the bandwidth advisor scenario:

Jane is waiting to depart on a business trip in an airport equipped with a wireless network. Contextual services are deployed at the airport providing information on the network and the physical layout of the airport. During her wait, Jane has been making some last minute changes to a very large graphically rich document she needs to email to her office. Shortly before her plane is scheduled to depart, she makes her final edits and clicks send. Unfortunately, a jumbo jet has arrived recently at an adjacent gate, and deplaning passengers are saturating the network cell in which Jane resides. Fortunately, Jane's mail client discovers from information gleaned from contextual services that she will miss her plane if she waits in her current location for her mail to finish sending. A quick scan of the surrounding area reveals that there is excellent bandwidth a short distance away. Following her device's suggestions, Jane switches locations, and is able to send her email before catching her flight.

As determining exactly when Jane needs to use large amounts of bandwidth is outside the scope of our discussion (a related effort is working on this task), we use an abridged version of this scenario consisting of the following steps:

1. Watch for low available bandwidth (i.e. high utilization) in the current cell (AccessPoint). (The identity of the current cell is determined locally on Jane's device.)

2. If the available bandwidth becomes low, find nearby locations where bandwidth is better (AccessPointCoverage, AccessPoint).

This scenario uses a smaller number of services than the presentation scenario; however, all steps require access to services that dynamically compute the results to queries. Hence, we make heavy use of attribute requirements.

Consider step 1 in detail. This step uses a trigger to relieve it of the burden of constantly polling available bandwidth as illustrated in the following pseudocode excerpt:

```
postCondTrigger(mbpsTotal,                // selected attrib
                AccessPointService,       // from service
                apName= "MyCellID"        // where clause
                mbpsTotal.sampleInterval // requirements
                     10 seconds,
                2 seconds,                // time limit
                10 seconds,               // execInterval
                trigger whenever          // trigger cond.
                     mbpsTotal > 2.0
                refToClient)
```

When the AccessPointService receives this query it knows from the sampleInterval requirement and execInterval specified that it should begin sampling access point bandwidth every 10 seconds. The trigger expression tells the service to inform the client whenever cell utilization is over 2.0 Mbps. When this happens, a callback is triggered on the client and it can proceed to look for a better access point (step 2).

Step 2 first uses a simple query to retrieve a list of nearby access points. For each of these access points, the sampleInterval and updateTime attribute requirements are used to specify that a prediction of bandwidth in the immediate future is required. The AccessPoint service can then use simple linear time series analysis to quickly provide a short term prediction. This is in contrast to the long term bandwidth prediction required in the presentation scenario which necessitated access to historical data, was computationally expensive, and far less accurate in the near term, but was needed for the that scenario. The ability to specify attribute requirements is essential in allowing the AccessPointService to decide which type of prediction is appropriate.

9.2 Bandwidth Advisor Performance

To determine what kind of performance we can expect from an active server, we measured the performance of our access point tracking services using tests similar to those in Section 6.2. During these tests, the services were actively gathering data from over 600 access points on CMU's wireless network. (The bandwidth information was gathered every 10 seconds while the cell population information was gathered every hour with the exception of a small number of cells for which it was gathered every two minutes.) For the Access Point Service test, we queried the total bandwidth at a single access point. For the Access Point Devices Service Test, we retrieved a list of all devices at a single access point (approximately 4 devices were present at the time the test was run).

Both the Access Point Service and the Access Point Devices service were run on the same machine: a 1.5 GHz machine with 256 MB of RAM. The client was run on a 300 MHz machine with 128 MB of RAM (these tests used JDK 1.4 beta 3). For these tests, we timed runs of 100 consecutive queries and averaged the measured time over the 100 queries. We then repeated this test approximately 20 times, computed a global average, and computed a confidence interval for that global average. Table 2 shows our results.

	Average	95% Confidence Interval
Access Point Service	13.87 ms	+/- 0.53 ms
Access Point Devices Service	16.04 ms	+/- 0.70 ms

Table 2: Access Point Services Performance

These results show that despite the large amount of work to constantly sample over 600 access points, our network services can provide timely network information to applications. These times will only improve as our implementation is refined.

10 Related Work

As discussed previously, the Contextual Service Interface performs a role that is somewhat similar to the attribute value retrieval facility provided by LDAP. Unlike LDAP systems, however, the Contextual Service Interface supports the notion that attribute values may be determined dynamically taking into consideration constraints (such as sample interval) required by applications; in addition, attributes may have uncertainty and other meta-attributes associated with them. We have also discussed how SQL databases can conveniently implement static services, but like LDAP are insufficient for dynamic attribute resolution. The Contextual Service Interface is, however, designed to be able to leverage SQL databases in certain circumstances without requiring them to be used in all cases.

There has also been work on adding support for uncertainty to static databases [12] [13]. Our work differs from efforts such as these in that we do not assume that all attributes of entities we are interested in are stored in a static database, rather they can be computed dynamically. Further, our interest with uncertainty is for the purposes of communicating some constraints on meta-attributes to services and allowing services to report to clients observed meta-attributes. We are not interested in extending SQL-like expressions to support uncertainty; in fact, our query expressions are restricted to be less powerful than SQL expressions.

The "Context Toolkit" [14] [15] uses a very similar approach to the approach we have taken, and supports entity attribute value retrieval via polling and callbacks. The largest difference between our work and the Context Toolkit is our explicit support for attribute requirements in requests and meta-attributes in results. Without these capabilities, much of the functionality that we are interested in supporting is lost. In addition, our focus is somewhat different. While the Context Toolkit focuses on developing an object oriented framework and allows use of multiple wire protocols, we focus on developing a standard interface for accessing services and force all services and clients to use the same wire protocol. This sacrifices flexibility, but increases interoperability.

Systems such as Jini [16] and UPnP [17] provide mechanisms for discovering and using arbitrary network deployed services. Our goals are rather different: we propose a small set of service classes for providing contextual information to applications via the particular interface that we have discussed.

11 Conclusion

We have shown how contextual services can be organized so that services are well focused and cleanly designed. This organization has allowed us to define and implement a Contextual Service Interface that provides applications with a uniform mechanism to query properties of physical entities and available resources in the local environment. We have discussed how providing information and allowing constraint specification on meta-attributes of "dynamic attributes" provides essential functionality required to deal with issues such as uncertainty and attribute sample time.

We have developed a number of contextual services using this interface, and have shown how the interface supports both simple service implementation using a readily available database as well as dynamic computation of service query results. We have further shown, via examples, how these services can be used to implement applications that adapt to provide users with behavior appropriate to their local environment.

References

[1] Garlan D., Siewiorek, D., Smailagic, A., Steenkiste, P., Project Aura: Towards Distraction-Free Pervasive Computing. IEEE Pervasive Computing, 1 (2):22-31, 2002.

[2] Wahl, M., Howes, T., Kille, S.: Lightweight Directory Access Protocol (v3), Dec. 1997. RFC 2251.

[3] Date, C.J., A Guide to the SQL Standard, Reading, Mass.: Addison-Wesley, 1987.

[4] Object Management Group, The Common Object Request Broker: Architecture and Specification, 2.3 ed., June 1999.

[5] World Wide Web Consortium, Simple Object Access Protocol (SOAP) 1.1, http://www.w3.org/TR/SOAP

[6] Sun Microsytems, Java Remote Method Invocation Specification, ftp://ftp.javasoft.com/docs/jdk1.2/rmi-spec-JDK1.2.pdf

[7] Srinivasan, R., RPC: Remote Procedure Call Specification Version 2, Aug. 1995. RFC 1831.

[8] Fielding, R., et. al., HTTP Transfer Protocol HTTP/1.1, June 1999, RFC 2616.

[9] World Wide Web Consortium, Extensible Markup Language (XML), http://www.w3.org/TR/REC-xml

[10] Case, J., et. al., A Simple Network Management Protocol (SNMP), May 1990, RFC 1157.

[11] Guttman, E., et. al., Service Location Protocol, Version 2, June 1999, RFC 2608.

[12] Dyreson, C., Snodgrass, R., Supporting Valid-Time Indeterminacy. ACM Transactions on Database Systems, 23 (1):1-57, 1998.

[13] Dekhtyar A., Ross R., Subrahmanian V.S. Probabilistic Temporal Databases: Algebra, January 1999, University of Maryland technical report CS-TR-3987, submitted to ACM Transactions on Database Systems.

[14] Salber D., Dey, A.K., Abowd, G.D. The Context Toolkit: Aiding the Development of Context-Enabled Applications. Proceedings of Conference on Human Factors in Computing Systems. Pittsburgh, PA, May 1999.

[15] Dey A.K., Salber, S., Futakawa M., Abowd, G. An Architecture to Support Context-Aware Applications GVU Technical Report GIT-GVU-99-23.

[16] Sun Microsystems, Jini Specifications, http://www.sun.com/jini/specs/

[17] UPnP Forum, Universal Plug-and-Play Documents, http://www.upnp.org/resources/documents.asp

[18] Jiang C., Steenkiste P., A Hybrid Location Model for Ubiquitous Computing. Proceedings of Ubicomp 2002. Goteborg, Sweden.. September 2002.

[19] The Apache Software Foundation, Xerces, http://xml.apache.org/xerces-c/index.html