Providing Contextual Information to Pervasive Computing Applications

Glenn Judd and Peter Steenkiste
Carnegie Mellon Univeristy
Pittsburgh, PA, USA
glennj@cs.cmu.edu, prs@cs.cmu.edu

Abstract

Pervasive 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 Contextual Information Service that provides applications with contextual information via a virtual database. Unlike previous efforts, our service provides applications a consistent, lightweight, and powerful mechanism for obtaining contextual information, and includes explicit support for the on demand computation of contextual information. We show, via example applications and a Contextual Information Service prototype that we have implemented, how this approach can be used to allow proactive applications to adapt their behavior to match a user's current environment.

1. Introduction

Fueled by advances in processing power, storage capacity, and battery life, the proliferation of mobile computing devices is rapidly turning the focus of computing away from personal computers and towards a collaboration between mobile devices, personal computers, and servers. Unfortunately, the increased usage of mobile devices has also increased the amount of user effort required to operate these devices. 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, thus providing users with more intelligent application behavior and allowing users to focus on higher level tasks.

To provide applications with environmental information, we have developed a Contextual Information Service that provides 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 etc. This contextual information is provided by several individual "Contextual Information Providers" that are organized into a virtual database. Synthesizing this contextual information allows applications to adapt to environmental and resource changes without user intervention.

To show how this Contextual Information Service enables 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. The Contextual Information Service will allow George to perform tasks such as selecting a conference room with both a video projector and enough wireless bandwidth for videoconferencing. It will also allow him to discover the whereabouts of late participants to the meeting. In the second example, we will show how the Contextual Information Service can assist a user, Jane, who has demanding network bandwidth requirements in a bandwidth scarce environment. In this example, the Contextual Information Service will allow Jane to move to a location where her bandwidth demands can be met.

Designing a contextual information service is, however, a difficult task both because of the diversity of the information involved and the complexity of the queries that must be supported. Consider some of the requests that might be used in the scenarios described above: What devices are in room 160? What is the expected bandwidth in room 160 between 2 p.m. and 3 p.m. tomorrow? Where is Jane now?

In addition, since we desire to support a wide variety of applications beyond the given scenarios, we consider requests such as: When will bandwidth be best, within the next hour, to flush my distributed file system's cache? Inform me whenever John moves more than 50 m. Where is the closest color printer with an empty print queue?

A simplistic approach to providing the information desired in the requests listed above is to write custom contextual information services, as needed. Unfortunately, using such an ad hoc approach will result in multiple services with multiple interfaces, which will complicate application development. Moreover, even services that function well

individually could become fractured and inefficient when used in conjunction with other contextual services.

An attractive alternative is to store contextual information in a database. Databases are a well-understood technology and they directly address the problems listed above by providing cleanly organized data via a single consistent interface. In addition, using a database allows clients to remain lightweight since they can issue powerful queries for contextual information from several sources using a lightweight interface. Unfortunately, a static database precludes on demand gathering of contextual information; this restriction has limited both the functionality and the scalability of previous efforts. Moreover, contextual information often has meta-data associated with it (such as accuracy and freshness) and databases do not directly support this.

To overcome these limitations, we have developed a Contextual Information Service (CIS) that is organized as a virtual database: it provides applications with an SQL-like query interface but the information is stored, or collected on demand, by a distributed infrastructure of contextual information providers. This approach allows us to retain the ability for applications to easily synthesize information from several sources of contextual information while avoiding the limitations of a static database. Moreover, contextual information providers that do not require features such as on demand computation of results are able to utilize an ordinary database for implementation.

Our discussion proceeds as follows: Section 2 outlines service interface requirements. Section 3 outlines our service and service interface architecture. Section 4 introduces major service interface functions. Section 5 discusses related work. Section 6 briefly describes our Contextual Service Interface implementations. Section 7 describes our deployed Contextual Information Service prototype. Sections 8 and 9 discuss implementation of the examples mentioned above, and Section 10 concludes our discussion.

A more complete version of this paper is available as a technical report [2].

2. Requirements of the CIS

In this section we discuss requirements and design guidelines for both the Contextual Information Service and the interface used to access the CIS. (Adding security and privacy is discussed in [3].) Sections 3 and 4 will then illustrate how these requirements are satisfied.

2.1. Allow Clients to Easily Synthesize Required Contextual Information

The CIS should provide clients with contextual information while requiring minimal effort on the part of the client. To accomplish this, we must allow clients to easily synthesize contextual information from several contextual information providers. This greatly simplifies the efforts of application developers since clients may issue rich queries for contextual information, and relieves developers of the burden of manual contextual information synthesis. In addition, support for rich queries shifts the burden of contextual information synthesis off of the client and into the CIS.

Moreover, to further reduce the burden on clients, the CIS should support callback functions that reduce the need for polling by clients. Lastly, CIS clients and contextual information providers should not be required to implement every aspect of the interface; clients and providers need only support the subset of the features in the interface that they desire or that are feasible to implement.

2.2. Facilitate Implementation of Efficient Information Providers

Given the diversity of the contextual information, the CIS should allow providers to use the most convenient means of implementation. In many instances, contextual information will be fairly static, e.g. information about building layout, personal information such as phone numbers, etc. In these cases the most convenient implementation will typically be a database. Therefore, it is important that the CIS allows providers of static contextual information to leverage databases in a straightforward manner.

In other instances, however, contextual information is highly dynamic. In these cases it is often undesirable or even impossible to statically store the information in a database. In situations such as this, it is most appropriate to actively compute the answer to a query. For instance, a person location provider should probably only compute the location of a person when a client is actually interested in that person's location. Moreover, such a provider should usually only retrieve information at the lowest accuracy and confidence level required by the clients. For example, finding out whether John is on campus will in general be easier than identifying the room that he is in. Lack of support for on demand retrieval of contextual information would force contextual information sources to constantly collect and store updates at the highest granularity and accuracy that might be desired by any client, which would be very inefficient.

Of course, providers supporting dynamic data should be able to cache information to improve performance and response time. Caching can be useful not only in the provider that collects the data but also in providers that resolve more complex queries or even in the client library.

2.3. Support for Dynamic Attributes

Contextual information that is dynamic typically has uncertainty associated with it. For these types of attributes, clients may require providers to support various meta-attributes. Examples include *accuracy* and *confidence*. For

example, if George is looking for John, a person location provider could inform George that John is at a particular location plus or minus some range. Similarly, the bandwidth information provider could tell Jane that bandwidth will be poor with a high degree of confidence.

This requirement affects the service interface in two ways. First, clients must be able to receive these meta-attributes in query results so that they can interpret the contextual information correctly. Second, clients must be able to specify requirements for the meta-attributes, so they can make sure that the information provided is useful to them, without requiring the CIS to always collect the most accurate and precise information that might possibly be needed. For example, if a client needs to know whether John is at work, there is no need for the Contextual Information Service to identify precisely what room he is in.

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.
- **Confidence.** Specifies to what degree of confidence the value and accuracy are known.
- **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.

3. System Architecture

3.1. Contextual Information Service Architecture

Figure 1 illustrates how the Contextual Information Service provides applications with a database abstraction of the contextual information providers. Clients issue queries using the Contextual Service Interface (CSInt - discussed in Section 4), the queries are decomposed by the Query Synthesizer, one or more lower-level queries are forwarded (via CSInt) to individual providers, and the results are synthesized and returned to the client application. This architecture enables client applications to focus on the information that they desire, and reduces the need to worry about how contextual information is retrieved. Thus even extremely thin clients can issue rich queries without incurring the large communication and computational expenses of such queries. Unlike previous techniques, our approach achieves this goal while allowing for efficient CIS implementation.

This approach also allows contextual information providers to be implemented efficiently without sacrificing essential functionality. For example, for providers of relatively static data, we have developed a CSInt-SQL wrapper that allows for straightforward implementation using a database without requiring any coding. For providers of dynamic information, a database may not be an appropriate implementation. Using a database-like interface, however, allows a consistent interface to these dynamic providers. For added efficiency, we explicitly support caching at every stage (client, query synthesizer, and information provider).

To illustrate how our approach simplifies communication with both providers that statically store data and those that actively compute the results to queries, consider Figures 1 and 2. Under the traditional model (Figure 2), synthesizing information from several providers requires individual communication with each provider using multiple incompatible interfaces. Applications must include support for these interfaces and know how to synthesize the information returned. Under our model (Figure 1), applications are able to synthesize contextual information from several sources with a single query and without incurring the computational cost of query decomposition and synthesis.

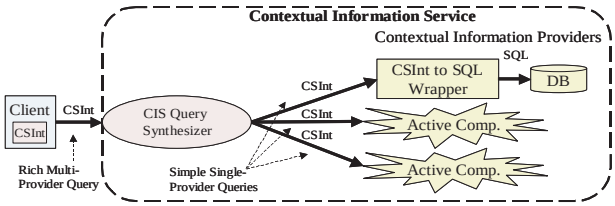

Figure 1. CIS architecture

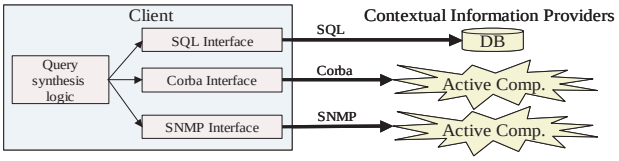

Figure 2. Traditional client-svc. Interaction

3.2. Contextual Information Provider Classes

We now consider the contents of the virtual database provided by the Contextual Information Service. A common design methodology used in database implementation is to consider various "entities" of interest as well as the relationships between these entities. We argue that providing information on the aspects of the contextual environment that are most relevant to mobile applications can be accomplished by providing information on entities and relationships 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 possibility 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 pervasive applications will need information on people, devices (e.g. printers), and physical spaces; hence, we define an information provider 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 locations in the world. We could also have defined a class for power sources; however, as these essentially amount to either an electrical outlet in a physical space or a battery on a device, we treat these as attributes of physical spaces and devices respectively.

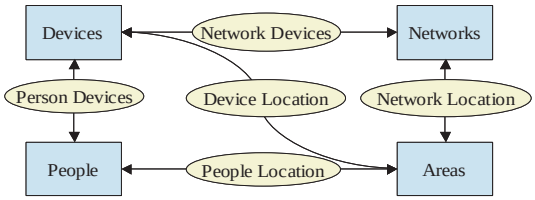

Figure 3. Provider classes

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 provider classes for each of the entity classes listed above, we define an information provider class that tracks relationships between each pair of entity classes above (with the sole exception of people and networks) as shown in Figure 3. For example, one instance of the Person Devices class might provide information on what devices are currently being used by particular individuals while another instance might provide information on what devices are owned by particular individuals.

There are several other potential classes of entities that might be useful. As the need for these classes is not clear, they are currently omitted from our architecture.

4. Contextual Service Interface Functions

We now discuss the functions defined by the Contextual Service Interface. Applications use this interface to communicate with the CIS query synthesizer which then uses this same interface to communicate with the individual contextual information providers (applications may contact contextual information providers directly if they desire).

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 contextual information providers will support. All other major functions are extensions of the Query function. As previously discussed, we want to make using an SQL database as a provider 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,
             providerNames, 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.

- **providerNames.** A list of provider(s) that should handle the query. This corresponds to an SQL "from" clause. Many providers will only support a single entry in this list. Allowing more than one name, however, is critical in allowing clients to express synthesis of information from multiple providers. These multi-provider queries can then be used by the query synthesizer to break this query into multiple single-provider queries. Multi-provider queries can also be used in situations where multiple providers are implemented together (such as those implemented via a database).

- **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, we do not require all providers to accept all expressions. So a person location provider might only accept expressions of the form "personID=x".

- **attributeReqs.** 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 in Section 2.3, clients may 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, not all providers need to allow clients to specify attribute requirements. However, for some providers it is critical to support this functionality.

- **timeLimit.** The time in which the client needs a reply. This argument can also be viewed as a hint to the provider on how much effort to expend in answering the query.

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 2.3.

In addition, the QueryResult contains a completion flag that indicates whether or not the provider 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 provider from satisfying both. In these cases, the provider may use 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 provider) at which the provider 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. For these situations we define a small number of additional functions: the PostQuery call which executes a query at a specified interval; the PostCondTrigger which acts like a postQuery that only returns a result if a given condition holds; and PostModTrigger which only returns a result if a given list of attributes have changed a given amount. These extensions are discussed more in the technical report [2].

5. Related Work

We now compare and contrast the design we have presented with previous approaches. Subsequent sections will then discuss our implementation.

5.1. Context Architectures

Many systems have been developed for providing applications with contextual information in a distributed environment. Schilit's Active Map system [4] [5] can be viewed as a location-based publish-subscribe system for contextual information dissemination. Under this system, location tagged contextual information is published to an Active Map Server which then disseminates the information to interested applications. Steggles and Harter [6] discuss a three tier contextual information architecture. The first tier consists of producers and consumers of contextual information which send updates and contextual queries to a set of second tier of CORBA-based servers. This second tier communicates with a database, which makes up the third tier, in order to process updates and queries. This third tier may be bypassed if performance needs require. Brown [7] and Schmidt [8] use a physical note metaphor for developing context aware applications. Applications post notes of interest, and an action triggers when a given condition holds. EasyLiving [9], stores contextual information in a single database. This allows applications to retrieve contextual information using powerful queries. The Context Toolkit [10] [11] uses three types of components (termed "widgets") to gather, observe, and process context. Hong's Context Framework [12] is an infrastructural approach that supports event and query based access to contextual information.

5.2. Contributions of the Aura CIS

Unlike previous work, we explicitly include strong support for contextual information providers that actively compute the results to requests for contextual information. Our explicit support allows dynamic computation of contextual information to be efficient and scalable. For instance, unlike previous systems, we explicitly support the caching of dynamically generated results, and provide means for caches to realize when the results that they contain are insufficient to satisfy a query.

A key feature, lacking in other systems, that enables the dynamic computation of query results is our support for meta-attributes such as accuracy, confidence, sample time, and sample interval duration as discussed in Section 2.3. The lack of support for meta-attributes in other systems hampers the expression of notions such as future and historical values of attributes. In addition, lack of support for meta-attributes mandates hand tuning of values such as sample interval, and, as a result, many previous systems cannot support on demand computation of continuously valued contextual information in a scalable manner.

For example, each new location sensor in Steggles and Harter [6] increases the update load on the network and the central database. Their system attempts to mitigate this load by reducing the sample interval and allowing updates to bypass the database, but these stopgap measures do not fundamentally change the fact that the load increases directly with the number of sensors. With our architecture, it is possible to create systems that only query sensors (or other sources of contextual information) that produce information that clients are actively interested in. In addition, these information sources need only be queried at a resolution that clients actually require (as opposed to always sam-

pling at the finest granularity that clients might possibly be interested in).

Another important contribution of our research is to leverage techniques commonly used in the database community in order to develop a powerful and efficient Contextual Information Service without actually requiring the use of a database for provider implementation. This approach allows applications to focus on the information that they desire while greatly reducing their need to worry about how it is obtained. In addition, our architecture reduces the load on mobile hosts by offloading query processing onto the CIS (without requiring manual construction of intermediate proxies as previous efforts have). Also, when databases are appropriate, our architecture allows them to be seamlessly integrated as contextual information providers without the need for any coding. Thus we are able to retain the benefits of an SQL-like query language while avoiding the limitations of mandating implementation in an actual database.

The Contextual Information Service is also the first context architecture to treat networks as first class contextual entities. As network connectivity can have tremendous impact on application performance, we enable applications to ascertain what type of network connectivity they can expect at a given location and time. This allows applications to intelligently adapt to current or expected network conditions. For example, a mobile device might realize that network connectivity may soon be lost and perform critical tasks while connectivity is still available.

Probably the closest efforts to ours are the Context Toolkit and Context Framework. The Context Toolkit, however, lacks critical features required for on demand generation of contextual information such as support for the previously mentioned meta-attributes. In addition, the Context Toolkit does not allow applications to automatically synthesize results from several contextual providers. Aggregation of results from several providers must be managed manually by providers and clients.

Like the Context Framework, we advocate an infrastructural approach to providing contextual information to clients. From the information published so far, the Context Framework appears to lack support for critical meta-attributes (though it does include support for confidence) that allow for on demand computation of contextual information. This limits both functionality and scalability as discussed previously. Also, while we target low-level contextual information (people location, device properties, etc.) and provide a powerful interface to automatically synthesize this low-level contextual information, the Context Framework appears to target higher level information and automatic conversion of complex data types (e.g. PDF to PostScript). As such, portions of the Context Framework are largely complimentary to our work.

Note that while provisions for security and privacy are critical in many situations, they are beyond the scope of our current discussion. A related effort [3] in the Aura Project has enhanced the contextual interface discussed here with a certificate based security and privacy system that allows users to control their contextual information.

6. Service Interface Implementation

6.1. Communication

Our solution for interface communication was to use an SQL-like query language encoded in XML 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 providers use for communication. By encoding queries in this manner, we can retain the benefits of SQL (ease of provider implementation for static providers and powerful queries) while avoiding the overhead requiring provider implementation via a database. In addition, by allowing providers to decide the types of queries they support, we allow for both simple clients and providers.

6.2. Available CIS Libraries

We currently have two Contextual Information Service libraries: a C implementation and a Java implementation. Our C-based implementation provides both client-side and service-side support for direct access to the Contextual Service Interface functions defined previously.

In addition to basic support, our Java implementation provides functionality to make developing clients and providers 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 providers, the Java implementation has support for writing providers 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 information provider; this allows basic providers 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.

6.3. Communication Performance

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

We measured the response time of CSInt compared to a mature communication protocol: RMI. The providers resided on a Pentium II 500 MHz machine attached to a wired 100 Mbps Ethernet while the clients resided on a Pentium III 600 MHz machine attached to a 11 Mbps wireless LAN. 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 an "empty" method on a provider. 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, and computed a global average and confidence interval for that global average. Table 1 shows our results.

Table 1. CSInt and RMI response time

	Average	95% Conf. Int.
CSInt	7.76 ms	+/- 0.29 ms
RMI	3.85 ms	+/- 0.23 ms

Our results show that the Contextual Service Interface (CSInt 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 (RMI is a binary protocol whereas HTTP/XML is text based), 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. Contextual Information Service Prototype

We have deployed a Contextual Information Service prototype consisting of several Contextual Information Providers and a prototype Context Synthesizer. As depicted in Figure 4, we have deployed one provider for each of the classes defined previously in Figure 3.

7.1. Context Synthesizer

As shown in Figure 1, the Context Synthesizer accepts queries from clients, decomposes them, queries the appropriate Contextual Information Providers, and then synthesizes the results for return to the client. Our current synthesizer is a simple model that does not yet implement more advanced functionality such as query optimization. Additional research is required to extend distributed query processing techniques to efficiently operate on Contextual Information Providers. Nevertheless, our current Context Synthesizer is already capable of allowing clients to synthesize large amounts of contextual information using very simple queries while imposing very little load on the client.

7.2. Static Contextual Information Providers

The contextual information providers shown in Figure 4 in normal typeface were implemented using a simple database. CIS's explicit support for provider implementation via a database allowed for trivial implementation of these static contextual information providers. For instance, the PersonDevices provider is implemented as an SQL relation with attributes personID and deviceID. In addition to easing provider implementation, implementing multiple providers with a single database reduced the amount of communication required since the synthesizer can issue a single multi-provider query rather than multiple simple queries.

7.3. Dynamic Contextual Information Providers

Providers depicted in Figure 4 in underlined italicized typeface required custom code to implement. For these services, the meta-attributes are a critical feature that enables these services to determine what type of information a user is requesting while minimizing the amount of effort expended to resolve queries. (The technical report [2] discusses each of these in more detail.)

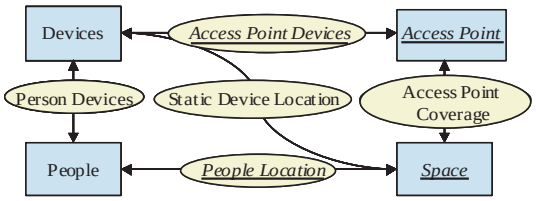

Figure 4. Deployed providers

To determine what kind of performance we can expect from providers of dynamic contextual information, we

measured the response time of typical queries on our access point providers. During these tests, the access point providers 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 Provider test, we queried the total bandwidth at a single access point. For the Access Point Devices Provider 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 Provider and the Access Point Devices Provider 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 20 times, computed a global average, and computed a confidence interval for that global average. Table 2 shows our results.

Table 2. AP providers query time

	Average	95% Conf. Int.
AP Provider	13.87 ms	+/- 0.53 ms
AP Devices Provider	16.04 ms	+/- 0.70 ms

These results show that despite the large amount of work to constantly sample over 600 access points, our access point information providers can provide timely network information to applications.

8. Presentation Scenario

We now illustrate how our Contextual Information Service prototype can be used to implement the scenarios mentioned earlier. (To simplify discussion, in these scenarios users interact with "Aura clients" that contact the CIS on behalf of users.) Consider in detail the presentation scenario briefly mentioned previously:

George works on a campus equipped with a Contextual Information Service 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.

Our CIS allows 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 participant is absent. The Contextual Information Service allows George to determine that this participant is not on campus.

This scenario consists of two independent tasks: 1 - Find a conference room that meets the requirements specified. 2 - Find the late user.

Both the network bandwidth and person location information required are dynamic attributes. To retrieve this information, the ability to specify requirements on meta-attributes is essential. Likewise, the meta-attributes discussed previously are important in interpreting the results returned by the providers. For example, specifying 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 Contextual Information Service can then return a prediction and also give an indication of how confident it is in this prediction via the confidence meta-attribute.

The following SQL-like psuedocode illustrates how George's client can find a conference room meeting his requirements using a single query() call (the psuedocode maps directly to the parameters required by query):

```
Select APCoverage.room, APCoverage.apName
From Space, Device, DeviceLocation,
     APCoverage, AccessPoint
Where Space.type = "Conference"
  and Space.ali within "ali://cmu/wean"
  and  DeviceLocation.room = Space.name
  and  DeviceLocation.id = Device.id
  and  Device.type = "Projector"
  and  APCoverage.room = Space.name
  and  AccessPoint.apName = APCoverage.apName
  and  AccessPoint.mbpsTotal < 1.0
Require mbpsTotal.sampleTime = start of meeting
        mbpsTotal.sampleInterval = meeting length
TimeLimit none
```

This query returns a list of conference rooms with projectors with access points likely to have ample bandwidth. A quick inspection of this list allows George's Aura client to find a conference room that is covered by multiple suitable access points.

To demonstrate the feasibility of this scenario, we implemented a simple client that queries our prototype CIS for this information. Limitations in our current synthesizer require the above query to be split into two parts: First obtain a list of candidate conference rooms and access points. Next, check the predicted bandwidth for each access point.

We measured the response time of the above queries on our CIS prototype. Each query was executed 10 times, and an average time was computed. This process was repeated 20 times, and then an overall average and confidence interval for that average were computed as shown in Table 3. These measurements show that our prototype CIS allows George to find a suitable location in approximately 1 second. (The disparity in times for the two queries relates to the complexities of the queries involved and the fact that

our Context Synthesizer does not attempt query optimization.)

Table 3. George scenario query time

	Average	95% Conf. Int.
Conference Room Query	1.187 s	+/- 0.013 ms
Bandwidth Query	0.077 s	+/- 0.013 ms

Now we illustrate how our CIS enables George to locate the late meeting participant. This query uses attribute requirements to specify that fresh location information is desired, but only a very rough accuracy is necessary (is John on campus or not).

```
Select location
From   PersonLocation
Where  PersonID = John's UID
Require location.updateTime
           within 2 minutes of present time
        location.accuracy
           within 500 meters of actual location
TimeLimit 1 minute
```

Now in resolving this query, our prototype Person Location Provider 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 location provider must use this information sparingly.

Table 4. Simple location query time

	Average	95% Conf. Int.
Location Query	0.446 s	+/- 0.014 ms

Table 4 shows the response time of our prototype Person Location Provider (the average shown was computed using 20 runs of 20 queries each). In this test, the location of the user to be located was known to the People Location Provider. Clearly, a simple location query can be executed quite quickly. Hence, a location provider can use any additional time provided by the TimeLimit parameter to perform a more exhaustive search. For example, the 1 minute TimeLimit parameter in the above location query, indicates both that George can afford to wait some time for the query to complete, and that George desires the location provider to expend a large amount of effort, if necessary, to locate John.

As can be seen in this example, the ability of clients to communicate attribute requirements is critical to providing clients with the information they desire in an efficient manner. We have further shown how our CIS allows simple contextual information providers to be efficiently implemented while still supporting more complex providers.

9. 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. A Contextual Information Service is 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 the CIS 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 (APCoverage, AccessPoint).

This scenario uses a smaller number of information providers than the presentation scenario; however, all steps require access to providers 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
Select mbpsTotal
From   AccessPoint
Where  apName= "MyCellID"
ExecInterval 10 seconds,
Require mbpsTotal.sampleInterval 10 seconds
Trigger whenever mbpsTotal > 2.0
TimeLimit none
```

When the Access Point Provider 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 provider 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 Provider then uses current utilization information to provide a simple near 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 that scenario. The ability to specify attribute requirements is essential in allowing the Access Point Provider to decide which type of prediction is appropriate.

10. Conclusion

We have developed a Contextual Information Service that provides applications with a virtual database view of physical entities and available resources in the local environment. Unlike previous efforts, this service provides explicit support for the on demand computation of contextual information while allowing ordinary databases to be used whenever possible.

We have implemented a query synthesizer and a number of contextual information providers for this service, and have shown, via examples, how this service can be used to create applications that adapt to provide users with behavior appropriate to their local environment.

References

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

[2] G. Judd, P. Steenkiste, Providing Contextual Information to Pervasive Computing Applications, CMU Technical Report CMU-CS-03-100.

[3] U. Hengartner, P. Steenkiste, Protecting People Location Information. Proceedings of Workshop on Security in Ubiquitous Computing. Goteborg, Sweden. September 2002.

[4] B. Schilit, M. Theimer, Disseminating Active Map Information to Mobile Hosts. IEEE Network, 8(5):22-32, September/October 1994.

[5] B. Schilit, N. Adams, R. Want, Context-Aware Computing Applications. IEEE Workshop on Mobile Computing Systems and Applications. Santa Cruz, CA. December 1994.

[6] A. Harter, A. Hopper, P. Steggles, A. Ward, P. Webster, The Anatomy of a Context-Aware Application. Proceedings of MOBICOM 1999, Seattle, WA. August 1999.

[7] P. Brown, J. Bovey, X. Chen, Context-Aware Applications: From the Laboratory to the Marketplace. IEEE Personal Communications, 4(5):58-64. 1997.

[8] A. Schmidt, M. Beigl, H. Gellersen, There Is More to Context than Location. Computer & Graphics, 23(6):893-901, December 1999.

[9] B. Brumitt, S. Shafer, Location Modeling for Ubiquitous Computing. Proceedings of Workshop on Location Modeling for Ubiquitous Computing. Atlanta, GA. September 2001.

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

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

[12] J. Hong, J. Landay, An Infrastructure Approach to Context-Aware Computing. Human-Computer Interaction (HCI) Journal, 16(2-3), 2001.

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

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

[15] S. Madden, M. Franklin, J. Hellerstein, W. Hong, TAG: A Tiny Aggregation Service for Ad-Hoc Sensor Networks. Proceedings of the OSDI 2002. Boston, MA. December 2002.

[16] P. Bonnet, J. Gehrke, P. Seshadri, Querying the Physical World. IEEE Personal Communications, 7 (5):10-15, 2000.