An Architecture To Support Context-Aware Applications

Anind K. Dey, Daniel Salber, Gregory D. Abowd
GVU Center, College of Computing
Georgia Institute of Technology
801 Atlantic Drive
Atlanta, GA, 30332-0280, USA
Tel: 1-404-894-5103
E-mail: anind, salber, abowd@cc.gatech.edu

Masayasu Futakawa
Hitachi Research Laboratory
7-1-1 Omika-cho
Hitachi-shi, Ibaraki-ken, 319-1221, Japan
Tel: 81-294-52-5111
E-mail: futakawa@hrl.hitachi.co.jp

ABSTRACT
Context is an important, yet poorly utilized source of information in interactive computing. It is difficult to use because, unlike other forms of user input, there is no common, reusable way to handle context. Most context-aware applications have been built in an *ad hoc* manner. We discuss the requirements for dealing with context and present an architectural solution we have designed and implemented to help application designers build context-aware applications more easily. We illustrate the use of the architecture through a context-aware application that assists conference attendees.

KEYWORDS: Context-aware computing, architecture, ubiquitous computing, toolkits, application development

MOTIVATION
In human-human interaction, a great deal of information is conveyed without explicit communication. Gestures, facial expressions, relationship to other people and objects in the vicinity, and shared histories are all used as cues to assist in understanding the explicit communication. These shared cues, or *context*, help to facilitate grounding between participants in an interaction [4]. We define context to be any information that can be used to characterize the situation of an entity, where an entity can be a person, place, or physical or computational object.

In human-computer interaction, there is very little shared context between the human and the computer. Context in human-computer interaction includes any relevant information about the entities in the interaction between the user and computer, including the user and computer themselves. Humans naturally provide context in the form of signs of frustration or confusion, for example, but computers cannot sense or use it. We define applications that use context to provide task-relevant information and/or services to a user to be *context-aware*. For example, a context-aware tour guide may use the user's location and interests to display relevant information to the user.

Submitted for review to UIST'99
Reviewers: please watch the accompanying video and for better screenshots, visit http://www.cc.gatech.edu/fce/contexttoolkit/uist

The increased availability of commercial, off-the-shelf sensing technologies is making it more viable to sense context in a variety of environments. The prevalence of powerful, networked computers makes it possible to use these technologies and distribute the context to multiple applications, in a somewhat ubiquitous fashion. So what has hindered applications from making greater use of context and from being context-aware?

A major problem has been the lack of uniform support for building and executing these types of applications. Most context-aware applications have been built in an *ad hoc* manner, heavily influenced by the underlying technology used to acquire the context. This results in a lack of generality, requiring each new application to be built from the ground up. To enable designers to easily build context-aware applications, there needs to be architectural support that provides the general mechanisms required by context.

This paper describes an architecture to support context-aware applications. We provide a discussion of how context has been handled in previous work and why we would like to handle it as a generalized form of user input. In earlier work [15], we presented the concept of context widgets, which allow us to handle context in a manner analogous to user input. This paper discusses the architectural support necessary for using context widgets. We derive the requirements for the architecture by examining the differences between the uses of context and input. Next, we present our architectural solution for supporting the design and execution of context-aware applications. We demonstrate the use of the architecture through an example application, the Conference Assistant. Finally, we discuss the benefits and limitations of the architecture, based on our experiences in building context-aware applications.

DISCUSSION OF CONTEXT HANDLING
We have demonstrated the importance of context in interactive computing. The reason why context is not used more often is that there is no common way to acquire and handle context. In this section, we discuss how context has previously been acquired and discuss some concepts that will allow us to handle context in the same manner as we handle user input.

Current Context Handling

In general, context is handled in an improvised fashion. Application developers choose whichever technique is easiest to implement, at the expense of generality and reuse. We will now look at two common ways in which context has been handled: connecting sensor drivers directly into applications and using servers to hide sensor details.

With some applications [7,14], the drivers for sensors used to detect context are directly hardwired into the applications themselves. In this situation, application designers are forced to write code that deals with the sensor details, using whatever protocol the sensors dictate. There are two problems with this technique. The first problem is that it makes the task of building a context-aware application very burdensome, by requiring application builders to deal with the potentially complex acquisition of context. The second problem with this technique is that it does not support good software engineering practices. The technique does not enforce separation of concerns between application semantics and the low-level details of context acquisition from individual sensors. This leads to a loss of generality, making the sensors difficult to reuse in other applications and difficult to use simultaneously in multiple applications.

The original Active Badge research took a slightly different approach [19]. In this work, a server was designed to poll the Active Badge sensor network and maintain current location information. Servers like this abstract the details of the sensors from the application. Applications that use these servers simply poll the servers for the context information that they collect. This technique addresses both of the problems outlined in the previous technique. It relieves application developers from the burden of dealing with the individual sensor details. The use of servers separates the application semantics from the low-level sensor details, making it easier for application designers to build context-aware applications and allowing multiple applications to use a single server.

However, this technique has two additional problems. First, applications that use these servers must be proactive, requesting context information when needed via a polling mechanism. The onus is on the application to determine when there are changes to the context and when those changes are interesting. The second problem is that these servers are developed independently, for each sensor or sensor type. Each server maintains a different interface for an application to interact with. This requires the application to deal with each server in a different way, much like dealing with different sensors. This may affect an application's ability to separate application semantics from context acquisition.

Current Input Handling

Ideally, we would like to handle context in the same manner as we handle user input. User interface toolkits support application designers in handling input. They provide an important abstraction to enable designers to use input without worrying about how the input was collected. This abstraction is called a widget, or an interactor. The widget abstraction provides many benefits. The widget abstraction has been used not only in standard keyboard and mouse computing, but also with pen and speech input [1], and with the unconventional input devices used in virtual reality [11]. It facilitates the separation of application semantics from low-level input handling details. For example, an application does not have to be modified if a pen is used for pointing rather than a mouse. It supports reuse by allowing multiple applications to create their own instances of a widget. It contains not only a polling mechanism but also possesses a notification, or callback, mechanism to allow applications to obtain input information as it occurs. Finally, in a given toolkit, all the widgets have a common external interface. This means that an application can treat all widgets in a similar fashion, not having to deal with differences between individual widgets.

Analogy of Input Handling to Context Handling

There have been previous systems which handle context in the same way that we handle input [2, 16]. These attempts used servers that support both a polling mechanism and a notification mechanism. The notification mechanism relieves an application from having to poll a server to determine when interesting changes occur. However, this previous work has suffered from the design of specialized servers, that result in the lack of a common interface across servers [2], forcing applications to deal with each server in a distinct way. This results in a minimal range of server types being used (e.g. only location [16]).

Previously, we demonstrated the application of the widget abstraction to context handling [15]. We showed that *context widgets* provided the same benefits as GUI widgets: separation of concerns, reuse, easy access to context data through polling and notification mechanisms and a common interface. Context widgets encapsulate a single piece of context and abstract away the details of how the context is sensed. We demonstrated their utility and value through some example applications.

The use of the widget abstraction is clearly a positive step towards facilitating the use of context in applications. However, there are differences in how context and user input are gathered and used, requiring a new architecture to support the context widget construct. The remainder of this paper will describe the requirements for this architecture and will describe our architectural solution.

ARCHITECTURE REQUIREMENTS

Applying input handling techniques to context is necessary to help application designers build context-aware applications more easily. But, it is not sufficient. This is due to the difference in characteristics between context and user input. The important differences are:

- the source of user input is a single machine, but context can come from many, distributed machines
- user input and context both require abstractions to separate the details of the sensing mechanisms, but

context requires additional abstractions because it is often not in the form required by an application
- widgets that obtain user input belong to the application that instantiated them, but widgets that obtain context are independent from the applications that use them

While there are some applications that use user input that have similar characteristics to context, (groupware [12] and virtual environments [5] deal with distributed input and user modeling techniques [8] abstract input, for example), they are not the norm. Because of the differences between input and context, unique architectural support is required for handling context and context widgets. We will now derive the requirements for this architecture.

Distribution of context sensing network

Traditional user input comes from the keyboard and mouse. These devices are connected directly to the computer they are being used with. When dealing with context, the devices used to sense context most likely are not attached to the same computer running the application. For example, an indoor infrared positioning system may consist of many infrared emitters and detectors in a building. The sensors must be physically distributed and cannot all be directly connected to a single machine. In addition, multiple applications may require use of that location information and these applications may run on multiple computing devices. As environments and computers are becoming more instrumented, more context can be sensed, but this context will be coming from multiple, distributed machines. Support for the distribution of context is our first high-level requirement.

The need for distribution has a clear implication on architecture design stemming from the heterogeneity of computing platforms and programming languages that could be used to both collect and use context. Unlike with user input widgets, the programming languages used by the application to communicate with context widgets and used by the context widgets themselves, may not be the same. The architecture must support interoperability of context widgets and applications on heterogeneous platforms.

Abstraction: Interpretation and Aggregation

There is a need to extend the existing notification and polling mechanisms to allow applications to retrieve context from distributed computers in the same way that they retrieve input from local widgets. There may be multiple layers that context data goes through before it reaches an application, due to the need for additional abstraction. For example, an application wants to be notified when meetings occur. At the lowest level, location information is interpreted to determine where various users are and identity information is used to check co-location. At the next level, this information is combined with sound level information to determine if a meeting is taking place. From an application designer's perspective, the use of these multiple layers must be transparent.

In order to support this transparency, context must often be interpreted before it can be used by an application. An application may not be interested in the low-level information, and may only want to know when a meeting starts. In order for the interpretation to be reusable by multiple applications, it needs to be provided by the architecture.

To facilitate the building of context-aware applications, our architecture must support the aggregation of context about entities in the environment. Our definition of context given earlier describes the need to collect context information about the relevant entities (people, places, and objects) in the environment. With only the context widget abstraction, an application must communicate with several different context widgets in order to collect the necessary context about an interesting entity. This has negative impacts on both maintainability and efficiency. Aggregation is an abstraction that allows an application to only communicate with one component for each entity that it is interested in.

Component Persistence and History

With most GUI applications, widgets are instantiated, controlled and used by only a single application. In contrast, our context-aware applications do not instantiate individual context widgets, but must be able to access existing ones, when they require. This leads to a requirement that context widgets must be executing independently from the applications that use them. This eases the programming burden on the application designer by not requiring her to maintain the context widgets, while allowing her to easily communicate with them. Because context widgets run independently of applications, there is a need for them to be persistent, available all the time. It is not known *a priori* when applications will require certain context information, consequently, context widgets must be running perpetually to allow applications to contact them when needed. Take the call-forwarding example from the Active Badge research [19]. When a phone call was received, an application tried to forward the call to the phone nearest the intended recipient. The application could not locate the user if the Badge server was not active.

A final requirement linked to the need for execution persistence is the desire to maintain historical information. User input widgets maintain little, if any, historical information. For example, a file selection dialog box keeps track of only the most recent files that have been selected and allows a user to select those easily. In general though, if a more complete history is required, it is left up to the application to implement it. In comparison, a context widget must maintain a history of all the context it obtains. A context widget may collect context when no applications are interested in that particular context information. Therefore, there are no applications available to store that context. However, there may be an application in the future that requires the history of that context. For example, an application may need the location history for a user, in order to predict his future location. For this reason, context widgets must store their context.

Requirements Summary

We have presented requirements for an architecture that supports context-aware applications. To summarize, these requirements are:

- allow applications to access context information from distributed machines in the same way they access user input information from the local machine
- support execution on different platforms and the use of different programming languages
- support for the interpretation of context information
- support for the aggregation of context information
- support independence and persistence of context widgets
- support the storing of context history

In the next section, we describe the architecture that we have built to address these requirements.

DESCRIPTION OF ARCHITECTURE

Our architecture was designed to address the requirements from the previous section. We used an object-oriented approach in designing the architecture. The architecture consists of three main types of objects:

- Widget, implements the widget abstraction
- Server, responsible for aggregation of context
- Interpreter, responsible for interpretation of context

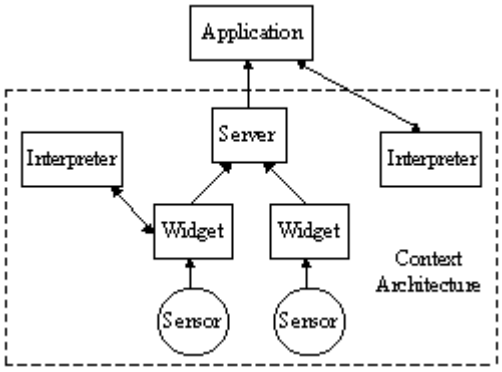

Figure 1. Relationship between applications and the context architecture. Arrows indicate data flow.

Figure 1 shows the relationship between the objects and an application. Each of these objects is autonomous in execution. They are instantiated independently of each other and execute in their own threads, supporting our requirement for independence. These objects can be instantiated all on a single computing device or on multiple computing devices. Although our base implementation is written in Java, the mechanisms used are programming language independent, allowing implementations in other languages as we will describe later.

It is important to note that the architecture provides scaffolding for context-aware computing. By this we mean that it contains important abstractions and mechanisms for dealing with context, but it is clearly not a complete solution, nor is it meant to be. The architecture supports the building of widgets and interpreters required by an application, but will not necessarily have them already available. Compared to input, there are a larger variety of possible sensors used to sense context and a larger variety of possible context. This makes it very difficult to provide all possible combinations of widgets and interpreters.

Context Widgets

A context widget, as mentioned earlier, has much in common with a user interface widget. It is defined by its attributes and callbacks. Attributes are pieces of context that it makes available to other components via polling or subscribing. Callbacks represent the types of events that the widget can use to notify subscribing components. Other components can query the widget's attributes and callbacks, so they don't have to know the widget capabilities at design time. A context widget supports both the polling and notification mechanisms to allow components to retrieve current context information. It also allows components to retrieve historical context information. The basic Widget object provides these services automatically for context widgets that subclass it.

Creating a new widget is very simple. A widget designer has to specify what attributes and callbacks the widget has, provide the code to communicate with the sensor being used, and when new data from the sensor is available, call 2 methods: `sendToSubscribers()`and `store()`. The Widget class provides both of these methods. The first method validates the data against the current subscriptions. Each time it finds a match, it sends the relevant data to the subscribing component. For example, multiple applications have subscribed to a Meeting Widget with different callbacks, attributes, and conditions. When the widget obtains new meeting information it sends it to the appropriate subscribers.

The second method adds the data to persistent storage, allowing other components to retrieve historical context information. This addresses our requirement for the storage of context history. The Widget class provides a default implementation for persistent storage using MySQL, a freeware database. The persistent storage mechanism is "pluggable". A widget designer not wanting to use the default mechanism can provide a class that implements a temporary cache and allows the storage and retrieval of information from some persistent storage. The name of the class is given to the widget at run time, allowing the new storage mechanism to be used.

Context Servers

Context servers implement the aggregation abstraction, which is one of our requirements. They are used to collect all the context about a particular entity, such as a person, for example. They were created to ease the job of an application programmer. Instead of being forced to subscribe to every widget that could provide information about a person of interest, the application can simply communicate with a single object, that person's context server. The context server is responsible for subscribing to every widget of interest, and acts as a proxy to the application.

The Server class is subclassed from the Widget class, inheriting all the methods and properties of widgets. It can be thought of, then, as a compound widget. Just like widgets, it has attributes and callbacks, it can be subscribed to and polled, and its history can be retrieved. It differs in how the attributes and callbacks are determined. A server's attributes and callbacks are "inherited" from the widgets to which it has subscribed. When a server receives new data, it behaves like a widget and calls `store()` and `sendToSubscribers()`.

When a designer creates a new server, she simply has to provide the names of the widgets to subscribe to. In addition, she can provide any attributes or callbacks in addition to those of the widgets and a Conditions object. The Conditions object is used in each widget subscription, so the server only receives information it is interested in. For example, the Anind User Server would have the subscription condition that the name must equal "Anind".

Context Interpreters

Context interpreters are responsible for implementing the interpretation abstraction discussed in the requirements section. Interpretation of context has usually been performed by applications. By separating the interpretation abstraction from applications, we allow reuse of interpreters by multiple applications. An interpreter does not maintain any state information across individual interpretations, but when provided with state information, can interpret the information into another format or meaning. A simple example of an interpreter is one that converts a room location into a building location (e.g. Room 343 maps to Building A). A more complex example is one that takes location, identity and sound information and determines that a meeting is occurring. Context interpreters can be as simple or as complex as the designer wants.

Context to be interpreted is sent to an interpreter's `interpretData()` method. It returns the interpreted data to the component that called the interpreter. Interpreters can be called by widgets, servers, applications and even by other interpreters. When a designer creates a new interpreter, she only has to provide the following information: the incoming attributes, the outgoing attributes, and an implementation of `interpretData()`.

Communications Infrastructure

All of the top-level objects (widgets, servers, and interpreters) used are subclassed from a single object called BaseObject. The BaseObject class provides the basic communications infrastructure needed to communicate with the distributed components and abstract away the details of heterogeneous platforms and programming languages, supporting heterogeneity and distribution. Applications use this class to communicate with the context architecture. The communications includes dealing with both the passing of high-level data and low-level protocol details.

High-level Communications A basic communications infrastructure is needed to support semantic or high-level communications. BaseObject provides methods for communicating with the widgets, servers, and interpreters. In particular it facilitates subscribing and unsubscribing to, querying/polling and retrieving historical context from servers and widgets, requesting interpretations from interpreters, and retrieving object-specific information like version numbers and attributes.

When a component (an application, for example) wants to subscribe to another component (a widget, for example), it uses a general subscription and notification mechanism, addressing our requirement to allow applications to access context data in the same way as user input. When a component wants to be notified of particular events in a context widget, it subscribes to the widget, adding itself as a "context handler", and implements a single method called `handle()`. In the subscription, it can specify a number of options to narrow the scope of the subscription:

- Callback: the event of interest to the component
- Attributes: the particular widget attributes of interest
- Conditions: the conditions under which the widget should return data to the subscribing component

These three options essentially act together as a filter to control which data and under which conditions context events are sent from a widget to a subscribing component to be handled. This is an extension of the general subscription mechanism, where only callbacks can be specified. This helps to substantially reduce the amount of communication, which is important in a distributed architecture for performance reasons. This mechanism also makes it easier for application programmers to deal with context events, by delivering only the specific information the application is interested in.

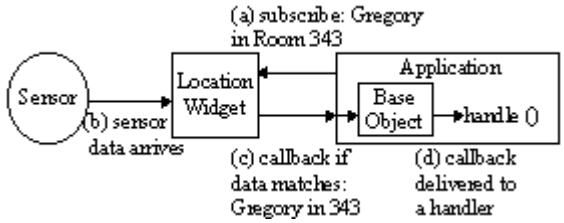

Figure 2. Example of application interacting with context widget. Arrows indicate communications flow.

When a widget sends callback data to a subscribing application, the application's BaseObject instance uses the data from the callback to determine which object(s) to send the data to. It calls that object's `handle()` method. An example of this is shown in Figure 2. An application has subscribed (2a) to the Location Widget, wanting to know when Gregory has arrived in Room 343. When the Location Widget has new location information available (2b), it compares the information to see if the information meets the subscription conditions. If it does, it sends the information to the application (2c) and the application's

BaseObject routes it to the `handle()` method (2d).

Communications Fundamentals The BaseObject acts as both a client and a server (in the client-server paradigm of distributed computing). It is used to both send and receive communications.

The BaseObject uses a "pluggable" communications scheme. By default, it supports communications using the HyperText Transfer Protocol (HTTP) for both the sending and receiving of messages. The language used for sending data is, by default, the eXtensible Markup Language (XML). The BaseObject does, however, support the use of other communications protocols and data languages. When a designer wants an object to use a different protocol, SMTP (Simple Mail Transfer Protocol) for example, she creates an object that "speaks" the SMTP protocol for outgoing communications and one for incoming communications. She then passes the names of these objects to the BaseObject at the time of instantiation. The BaseObject uses these objects rather than the default HTTP objects. In a similar fashion, a designer can replace XML with use his own data encoding scheme.

XML and HTTP were chosen for the default implementation because they support lightweight integration of distributed components and allow us to meet our requirement of architecture support on heterogeneous platforms with multiple programming languages. XML is simply ASCII text and can be used on any platform or with any programming language that supports text parsing. HTTP requires TCP/IP and is ubiquitous in terms of platforms and programming languages that support it. Other alternatives to XML and HTTP include CORBA and RMI. Both were deemed too heavyweight, requiring additional components (an ORB or an RMI daemon) and in the case of RMI, would have forced us to use a specific programming language – Java.

EXAMPLE APPLICATION: CONFERENCE ASSISTANT

We will now demonstrate the use of our architecture through an example application that we have built. We describe the Conference Assistant application using a scenario we have executed in our lab. Following this, we describe how context is used to provide new services to the user. Finally, we describe how features in the architecture were exploited to make this application easy to build.

Application Scenario

A user is attending a conference. When she arrives at the conference, she registers, giving her personal information as well as a list of interests. In return, she receives a copy of the conference proceedings and a Personal Digital Assistant (PDA). The application running on the PDA automatically displays a copy of the conference schedule, showing the multiple tracks of the conference, including both paper tracks and demonstration tracks. On the schedule (Figure 3), certain papers and demonstrations are highlighted (light gray) to indicate that they may be of particular interest to the user.

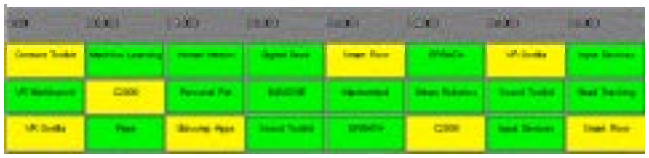

Figure 3. Screenshot of the augmented schedule, with suggested papers and demonstrations highlighted (light-colored boxes) in the three tracks.

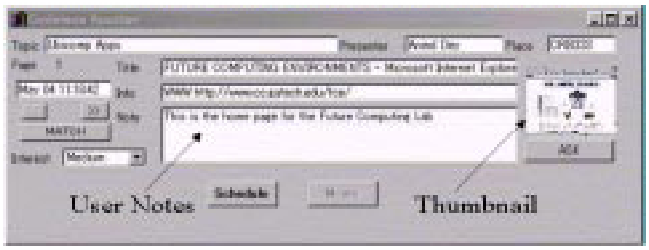

Figure 4. Screenshot of the Conference Assistant note-taking interface.

The user takes the advice of the application and walks towards the room of a suggested paper presentation. When she enters the room, the PDA automatically displays the name of the presenter and the title of the presentation. The presenter is using a combination of PowerPoint and Web pages for his presentation. A thumbnail of the current slide or Web page is displayed on the PDA. The application allows the user to create notes of her own to "attach" to the current slide or Web page (Figure 4). As the presentation proceeds, the PDA displays updated information for the user. The user takes notes on the presented slides and Web pages using the PDA. The presentation ends and the presenter opens the floor for questions. The user has a question about the presenter's tenth slide. She uses her PDA to control the presenter's display, bringing up the tenth slide, allowing everyone in the room to view the slide in question. She uses the displayed slide as a reference and asks her question. She adds her notes on the answer to her previous notes on this slide.

The user looks back at the schedule on the PDA and notices that the application has suggested a demonstration to see based on her interests. She walks to the room where the demonstrations are being held. As she walks past demonstrations in search of the one she is interested in, the PDA displays the name of each demoer and the corresponding demonstration. She arrives at the demonstration she is interested in. The application displays any PowerPoint slides or Web pages that the demoer uses during the demo. The user continues to use the PDA throughout the conference for taking notes on both demonstrations and paper presentations.

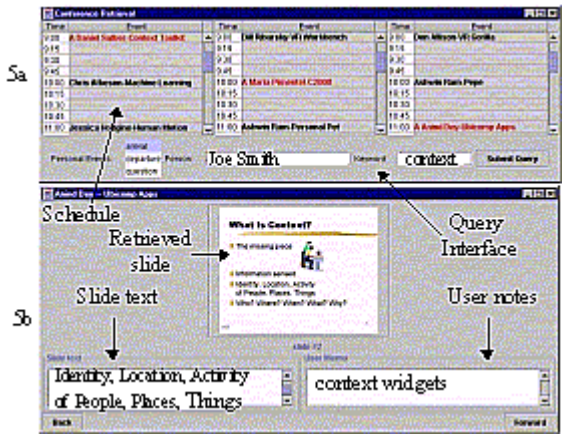

Figure 5. Screenshots of the retrieval application: timeline (5a) and slideshow (5b). (Image was touched up for clarity).

She returns home after the conference and wants to retrieve some notes about a particular presentation. The user starts her web browser and loads a retrieval applet using a URL provided by the conference. The application shows her a timeline of the conference schedule with the presentation and demonstration tracks. The application uses a feature known as context-based retrieval [10]. It provides a query interface that allows the user to populate the timeline with various events: her arrival and departure from different rooms, when she asked a question, when other people asked questions or were present or when a presentation used a particular keyword. By selecting an event on the timeline (Figure 5a), the user can view (Figure 5b) the slide or Web page presented at the time of the event, information about the event, and any personal notes she may have taken on the presented information. She can then continue to view the current presentation, moving back and forth between the presented slides and Web pages.

In a similar fashion, a presenter can retrieve information about their presentation including: names of the audience members, names of people who asked questions and the slide relevant to the question.

Use of Context

The application features in the scenario presented have all been implemented. The application makes use of a wide range of context. In this section, we discuss the types of context used and how they were used to provide benefits to the user.

When the user is attending the conference, the application first uses information about what is being presented at the conference and her personal interests to determine what presentations might be of particular interest to her. The application uses her location, the activity (presentation of a Web page or slide) in that location and the presentation details (presenter and title) to determine what information to present to her. The context of the presentation facilitates the user's asking of a question. The context is used to control the presenter's display, changing to a particular slide for which the user had a question.

After the conference, the retrieval application uses the conference context to retrieve information about the conference. The context includes public context such as the time when presentations started and stopped, the names of the presenters and the presentations and the rooms in which the presentations occurred and any keywords the presentations mentioned. It also includes the user's personal context such as the times at which she entered and exited a room, the rooms themselves, when she asked a question and what presentation and slide or Web page the question was about. The application also uses the context of other people, including their presence at particular presentations and questions they asked, if any. The user can use any of this context information to retrieve the appropriate slide or Web page associated with the context.

Use of the Architecture

In this subsection, we discuss how the architecture and widget library were used to develop this application. Figure 6 presents a snapshot of the architecture when a single user is viewing a presentation. For multiple users and presentations, the user and presentation architecture segments are replicated appropriately. Table 1 lists all the components and their use.

Table 1: Architecture components and responsibilities: S = Servers, W = Widgets, I = Interpreters

Component	Responsibility
Registration (W)	Acquires user registration and interests
Memo (W)	Acquires user's notes and relevant presentation info
Recommender (I)	Locates interesting presentations
User (S)	Aggregates all information about the user
Question (W)	Acquires audience questions and relevant presentation info
Location (W)	Acquires arrivals/departures of users
Content (W)	Monitors PowerPoint or Web page presentation, capturing content changes
Presentation (S)	All information about a presentation

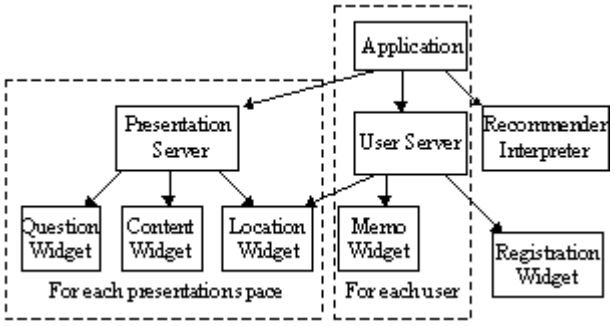

Figure 6. Conference Assistant architecture.

During registration, a User Server is created for the user. It is responsible for aggregating all the context information about the user and acts as the application's interface to the user's personal context information. It subscribes to information about the user from the public Registration

Widget, the user's Memo Widget and the Location Widget in each presentation space. The Memo Widget captures the user's notes and also any relevant context (relevant slide, time, presenter name).

We use two different sensors in different implementations of Location Widgets: a JavaRing™ and reader, and a PinPoint 3D-iD™ indoor positioning system. The servers and applications that use these widgets required no change and did not need restarting when sensors were changed.

There is a Presentation Server for each physical location where presentations/demos are occurring. A Presentation Server is responsible for aggregating all the context information about the local presentation and acts as the application's interface to the public presentation information. It subscribes to the widgets in the local environment, including the Content Widget, Location Widget and Question Widget.

When an audience member asks a question using their PDA, the Question Widget captures the context (relevant slide, location, time) and notifies the local Presentation Server of the event. The server stores the information and also uses it to access a service provided by the Content Widget, displaying the slide or Web page relevant to the question.

The application being executed on the PDA does not communicate with any widget directly, but instead communicates only with the user's User Server and the local Presentation Server. It subscribes to the User Server for changes in location and interests. It subscribes to the local Presentation Server for changes in a presentation slide or Web page when the user enters a presentation or demo space and unsubscribes when the user leaves.

In the context-based retrieval application, all the necessary information has been stored in the user's User Server and the public Presentation Servers. As shown in Figure 4, the application allows the user to retrieve slides (and the entire presentation) using context via a query interface. If personal context is used as the index into the conference information, the application polls the User Server for the times and location at which a particular event occurred (user entered or left a location, or asked a question). This information can then be used to poll the correct Presentation Server for the related presentation information. If public context is used as the index, the application polls all the Presentation Servers for the times at which a particular event occurred (use of a keyword, presence or question by a certain person). As in the previous case, this information is then used to poll the relevant Presentation Servers for the related presentation information.

EVALUATION OF ARCHITECTURE

In the previous section, we described the Conference Assistant application and how it was designed. In this section, we describe the benefits and limitations of the architecture in the context of the Context Assistant and other context-aware applications that we have built.

We have built multiple types of each object (widgets, servers, and interpreters) and included this library of objects in our Context Toolkit[1]. Our default implementation of the architecture was developed in Java. However, the abstractions and mechanisms that address our requirements are both platform and programming language independent. To demonstrate this, we have also written applications and widgets in C++, Python, and Frontier. We have components of the architecture executing on Windows CE, 95 and NT, Linux, Solaris, IRIX, and the Macintosh platforms, including mobile, wearable, and desktop machines.

Benefits

The benefits of the architecture are that it:

- supports lightweight integration of components
- makes it easy to add the use of context to existing applications that don't use context
- makes it easy to add more context to applications that already use context
- hides the details of the context sensors
- supports reusability by multiple applications

In this section, we will discuss these benefits, using the applications we have developed as examples.

The use of lightweight communications and integration mechanisms supports the deployment of the architecture and applications on multiple platforms using multiple programming languages. We have implemented an In/Out Board application that keeps track of who is in our building and when they were last seen. This application has been implemented in a stand-alone mode in Java, and on the web[2] using Frontier on a Macintosh and Python on Linux. The Conference Assistant application was executed on a Windows CE handheld device. The retrieval portion was in Java executing in a web browser. The numerous architecture components shown in Figure 5 for the application ran on Windows CE, Windows 95 and NT.

From an application builder's perspective, the architecture makes it simple to add the use of context to applications that previously did not use context. The DUMMBO [3] application supports the capture of informal whiteboard meetings. The original version did not use context and began to capture audio and pen strokes when someone started writing on the whiteboard. The application was modified to initiate data capture when people were gathered around the whiteboard, but had not yet started writing. The change required the modification/addition of 25 lines of Java code. The significant changes included 2 lines added to use the architecture, 1 line modified to enable the class to handle callbacks, and 17 lines that are application specific.

Based on these results, we argue that it is also simple to add additional context to applications that already use the

[1] The Context Toolkit may be downloaded at http://www.cc.gatech.edu/fce/contexttoolkit/

[2] See http://fire.cc.gt.atl.ga.us/inout for the web version

architecture. In the Conference Assistant application, the use of context was made much simpler through the use of the User and Presentation context servers. From the application builder's viewpoint, the servers make accessing context information much easier than dealing with the multiple widgets individually. If a new type of user context was added to this application, the application could continue to use a context server as a proxy to the context information, requiring minimal changes.

A benefit of the architecture is that it hides the details of the context sensors. This allows the underlying sensors to be replaced, without affecting the application. The In/Out Board and Conference Assistant have been used with JavaRings and the PinPoint system.

An important feature of the architecture is independent execution. This lets the context architecture run independently of applications, which allows multiple applications to use the architecture simultaneously. We have built a number of applications that leverage off of the context architecture we have running in our lab, including the In/Out Board, a context-aware tour guide, the Conference Assistant, and a context-aware mailing list that only delivers mail to the current occupants of the building.

Current Limitations and Future Work

Although our architecture eases the building of context-aware applications, some limitations remain. In particular, the architecture does not currently support dealing with unreliable sensor data, transparently acquiring context from distributed components, and dealing with component failures. We discuss each of these issues and propose possible solutions.

All sensors have failure modes, making the data they produce unreliable at some point in time. To add to this problem, the data from many sensors must be interpreted to make sense of it. In much the same way as speech input must be recognized without a 100% accurate recognizer, context from sensors must also be understood. The problem with context is greater due to the fact that with speech input, the user has the opportunity to give feedback about incorrect recognition results. With context, this isn't the case. Recognized context is often used without being displayed to the user. The architecture must provide support for applications that may be using unreliable context information. This support many include the use of sensor fusion from multiple, heterogeneous sensors (with different failure modes) to increase reliability, as well as the passing of a reliability measure with each piece of context. We currently perform sensor fusion in an *ad hoc* fashion, but we are exploring a general mechanism for supporting it.

The architecture does not completely support the transparent acquisition of context for applications. In order for an application to use a widget, server, or interpreter, it must know both the hostname and port the component is being executed on. When the architecture supports a form of resource discovery [17], it will be able to effectively hide these details from the application. This will allow the application to really treat local context widgets like user interface widgets. When an application is started, it could specify the type of context information required and any relevant conditions to the resource discovery mechanism. The mechanism would be responsible for finding any applicable components and for providing the application with ways to access them. We have investigated many existing techniques for supporting resource discovery and are currently in the process of selecting one to implement.

Another limitation of the architecture is dealing with component failures. When a component fails, it has to be manually restarted. With a requirement for perpetual execution, this is clearly not an admirable property. The architecture does keep track of existing subscriptions between component restarts. So, when a component fails and is restarted, it knows what components were subscribed to it so it can continue notifying them of context events. But, the architecture needs a facility for automatically restarting components when they fail. We are currently investigating the use of watchdog processes and redundant components for this purpose.

RELATED ARCHITECTURE WORK

In the discussion on context handling at the beginning of the paper, we discussed previous work that influenced us in our decision to treat context like input. We will now look at other related work that presents architectures that extend beyond supporting the abstraction of sensor details.

The metaDesk project [18] used sensor proxies, a concept similar to context widgets, to separate the details of individual sensors from the application. This system architecture supported distribution and a simple form of resource discovery. Only a polling mechanism was provided and interpretation was left to individual applications.

The proposed Situated Computing Service [9] has an architecture that is similar in intent to ours. It insulates applications from sensors used to acquire context. A Situated Computing Service is a single server that is responsible for both context acquisition and abstraction. It provides both polling and notification mechanisms for accessing relevant information. It differs from our research in that it uses a single server to act as the interface to all the context available in an environment as opposed to our modular architecture. A single prototype server has been constructed as proof of concept, using a single sensor type.

The AROMA system [13] was exploring awareness in media spaces. Its architecture used the concepts of sensor abstraction and interpretation to provide awareness of activity between remote sites. The interpreted context at each site was sent to the other sites for display.

The CyberDesk system [6] used minimal context to provide relevant services to a desktop computer user. Its architecture was modular, separating context acquisition,

abstraction, and notification. It did not use a distributed architecture and provided no support for aggregation.

CONCLUSIONS

We have presented an architecture for supporting the software design and execution of context-aware applications. Our architecture builds upon our previous work [15] that introduced the idea of a context widget for treating context like user input. We generated requirements for the architecture based on the differences between dealing with context and input. Using these requirements, we designed and built an architecture to make it easier for applications to deal with context. We demonstrated the use of the architecture through a complex application, the Conference Assistant. We discussed the benefits of the architecture through example applications that we have built. Finally, we described the limitations of the current architecture and, as part of our future work, plan to address these with the suggested improvements. We intend to test the architecture through the use of a larger variety of context and the building of more complex applications.

ACKNOWLEDGMENTS

We would like to thank Jen Mankoff for her help with the context widget concept, Jason Brotherton for his work on DUMMBO, and the other members of the Future Computing Environments research group for their ideas and for providing a testbed for our applications. This work was supported by NSF CAREER Grant # 9703384, NSF ESS Grant EIA-9806822, a Motorola UPR and a Hitachi grant.

REFERENCES

1. Arons, B. The design of audio servers and toolkits for supporting speech in the user interface. *Journal of the American Voice I/O Society 9* (Mar. 1991), 27-41.

2. Bauer, M. *et al*. A collaborative wearable system with remote sensing. In *Proceedings of International Symposium on Wearable Computers* (October, Pittsburgh, PA), IEEE, 1998, pp. 10–17.

3. Brotherton, J.A., Abowd, G.D. and Truong, K.N. Supporting capture and access interfaces for informal and opportunistic meetings. Georgia Tech Technical Report, GIT-GVU-99-06. Dec, 1998.

4. Clark, H.H. & Brennan, S.E. Grounding in communication. In L.B. Resnick, J. Levine, & S.D. Teasley (Eds.), *Perspectives on socially shared cognition*. Washington, DC. 1991.

5. Codella, C.F. *et al*. A toolkit for developing multi-user, distributed virtual environments. In *Proceedings of Virtual Reality Annual International Symposium* (Sept. 1993, Seattle, WA), IEEE, 1993, pp. 401–407.

6. Dey, A.K., Abowd, G.D. and Wood, A. CyberDesk: A framework for providing self-integrating context-aware services. *Knowledge-Based Systems 11*, 1999, pp. 3-13.

7. Harrison, B.L. *et al*. Squeeze me, hold me, tilt me! An exploration of manipulative user interfaces. In *Proceedings of CHI'98* (April 18-23, Los Angeles, CA), ACM/SIGCHI, N.Y., 1999, pp. 17–24.

8. Horvitz, E. *et al*. The Lumiere Project: Bayesian user modeling for inferring the goals and needs of software users. In *Proceedings of the 14th Conference on Uncertainty in Artificial Intelligence* (July, Madison, WI), 1998, pp. 256-265.

9. Hull, R., Neaves, P., and Bedford-Roberts, J. Towards situated computing. In *Proceedings of International Symposium on Wearable Computers* (October, Boston, MA), IEEE, 1997, pp. 146–153.

10. Lamming, M. *et al*. The design of a human memory prosthesis. *Computer Journal 37*, 3 (1994), 153-163.

11. MacIntyre, B. and Feiner, S. Language-level support for exploratory programming of distributed virtual environments. In *Proceedings of UIST'96* (Nov. 5–8, Seattle, WA), ACM/SIGCHI, N.Y., 1996, pp. 83–94.

12. Greenberg, S. (1990). Sharing views and interactions with single-user applications. In *Proceedings of the ACM/IEEE Conference on Office Information Systems* (Cambridge, MA), pp. 227–237.

13. Pederson, E.R. and Sokoler, T. AROMA: Abstract representation of presence supporting mutual awareness. In *Proceedings of CHI'97* (March 22–27, Atlanta GA), ACM/SIGCHI, N.Y., 1997, pp. 51–58.

14. Rekimoto, J. Tilting operations for small screen interfaces. In *Proceedings of UIST'96* (Nov. 5–8, Seattle, WA), ACM/SIGCHI, N.Y., 1996, pp. 167-168.

15. Salber, D., Dey, A.K., and Abowd, G.D. The Context Toolkit: Aiding the development of context-enabled applications. To appear in *Proceedings of CHI'99* (May 15–May 20, Pittsburgh, PA), ACM/SIGCHI, N.Y., 1999.

16. Schilit, W.N. System architecture for context-aware mobile computing. Ph.D. Thesis, Columbia University. May 1995.

17. Schwartz, M.F. *et al*. A comparison of internet resource discovery approaches. *Computing Systems*, (Fall 1992), 461-493.

18. Ullmer, B. and Ishii, H., The metaDESK: Models and prototypes for tangible user interfaces. In *Proceedings of UIST'97* (Oct. 14–17, Banff, Alberta, Canada), ACM/SIGCHI, N.Y., 1997, pp. 223-232.

19. Want, R. *et al*. The Active Badge location system. *ACM Transactions on Information Systems 10*, 1 (Jan. 1992), 91–102.