# A Context-Based Infrastructure for Smart Environments

Anind K. Dey, Gregory D. Abowd and Daniel Salber

Graphics, Visualization and Usability Center and College of Computing,
Georgia Institute of Technology, Atlanta, GA, USA 30332-0280
{anind, abowd, salber}@cc.gatech.edu

**Abstract.** In order for a smart environment to provide services to its occupants, it must be able to detect its current state or *context* and determine what actions to take based on the context. We discuss the requirements for dealing with context in a smart environment and present a software infrastructure solution we have designed and implemented to help application designers build intelligent services and applications more easily. We describe the benefits of our infrastructure through applications that we have built.

## 1 Introduction

One of the goals of a smart environment is that it supports and enhances the abilities of its occupants in executing tasks. These tasks range from navigating through an unfamiliar space, to providing reminders for activities, to moving heavy objects for the elderly or disabled. In order to support the occupants, the smart environment must be able to both detect the current state or *context* in the environment and determine what actions to take based on this context information. 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. This information can include physical gestures, relationship between the people and objects in the environment, features of the physical environment such as spatial layout and temperature, and identity and location of people and objects in the environment.

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. A smart environment, therefore, will be populated by a collection of context-aware applications or services. The increased availability of commercial, off-the-shelf sensing technologies is making it more viable to sense context in a variety of environments. These sensing technologies enable smart environments to interpret and begin to understand the contextual cues of its occupants. 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 context-aware 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. There has been little attempt to insulate the sensing of context in the physical world from using the context in applications. This makes it difficult to build new applications and to transfer existing applications to new environments with different or changing sensing technologies. To enable designers to easily build context-aware applications, there needs to be architectural support that provides the general mechanisms required by all context-aware applications. This support should include an insulation layer that on one side can hide the details of how context is sensed from applications, and on the other side, that can provide persistent context in a flexible manner without worrying about what, if any, applications require it.

At Georgia Tech, we have begun the construction of an experimental home to serve as a living laboratory for the experimentation of a number of ubiquitous computing research problems, including context-aware application development. One of the goals is to produce an "Aware Home" that will know contextual information about itself as well as information about its inhabitants [1][11].

There are a number of human-centered arguments for why an aware home is both desirable and worrisome. For instance, it may allow elderly to "age in place", remaining in familiar surroundings of their domicile as an alternative to more expensive and potentially alienating assisted living centers. But an environment that is instrumented to know too much about occupants' activities might not be a relaxing place to reside. Investigating these human-centered themes is a central focus of the long-term research plan. A necessary stepping-stone is the software infrastructure to allow us to rapidly prototype intelligent, or context-aware, services within such an environment.

This paper describes a distributed software infrastructure to support context-aware applications in the home environment. 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 [17], we presented the concept of context widgets, which allow us to handle context in a manner analogous to user input. This paper discusses the infrastructure support necessary for using context and context widgets. We derive the requirements for the infrastructure by examining the differences between the uses of context and input. Next, we present our solution for supporting the design and execution of context-aware applications in smart environments. Finally, we discuss the benefits and limitations of the infrastructure, based on our experiences in building context-aware applications.

## 2 Discussion of Context Handling

We have discussed the importance of context in smart environments. 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

[1] More information on the Aware Home is available at http://www.cc.gatech.edu/fce/house

discuss some concepts that will allow us to handle context in the same manner as we handle user input.

### 2.1 Current Context Handling

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

With some applications [8,16], the drivers for sensors used to detect context are directly hardwired into the applications themselves. Here, 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 [20]. Here, 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 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 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.

### 2.2 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 [13]. 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.

### 2.3 Analogy of Input Handling to Context Handling

There have been previous systems which handle context in the same way that we handle input [2, 18]. 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 [18]).

Previously, we demonstrated the application of the widget abstraction to context handling [17]. 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 infrastructure to support the context widget construct. The remainder of this paper will describe the requirements for this architecture and will describe our architectural solution.

## 3 Infrastructure 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 in a smart environment can come from many, distributed sources
- 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 [14] and virtual environments [5] deal with distributed input and user modeling techniques [9] abstract input, for example), they are not the norm. Because of the differences between input and context, unique infrastructure support is required for handling context and context widgets. We will now derive the requirements for this infrastructure.

### 3.1 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 in an instrumented smart environment, 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 infrastructure design stemming from the heterogeneity of computing platforms and programming languages that can 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 widgets themselves, may not be the same. The infrastructure must support interoperability of context widgets and applications on heterogeneous platforms.

### 3.2 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 infrastructure.

To facilitate the building of context-aware applications, our infrastructure 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.

### 3.3 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 [20]. 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.

### 3.4 Requirements Summary

We have presented requirements for a software infrastructure 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; and
- support the storing of context history.

In the next section, we describe the context-based infrastructure that we have built to address these requirements.

## 4 Description of Infrastructure

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

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

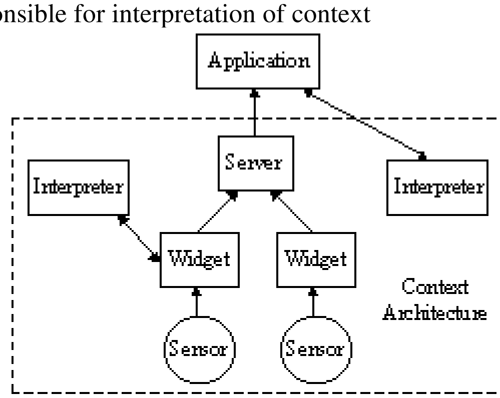


**Figure 1. Data flow between applications and the context-based infrastructure.**

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 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.

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

### 4.1 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 allows components to retrieve historical context information. The basic Widget object provides these services 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.

### 4.2 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".

### 4.3 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()`.

### 4.4 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 infrastructure. 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

- 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 infrastructure 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.

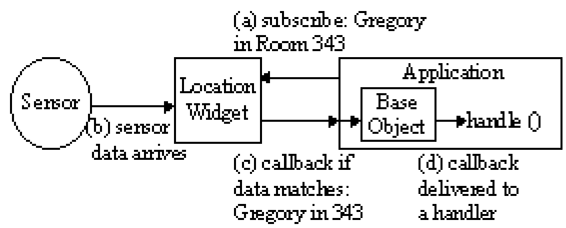


**Figure 2. Example of application interacting with context widget. Arrows indicate communications flow.**

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 to send the data to. It calls that object's `handle()` method, as 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), sending and receiving communications. The BaseObject uses a "pluggable" communications scheme. By default, it supports communications using the HyperText Transfer Protocol (HTTP) for both sending and receiving messages. The language used for sending data is, by default, the eXtensible Markup Language (XML). The BaseObject does 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 and incoming communications. She then passes the name of this object to the BaseObject at the time of instantiation. The BaseObject uses this object rather than the default object. 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 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.

# 5 Experience and Contributions

In this section, we describe the benefits and limitations of the context-based infrastructure for building context-aware applications. We will discuss these issues using context-aware applications that have already been created with this infrastructure.

We have built multiple types of each object (widgets, servers, and interpreters) and included this library of objects in our Context Toolkit[2]. Our default implementation of the infrastructure 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 infrastructure executing on Windows CE, 95 and NT, Linux, Solaris, IRIX, and the Macintosh platforms, including mobile, wearable, and desktop machines.

## 5.1 Benefits

The benefits of the context-based infrastructure are that it:
- hides the details of the context sensors; and
- supports lightweight integration of components;
- makes it easy to add context to existing applications that don't use context;
- makes it easy to add more context to applications that already use context; and
- supports reusability by multiple applications.

We will discuss these benefits, using the applications we have developed as examples.

*Hides Context-Sensing Details* A benefit of the infrastructure is that it hides details of the context sensors. This allows the underlying sensors to be replaced, without af-

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

fecting the application. We have built multiple applications that use location and identity context. In each case, when we change the sensor used to provide this context information, the applications do not have to be modified. We have built context widgets that sense location and identity around JavaRings (www.ibuttons.com), the PinPoint indoor positioning system (www.pinpointco.com), and the Texas Instruments TIRIS RFID system (www.ti.com). We have also built context widgets that determine activity context, using sound-level information, motion sensors, and door open/closed status. For a third type of context, we have built context widgets that can capture presentation information from a Web browser or PowerPoint.

*Lightweight Integration* The use of lightweight communications and integration mechanisms supports the deployment of the infrastructure and applications on multiple platforms using multiple programming languages. We have implemented an In/Out Board application [17] 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[3] using Frontier on a Macintosh and Python on Linux. We built a notetaking application [7] that assists conference attendees in choosing which presentations they wanted to attend based on their personal interests and recommendations of colleagues and in taking notes on the presentations by automatically monitoring and capturing presentation information. The Conference Assistant application was executed on a Windows CE device, with widgets, servers, and interpreters executed in C++ and Java on Solaris and Windows NT and 95. A combination of motion sensors placed around entrances/exits to our research lab and magnetic reed switches that determine when a door is open/closed are being used to keep track of the number of people in our laboratory. This application was built in C++ and Java on Linux.

*Simple to Use Context* From an application builder's perspective, the infrastructure makes it simple to add the use of context to applications that previously did not use context. The DUMMBO [3] application is an augmented whiteboard that stores what is written on it and spoken around it, to aid in 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. This enables capture of spoken information that would otherwise be lost. The change required the addition of 25 lines of Java code. The significant changes included 2 lines added to use the infrastructure, 1 line modified to enable the class to handle callbacks, and 17 lines that are application specific.

*Simple to Add Context* Based on these results, we argue that it is also simple to add additional context to applications that already use the infrastructure. In the Conference Assistant application, the use of context was made much simpler through the use of context servers (for each user and presentation room). From the application builder's viewpoint, the servers make accessing context information much easier than

---

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

dealing with the multiple widgets individually. If a new type of 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.

*Supports Multiple Simultaneous Applications* An important feature of the infrastructure is independent execution. This lets the context-based infrastructure run independently of applications, which allows multiple applications to use the infrastructure simultaneously. We have built a number of applications that leverage off of the context-based infrastructure 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.

### 5.2 Current Limitations and Future Work

Although our infrastructure eases the building of context-aware applications, some limitations remain. In particular, it does not currently support continuous context, 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.

Currently, the infrastructure only supports discrete context, and not continuous context such as a video feed or GPS location information. Given that a large number of sensors in a smart environment will provide continuous data, we need to support the collection of this type of context. We are investigating two possible solutions. The first is to provide a special mechanism that supports reliable, fast streaming of continuous context from widgets to applications. The second solution is to interpret the continuous data in real-time and have context widgets provide the discrete interpretations to applications.

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 is able 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 infrastructure 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 perform sensor fusion in an *ad hoc* fashion, but we are exploring a general mechanism for supporting it.

The infrastructure 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 infrastructure supports a form of resource discovery [19], 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 infrastructure 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 infrastructure 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 infrastructure 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.

## 6 Related Work

In our previous discussion on context handling, we discussed work that influenced us in our decision to treat context like input. We can see from the recent CoBuild workshop and AAAI Symposium on Intelligent Environments that there has been a lot of related work in smart environments. Rather than review the current state of smart environments, we will look at work that supports the use of context.

The proposed Situated Computing Service [10] has an infrastructure that is similar in intent to ours. It insulates applications from context sensors. It 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 infrastructure. A single prototype server has been constructed as proof of concept, using a single sensor type.

The AROMA system [15] explored awareness in media spaces. Its architecture used sensor abstraction and interpretation to provide awareness of activity between remote sites. Interpreted context at each site was displayed at the other sites.

The CyberDesk system [6] used minimal context to provide relevant services to a desktop computer user. Its modular architecture separated context sensing, abstraction, and notification. It did not use a distributed architecture or support aggregation.

## 7 Conclusions

We have presented a context-based infrastructure for supporting the software design and execution of context-aware applications in the Aware Home, a prototype smart environment. Our infrastructure builds upon our previous work [17] that introduced the idea of a context widget for treating context like user input. We generated requirements for the infrastructure based on the differences between dealing with context and input. Using these requirements, we designed and built an infrastructure to make it easier for applications to deal with context. We discussed the benefits of the infrastructure through example applications that we have built. Finally, we described the limitations of the current infrastructure and, as part of our future work, plan to address these with the suggested improvements. We intend to test the context-based infrastructure through the use of a larger variety of context and the building of more complex applications within the Aware Home.

## 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 (1991) 27-41.
2. Bauer, M. *et al*. A collaborative wearable system with remote sensing. In Proceedings of International Symposium on Wearable Computers (1998) 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. (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 (1993) 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) 3-13.
7. Dey, A.K. *et al*. The Conference Assistant: Combining context-awareness with wearable computing. To appear in the Proceedings of the International Symposium on Wearable Computers '99.
8. Harrison, B.L. *et al*. Squeeze me, hold me, tilt me! An exploration of manipulative user interfaces. In Proceedings of CHI'98 (1998) 17–24.
9. Horvitz, E. *et al*. The Lumiere Project: Bayesian user modeling for inferring the goals and needs of software users. In 14$^{th}$ Conference on Uncertainty in Artificial Intelligence (1998) 256-265.
10. Hull, R., Neaves, P., and Bedford-Roberts, J. Towards situated computing. In Proceedings of International Symposium on Wearable Computers (1997) 146–153.
11. Kidd, C, et al. The Aware Home: A living laboratory for ubiquitous computing research. To appear in the Proceedings of CoBuild'99 (1999).
12. Lamming, M. *et al*. The design of a human memory prosthesis. Computer 37, 3 (1994) 153-163.
13. MacIntyre, B. and Feiner, S. Language-level support for exploratory programming of distributed virtual environments. In Proceedings of UIST'96 (1996) 83–94.
14. Greenberg, S. Sharing views and interactions with single-user applications. In Proceedings of the ACM/IEEE Conference on Office Information Systems (1990) 227–237.
15. Pederson, E.R. and Sokoler, T. AROMA: Abstract representation of presence supporting mutual awareness. In Proceedings of CHI'97 (1997) 51–58.
16. Rekimoto, J. Tilting operations for small screen interfaces. In UIST'96 (1996) 167-168.
17. Salber, D., Dey, A.K., and Abowd, G.D. The Context Toolkit: Aiding the development of context-enabled applications. In Proceedings of CHI'99 (1999) 434-441.
18. Schilit, W.N. System architecture for context-aware mobile computing. Ph.D. Thesis, Columbia University (1995).
19. Schwartz, M.F. *et al*. A comparison of internet resource discovery approaches. Computing Systems (1992) 461-493.
20. Want, R. *et al*. The Active Badge location system. ACM TOIS 10, 1 (1992) 91–102.