Distributed mediation of ambiguous context in aware environments

Anind Dey[1, 2], Jennifer Mankoff[2], Gregory Abowd[3] and Scott Carter[2]

[1]Intel Research, Berkeley
Intel Corporation
Berkeley, CA 94720-1776
anind@intel-research.net

[2] EECS Department
UC Berkeley
Berkeley, CA 94720-1776
{jmankoff, sacarter}@cs.Berkeley.edu

[3]College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280
abowd@cc.gatech.edu

ABSTRACT

Many context-aware services make the assumption that the context they use is completely accurate. However, in reality, both sensed and interpreted context is often ambiguous. A challenge facing the development of realistic and deployable context-aware services, therefore, is the ability to handle ambiguous context. In this paper, we describe an architecture that supports the building of context-aware services that assume context is ambiguous and allows for mediation of ambiguity by mobile users in aware environments. We illustrate the use of our architecture and evaluate it through three example context-aware services, a word predictor system, an In/Out Board, and a reminder tool.

Keywords: context-aware computing, ambiguity, aware environments, ubiquitous computing, mediation, error handling.

INTRODUCTION

A characteristic of an aware, sensor-rich environment is that it senses and reacts to *context*, information sensed about the environment's mobile occupants and their activities, by providing context-aware services that facilitate the occupants in their everyday actions. Researchers have been building tools and architectures to facilitate the creation of these context-aware services by providing ways to more easily acquire, represent and distribute raw sensed data and inferred data [16]. Our experience shows that though sensing is becoming more cost-effective and ubiquitous, the interpretation of sensed data as context is still imperfect and will likely remain so for some time. A challenge facing the development of *realistic* and *deployable* context-aware services, therefore, is the ability to handle imperfect, or *ambiguous*, context. This paper presents a runtime architecture that supports programmers in the development of multi-user, interactive, distributed applications that use ambiguous data.

Researchers in aware environments have used techniques from the artificial intelligence (AI) community, including Bayesian networks and neural networks [7,17], to deal with imperfect context. However, the techniques cannot remove all the ambiguity in the sensed data, leaving it up to the aware environment programmer and occupants to deal with. To alleviate this problem, we propose to leverage off any useful AI techniques for reducing the ambiguity and involve end users in removing any remaining ambiguity, through a process called *mediation* [14].

In graphical user interface (GUI) design, mediation refers to the dialogue between the user and computer that resolves questions about how the user's input should be interpreted in the presence of ambiguity. A common example of mediation in recognition-based GUIs is the *n*-best list. Ambiguity arises when a recognizer is uncertain as to the current interpretation of the user's input, as defined by the user's intent. An application can choose to ignore the ambiguity and just take some action (*e.g.* act on the most likely choice), or can use mediation techniques to ask the user about her actual intent. Ambiguous context, from an aware environment, can produce errors similar to those in recognition-based interfaces.

In previous work, we presented an architecture for the development of context-aware services, that assumed context to be unambiguous [8]. We also developed an architecture to support the mediation of ambiguity in recognition-based GUI interfaces [14]. While we build on this past work, our contribution in this paper is to solve the *additional* architectural requirements (justified in the next section) that arise as a result of requesting highly mobile users to mediate ambiguous context in *distributed, interactive,* sensing environments. We support:

- Timely delivery and update of ambiguous events across an interactive distributed system;
- Delayed storage of context once ambiguity is resolved;
- Delivery of ambiguous context to multiple applications that may or may not be able to support mediation;
- Pre-emption of mediation by another application or component;
- Applications or services in requesting that another application or service mediate; and,
- Distributed feedback about ambiguity to users in an aware environment.

Our runtime architecture addresses these issues and supports our goal of building more realistic context-aware applications that can handle ambiguous data through mediation.

UIST'02, October 27-30, 2002, Paris, FRANCE.

Overview

We begin by presenting a motivating example used to illustrate the requirements of mediation in a context-aware setting. In the next section, we present brief overviews of previous work that we have extended. We show how they were combined to deal with ambiguous context, and describe additional architectural mechanisms that were developed for the requirements unique to mediation of context in a distributed setting. Our next section demonstrates how our architecture and these mechanisms support the implementation of our motivating example. We then evaluate the architecture by describing what was required on the part of a programmer to modify two existing context-aware applications to support mediation. We complete our evaluation by describing how our architecture supports experimenting with multiple mediators, in the context of the motivating example, implemented entirely with our architecture. We conclude the paper with related work and a discussion of further challenges in mediating interactions in context-aware applications.

MOTIVATING EXAMPLE

We have developed three applications as demonstrations of our architecture. One in particular, a context-aware communication system, the Communicator, will be used to illustrate key points throughout this paper, and we introduce it here.

The Communicator is designed for people with motor and speech impairments. For these people, exemplified by Stephen Hawking, computers can provide a way to communicate with the world and increase both independence and freedom. Many people with severe motor impairments can control only a single switch, triggered by a muscle that is less spastic or paralyzed than others. This switch is used to scan through screen elements, such as the keys of a soft keyboard. Input of this sort is very slow and is often enhanced by word prediction.

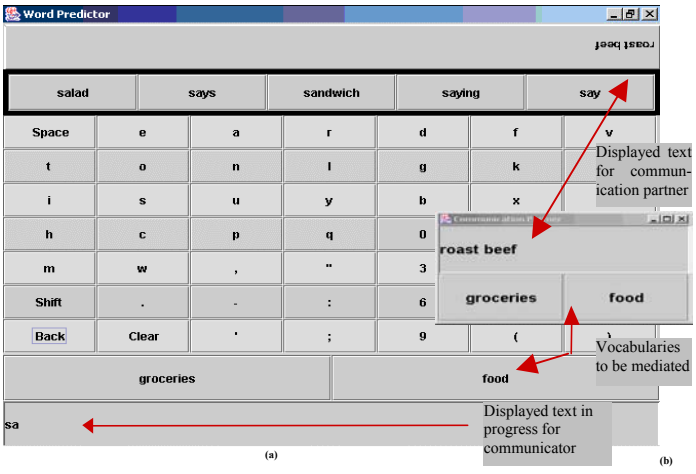

Figure 1 (a) Communicator (back) and partner (right, front) interfaces.

The Communicator, shown in Figure 1, is based on a *word* predictor that attempts to predict what word a user is typing from the letters that have been typed so far. The non-speaking individual uses the interface shown Figure 1a. The keyboard layout shown was chosen for optimal efficiency for scanning users. Text is displayed to the (abled) communication partner at top, reversed for easy readability by someone facing the user, across a flat display and in a separate interface (Figure 1b). Word predictors are very inaccurate, and because of this, they usually display a list of possible predictions that the user scans through for the correct choice, often to no avail. Word prediction is especially difficult to use for spoken communication because the speed of conversational speech often reaches 120 words per minute (wpm) or more, while users of word prediction rarely go above 10 wpm.

The goal of the Communicator is to facilitate conversational speech through improved word prediction. We augment word prediction by using a third party intelligent system, the Remembrance Agent [18], to select conversation topics, or *vocabularies,* based on contextual information including recent words used, a history of the user's previous conversations tagged with location and time information, the current time and date and the user's current location. These vocabularies help to limit the set of predicted words to those that are more relevant and thus improve prediction. For example, when in a bank, words such as "finance" and "money" should be given priority over other similar words. This has been effective for predicting URLs in Netscape™ and Internet Explorer™ and, in theory, for non-speaking individuals [12,15]. Our goal was to build an application to support context-aware word prediction for non-speaking individuals.

Unfortunately, it is hard to accurately predict the topic of a user's conversation, and because of this, the vocabulary selection process is ambiguous. We experimented with several mediation strategies, ranging from simply and automatically selecting the top choice vocabulary without user intervention to stopping the conversation in order to ask the user which vocabulary is correct.

REQUIREMENTS

The focus of this paper is to support programmers in building realistic context-aware applications by first addressing the architectural issues needed to support mediation of ambiguous input. The first two issues are obvious: there must exist a system that is able to capture context and deliver it to interested consumers, and there must be mediation techniques for managing ambiguity. These issues were dealt with in our previous work. In the following subsections we discuss the interesting additional challenges that arise from mediating ambiguous context, all of which are supported by the architecture presented here.

Context Acquisition and Mediation

One common characteristic of context-aware applications is the use of sensors to collect data. In the Communicator, location and time information is used to help improve word prediction. A user's location can be sensed using Active Badges, radar, video cameras or GPS units. All of these sensors have some degree of ambiguity in the data they

sense. A vision system that is targeted to identify and locate users based on the color of the clothing they wear will produce inaccurate results if multiple users are wearing the same color of clothing. The ambiguity problem is made worse when applications derive implicit higher-level context from sensor data. For example, an application may infer that a meeting is occurring when a number of users move into the same room in a given time interval. However, there may be other explanations for this phenomenon, including random behavior, lunchtime or the workday has started and multiple people are arriving at their desk. Even with the use of sophisticated Bayesian networks or other AI techniques, low- and high-level inferences are not always correct, resulting in ambiguity.

Distribution

Most context-aware applications are distributed across multiple computing devices. Applications or system components that are interested in context (often called *subscribers*) are running on devices that are remote from components that are gathering context. The component gathering the context may not be the component that mediates it, since it may not have an interface. In the Communicator, the user's interface and the communication partner's interface are running on separate devices. It is important to minimize the number and duration of network calls in an interactive distributed system, and thus, to only send the information absolutely needed for mediation to only those components that are performing the mediation.

Storage

Because context-aware systems are often distributed and asynchronous, and because sensor data may be used by multiple applications, it is beneficial to store data being gathered by sensors. The Communicator takes advantage of stored information by accessing past conversations that match the user's current location and time. Storing context data allows applications that were not running at the time the data was collected to access and use this historical data. When that data is ambiguous, several versions must be saved, making the storage requirements prohibitive. Interesting issues to address are when should we store data (before or after ambiguity is resolved) and what should we store (ambiguous or unambiguous context).

Multiple Subscription Types

In many context-aware systems, multiple subscribers are interested in a single piece of sensed input. An interesting issue is how to allow individual components to "opt in" to ambiguous context while allowing others to "opt out". Some components may wish to deal with ambiguity while others may not. For example, non-interactive components such as a data logging system may not have any way to interact with users and therefore may not support mediation. Other components, like the Communicator interface, may only wish to receive unambiguous data. In the same manner, a logging system might wish to only record data that is certain. A second issue to deal with is allowing components to deal with ambiguous data while not requiring them to perform mediation. Later in the paper, we will discuss a word predictor widget in the Communicator that has this same property.

Pre-emption of Mediation

In our system, multiple, completely unrelated components may subscribe to the same ambiguous data source. Both Communicator interfaces have the ability to mediate ambiguous vocabularies, for example. An important concern to resolve is what to do when these components start to mediate that data at the same point in time.

Forced mediation

There are cases where a subscriber does not wish to mediate ambiguous data itself, but may still wish to exert some control over the timing of when another subscriber completes mediation. One way of doing this is allowing it to request immediate mediation by others. In the Communicator, when a conversation ends, a component responsible for managing past conversations wants to store this conversation in an appropriate vocabulary. This component does not have an interface, so it requests that the application mediate the possible vocabularies.

Feedback

When distributed sensors collect context about a user, a context-aware system needs to be able to provide feedback about the ambiguous context to her, particularly when the consequences are important to her. In a typical aware environment, users are mobile and may interact with multiple devices throughout their interaction with the environment. For this reason, the architecture needs to support the use of remote feedback, providing feedback (visual or aural, in practice) on a device that may be remote from both the subscribing component and the sensing component. Take the previous example of a user's motion being monitored by a video camera to provide identity and location-based services. As the user moves down a hallway, a device on the wall may display a window or use synthesized speech to indicate who the video camera system thinks the user is. This device is neither a subscriber of the context nor the context sensor, but simply has the ability to provide useful feedback to users about the state of the system. We will present an implemented example of feedback in our discussion of the reminder application.

In the next section, we will discuss the architecture we designed and implemented to deal with these requirements.

MEDIATING AMBIGUOUS CONTEXT

We built support for mediation by extending an existing toolkit, the Context Toolkit [8]. The Context Toolkit is a software toolkit for building context-aware services that support mobile users in aware environments. There are two basic building blocks that are relevant to this discussion: context widgets and context interpreters. Figure 2 shows the relationship between context components and applications.

Context widgets, presented elsewhere [8] and based on an analogy to GUI widgets, are responsible for collecting

contextual information about the environment and its occupants. They provide a uniform interface to components or applications that use the context, hiding the details of the underlying context-sensing mechanism(s). These widgets allow the use of heterogeneous sensors that sense redundant input, regardless of whether that input is implicit or explicit. Widgets maintain a persistent record of all the context they sense. They allow applications and other widgets to both query and subscribe to the context information they maintain.

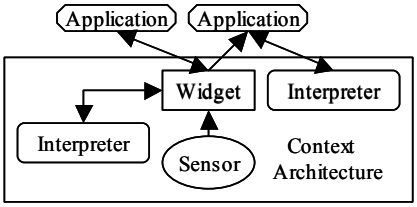

Figure 2 Context Toolkit components: arrows indicate data flow.

A *context interpreter* is used to abstract or interpret context. For example, a context widget may provide location context in the form of latitude and longitude, but an application may require the location in the form of a street name. A context interpreter may be used to provide this abstraction. A more complex interpreter may take context from many widgets in a conference room to infer that a meeting is taking place. Both interpreters and widgets are sources of ambiguous data.

Modifications for Mediation

In order to explain how we met the requirements given in the previous section, we must first introduce the basic abstractions we use to support mediation. We chose to base our work on the abstractions first presented in the OOPS toolkit [14], a GUI toolkit that provides support for building interfaces that make use of recognizers (*e.g.* speech, gestures) that interpret user input. We chose OOPS because it explicitly supports mediation of single-user, single application, non-distributed, ambiguous desktop input, a restricted version of our problem.

OOPS provides an internal model of recognized input. This model encapsulates information about ambiguity and the relationships between input and interpretations of that input that are produced by recognizers in a graph (See Figure 3). The graph keeps track of source events, and their interpretations (which are produced by one or more recognizers).

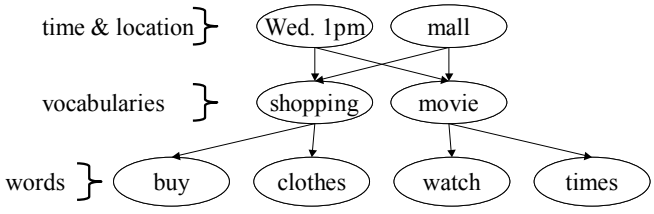

Figure 3 An event graph representing predicted words from context.

Like OOPS, our toolkit automatically identifies ambiguity in the graph and intervenes between widgets and interpreters and the application by passing the directed graph to a mediator. A mediator displays a portion of the graph to the user. Based on the user's response, the mediator *accepts* or *rejects* events (*i.e.* keeps correct interpretations or removes incorrect interpretations) in the graph. Once the ambiguity is resolved (all events in the graph are accepted or rejected), the toolkit allows processing of the input to continue as normal. Figure 4 shows the resulting changes. The gray boxes indicate components that have been added to the Context Toolkit architecture illustrated in Figure 2 to support mediation of ambiguous context.

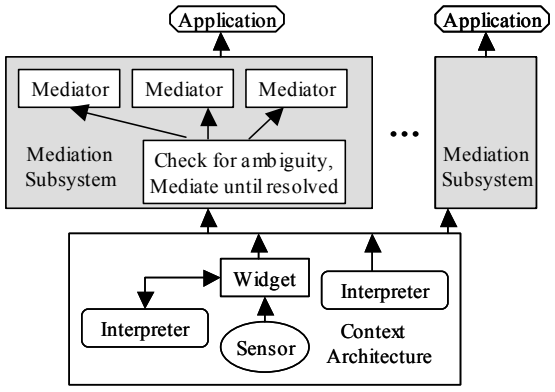

Figure 4 The architecture for the extended Context Toolkit. Everything in the gray box is new.

Example

Before discussing the additional changes necessary to support the requirements listed above, we illustrate the use of ambiguous hierarchical events in the Context Toolkit with an example. In the Communicator system, time and location information is used to choose relevant vocabularies. An intelligent recognition system provides the most likely vocabularies and then these are interpreted into the words the user is most likely to be typing. The set of vocabularies and the set of words are stored as sets of alternatives with associated confidences (a fairly common representation). Each of these alternatives becomes an ambiguous event in our system. The result is a directed graph, like that shown in Figure 3.

Often only one path through this graph is correct from the user's perspective (*e.g.* mall & Wednesday → shopping → clothes). We call this situation *ambiguous* and mediation is used to resolve the ambiguity. In particular, a mediator will display feedback about one or more interpretations to the user, who will then select one or repeat her input.

Now suppose that an application subscribes to this data. All three of the applications we present later make use of location data. Subscribers to location data may then:

1) Wait until all the ambiguity has been resolved before taking any action on a location update; or,

2) Take action on the ambiguous data by:

   a) Asking for a user to help mediate the data;

   b) Picking one of the alternatives (usually the one with the highest confidence) and acting on it; or

c) Performing some higher-level inference (such as the word a user is typing) with its own ambiguity. This increases the depth and complexity of the event graph.

Modifications for New Requirements

The previous subsections described the basic abstractions used to support mediation: widgets, interpreters, applications, mediators and the event graph. We now explain the additional architectural mechanisms needed to support the unique problems faced by mediation of ambiguous context, introduced above.

Distribution

The original OOPS toolkit was designed to support mediation in non-distributed GUI applications. It always passed a pointer to the entire graph to mediators.

In order to support appropriate response times in the distributed environment of the Context Toolkit, only those portions of the event graph that have subscribers are passed across the network. Otherwise, each time a new event is added or an existing event is accepted or rejected, every component interested in the ambiguous context would have to be notified. In a distributed system, this would impede our ability to deliver context in a timely fashion, as is required to provide feedback and action on context.

No one component contains the entire graph being used to represent ambiguity of a particular piece of context. The graph is, instead, distributed across multiple components (widgets and interpreters) and copies of particular graph levels are provided to applications, as needed. Each event or element in the graph has a list of its source events (parent(s)) and its interpretations (children). Rather than having the lists contain full representations of the sources and interpretations, the lists instead contain event proxies. An event proxy consists of an event id, the status (accepted, rejected or undetermined) of the event and communication information (hostname, port number, component name) for the component that produced the event and contains its full representation. Because components mostly care about the status of their sources and interpretations, the proxies allow components to operate as if they had local access to the entire graph and to request information about parts of the event graph that they do not have locally.

Storage

As described above, storage of context data is a useful feature of a context-aware architecture. However, when context is ambiguous, it is not immediately obvious what should be stored and when. One option is to store all data, regardless of whether it is ambiguous or not. This option provides a history of user mediation and system ambiguity that could be leveraged at some later time to create user models and improve recognizers' abilities to produce interpretations. We chose to implement a less complex option: By default, every widget stores only unambiguous data. Another dimension of storage relates to when data is stored. Since we are storing unambiguous data only, we store context data only after it has been mediated.

The reason for our choice is two-fold: the storage policy is easier to deal with from an access standpoint and we gain the benefits offered by knowledge of ambiguity during the mediation process, just not at some arbitrary time after mediation (when the record of ambiguity has been discarded). In any case, it would be relatively simple to modify the architecture to support the first option as a default.

Multiple Subscription Types

Because multiple components may be interested in the same piece of context, and only some may be interested in ambiguous data, components need a way of specifying whether they want to handle ambiguous data. In our architecture, they simply set a Boolean flag to specify this.

Components that accept ambiguous data are not required to perform mediation. They can take any action they wish with the unmediated data. Components that accept unambiguous data also are not required to perform mediation, but they must wait until another component does (or force mediation, as described below) before they receive the data.

In either case, when a component successfully mediates data, other components interested in the data are notified. The architecture keeps track of all the recipients of the ambiguous data and updates them. As well, it keeps track of any components waiting for unambiguous versions of the data and passes the mediated data to them. Finally it notifies the components that produced the ambiguous data who can use the data to improve their ability to produce new data.

Pre-Emption of Mediation

Because multiple components may subscribe to the same ambiguous data, mediation may actually occur simultaneously in these components. If multiple components are mediating at once, the first one to succeed "interrupts" the others and updates them. This is handled automatically by the architecture when the successful mediator accepts or rejects data. The architecture notifies any other recipients about the change in status. Each recipient determines if the updated data is currently being mediated locally. If so, it informs the relevant mediators that they have been pre-empted and should stop mediating. Past work did not handle mediation in multiple distributed components. Other strategies for handling simultaneous mediation are discussed in the future work section.

Forced Mediation

In cases where a subscriber of ambiguous context is unable to or does not want to perform mediation, it can request that another component perform it. The subscriber simply passes the set of ambiguous events it wants mediated to a remote component and asks that remote component to perform mediation. If the remote component is unable to do so, it notifies the requesting component. Otherwise, it performs mediation and updates the status of these events, allowing the requesting component to take action.

Feedback

Since context data may be gathered at locations remote from where the active application is executing and at times remote from when the user is interacting with the active application, there is a need for distributed feedback services that are separate from applications. To support distributed feedback, we have extended context widgets to support feedback and actuation via output services. Output services are quite generic and can range from sending a message to a user to rendering some output to a screen to modifying the environment. Some existing output services render messages as speech; send email or text messages to arbitrary display devices; and control appliances such as lights and televisions. Any application or component can request that an output service be executed, allowing any component to provide feedback to a user.

In this section, we described modifications to the Context Toolkit that will allow for human-driven distributed mediation of imperfectly sensed and interpreted context. In the next two sections, we demonstrate how the architectural solutions provided by the modified Context Toolkit were used to implement our motivating example and to modify two existing applications so that they can support mediation for ambiguity.

USE OF ARCHITECTURE

In this section, we use the Communicator to illustrate the runtime behavior of the architecture. We also use it to show how, in practice, a programmer designing a context-aware application uses the architecture.

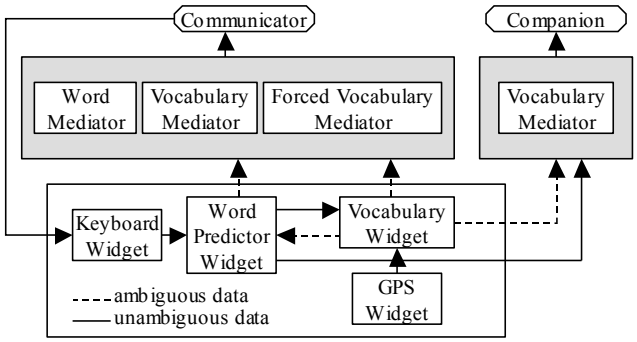

Figure 5 Architecture for the Communicator System

Runtime description of Communicator architecture

In order to illustrate how the toolkit works at runtime, we first need to describe some details of the Communicator system. Figure 5 shows the architecture described below.

Applications and widgets

The Communicator makes direct use of data from three widgets: a soft keyboard, a word predictor and a vocabulary selector. The keyboard widget produces unambiguous data and simply lets other components know what the user is typing. The word predictor widget produces ambiguous data and uses the current context to predict what word the user is trying to type. It uses a unigram, frequency-based method common in simple word predictors, as well as a history of recent words. It subscribes to the keyboard to get the current prefix (the letters of the current word that have been typed so far). As each letter is typed, it suggests the most likely completions. The word predictor also uses weighted vocabularies to make its predictions. It subscribes to the vocabulary widget to get a list of ambiguous, probable vocabularies and uses the probability associated with each suggested vocabulary to weight the words from that vocabulary. As described earlier, the vocabulary widget uses the Remembrance Agent [18] to suggest relevant, yet ambiguous vocabularies for the current conversation.

If the person the user is communicating with also has a display available, a companion application can be run. This application presents an interface (see Figure 1), showing the unambiguous words selected by the user and the current set of ambiguous vocabularies.

In summary, this application uses two unambiguous widgets (GPS and keyboard), and two widgets that generate ambiguous data, one based on a third party recognizer (vocabulary), and one based on an in-house recognizer (word). Unlike typical context-aware systems, ambiguity in our systems is retained, and, in some cases, displayed to the user.

Mediation

Ambiguous information generated in our system includes potential vocabularies and potential words. The architecture allows a component to mediate ambiguous context, use it as is, or use it once something else has mediated it. All three cases exist in this system. The application mediates both ambiguous words and vocabularies. The word predictor uses ambiguous vocabularies. The vocabulary widget uses unambiguous words after the user has mediated them. The word mediator is graphical and it displays ambiguous words as buttons in a horizontal list, shown *in situ* near the bottom of Figure 1a. A word may be selected by the user or ignored. The mediator replaces all the displayed words whenever it receives new words from the word predictor.

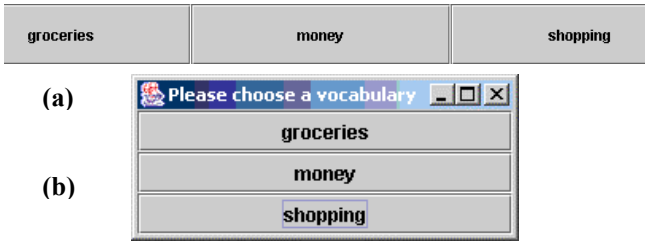

Figure 6 Screenshots of mediators (a) choice mediator for words or vocabularies and (b) required mediator for vocabularies.

We experimented with four different strategies for mediating ambiguous vocabularies. The first simply accepts the vocabulary with the highest probability without user input (equivalent to no mediation at all). The second (see Figure 6a) displays the choices similarly to words, and allows the user to ignore them. The last two require the user to choose a vocabulary at different points in the conversation (Figure 6b). The third requires a choice when a new conversation starts and new ambiguous vocabularies are suggested. The fourth displays the choices, but only

requires that the user choose one when a conversation has ended. The mediated vocabulary name is used to append the current conversation to the appropriate vocabulary file, which then improves future vocabulary/word prediction. These approaches demonstrate a range of methods whose appropriateness is dependent on recognizer accuracy. The architecture easily supports this type of experimentation by allowing programmers to easily swap mediators.

Event Graph

We will now describe how the architecture works from a system perspective. When all of the widgets and user interfaces are started, the word predictor generates an initial set of guesses of likely words, based on an "empty" prefix from the keyboard widget.

The source event (the empty prefix) is sent to the word predictor for interpretation and the interpretations (predicted words) are passed to a handler in the user interface (UI), which immediately routes them to the word mediator for display because they are ambiguous. The user may select one, in which case, the mediator accepts that word and rejects all of the others. The toolkit then proceeds to notify the interpretations' and source event's producers (word predictor and keyboard widgets, respectively), and all the recipients. The word predictor adds the accepted word to a "recent words" list used to enhance prediction. The Communicator UI and the companion application's UI (Figure 1) display the word to the user and companion.

If the user types a letter with the soft keyboard, that letter is passed to the Communicator UI (which displays it at the bottom) and to the word predictor. The word predictor uses that and all subsequent letters as sources of its predictions and once again the user may resolve the predictions by selecting a word.

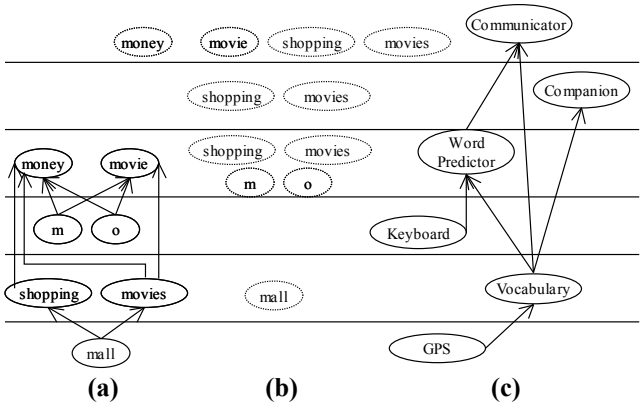

Figure 7 (a) Sample event graph and distribution across components (c). Events exist both in the components that created them and the (b) components they were sent to (*e.g.* the "money" word event exists in the Word Predictor Widget and the Communicator interface).

Meanwhile, the vocabulary widget attempts to find relevant vocabularies every time the user enters a new word or the user changes location. These ambiguous vocabularies are received by the word predictor widget, which then predicts new words (see Figure 7). The potential vocabularies are displayed by a vocabulary mediator in both the Communicator UI and the companion's UI. If either person selects a vocabulary, the architecture notifies the other mediator that it has been pre-empted. When using the fourth vocabulary mediation strategy, the vocabulary widget forces mediation to request selection of a vocabulary at the end of a conversation (signaled by a long break in keyboard use). The architecture passes this request on to each subscriber to see if it can perform mediation.

The Communicator receives the request and creates the dialog box mediator shown in Figure 6b. The user interacts with the mediator and selects a vocabulary. The event hierarchy is updated, and the vocabulary widget is notified that an event it created has been accepted. The widget writes the conversation out to disk in the appropriate vocabulary file.

Writing a Program

Two features define an application in our system: the data it subscribes to, and the mediators it uses. From the programmer's perspective, a new context-aware system may entail the creation of an application, mediators and widgets (the last two only if mediators and widgets from the available library of components is not sufficient). Figure 8 illustrates the demands on a programmer when creating each of these components from scratch.

Application:
- Specify whether to handle ambiguous data or not
- Create subscriptions to widgets
- Retrieve data from storage, if necessary
- Install mediators
- Handle results of subscriptions

Mediator:
- Produce some feedback about the data being mediated
- Request info about data's parents or children, if needed for mediation
- Allow the user to interact
- Accept or Reject events based on user interaction (*i.e.* mediate)
- Take mediator-specific action if pre-empted or if forced to mediate

Widget:
- Specify the data you provide
- If ambiguous data, create an event graph to send to subscribers
- Garbage collect and perform widget-specific actions on mediated data

Figure 8 Steps for building system components

The programmer can create *subscriptions*[1] in the initialization code of the application. No distinction is made between subscribing to ambiguous or unambiguous context. Any subscription data that arrives is left for the programmer to handle, as this is application-specific. The programmer also specifies whether the application wishes to receive ambiguous data using a simple Boolean flag. In the Communicator, the flag is set to false, meaning that it will only receive unambiguous data. The word prediction widget sets this flag to true, since it is able to use the likelihood of ambiguous vocabularies to improve its prediction accuracy. If an application needs access to historical context, it simply asks the relevant widget for it. Information retrieved from *storage* is never ambiguous, as stated earlier.

[1] *Italics* highlight issues impacting program design.

The programmer also specifies which mediators to install during initialization, thus allowing him to experiment with mediators directly. The programmer may wish to extend an existing mediator (from our library of mediators) in some way to be more specific to his application. In the case of the Communicator, this means modifying a reusable graphical mediator to extract the names of vocabularies or words from the ambiguous events in order to display meaningful choices to the user.

Each mediator must support the acquisition of user feedback about ambiguous data. This is usually done through the application's user interface. If this is not appropriate, the mediator can ask another component, such as the data's producer, to present *feedback* to the user. Remote feedback is used in the reminder application we discuss in the next section.

When an event is accepted or rejected by the user, the mediator updates the local part of the event graph and notifies any recipients and the event producer that the event was accepted. Due to the issues involved in *distribution* described earlier, only a portion of the event hierarchy is sent to a mediator. If necessary, a mediator may request additional events such as the sources or interpretations of the events it is mediating. In practice, we have found that the events the application subscribed to are sufficient for mediation to proceed in most cases. This is because applications tend to subscribe to events that are of interest to the user and appropriate to be displayed during mediation.

A mediator must also provide code indicating what to do when it is *pre-empted* by another mediator. For example, both the Communicator UI and the companion UI include a vocabulary mediator. If either user selects a vocabulary, the other is pre-empted. The mediator should clean things up visually and notify the user that someone else has completed mediation. The Communicator's word mediator supports this by removing the choices it has presented to the user.

Finally, because an unrelated subscriber may *force mediation*, a mediator must provide code for when it is asked to immediately mediate. If the mediator is able to mediate, nothing special is required. However, if it does not have the necessary information to resolve ambiguity itself, it should clean up its display and pass control to the next mediator in line. The Vocabulary widget forces the Communicator's vocabulary mediator to mediate when a conversation has ended because it needs to know which vocabulary file the conversation should be appended to.

A programmer may need to create a new widget to encapsulate a new source of context data. The widget must specify the type of data it produces. When it produces new data, it notifies all subscribers. If the data is ambiguous, a new event graph is created with the data as the root. If it is unambiguous, no event graph is created. When the data is mediated, the widget is notified so it can garbage collect the event and take actions on the event (such as appending a conversation to the appropriate vocabulary file).

Summary

We have shown how the architectural changes described in this paper are used in practice. First, we illustrated the runtime behavior of the Communicator application. Then we described what a programmer needs to know to create components and applications in our architecture. Now we discuss the impact our architecture had on the design of three applications.

EVALUATION OF ARCHITECTURE

We evaluated our architecture by building three applications. The first two were simple modifications of existing applications to include ambiguous sensors and mediation. The third application was built from scratch, using both ambiguous and unambiguous data sources.

In/Out Board

The first application we modified is the In Out Board [8], an application that displays the current in/out status for a group of building occupants. In the original system, an unambiguous location widget informed the application when a user entered or left a room. Users indicated their status by docking a Java iButton®.

We substituted an ambiguous location widget for the original widget. Rather than requiring explicit action from the user to determine in/out status, the new widget uses historical information combined with a motion detector to guess who is entering or leaving.

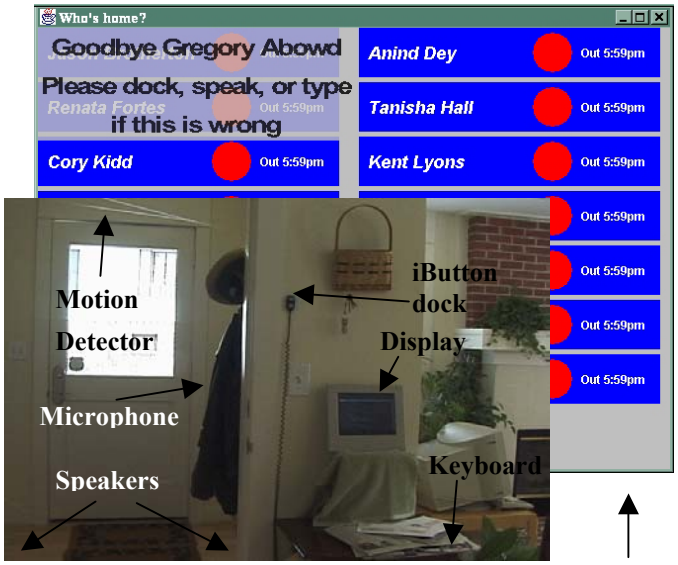

Figure 9 (a) Photograph of In-Out Board physical setup; **(b)** In-Out Board with transparent graphical feedback.

Because the original system did not support ambiguity, it ignored the fact that docking an iButton® merely provides information about user presence and not about user arrival or departure from a room. The new widget not only introduces this ambiguity about user state, but, in an attempt to require less explicit user action, also introduces

additional ambiguity about the user's identity. We added a mediator that handles both types of ambiguity. The mediator displays the current best guess to the user (Figure 9b), and allows her to correct it in a variety of modalities ranging from lightweight to heavyweight, including speech, docking her iButton®, and typing at a keyboard.

The application was modified as follows. A total of 22 lines were changed or added. 14 were minor substitutions where references were changed from the unambiguous widget to the ambiguous one and three were new library imports. Two new class variables were created to hold pointers to the mediator and three lines of code were added to create the mediator and pass it one piece of necessary information about the application, a pointer to its user interface.

The new widget used by the In/Out Board is reusable and is in fact used by the next application as well. The mediator we use is an extension of a mediator from our library of mediators, modified to display application-specific text.

CybreMinder

The second application we modified is CybreMinder [9], a situation-aware reminder system. The original application subscribes to every widget that is running and allows the user to create reminders triggered by any combination of events that these widgets might generate. For example, a user might set up a reminder to go to a meeting when at least three other people are present in the meeting room at the right time. Delivery of reminders is performed whenever the current context appears to match the triggers specified by the user. The application assumes that the reminder has been successfully delivered and acted upon.

We modified the delivery mechanism by adding a mediator to remove this assumption. The mediator gives the user the opportunity to reject a reminder within a certain time after its delivery. This indicates that the reminder should be re-delivered the next time the current context matches the trigger. If the user does not reject it, the system proceeds to change its status to 'delivered' just as it would have done immediately in the original application.

The original application was modified to subscribe to all iButton® widgets so the application would be notified when a user docked to mediate a reminder and to install a custom mediator. The mediator associated with CybreMinder makes use of remote widget feedback services to display feedback about the reminder status. It is an extension of a timer mediator modified to display application-specific messages. The application modifications required the addition of 3 library imports and 27 lines of code either modified or added.

Word Predictor

Our third application, the Communicator, was built from scratch. The system we built consists of four widgets, four mediators, two ambiguous recognizers (one off the shelf and the other homegrown), and two interfaces. We have shown the architecture in Figure 5 and the application in Figure 1. This application demonstrates two important features of the architecture. First, it shows that it supports experimentation with mediation by making it trivial to swap mediators in and out. Adding or replacing a mediator only requires two lines of code. Second, it shows it is not difficult to build a compelling and realistic application. The main Communicator application consists of only 435 lines of code, the majority dealing with GUI issues. Only 19 lines are for mediation and 30 deal with context acquisition.

In Summary

We modified two existing applications and built one more from scratch. Between them, they demonstrate all the required features of the architecture. They use (and reuse) four ambiguity-generating widgets. The first two applications required minor modifications to deal with ambiguity. The third application was built from scratch and very little of its code was dedicated to dealing with mediation or context acquisition. All three applications involve distributed event hierarchies and use reusable mediators to resolve ambiguity.

RELATED WORK AND CONCLUSIONS

Over the past several years, there have been a number of research efforts aimed at creating a ubiquitous computing environment, as described by Weiser [20]. *Aware environments* are environments that can automatically or implicitly sense information about their state and users who are present and take action on this context. Past work such as the Reactive Room [6], Neural Network House [17], Intelligent Room [4], and KidsRoom [1] do not provide explicit reusable support for users to handle or correct uncertainty in the sensed data and its interpretations. A number of architectures that facilitate the building of context-aware services, such as those found in aware environments, have been built [2,8,10,11,19]. As in the case of the aware environments, a simplifying assumption is made that the context being sensed is unambiguous.

There are exceptions to this assumption. For example, the Remembrance Agent uses context to retrieve information relevant to the user and explicitly addresses ambiguity in its interface [18] by showing users multiple potentially relevant pieces of information and letting him select those that are interesting. Multimodal Maps, a map-based application for travel planning, also addresses ambiguity by using multimodal fusion to combine direct manipulation, pen-based gestures, handwriting and speech input, and then prompts the user for more information to remove any remaining ambiguity [3]. QuickSet, another multi-modal map application also prompts the user for disambiguating information [5]. These services demonstrate mediation techniques that allow the user to correct ambiguity in sensed input. They all require explicit input on the part of the user before they take action. Our goal is to provide an architecture that supports a variety of techniques, ranging from implicit to explicit, that can be applied to context-aware services. By removing the simplifying assumption that all context is certain, we are attempting to facilitate the building of more realistic services.

A valid question is why not use sensors that can be more accurately interpreted. Unfortunately, *in practice*, due to both social and technological issues, there are few sensors that are both reliable and appropriate. As long as there is a chance that the sensors may make a mistake, we need to provide the users with techniques for correcting these mistakes. None of the sensors we chose are foolproof either, but the combination of all the sensors and the ability to correct errors before applications take action is a satisfactory and necessary alternative.

Future Work

The extended Context Toolkit supports the building of more realistic context-aware services that are able to make use of ambiguous context. But, we have not yet addressed all the issues raised by this problem. Although we have implemented a basic algorithm for handling multiple applications attempting to mediate simultaneously, we would like to add a more sophisticated priority system that allows mediators to have control over the global mediation process.

We also plan to build more context-aware services using this new architecture and put them into extended use. This will lead to both a better understanding of how users deal with having to mediate their implicit input and a better understanding of the design heuristics involved in building these context-aware services.

Finally, this work does not attempt to answer the question of how best to handle mediation in such settings. The design of mediation for distributed multi-user settings and in settings with implicit input is still an open question. Our architecture makes it easy for programmers to experiment with mediation techniques and we hope it enables us to learn more about appropriate ways of handling mediation.

The extended Context Toolkit supports the building of context-aware services that deal with ambiguous context and allow users to mediate that context. When users are mobile in an aware environment, mediation is distributed over both space and time. The toolkit extends the original Context Toolkit providing mediators that provide the timely delivery of context via partial delivery of the event graph and distributed feedback via output services in context widgets. We demonstrated and evaluated the use of the extended toolkit by modifying two example context-aware applications and the creation of a new context-aware application. We showed that our architecture made it relatively simple to create more realistic context-aware applications that can handle ambiguous context.

REFERENCES

1. Bobick, A. et *a*l. The KidsRoom: A perceptually-based interactive and immersive story environment. *PRESENCE 8*, 4 (1999), 367-391.
2. Brown, P.J. The stick-e document: A framework for creating context-aware applications, in Proc. of Electronic Publishing (1996), 259-272.
3. Cheyer, A. & Julia, L. Multimodal maps: An agent-based approach, in Proc. of the International Conference on Cooperative Multimodal Communication (1995), 103-113.
4. Coen, M. The future of human-computer interaction or how I learned to stop worrying and love my intelligent room. *IEEE Intelligent Systems 14*, 2 (1999), 8-10.
5. Cohen, P.R. *et al*. QuickSet: Multimodal interaction for distributed applications, in Proc. Of Multimedia ’97, 31-40.
6. Cooperstock, J. *et al.* Reactive environments: Throwing away your keyboard and mouse. *CACM 40*, 9 (1997), 65-73.
7. Cutrell, E., Czerwinski, M. & Horvitz, E. Notification, disruption and memory: Effects of messaging interruptions on memory and performance. In Proc. of Interact ’01, (2001), 263-269.
8. Dey, A.K. *et al*. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Human-Computer Interaction Journal 16, 24* (2001), 97-166.
9. Dey, A.K. & Abowd, G.D. CybreMinder: A context-aware system for supporting reminders, in Proc. of HUC 2000, 172-186.
10. Harter, A. *et al*. The anatomy of a context-aware application, in Proc. of Mobicom ’99 (1999), 59-68.
11. Hull, R., *et al.* Towards situated computing, in Proc. of ISWC ’97 (1997), 146-153.
12. Lesher, G. W., *et al.* Techniques for augmenting scanning communication. *Augmentative and Alternative Communication 14*, 81-101.
13. Lesher, G.W. & Rinkus, G.J. Domain-specific word prediction for augmentative communications. Proceedings of the RESNA 2002 Annual Conference, Reno (2002).
14. Mankoff, J. *et al*. OOPS: A Toolkit Supporting Mediation Techniques for Resolving Ambiguity in Recognition-Based Interfaces. *Computers and Graphics 24*, 6 (2000), 819-834.
15. McKinlay, A., et al. Augmentative and alternative communication: The role of broadband telecommunications.. IEEE Transactions on Rehabilitation Engineering. 3(3), September 1995.
16. Moran, T.P and Paul Dourish, editors. Special Issue on Context-Aware Computing. *Human-Computer Interaction Journal 16, 2-4* (2001), 87-420.
17. Mozer, M. C. The neural network house: An environment that adapts to its inhabitants, in Proc. of the AAAI Spring Symposium on Intelligent Environments (1998), 110-114.
18. Rhodes, B. The Wearable Remembrance Agent: A system for augmented memory. *Personal Technologies 1*, 1 (1997), 218-224.
19. Schilit, W.N., System architecture for context-aware mobile computing, Ph.D. Thesis, Columbia University, May 1995.
20. Weiser, M. The computer for the 21st century. *Scientific American 265*, 3 (1991), 66-75.