

Designing Mediation for Context-Aware Applications

ANIND K. DEY and JENNIFER MANKOFF
Carnegie Mellon University

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. Although some of this ambiguity may be resolved using automatic techniques, we argue that correct handling of ambiguous context will often need to involve the user. We use the term *mediation* to refer to the dialogue that ensues between the user and the system. In this article, 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 present design guidelines that arise from supporting mediation over space and time, issues not present in the graphical user interface domain where mediation has typically been used in the past. We illustrate the use of our architecture and evaluate it through an example context-aware application, a word predictor system.

Categories and Subject Descriptors: H.5.2 [**Information Interfaces and Presentation**]: User Interfaces—*Graphical user interfaces; interaction styles*; D.2.11 [**Software Engineering**]: Software Architectures—*Domain-specific architectures*

General Terms: Human Factors

Additional Key Words and Phrases: Context-aware computing, ambiguity, aware environments, ubiquitous computing, mediation, error handling

1. 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

This material is based on work supported by the National Science Foundation under Grant No. IIS-0205644.

Authors' address: Human-Computer Interaction Institute, Carnegie Mellon University, 5000 Forbes Ave., Pittsburgh, PA 15213; email: anind@cs.cmu.edu.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 1515 Broadway, New York, NY 10036 USA, fax: +1 (212) 869-0481, or permissions@acm.org.

© 2005 ACM 1073-0616/05/0300-0053 \$5.00

inferred data [Moran and Dourish 2001]. 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. Yet historically this issue has often been ignored or glossed over. A challenge facing the development of *realistic* and *deployable* context-aware services, therefore, is the ability to handle imperfect, or *ambiguous*, context.

Although some of this ambiguity may be resolved using automatic techniques, we argue that correct handling of ambiguous context will often need to involve the user. We and others have used the term *mediation* to refer to the dialogue that ensues between the user and the system [Mankoff et al. 2000; Heer et al. 2004; Saund and Lank 2003]. This can be seen as an application of *mixed-initiative* interaction to the problem of correcting ambiguity [Ferguson and Allen 1998; Horvitz et al. 2003; Horvitz 1999; Paek and Horvitz 2000], and was originally inspired by the concept of *grounding*, the signals that humans use during conversation to disambiguate potential misunderstandings [Clark and Brennan 1991]. *Mediation techniques* are interface elements that help the user to identify and fix system actions that are incorrect, or potentially involve the user in helping the system to avoid making those mistakes in the first place. This touches on two key challenges for designing the communication aspects of sensing-based systems, highlighted recently by Bellotti et al. [2002] who identify five challenges in all: how to address individual devices in a sensor-rich environment, how to know that the system is attending to the user, how to take action, how to know that the system has taken the correct action, and how to avoid mistakes. Mediation addresses the issues of providing *feedback* to support users in knowing that the system is attending to them and in knowing what the system has done and providing the ability to *disambiguate* sensed input to avoid the system taking incorrect actions.

This article includes two contributions relating to ambiguity management. First, we present *design guidelines* for mediation in sensing-based systems that address design issues beyond those normally dealt with in graphical user interfaces (GUIs). We illustrate these guidelines using an example context-aware application. Second, we discuss our architectural support for mediation, feedback, and disambiguation. We have built a *runtime architecture* that supports programmers in the development of multi-user, interactive, distributed applications that use ambiguous data.

1.1 Designing for Ambiguity

During the course of our research in sensor-based interactions, we have built a number of applications that use ambiguous context and support users in mediating this ambiguity. Ambiguous context, from an aware environment, can produce errors similar to those in desktop recognition-based interfaces. Just as a speech recognizer can incorrectly recognize a user's utterances, a positioning system using Wi-Fi signal strength can incorrectly locate a user. In both cases, a user's actions may be misinterpreted, or missed entirely. Additionally, lack of input may be mistaken as an action. 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 it can use mediation techniques to ask the user about her actual intent. In our past work in desktop interface design, we used mediation to refer to the dialogue between the user and the computer that resolves questions about how the user's input should be interpreted in the presence of ambiguity [Mankoff et al. 2000]. A common example of mediation in recognition-based desktop interfaces is the n -best list, where the n most likely interpretations of some ambiguous input are presented to the user to choose from. Mediation can be seen as an application of mixed initiative interaction [Horvitz 1999], or humans and computers solving problems together, to resolve the problem of ambiguity and error correction.

While a known set of mediation techniques can be applied in most instances of desktop-based interfaces [Mankoff et al. 2000], in the case of ambiguous context, there are additional challenges that arise for several reasons. In particular, humans in off-the-desktop environments are mobile and may be involved in much more complex situations than they are in desktop environments. Additionally, input is often *implicit*, and a person may not be interacting with a computer at all when ambiguity needs to be resolved. These and other issues led us to develop the following guidelines for mediation of context-aware applications:

- Applications should provide *redundant mediation techniques* to support more natural and smooth interactions;
- Applications should facilitate providing input and output that are *distributed both in space and time* to support input and feedback for mobile users;
- Interpretations of ambiguous context should have *carefully chosen defaults* to minimize user mediation, particularly when users are not directly interacting with a system;
- Ambiguity should be retained* until mediation is necessary for an application to proceed.

A valid question is: why involve users in mediation at all? Certainly, mediation puts a burden on the user by asking him to help the system out. Why not simply improve recognition, extend the number of sensors in use, and save the user the effort? Our answer is that currently the information available to recognizers is simply too limited to support the level of certainty necessary to eliminate the need for user feedback. Indeed, the ultimate sensing-based recognition system, the human being herself, often uses dialogue to resolve uncertainty when conversing with other humans.

1.2 System Architecture

Designing correction strategies that meet these requirements can be facilitated by architectural support. In previous work, we presented an architecture for the development of context-aware services that assumed context to be unambiguous [Dey et al. 2001]. We also developed an architecture to support the mediation of ambiguity in recognition-based GUI interfaces [Mankoff et al. 2000]. Building on this past work, we developed support for the *additional* architectural

requirements that arise as a result of requesting highly mobile users to mediate ambiguous context in *distributed, interactive*, sensing environments [Dey et al. 2002]. Our architecture supports the building of applications that allow humans in an aware environment to detect errors in sensed information about them and their intentions, and to correct those errors in a variety of ways. In particular it supports:

- acquisition of ambiguous context;
- context mediation;
- 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;
- distributed feedback about ambiguity to users in an aware environment; and,
- delayed storage of context once ambiguity is resolved.

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

1.3 Overview

We begin by presenting related work. In the next section, we present a motivating example used to illustrate the requirements of mediation in a context-aware setting, followed by a discussion of mediation from an application designer's perspective, including design guidelines for mediation. Next, we present the requirements for our architecture, followed by brief overviews of previous work that we have extended: the Context Toolkit, an infrastructure to support the rapid development of context-aware services, which has assumed perfect context sensing in the past; and OOPS (Organized Option Pruning System), an architecture for the mediation of errors in recognition-based interfaces. 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. We then present a case study of a word prediction system for the disabled (our motivating example) that illustrates how the architecture supports mediation of ambiguous context. We conclude the article with a discussion of further challenges in mediating interactions in context-aware applications.

2. RELATED WORK

Over the past several years, there have been a number of research efforts aimed at creating a ubiquitous computing environment as described by Weiser [1991]. In this section, we first define mediation, and then focus on those efforts dealing with context-aware systems and ambiguity. In particular, we divide these efforts into three categories: context-aware applications; architectures to support context-aware services; and guidelines for dealing with ambiguity.

2.1 Defining Mediation

We define mediation as a dialog between a human and computer that resolves ambiguity. Mediation can conceptually be applied whenever misunderstandings arise between application and user (or even as a way of avoiding such misunderstandings), and it has played an important role in interface design since the development of the earliest computer systems. We are particularly interested in the forms such a dialog can take (such as repeating input, selecting a choice from a list, and so on) and the architectural support needed to support it.

Particularly difficult problems arise when human input is misinterpreted by an application, an increasingly common phenomenon in interfaces, and one that poses serious usability and adoptability issues for users. Thus, our work is focused particularly on the design of, and architectural support for, mediators for resolving ambiguity caused by flaws in an *application's understanding of its user*.

2.2 Current Support for Sensing and Ambiguity in Context-Aware Applications

A system is context-aware if it uses context to provide relevant information and/or services to the user, where relevancy depends on the user's task. Context refers to information that can be used to characterize the situation of a person, place, or object relevant to the task at hand. Context may include information about activity, identity, location, and time. A classic example of a context-aware application is a tour guide that provides relevant information and/or services to a user based on her location [Abowd et al. 1997; Brown et al. 1997; Cheverst et al. 2000].

Table I presents a collection of existing context-aware applications, a short description of each system, the number of types of context a system senses, and how each system handles ambiguity. As shown in Table I, a typical context-aware application includes only a small variety of sensed context (average is 2.35 and median is 2) [Dey and Abowd 2000]. An exception to this is Horvitz's Notification Platform [2004] which uses seven separate sources of context. More typical is a tour guide, such as Cyberguide [Abowd et al. 1997], which uses a location sensor and the identity of the current user (sensed statically via user login) as context.

The sensing of context is typically *implicit* where sensors are used to gather information without requiring action by the user. The more implicit the sensing, the more likely the chance there will be an error in its interpretation. However most of the applications featured in Table I ignore any uncertainty in the sensed data and its interpretations (these are labeled "Ignored" in the final column). In these applications, if the environment takes an action on incorrectly sensed input, it is the occupant's responsibility to undo the incorrect action (if this is possible) and to try again. There is no explicit support for users to handle or correct uncertainty in the sensed data and its interpretations.

However, some exceptions exist. The KidsRoom project uses computer vision to determine the location and activity of children in an interactive storytelling environment [Bobick et al. 1999]. It attempts to constrain user interactions

Table I. Uses of Context and Support for Ambiguity in Representative Systems

System Name	System Description	Number of Types of Context	Handling of Ambiguity
Classroom 2000 [Abowd 1999]	Capture of a classroom lecture	1	Ignored
GUIDE [Cheverst et al. 2000]	Tour guide	1	Ignored
NETMAN [Kortuem et al. 1998]	Network maintenance	1	Ignored
Active Badge [Want et al. 1992]	Call forwarding	1	Ignored
Fieldwork [Pascoe et al. 1998]	Fieldwork data collection	1	Ignored
Stick-e Documents [Brown 1996b; Brown et al. 1997]	Tour guide	1	Ignored
Context Toolkit [Dey et al. 2001]	In/out board	1	Ignored
Context Toolkit [Dey et al. 2001]	Capture of serendipitous meetings	2	Ignored
Stick-e Documents [Brown 1996b; Brown et al. 1997]	Paging and reminders	2	Ignored
Augment-able Reality [Rekimoto et al. 1998]	Virtual post-it notes	2	Ignored
Cyberguide [Abowd et al. 1997]	Tour guide	2	Ignored
Teleport [Brown 1996a]	Migrating desktop environment	2	Ignored
Responsive Office [Elrod et al. 1993]	Office environment control	4	Ignored
Reactive Room [Cooperstock et al. 1997]	Intelligent control of audiovisuals	6	Ignored
CyberDesk [Dey et al. 1999]	Automatic integration of user services	6	Ignored
KidsRoom [Bobick et al. 1999]	Interactive narrative space	2	Automatic Mediation
Remembrance Agent [Rhodes 1997]	Suggests documents related to active typing	1	Mediation
QuickSet [Cohen et al. 1997]	Interactive maps	2	Mediation
LookOut [Horvitz 1999]	An email-based appointment scheduling system	2	Mixed-initiative, Mediation
Notification Platform [Horvitz et al. 2003]	A notification system	7	Mixed-initiative

and uses automatic mediation techniques to resolve ambiguity when necessary. Automatic mediation is used to handle the uncertainty in the computer vision system, where an activity is selected when the probability of that activity crosses a prespecified threshold. While automatic mediation is effective in many settings, in situations that are highly uncertain, it may not accurately interpret user input. Thus, it is often the case that a dialog with the user is required to remove uncertainty. We now introduce a number of systems that use mediation to handle ambiguity in user input.

The Remembrance Agent selects relevant documents based on a user's typing activity, displays an n -best list of top documents (a standard form of mediation used in desktop applications) [Rhodes 1997]. QuickSet, a multimodal map, application, prompts the user for more information where uncertainty exists [Cohen et al. 1997]. These applications all include mediation of some form, and we label them "Mediation" in Table I. Work in the AI community on "mixed-initiative" interaction, which most broadly refers to humans and computers solving problems together, has often focused on handling of ambiguity or uncertainty [Ferguson and Allen 1998; Horvitz et al. 2003; Horvitz 1999; Paek and Horvitz 2000]. For example, the LookOut system watches a user's email for potential appointments [Horvitz 1999]. If it is fairly certain of an appointment, it will take the initiative to complete as much of the scheduling task as possible for a user. Alternatively, the user may take the initiative, invoking LookOut when the system chooses not to act due to uncertainty. LookOut also includes intermediate-level actions such as simply confirming a potential appointment with the user. Before selecting an action, the system attempts to predict when it is appropriate to engage the user in dialog based on a measure of user attention. In no case does LookOut completely schedule an appointment without some sort of user confirmation. Mixed-initiative computing has also been applied in the context-aware computing domain. Examples include the Bayesian Receptionist, a speech-based software receptionist [Paek and Horvitz 2000], and the Notification Platform [Horvitz et al. 2003], a context-aware system that acts as a clearinghouse for incoming messages, dispatching them to a variety of devices based on context about a user's activity level and business.

2.3 Systematic Architectural Support for Context and Ambiguity

A number of architectures that facilitate the building of context-aware services, such as those shown in Table I, have been built [Brown 1996b; Davies et al. 1997; Dey et al. 2001; Harter et al. 1999; Hull et al. 1997; Schilit 1995]. Unfortunately, as in the case of most context-aware applications, a simplifying assumption is made in *all* of these architectures that the context being implicitly sensed is 100% certain. Context-aware services that are built on top of these architectures act on the provided context without any knowledge that the context is potentially uncertain. Our goal is to provide a general, reusable architecture that supports ambiguity and a variety of mediation techniques, ranging from implicit to explicit, that can be applied to context-aware applications. By removing the simplifying assumption that all context is certain, we are attempting to facilitate the building of more realistic services.

2.4 Guidelines for Dealing with Ambiguity

Although ambiguity is rarely addressed in context-aware systems, it has received some attention. In particular, Bellotti et al. [2002] raise multiple challenges relating to whether a user can find out if the system has done what they intended, find out about mistakes that have occurred, and correct mistakes in a timely manner. They suggest two main solutions relating to feedback and control. In terms of feedback, they advocate the following: make sure that users

can tell or ask what state a system is in; make feedback both timely and appropriate; make sure that feedback about the state, versus action or response, are easily differentiable. In terms of control, they advocate the following, both “in time” to avoid crucial system errors: make sure that users can cancel or undo an action; help users disambiguate. Both feedback and control are the heart of what mediation is about and the goal of our work is to provide better systematic support for mediation and to suggest guidelines for effective mediation.

Prior to Bellotti et al.’s [2002] work, Horvitz [1999] proposed guidelines for mixed-initiative interaction, many of which are applicable to the problem of dealing with uncertainty or ambiguity. As he suggests in principle (5) one may “[employ] dialog to resolve key uncertainties,” and (9) “[provide] mechanisms for efficient agent-user collaboration to refine results.” This is what mediators are intended to do. He also makes some suggestions about how the impact of uncertainty on the user should be minimized, including (7) “minimizing the cost of poor guesses. . . including appropriate timing out. . .” and (8) “giving agents the ability to gracefully degrade” and “scoping precision of service to mask uncertainty.” While these principles for mixed-initiative computing hold true for ambiguous, context-aware applications, there are additional guidelines that specifically relate to systems with mobile users and implicit input that we have discovered based on our work in this area. We present our guidelines in Section 4.

2.5 Summary of Related Work

It is the canonical and very common context-aware application which includes two pieces of sensed data, simple intelligence if any, and (currently) no concept of ambiguity that we are hoping to better support with our approach. By empowering the developers of such applications to consider and deal with ambiguity appropriately, we believe we can improve the experience of users of these systems. We believe that this can only happen when systematic architectural support exists for handling ambiguity and experimenting with mediation strategies in context-aware applications.

3. 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 article, 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. Our system is intended to support communication for a mobile user, both with colocated communication partners and with remote communication partners.

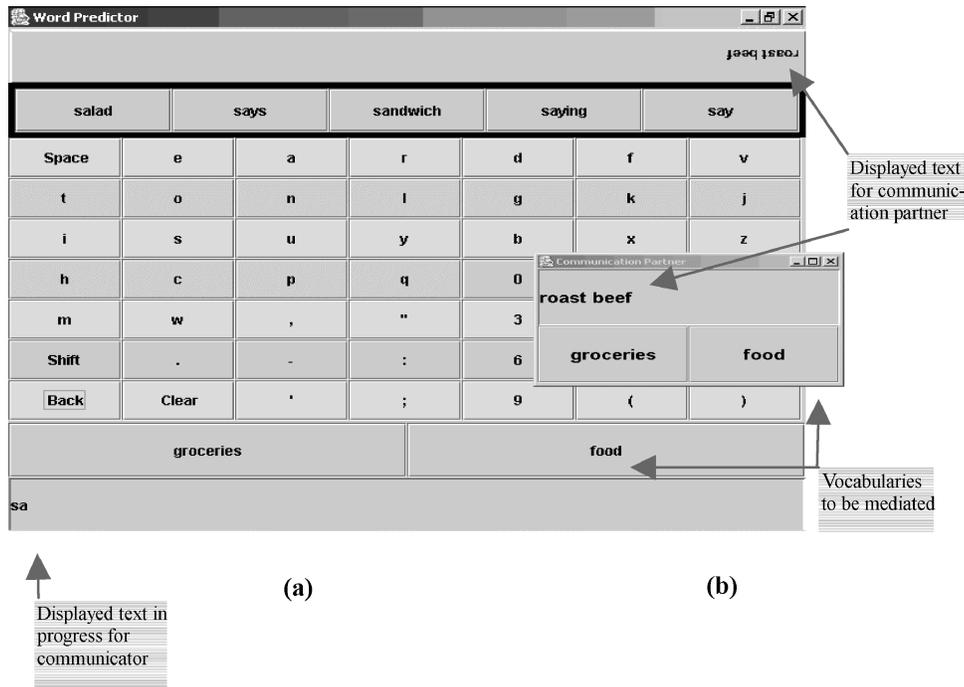


Fig. 1. (a) Communicator (back) and (b) partner (right, front) interfaces.

Ideally it will give its user the same freedom to communicate that is currently enjoyed by a person who can speak to other nearby people or call a friend with a cellphone.

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 nonspeaking individual uses the interface shown in Figure 1(a). The keyboard layout shown was chosen for optimal efficiency for scanning input [Leshner et al. 1998]. Text is displayed to the (abled) communication partner either at the top, reversed for easy readability by someone facing the user, across a flat display (Figure 1(a) top), or on the display of a remote computer (Figure 1(b)). 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 [Rhodes 1997], 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 shown to be effective for predicting URLs in Netscape™ and Internet Explorer™ and, in theory, for nonspeaking individuals [Leshner et al. 1998; McKinlay et al. 1995]. Our goal was to build an application to support context-aware word prediction for nonspeaking 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 to ask the user, or the communication partner, which vocabulary is correct.

We choose to use the Communicator as a running example because it is representative of the context-aware applications (discussed in the previous section) that we want to support. It includes three types of context, including sensed context (location), “sensor-fusion” (word prediction combines context about the optimal vocabulary with keyboard input), and “soft” context (conversational history, recent words), and it is mobile, distributed, and involves multiple users using different applications to view the same data stream. Additionally, it involves two ambiguous data sources: location (where ambiguity is inherent in the sensor), and associated vocabularies and word prediction (where ambiguity is inherent in the interpretation of sensed input).

4. APPLICATION DESIGN CONSIDERATIONS: EXPLORING DISTRIBUTED MEDIATION IN PRACTICE

This section presents a practical description of the key actions a programmer must take in developing a mediation-enriched application, including design guidelines for mediating sensor-based applications. Our past work with the Context Toolkit led to an architecture that supported the following design process for building a context-aware application [Dey et al. 2001]:

- (1) *Specification*. Specify the problem being addressed at a high level, including context-aware behaviors and necessary context.
- (2) *Acquisition*. Determine what new hardware and sensors are necessary to provide that context, write any necessary software, and so on. This is only necessary if sensors are not already supported.
- (3) *Action*. If appropriate, activate software to perform context-aware behavior.

In the Context Toolkit, Step 3, action, is left entirely in the hands of the application designer and is not supported by any toolkit mechanisms. Action is essentially what happens after context is handed off to an application or any intermediary component that will act on context. Such an application or component may or may not directly involve a user in its action. It might explicitly involve the user as in the case of the Communicator. It might change the environment by implicitly conveying information to the user by displaying something visually or audibly. Or it might simply update its state, or that of the

environment, without any intent of notifying the user (e.g., by modifying the climate control mechanisms in the environment).

But how should the design process change if the context an application receives is ambiguous? Specification should remain unchanged except for the choice to use ambiguous context. Acquisition also remains the same, with the sole requirement that the system not throw away ambiguity but pass it to the application to mediate instead. Only action is really affected. In particular, an application designer should include a plan for dealing with ambiguity in her application design. The elements of this plan can be conceptualized as mediators and can be systematically supported by the toolkit. This approach is based on our past work on mediation of Graphical User Interfaces [Mankoff et al. 2000]. From a practical perspective, this means that the application designer should separate her handling of ambiguity from the rest of the application interface. A *mediator*, then, is a component that represents one particular way of handling ambiguity in one or more context sources. In the Communicator, the row of buttons labeled “Vocabularies to be mediated” is a mediator. A mediator represents a dialog. This is appropriate in a setting where user involvement is explicit. When the connection to a user is implicit, a mediator may simply display information at a different level of precision appropriate to the amount of ambiguity present. If no user is present, a mediator may choose to pause action until uncertainty is resolved, try to contact the user, or do its best to automatically determine the top choice.

Based on our work in developing three ambiguous, context-aware applications, we present the following practical guidelines for designing mediation into context-aware applications.

- Applications should provide *redundant mediation techniques* to support more natural and smooth interactions;
- Applications should provide facilities for providing input and output that are *distributed both in space and time* to support input and feedback for mobile users;
- Interpretations of ambiguous context should have *carefully chosen defaults* to minimize user mediation, particularly when users are not directly interacting with a system;
- Ambiguity should be retained until mediation is necessary for an application to proceed.

In the following, we present explanations of each guideline. Note that while these may seem obvious in retrospect, our contribution lies not only in identifying them, but also in illustrating how the unique needs of mobile users and context-aware applications require that these issues be addressed.

4.1 Providing Redundant Mediation Techniques

One of the attractive features of context-aware computing is the promise that it will allow users to carry out everyday tasks without having to provide additional explicit cues to some computational service. Our experience shows, however, that the more implicit the gathering of context, the more likely it is

to be in error. In the GUI domain, typically only two mediation techniques are provided at a time, an n -best list of some sort, and a way to delete or undo incorrect interpretations and reenter them. In designing mediation techniques for correcting context, a variety of redundant techniques should be provided simultaneously. This redundant set not only provides a choice on the form of user input and system feedback, but also the relative positioning and accessibility to the user should be carefully thought out to provide a smooth transition from most implicit (and presumably least obtrusive) to most explicit [Rhodes 1997]. This gives the user the freedom to select the most appropriate level of interaction based on the seriousness of any errors and her own level of engagement in the task. This is similar to the set of social interactions that may occur when someone knocks on one's office door. If you are on the phone, you might talk louder (or more quietly) and ignore the knock. If you are not at all busy, you might walk to the door to answer it. A range of intermediate responses exists as well. Additionally, because recognition is less accurate in unconstrained mobile settings, it is particularly crucial to provide redundancy. If one mediation technique fails due to recognition errors, other options are still available.

4.2 Spatio-Temporal Relationship of Input and Output

Some input must be sensed before any interpretation and subsequent mediation can occur. Because we are assuming user mobility, this means that the spatial relationship of initial input sensors must mesh with the temporal constraints to interpret that sensed input before providing initial feedback to the user. Should the user determine that some mediation is necessary, the feedback needs to be located within physical range of the sensing technologies used to mediate the context and the space through which the user is moving. In contrast, both the user's attention and location are relatively fixed in the GUI domain. Mediating context should occur along the natural path that the user would take. In some cases, this might require duplicate sensing technologies to take into account different initial directions in which a user may be walking. In addition, the mediation techniques may need to have a carefully calculated timeout period after which mediation is assumed not to happen because a user may not have noticed or may have moved past a mediator.

4.3 Effective Use of Defaults

Sometimes the most effective and pleasurable interactions are ones that do not have to happen. Prudent choices of default interpretations can result in no additional correction required by the user. These defaults could either provide some default action or provide no action, based on the situation. For example, in a situation involving highly ambiguous context such as word prediction, it may be best to do nothing by default and only take action if the user indicates the correct interpretation through mediation. This guideline is important in the GUI domain, but crucial in the context domain because the amount of ambiguous context is likely to be more than a user could be reasonably expected to handle.

4.4 Ambiguity Should Be Retained

If every ambiguously sensed event was mediated, a user could be overwhelmed with requests for confirmation. However, ambiguity may be retained in two circumstances. First, there is no reason to ask a user for input until an application needs to act on the sensed data. Second, if the application can use the data, even though ambiguous, it should. An example is the use of vocabularies in the communicator. Vocabularies for the top choices can all be merged in the absence of user feedback about which is correct. Again, these issues exist in the GUI domain, but are more crucial in the context domain because it can reduce the number and frequency of mediation requests with which the user is faced.

5. ARCHITECTURE REQUIREMENTS

The focus of this article is on support for building context-aware applications that deal realistically with ambiguity. This includes providing design guidelines for building applications and addressing the architectural issues needed to deliver ambiguous context and support its mediation. On the architecture side, a system must exist that is able to capture context and deliver it to interested consumers, and there must be support for managing ambiguity. Our architecture addresses seven challenges that arise from mediating ambiguous context.

(1) *Context Acquisition and Ambiguity.* 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 clothing. The ambiguity problem is made worse when applications derive implicit higher-level context from sensor data. This issue arises in the Communicator's inference about vocabulary from (already ambiguous) location and time. Even with the use of sophisticated AI techniques, low- and high-level inferences are not always correct, resulting in ambiguity.

(2) *Context Mediation.* As argued previously, mediation is one way of resolving ambiguity. Architecturally, this leads to the following requirements: the architecture must have a model of ambiguity in sensed and interpreted data; it must be able to identify ambiguity when it is present; it must provide support for selecting among potential mediators and automatically instantiating them when ambiguous data arrives in an application. Additionally, an application developer must have some way to specify the relationship between data and mediator selection. Finally, a basic mediator class must be provided that includes simple functionality such as selecting a specific event as correct. In the GUI realm, we found that architectural support for these activities significantly reduces the amount of custom code a developer must write for each mediator [Mankoff et al. 2000].

(3) *Multiple Subscription Options.* 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, noninteractive components such as a data logging system may not have any way to interact with users and, therefore, may not support mediation. The ability to opt out also makes our toolkit compatible with applications that cannot handle ambiguous context such as previously existing Context Toolkit applications.

Other components may want to receive only unambiguous data. For example, a logging system might want 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 article, we will discuss a word predictor widget in the Communicator that has this property.

(4) *Preemption 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. What should happen if one person selects one vocabulary while another selects a different one? We believe that multiple conflicting disambiguations should not be allowed to exist simultaneously. Rather, disambiguation is meant to represent a human decision about the correct interpretation for ambiguous data. Where conflicts exist due to the presence of multiple users, we argue that it becomes a social issue and should be dealt with as such. Note that we are not advocating multiple competing mediators within a single application, but rather that we need a way to handle conflicts when they occur across applications.

(5) *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.

(6) *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, even where no interactive application is present. For this reason, the architecture needs to support the use of remote feedback, providing feedback (visual or aural, in practice) on any nearby device. For example, in a scenario where a user is being tracked by video, 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. Note that we are not advocating constant feedback about all sensed events but simply architectural support for the ability to provide feedback when something the user would want to know about is occurring as determined by the application developer.

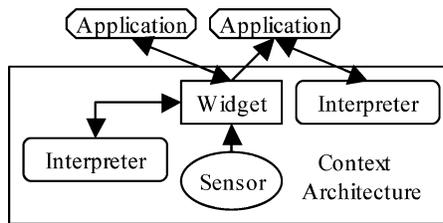


Fig. 2. Context Toolkit components: arrows indicate data flow.

(7) *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).

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

6. MEDIATING AMBIGUOUS CONTEXT

We built support for mediation of imperfectly sensed context by extending an existing toolkit, the Context Toolkit [Dey et al. 2001]. The Context Toolkit is a software toolkit for building context-aware services that support mobile users in aware environments, using context it assumes to be perfect. The toolkit makes it easy to add the use of context or implicit input to existing applications that do not use context. 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.

Widgets and interpreters are intended to be persistent, running 24 hours a day, 7 days a week. They are instantiated and executed independently of each other in separate threads and on separate computing devices. The Context Toolkit makes the distribution of the context architecture transparent to context-aware applications, handling all communications between applications and components.

Context widgets are based on an analogy to GUI widgets. They encapsulate information about a single piece of context such as location or activity and provide a uniform interface to components that use the context. This makes it possible to use heterogeneous sensors to sense redundant input. Widgets maintain a persistent record of the context they sense. They allow applications and other widgets to both query and subscribe to the context information they maintain. The existing toolkit includes an extensible library of such widgets for sensing motion, temperature, identity (iButtons), volume level, and many others. Applications may use any combination of these sensors.

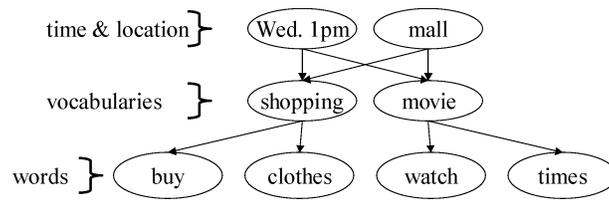


Fig. 3. An event graph representing predicted words from context.

A *context interpreter* is used to abstract or interpret context. For example, if a GPS widget provides location context in the form of latitude and longitude, a context interpreter would be used to translate this to a street name. 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. A sensor may have inherent inaccuracies which affect the way a widget encapsulates its data while an interpreter, because it is making inferences, may naturally include uncertainty.

6.1 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 [Mankoff et al. 2000], a GUI toolkit that provides support for building interfaces that make use of recognizers (e.g., speech, gestures) that interpret user input. Like sensing context, recognition is ambiguous and OOPS provides support for tracking and mediating uncertainty. We chose OOPS because it explicitly supports mediation of single-user, single application, nondistributed, ambiguous desktop input, a restricted version of our problem.

OOPS provides an internal model of recognized input based on the concept of hierarchical events [Myers and Kosbie 1997] which allows separation of mediation from recognition and from the application. This is a key abstraction that we will use in the extended Context Toolkit. 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 depicts relationships between source events and their interpretations (which are produced by one or more recognizers). Note that this graph is not intended to encode semantic relationships or to support reasoning about those relationships. Thus the authoring of the graph is a simple matter of recording the source events from which interpretations are derived, and no explicit design is required to create the graph. The graph is used simply to identify the presence of ambiguity and to ensure that the appropriate components are updated when decisions about ambiguity are made.

In most existing context-aware applications, the relationships represented in the graph are typically discarded along with any information about ambiguity. However, the creation of the graph is very straightforward since context events are being created each time an event is interpreted: when a new interpretation is created in a widget or interpreter, it must be passed a set containing any

source events that were used to create it. This is done at creation time when that information is still easily accessible and simply requires an additional argument to the event constructor. More importantly, when ambiguity is present, multiple events must be created for each ambiguous interpretation. This postpones the work of selecting the most likely choice for later and moves that work to the mediation phase of event handling.

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 is a component in the application that either resolves ambiguity automatically or allows the user and computer to communicate about ambiguity. Mediators generally fall into three major categories. *Choice* mediators give the user a choice of possible interpretations of her input. *Repetition* mediators support the user in repeating her input, usually in an alternate and less error-prone modality. Both types of mediators are very similar to a GUI widget such as a button or menu, with the exception that they only appear when ambiguity that they are designed to resolve is present. *Automatic* mediators select the most likely choice without user input and may vary widely in complexity and sophistication. *Metamediators* are used to select the appropriate mediator based on context type, application state, and so on, and help to decide whether mediation should involve the user or not.

A mediator typically handles or displays the same portion of the graph that an application would have acted on had no ambiguity been present. Thus, a mediator may be created by simply extending an existing mediator from our toolkit library to display events that an application was already designed to support, or to resolve ambiguity that would have been resolved before an event reached the application in the past. Mediators all include architectural support for *accepting* or *rejecting* events with the side effect that correct interpretations are kept and incorrect interpretations removed from the graph. Once the ambiguity is resolved, the toolkit allows processing of that portion of the input graph to continue as normal. Typically, a mediator will focus on the highest level interpretations (leaf nodes) of the graph, and accepting a leaf node is sufficient to disambiguate the remainder of the graph because the entire path from the root to that leaf must also be accepted while any conflicting paths must be rejected [Mankoff et al. 2000].

Similar to OOPS, our toolkit automatically handles the tasks of routing input to mediators when it is ambiguous and of informing recognizers and the application about the correct result when ambiguity is resolved. This separation of mediation from recognition and from the application means that the basic structure of an application and its interface does not need to be modified in order to add to or change how recognition or mediation is done. Additionally, the directed graph provides a consistent internal model that makes it possible to build mediators that are completely independent of recognizers. Note that an application may opt to bypass mediation and receive ambiguous events directly or ask to only be informed about events that are unambiguous or have had all ambiguity resolved. A noninteractive component that cannot ask the user for input may select this option for example.

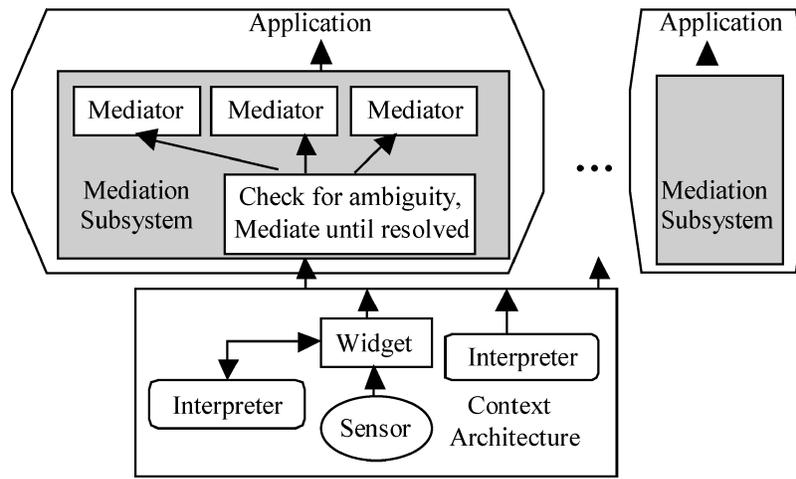


Fig. 4. The architecture for the extended Context Toolkit. Everything in the gray boxes is new.

Figure 4 shows the resulting changes. The light gray boxes indicate components that have been added to the Context Toolkit architecture illustrated in Figure 2 to support mediation of ambiguous context.

6.2 Example

Before discussing the additional changes necessary to support the requirements listed in Section 5, 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. Eventually, the user will need to select the correct path through this graph (e.g., mall & Wednesday → shopping → clothes).

Now suppose that an application subscribes to this data. Subscribers to ambiguous data, using our toolkit, may wait until all the ambiguity has been resolved before taking any action on a location update, or take action on the ambiguous data using an automatic or interactive mediator. In the case of the Communicator, we have chosen to let the user mediate the graphs at two levels: that of vocabularies (which change only when the user changes location), and that of words. A single mediator that displays input events as a row of buttons must be created, inheriting from our basic choice mediator. Two instances of this mediator are instantiated and passed to the architecture. One is assigned to handle vocabulary events when they arrive using the same subscription mechanisms that an application would use to select that particular piece of context, while the other is assigned to handle word events when they arrive. Should the user select a vocabulary or word, the event graph will be updated by a call to the

system-provided `accept()` method. When a new set of vocabularies or words is delivered to the application, the architecture automatically routes them to the appropriate mediator, asking the mediator to update its display by replacing the previous set with the new set. When ambiguity is resolved, the main application receives the final choice and may act on it (in this case, by displaying it to the communication partner).

6.3 Modifications for New Requirements

The previous sections described the basic abstractions used to support mediation: widgets, interpreters, applications, mediators, and the event graph. These abstractions, taken from our past work [Mankoff et al. 2000], were sufficient to handle the first three requirements listed in Section 5 (*context acquisition and ambiguity*, *context mediation*, and *multiple subscription options*). We now explain the additional architectural mechanisms needed to support the remaining three requirements which all represent unique problems faced by mediation of ambiguous context in a distributed, multi-user setting, introduced previously.

(4) *Preemption of Mediation*. Because multiple applications may subscribe to the same ambiguous data, mediation of the same data may actually occur simultaneously. If multiple components are mediating at once, the first one to succeed “interrupts” the others and updates them with the mediated data. 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. The architectural stub in each recipient component that handles communication and mediation determines if the updated data is currently being mediated locally. If so, it informs the relevant mediators that they have been preempted and should stop mediating. Our solution is particularly important to supporting mediation over space and time (our second design guideline) since it allows multiple mediation opportunities to be presented simultaneously at different locations along the user’s expected path. It is unique architecturally because it handles mediation in multiple distributed components.

(5) *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 mediation by passing the set of ambiguous events it wants mediated to a remote component and have that remote component perform the 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. Currently, there is no way to request mediation without specifying who should do it. Forced mediation may be used when ambiguity has been retained (the first design guideline), and now needs to be resolved.

(6) *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 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 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. This issue is also particularly relevant to supporting mediation over space and time as well as retaining ambiguity (our second and fourth design guidelines). If and when mediation is necessary, a user may not be in the location where the data was originally sensed, nor where the application using it is located. Remote feedback can help with this.

(7) *Storage*. 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. As a consequence of this choice, if no applications have subscribed to a sensor, its data will not be stored and, therefore, will not be available to applications later. One solution available to a widget designer is to ask users to disambiguate data even in the absence of an application. This represents a burden to the user. Another alternative is to automatically choose an interpretation (essentially what is done by default in most toolkits).

Another dimension of storage relates to when data is stored. Since we only store unambiguous data, we store data only after it has been mediated. This is for two reasons: the storage policy is easier to deal with from an access standpoint because applications can treat any retrieved data as certain, and we gain the benefits offered by knowledge of ambiguity during the mediation process without bearing the cost of long-term storage. All that is lost is access to information about ambiguity at some arbitrary time after mediation (when the record of ambiguity has been discarded). It would be relatively simple to modify the architecture to store all information about ambiguity.

6.4 Summary of Architecture

In summary, we have created an architecture that combines the context and distribution capabilities of the Context Toolkit [Dey et al. 2001] with the ambiguity-handling and mediation capabilities of OOPS [Mankoff et al. 2000]. Our combined architecture addresses seven requirements for mediating ambiguous context. The resulting architecture simplifies the previously ad-hoc process of handling ambiguous data from an application developer's perspective to the steps shown in Figure 5. Items in *italics* are additions caused by our support for mediation and ambiguity. Note that Mediators (Figure 5(b)) and Widgets (Figure 5(c)) are intended to be reused and may not need to be created from scratch for every application. This leaves only two crucial steps relating to mediation to the typical application developer (Figure 5(a)): he must decide whether or not the application should receive and handle ambiguous data or only receive data once mediation has occurred (this requires a single change to the subscription code); and he should select which mediators should be used

(a) Application	(b) Mediator
<i>Specify whether to handle ambiguous data or not</i> Create subscriptions to widgets Retrieve data from storage, if necessary <i>Install mediators</i> Handle results of subscriptions	(if interactive) <i>Produce some feedback about the data being mediated</i> <i>Accept or Reject events</i> (based on user interaction if interactive) <i>Take mediator-specific action to maintain valid state if pre-empted or forced to mediate</i>
(c) Widget	
Specify the data you provide <i>When creating events, specify their sources if appropriate</i> <i>When creating events, if ambiguous, specify each possible interpretation</i> <i>Garbage collect and perform widget-specific actions on mediated data</i>	

Fig. 5. Steps for building system components. *Italics* represent new steps due to our support for ambiguity.

to resolve ambiguity and install them (typically requires two lines of code per mediator).

Mediators are designed to support reuse in multiple applications. However, should the existing mediators in the toolkit library not suffice, an application developer may write a new mediator. This task is separated from application development so that mediators may be easily replaced or modified, allowing application developers to easily experiment with appropriate mediation strategies. However, the basic task of writing a mediator is no more complicated than dealing with ambiguity directly in the application would be, while the benefits in terms of the support for mediation across multiple applications are great. The new mediator can be reused by other applications and the architecture handles the job of communicating the results of the mediation to other interested components. Additionally, the architecture automatically determines when ambiguity is present and mediation is needed and routes ambiguous events to the appropriate mediator (based on subscriptions created by the application developer).

The designer must find some way to provide feedback to the user about the data being mediated. When the user responds to that feedback, the mediator uses that information to call a single method (*accept* or *reject*) informing the system of the final correct interpretation. Finally, the designer must decide how the mediator should act when pre-empted or forced to mediate.

As with mediators, a developer may find himself writing a new widget. This task is fairly straightforward even when data is ambiguous. Rather than creating a single event to represent a piece of sensed data, one event is created for each ambiguous interpretation. If any of these events was derived from pre-existing events (this is more common in interpreters than widgets), those source events must be made explicit to the system by passing them as arguments to the new event's constructor. Finally, when the widget is notified that an event has been mediated, it may wish to perform special actions such as updating a learning algorithm. However, this last step is not required. Note that giving the widgets access to information about ambiguity may allow them to do other

interesting things such as fusing data from multiple sensors to create more sophisticated interpretations of user input. However this is not a focus of the current work.

Note that it is straightforward to integrate existing applications that cannot handle ambiguity into this architecture. No mediators need be installed, so the applications themselves only require a change to one constructor. Existing widgets, as just described, must specify the source of any events they create, again by simply changing the arguments to a constructor.

While we have given some details about how an application programmer would use our architecture, more information on those requirements can be found in our prior work on this topic [Dey et al. 2002]. In the next section, we describe an application that we built with this architecture.

7. CASE STUDY

In this section, we describe the building of a context-aware application, the Communicator (see the online Appendix in the ACM Digital Library for two more case studies: an In/Out Board and a reminder system). This application was built entirely using our new architecture and includes both ambiguous and unambiguous data sources. The application validates our architecture and illustrates how our design guidelines can be applied to real systems. Our case studies represent a range of complexity in terms of the number of types of context they use, the mobility of their users, and the extent to which sensing is implicit or explicit. Because each application represents a range of characteristics typical of existing context-aware applications, we believe they are of practical use to designers.

We introduced the Communicator system as our motivating example (see Figure 1). Here we describe the physical setup of the application, the interaction supported, the system architecture, and its use of the design guidelines presented earlier.

This application demonstrates two important features of the architecture. First, the architecture 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 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 for context acquisition. Additionally, it illustrates the need for appropriate use of defaults and for retaining ambiguity.

7.1 Physical Setup

The Communicator runs on a laptop computer with a GPS unit attached to a wheelchair. As the user moves about in a downtown city environment, the Communicator suggests appropriate vocabularies based on the current location and time, shown near the bottom of the interface. The user can select a vocabulary at any time during a conversation to help the system predict appropriate words. As the user starts entering characters with the scanning interface, the system predicts potential word completions shown near the top of the interface, using

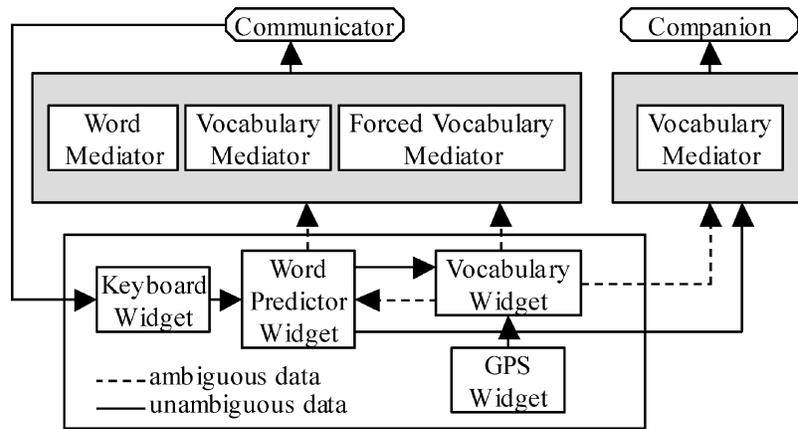


Fig. 6. Architecture for the Communicator System.

words from the system-suggested vocabularies or the user-selected vocabulary, to support him in maintaining a conversation. The user can select a suggested word or continue to type individual characters. A companion, using a partner device, can also select the appropriate vocabulary on behalf of the user.

7.2 Implementation

The Communicator directly uses data from three widgets: a soft keyboard, a word predictor, and a vocabulary selector as shown in Figure 6. The keyboard widget produces unambiguous data and lets other components know is being typed. The word predictor widget produces ambiguous data and uses current context to predict what word the user is typing. It uses a unigram, frequency-based method common in simple word predictors, along with a history of recent words. It subscribes to the keyboard to get the current prefix (the letters typed so far for the current word). 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 vocabulary to weight the words from that vocabulary. The vocabulary widget uses the Remembrance Agent [Rhodes 1997] 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.

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.

Ambiguous information generated in our system includes potential vocabularies and potential words. The architecture allows a component to mediate



Fig. 7. Screenshots of mediators (a) choice mediator for words or vocabularies and (b) required mediator for vocabularies.

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 1(a). 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.

The application presented here and the other two discussed in the online Appendix represent a range of complexity within the context-aware domain and demonstrate all the required features of the architecture. The applications help to illustrate the validity of the underlying architecture and to demonstrate how the design guidelines can be analyzed and applied in real settings.

7.3 Design Issues

The Communicator supports mediation of both words and vocabularies. For the mediation of words, users can either select one of the words suggested or can continue typing causing a new set of suggested words to appear. 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 7(a)) displays the choices similar 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 7(b)). 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 readily supports this type of experimentation by allowing programmers to easily swap mediators. Providing this range of mediation techniques offers flexibility to the end user.

The variety of mediators we experimented with allowed us to explore two design heuristics: the appropriate use of defaults and retaining ambiguity.

Appropriate Use of Defaults. Providing appropriate defaults and reducing keyboard input is the main purpose of this application. The different mediators we experimented with supported different defaults. The mediator that automatically chose the most likely vocabulary is most appropriate when there is little ambiguity, whereas the mediator that does not require the user to select a vocabulary is most appropriate when there is a lot of ambiguity. Because each selection act by the user may take multiple seconds, the appropriate use of defaults is critical in this application.

Retaining Ambiguity. The fourth mediator described earlier only requires mediation when a conversation ends. This demonstrates how an application can postpone mediation as long as possible. In particular, the conversation needs to be recorded to disk in the correct vocabulary category, so vocabulary ambiguity must be resolved at this time. Again, this guideline supports the goal of minimizing the number of selection requests for the user. Additionally, by allowing the user to postpone mediation as long as possible, we minimize the chance that mediation will interrupt an already slow conversation.

8. 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.

Because multiple components may subscribe to the same ambiguous events, mediation may actually occur simultaneously in these components. When one component successfully mediates the events, the other components need to be notified. We have already added the ability for input handlers to keep track of what is being mediated locally in order to inform mediators when they have been preempted. We would like to add a more sophisticated priority system that will allow mediators to have control over the global mediation process. This could also support more sophisticated ways of dealing with conflicts when multiple applications or users are mediating the same data.

Related to this issue of multiple applications mediating is the need to examine whether a single, final mediated result is appropriate for multiple applications. An alternative would be to modify our architecture to maintain information about multiple disambiguations for different applications or groups of applications. There may be situations where a user may want different mediated results to be sent to different applications to protect her privacy, for example. In an application located within her home, she may be willing to provide exact information about her activities, but may wish to provide less fine-grained, or even false, information to the application for applications and users outside her home.

An additional issue we need to further explore is how events from different interactions can be separated and handled. For example, in the In/Out Board service, it is assumed that only one user is mediating their occupancy status at any one time. If two people enter together, we need to determine which input event belongs to which user in order to keep the mediation processes separate.

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

9. CONCLUSIONS

The extended Context Toolkit supports the building of realistic context-aware services, ones 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. As a result of this, the design of mediation differs from the GUI domain. We introduce design guidelines for mediating distributed, context-aware applications.

- Applications should provide *redundant mediation techniques* to support more natural and smooth interactions.
- Applications should provide facilities for providing input and output that are *distributed both in space and time* to support input and feedback for mobile users.
- Interpretations of ambiguous context should have *carefully chosen defaults* to minimize user mediation, particularly when users are not directly interacting with a system.
- *Ambiguity should be retained* until mediation is necessary for an application to proceed.

To support the mediation of ambiguous context, we extended the Context Toolkit to support seven key features:

- acquisition of ambiguous context;
- context mediation;
- delivery of ambiguous context to multiple applications that may or may not be able to support mediation;
- preemption of mediation by another application or component;
- applications/services in requesting that another application or service mediate;
- distributed feedback about ambiguity to users in an aware environment; and,
- delayed storage of context once ambiguity is resolved.

We demonstrated and evaluated the use of the extended toolkit by modifying two example context-aware applications and by creating 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 and demonstrated the use of the design guidelines for creating these types of application .

ELECTRONIC APPENDIX

Two additional case studies can be found in the electronic appendix to this article in The ACM Digital Library.

ACKNOWLEDGMENTS

We would like to thank colleagues at Georgia Tech and UC-Berkeley who helped us build and use the applications described in this article and who provided guidance on the design of the architecture.

REFERENCES

- ABOWD, G. D. 1999. Classroom 2000: An experiment with the instrumentation of a living educational environment. *IBM Syst. J.* 38, 4, 508–530.
- ABOWD, G. D., ATKESON, C. G., HONG, J., LONG, S., KOOPER, R., AND PINKERTON, M. 1997. Cyberguide: A mobile context-aware tour guide. *Balzer/ACM Wireless Netw.* 3, 5, 421–433.
- BELLOTTI, V. BACK, M., EDWARDS, W. K., GRINTER, R. E., HENDERSON, A., AND LOPES, C. 2002. Making sense of sensing systems: Five questions for designers and researchers. In *Proceedings of Computer Human Interaction (CHI 2002)*, 415–422.
- BOBICK, A. F., INTILLE, S. S., DAVIS, J. W., BAIRD F., PINHANEZ, C. S., CAMPBELL, L. W., IVANOV, Y. A., SCHUTTE, A., AND WILSON, A. 1999. The KidsRoom: A perceptually-based interactive and immersive story environment. *Presence* 8, 4, 367–391.
- BROWN, M. 1996a. Supporting user mobility. In *Proceedings of IFIP World Conference on Mobile Communications*, 69–77.
- BROWN, P. J. 1996b. The stick-e document: A framework for creating context-aware applications. In *Proceedings of Electronic Publishing '96*, 259–272.
- BROWN, P. J., BOVEY, J. D., AND CHEN, X. 1997. Context-aware applications: From the laboratory to the marketplace. *IEEE Pers. Comm.* 4, 5, 58–64.
- CHEVERST, K., DAVIES, N., MITCHELL, K., FRIDAY, A., AND EFSTRATIOU, C. 2000. Developing a context-aware electronic tourist guide: Some issues and experiences. In *Proceedings of Computer Human Interaction (CHI 2000)*, 17–24.
- CLARK, H. AND BRENNAN, S. E. 1991. Grounding in communication. *Perspectives on Socially Shared Cognition*. Resnick, L., Levine, J., and Teasley, S. Eds. American Psychological Society. 127–149.
- COHEN, P. R., JOHNSON, M., MCGEE, D. R., OVIATT, S., PITTMAN, J., SMITH, I., CHEN, L., AND CLOW, J. 1997. QuickSet: Multimodal interaction for distributed applications. In *Proceedings Of Multimedia '97*, 31–40.
- COOPERSTOCK, J., FELS, S., BUXTON, W., AND SMITH, K. 1997. Reactive environments: Throwing away your keyboard and mouse. *Comm. ACM* 40, 9, 65–73.
- DAVIES, N., WADE, S., FRIDAY, A., AND BLAIR, G. 1997. Limbo: A tuple space based platform for adaptive mobile applications. In *Proceedings of Conference on Open Distributed Processing / Distributed Platforms '97*.
- DEY, A. K., MANKOFF, J., ABOWD, G. D., AND CARTER, S. 2002. Distributed mediation of ambiguous context in aware environments. In *Proceedings of the ACM Symposium User on Interface Software and Technology (UIST 2002)*, 121–130.
- DEY, A. K., SALBER, D., AND ABOWD, G. D. 2001. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Hum.-Comput. Interac. J.* 16, 24, 97–166.
- DEY, A. K. AND ABOWD, G. D. 2000. Towards a better understanding of context and context-awareness. In *Computer Human Interaction 2000 Workshop on the What, Who, Where, When, Why and How of Context-Awareness*.
- DEY, A. K., ABOWD, G. D., AND WOOD, A. 1999. CyberDesk: A framework for providing self-integrating context-aware services. *Knowledge-Based Syst.* 11, 3–13.
- ELROD, S., HALL, G., COSTANZA, R., DIXON, M., AND DES RIVIERES, J. 1993. Responsive office environments. *Comm. ACM* 36, 7, 84–85.
- FERGUSON, G. AND ALLEN, J. F. 1998. TRIPS: An intelligent integrated problem-solving assistant. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98)*, 567–573.

- HARTER, A., HOPPER, A., STEGGLES, P., WARD, A., AND WEBSTER, P. 1999. The anatomy of a context-aware application. In *Proceedings of Mobicom '99*, 59–68.
- HEER, J., GOOD, N. S., RAMIREZ, A., DAVIS, M., AND MANKOFF, J. 2004. Presiding over accidents: System mediation of human action. In *Proceedings of Computer-Human Interaction 2004 (CHI04)*, 463–470.
- HORVITZ, E., KADIE, C. M., PAEK, T., AND HOVEL, D. 2003. Models of attention in computing and communications: From principals to applications. *Comm. ACM* 46, 3, 52–59.
- HORVITZ, E. 1999. Principles of mixed-initiative interaction. In *Proceedings of Computer Human Interaction (CHI99)*, 159–166.
- HULL, R., NEAVES, P., AND BEDFORD-ROBERTS, J. 1997. Towards situated computing. In *Proceedings of the International Symposium on Wearable Computers*, 146–153.
- KORTUEM, G., SEGALL, Z., AND BAUER, M. 1998. Context-aware, adaptive wearable computers as remote interfaces to ‘intelligent’ environments. In *Proceedings of the International Symposium on Wearable Computers*, 58–65.
- LESHER, G. W., MOULTON, B. J., AND HIGGINBOTHAM, J. 1998. Techniques for augmenting scanning communication. *Augment. Altern. Comm.* 14, 81–101.
- PAEK, T. AND HORVITZ, E. 2000. Conversation as action under uncertainty. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence (UAI 2000)*, 455–464.
- MANKOFF, J., ABOWD, G. D., AND HUDSON, S. E. 2000. OOPS: A Toolkit Supporting Mediation Techniques for Resolving Ambiguity in Recognition-Based Interfaces. *Comput. Graph.* 24, 6, 819–834.
- McKINLAY, A., BEATTIE, W., ARNOTT, J. L., AND HINE, N. A. 1995. Augmentative and alternative communication: The role of broadband telecommunications. *IEEE Trans. Rehabil. Eng.* 3, 3, 254–260.
- MORAN, T. P. AND DOURISH, P. 2001. Eds. Special Issue on Context-Aware Computing. *Hum.-Comput. Interact. J.* 16, 2–4, 87–420.
- MYERS, B. A. AND KOSBIE, D. S. 1997. Reusable hierarchical command objects. In *Proceedings of Computer Human Interaction (CHI '97)*, 260–267.
- PASCOE, J., RYAN, N. S., AND MORSE, D. R. 1998. Human-Computer-Giraffe Interaction – HCI in the Field. In *Proceedings of the Workshop on Human Computer Interaction with Mobile Devices*.
- REKIMOTO, J., AYATSUKA, Y., AND HAYASHI, K. 1998. Augment-able reality: Situated communication through physical and digital spaces. In *Proceedings of the International Symposium on Wearable Computers*. 68–75.
- RHODES, B. 1997. The Wearable Remembrance Agent: A system for augmented memory. *Pers. Technol.* 1, 1, 218–224.
- SAUND, E. AND LANK, E. 2003. Stylus input and editing without prior selection of mode. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST'03)*, 213–216.
- SCHLIT, W. N. 1995. System architecture for context-aware mobile computing, Ph.D. Thesis, Columbia University (May).
- WANT, R., HOPPER, A., FALCAO, V., and Gibbons, J. 1992. The Active Badge location system. *ACM Trans. Inform. Syst.* 10, 1, 91–102.
- WEISER, M. 1991. The computer for the 21st century. *Scient. Amer.* 265, 3, 66–75.

Received February 2003; revised August 2003, March 2004; accepted February 2004 by Shumin Zhai and Victoria Bellotti