Join and Capture: A Model for Nomadic Interaction

Dan R. Olsen Jr., S. Travis Nielsen, David Parslow
Computer Science Department, Brigham Young University, Provo, Utah, 84602, USA
{olsen, nielsent, dparslow}@cs.byu.edu

ABSTRACT

The XWeb architecture delivers interfaces to a wide variety of interactive platforms. XWeb's SUBSCRIBE mechanism allows multiple interactive clients to synchronize with each other. We define the concept of Join as the mechanism for acquiring access to a service's interface. Join also allows the formation of spontaneous collaborations with other people. We define the concept of Capture as the means for users to assemble suites of interactive resources to apply to a particular problem. These mechanisms allow users to access devices that they encounter in their environment rather than carrying all their devices with them. We describe two prototype implementations of Join and Capture. One uses a Java ring to carry a user's identification and to make connections. The other uses a set of cameras to watch where users are and what they touch. Lastly we present algorithms for resolving conflicts generated when independent interactive clients manipulate the same information.

Keywords

Multimodal interaction, nomadic, mobile, ubiquitous

INTRODUCTION

In 1993 Mark Weiser published a vision of ubiquitous computing where computation and information would be distributed throughout our working environment [WEIS 93]. This vision is becoming reality with smaller computing, cheaper communications and more sensors [BORR 00]. The goal is for users to move through their environment, finding resources and services wherever they are and to have those services provided in the context of their physical environment. A part of that vision is that small computing devices be carried by the user to supply various pieces of interactive information.

We are interested in three aspects of this mobile or nomadic computing problem. The first is the problem of size vs. ease of interaction. A device that is small enough to comfortably carry is frequently too small to comfortably use. We believe that truly effective interaction will exploit interactive resources encountered in the environment. For example, when one walks into a strange conference room, one will want to place an interactive task on the wall display. When we walk into someone else's office we may want to transfer a task from our palm device onto the workstation in that office. The goal is to exploit interactive resources as they are encountered rather than forcing users to carry such devices with them.

A second goal is multimodal exploitation of interactive resources. While accessing an information service over a cell phone, one may walk into the family room and place the task onto the television to get a better view of the data. Speech interfaces may be fine for entering data, but a visual representation on the screen can greatly increase usability. A small number of buttons sewn into a glove could be integrated with an interface projected on the wall. Small controlling devices can dynamically integrate with large display services encountered in the world. The goal is to empower opportunistic assemblies of interactive resources to accomplish a particular task.

Lastly we are interested in obtaining interactive control of a device by physical reference to it. For example when walking into a strange conference room one could touch the projector and have its control interface transferred to the user's palm device. The goal is if one sees it and touches it, one can interact with it.

This vision is made possible by drastic reductions in the cost of computing and interactive devices and by the pervasiveness of the Internet. The world has changed from people per device to devices per person. With this abundance of interactive computing it is unnecessary to convert users into computational pack animals.

To make this vision work an interactive model for acquiring and using interactive resources is required along with an infrastructure to tie it all together. Our interactive model is based on the two concepts of Join and Capture and the substrate is provided by the distributed interface mechanisms of XWeb.

We will first review the work of others in nomadic interaction. This will be followed by an overview of those components of the XWeb architecture, which support dynamically assembling interactive resources. We will then define the Join and Capture primitives in terms of the XWeb architecture. We will then show how we have implemented these concepts using Java rings and cameras to create a smooth and natural environment for interactivity. Lastly we will discuss the key algorithms for managing interaction with multiple users and clients.

PRIOR WORK

One of our goals is that users will be empowered to use multiple interactive devices in consort on a particular task. Rekimoto has shown how a PDA in conjunction with a whiteboard can produce more effective collaborative interactions than either device alone [REKI 98]. We agree with these goals but have greater ambitions in supporting more nomadic interactions that involve more diversity of interactive devices than just a whiteboard/PDA pair. The QuickSet project [COHE 97] integrates speech and pointing modalities. QuickSet also pioneered a unification model for integrating multimodal input. Our multimodal approach is less tightly integrated than QuickSet. In our model each interactive platform must independently resolve user intent rather than merging inputs from a variety of modalities. Our larger grained integration leaves fuzzy inputs like speech and gesture to work out their problems independently, but provides a more open model for users to dynamically integrate modalities for a particular problem.

Embedding interaction in the physical world has revolved around the tagging of objects in the world so that their identity can be translated into a behavior. Rekimoto used color codes [REKI 95] and visual tags [REKI 00] to identify objects in the world and to associate information with those identities. We desire to go beyond simple information to interaction services and interactive devices. Our work is closer to the active objects marked by RFID tags done at Xerox [WANT 99]. In contrast with both of these cases we desire that a user need not carry any device other than a ring or card for identification. All other interactive resources can be encountered in the environment. If the user does carry devices such as a PDA or laptop, these can be seamlessly integrated with other devices the user may find. We will show how such tagging-based techniques can fit within the Join/Capture paradigm.

In supporting nomadic computing it is important to be able to identify where a user is located relative to resources and services. The EasyLiving project [BRUM 00b] has focused on geometry as the enabling substrate for such interactions [BRUM 00a]. While EasyLiving used cameras to deduce geometric position, AT&T has used ultrasonic beacons [HART 99]. Each of these projects has explored computing models that follow users wherever they go. Our work is in a similar vein. However, in XWeb geometry is replaced by information about identity and adjacency as a simpler model for driving the interaction.

XWEB

The XWeb architecture draws its inspiration from the client-server model of the WWW. The primary difference is XWeb's focus on interactivity rather than publication. XWeb primarily views servers as maintaining trees of XML structured data. In XWeb, a client may not only get data from a server, but also change that data and monitor the changes made by others.

XWeb terms

At the heart of XWeb are <u>*servers*</u>. These behave a lot like HTTP servers in that they and their contents can be accessed via URLs and they support the same HTTP GET method for retrieving information. To the outside world, an XWeb server appears to be a large, dynamic XML tree full of objects with types, attributes and children objects.

Unlike the WWW, XWeb servers also support a CHANGE method that accepts XML descriptions of changes to be made to the server's tree of data. The change language supports the usual insertion, deletion and modification operators. Any possible modification of an XWeb tree can be encoded in the change language. This CHANGE method is what makes XWeb interactive.

Services

Within an XWeb server there can be many <u>*services*</u> embedded in its tree. As with the WWW, servers can be a façade for data and services of infinite variety. So in a home a server might be housed in the basement and connected to the Internet. On this server might be informational services such as email or shopping. There might also be a thermostat service that uses X10 to control the heating and air conditioning. Such a service might be a simple XML object that looks like:

```
    <thermostat
        wakeTime=”6:30AM” wakeTemp=”74”
        sleepTime=”11:00PM” sleepTemp=”70”
        currentTime=”4:00PM”
    />
```

To client software accessing the server, this is simply XML data that can be changed. However, within the service implementation a change of currentTime will modify the thermostat's clock, and changing wakeTemp will modify the temperature setting for the morning. In the Internet world, this service resides on the processor in the basement. In the user's physical world this service resides in a box on the hallway wall.

Clients

Services are accessed via <u>*clients*</u>. In WWW terms, a client is like a browser except that it can modify and monitor data as well as retrieve it. In the XWeb project we have built clients for desktops, televisions, pens on a wall, speech, laser pointers and gloves. Each client implements an interactive behavior that is appropriate to the interactive devices that the client has available. In the Internet world, a client is a source of GET and CHANGE requests and is a computer with a particular IP address. To a user in the physical world a client is an interactive tool with a collection of input and display devices that together can manipulate any XWeb service. For example a PDA with its screen and stylus, a cell phone dialed into an Internet connected computer, a television with an embedded processor and a special remote control, and a wall display controlled by a laser pointer[OLSE 01] might all be clients. The software for supporting the interaction is all embedded in the client device, which is referenced, by a domain name or IP address.

Views and tasks

Raw XML is not an appropriate mechanism for interaction. The key to XWeb interfaces are *views*. A view is an abstract definition of a particular interaction. Views are encoded in XML using the XView language. The abstract nature of XView allows interfaces to be distributed to diverse kinds of interactive clients[OLSE 00]. An interactive *task* is defined by a special two-part URL of the form *dataReference::viewReference*. A client receives its task in much the same way as a WWW browser. The user either enters a URL or follows links to reach the task.

A view is simply another piece of XML residing on an XWeb server. This may be the same server as the desired service or some other server entirely. A view for our thermostat service might contain better names than "wakeTemp" as well as icons and synonyms to help clients present a more effective interface. The normal thermostat view might not expose the currentTime, leaving that to a configuration view.

Combining a view with compatible data within a client forms a complete XWeb interaction with a particular XWeb service. Views are defined as a tree of *interactors*. A particular interactor can be referenced by a *path-name* from the root of the view. Our thermostat service may be integrated with other services in a single home automation view that contains the VCR control, cable TV, home lighting, sprinkler timer, and microwave services. The path-name specifies which part of which service within home automation, the user is currently manipulating. The data and view of its current task as well as the path-name of the interactor that currently has the focus can characterize the current state of a client.

SUBSCRIBE

One of our goals is to combine multiple clients working on the same task. For example our work with the speech client has shown that speech is awkward for navigation. The laser pointer client is excellent at navigation by pointing, but awkward in changing values. Dynamically integrating these two on the same task provides a much better interaction. A user might carry a speech client embedded in a cell phone but would need to encounter a wall display with laser pointer support and then dynamically combine the two.

To support multi-client interaction, we introduced the XWeb SUBSCRIBE method. When a client begins a task it subscribes to the associated data. When an item of data is changed, the server will forward that change to all clients that are currently subscribed to that data. Upon receiving a subscription change notice each client will update its display appropriately. Thus any set of clients that are subscribed to the same data will have synchronized copies of that data. Since multiple clients can change data, there are potential conflicts that must be resolved. We will discuss those techniques later in the paper.

As an example of the use of SUBSCRIBE, consider a home thermostat. The living room television might have an embedded XWeb client, which could be directed to the thermostat's URL and thus present the thermostat on the television. When the television client first references the thermostat service it not only gets the data but also subscribes to that data by means of the SUBSCRIBE protocol method. The user may also reference the thermostat through an intelligent wireless phone that implements a different XWeb client based on speech recognition. Now both the phone and the television are subscribed to the thermostat. If the user says "Wake up time – set to six thirty A.M." not only will the thermostat change but the subscribed television client will also be notified so that the time changes on the screen. Conversely the user might use the remote control to change the time using the television client and then would hear "wake up time set to six thirty A.M." spoken through the subscribed telephone client.

Neither the television, nor the telephone clients are aware of each other, but they both are integrated on the same task. The XWeb protocol and its SUBSCRIBE mechanism form the framework that ties these interactive modalities together. The problem that Join and Capture addresses is to simplify bringing together the thermostat, television and telephone without a lot of configuration effort on the part of the user.

Sessions

Since XWeb behaves much like the WWW, interactive tasks are linked together and the user may move around freely. For clients to cooperate closely with each other it is important that we be able to synchronize their navigation as well as their data. This synchronization of tasks it not only helpful for multi-client interaction but also for multi-user collaboration. The ways in which people may want to organize their collaborations are very broad. We therefore wanted a single mechanism that could freely tailored. To do this we introduced the notion of a *session*.

A session is a piece of XML data with the special tag <XWeb:session>. A session has three parts:

- URL for the data
- URL for the view
- Path name to the current interactor

When a client's URL references a session object, that client joins the session. Because a session is just a data object on an XWeb site, a client can subscribe to that object and be notified of any changes. Whenever a user changes the current interactor of a client (moves from "sleep time" to "sleep temperature"), if that client has joined a session it will change the interactor path-name for that session. The server forwards this interactor change to all subscribed clients who then change their current interactor in a consistent manner. For example if the user said "sleep temperature" to the telephone client, that client would move to the new interactor and also change the session's interactor reference to be "thermostat/sleep temperature." Because the television client is subscribed not only to the thermostat data but also to the user's session, the television would move its selected interactor to sleep temperature. If the user then used the remote control to scroll to "wake up time", the subscribed telephone client would receive the session notification and would speak the wake up time.

A similar session modification and notification occurs if a client follows a link to another user interface. The session's data and view URLs are changed to the URLs of the link. This causes all clients subscribed to that session to similarly change the data and interface that they are using. XWeb's SUBSCRIBE is similar to the subscription mechanisms for multimodal interaction used in QuickSet [COHE 97]. We do not, however, attempt the fine grained unification of multiple modes of interaction found in QuickSet. We merely coordinate the data changes rather than support multimodal disambiguation of user inputs.

Subscription to sessions keeps clients synchronized. Because sessions are just data objects on servers, an unlimited number of sessions with any organization, naming, editing or management strategies being possible. A client finds and joins a session in the same way that a WWW browser might locate and access a web-based chat room. Most of the session management joining and departure strategies proposed for collaborative work are quite simply implemented using XWeb sessions. Sessions will form our basic mechanism for assembling various interactive resources around a particular user's task. In Join and Capture we will associate each user with a session object. The user's session can be anywhere on the Internet and need not be directly associated with the user's current physical location.

Connecting XWeb to the physical world

Most of the XWeb concepts have an Internet identity. Services, views and sessions all have URLs. If one knows the URL for each of these, one can GET or CHANGE them as well as SUBSCRIBE to them. Clients are identified by their domain name or IP address. Our first problem is that clients, services, views and sessions are all scattered around the Internet and we need a convenient mechanism for users in the physical world to bring them together to address a particular user task. A related problem is that when a user is interacting in the physical world, entering URLs directly into a client is not an effective experience. This is particularly true if no keyboard is available. The nice thing about the real world is that we can reference things by touching, pointing, or standing near physical objects rather than describing, selecting or searching. This stands in sharp contrast to most Internet discovery tools such as JINI [WALD 99], which assume that the user will use the Internet to establish connections.

To accomplish these goals we need to make the following connections between the physical and Internet worlds.

- Physical object that has an associated service -> two part URL for the interface to that service
- Physical object that has an associated client -> Domain name or IP address for that client
- A physical user -> URL for that user's current session

JOIN AND CAPTURE

Cut, Copy and Paste are three simple operations for integrating the work product of a wide variety of applications. They have become deeply ingrained in the culture of graphical user interfaces. Similarly we propose two commands for dynamically assembling multimodal interactive resources: Join and Capture.

Our model makes several assumptions. The first is that all interactive devices are connected to a shared network. This need not be the Internet. A local network of Bluetooth devices would be sufficient. The current task for each user is represented by the URL for a user session that is hosted on some XWeb server. The session may exist on the user's PDA that has a wireless connection or on her home server back at the office.

When a user walks into a room there may be several XWeb devices in the room, either clients or services, and the user may be carrying one or more client devices. A desktop workstation might be an interactive client. A microphone headset or a pen-enabled whiteboard may also implement XWeb clients. Each of these interactive clients implements XView interfaces in a manner that is appropriate to their own interactive modality.

Example interactive services might be lights that can be remotely controlled, the room thermostat, or a television. There may also be information services such as a news feed or email access. These services may be provided outside the room but their access must be physically manifest in the room.

We also assume that the user can readily recognize physical objects that are XWeb clients or services. This might be because they have special devices or tags on them or perhaps a standard XWeb icon that is easily recognized. It would be very annoying if the user needed to walk around the room waving at things to see which objects are active.

Join

An interactive service can be characterized by its URL. An interactive client may already have an active task, which also can be characterized by its two-part URL. For example, because of some prior use our television client may already be pointed at the thermostat service's interface. A user can JOIN a service or the active task of a client. The user with the telephone client might JOIN the thermostat directly or it indirectly by joining the current task of the television client. To execute a JOIN we must know the URL of the user's session and the URL of the task being joined. The task's URL can be inserted into the user's session by means of an XWeb change transaction, which will then be propagated to all clients that are subscribed to that session. Thus all of the user's currently attached clients will now be working on the interactive task that the user has joined.

Our user might walk into a conference room where a budget meeting is in progress, while carrying a laptop that is already subscribed to the user's session. By JOINing the budget task already on the conference room's whiteboard, the user's laptop is now collaborating with the rest of the participants in the room. The budget task's URL would be placed into the user's session to which the laptop was subscribed. The subscribing laptop client would be notified and would redirect itself to the new data and view. From now until it is detached, the laptop client and the other clients working on the budget task are synchronized.

Formally a Join of user U with client or service S would be

$$\begin{aligned} &U.session.dataURL \leftarrow S.dataURL \\ &U.session.viewURL \leftarrow S.viewURL \end{aligned}$$

The concept of joining carries with it the ability to create spontaneous collaborations. Whenever we encounter someone with whom we want to share a task, we can join their task, by referring to one of their clients or they can join ours. The remaining technical problem is to identify who is joining whom and when this is to be done. This will be discussed later in the paper.

All XWeb clients have the ability to detach from a session. When detached from a session, the client is still pointed at the same data and view, but is no longer subscribed to the session. This means that data changes will be propagated but navigation through the interface will not. This allows users to connect to the same task and then work independently. They will see each other's changes if they are still looking at the same data, but their interfaces will not automatically follow each other around.

Capture

To interact with anything, a user must have control of one or more interactive clients. To obtain control of a client the user must Capture the client. A capture operation causes the client to subscribe to the user's session. The client will now be using the user's session as the source for its data URL, view URL and interactor path-name. Any changes made by the captured client will not only change the data, but also update all other clients subscribed to the user's session.

In our thermostat example the user is already working with the television client. By picking up and identifying himself to the telephone client he can capture the telephone for his use. The telephone now references the user's session and its behavior is synchronized with the television by means of the subscription mechanism.

A user may walk into a room and capture a client that uses a laser pointer to control a wall display [OLSE 01]. The user may then pick up the microphone headset and capture its associated client. Now both clients are subscribed to the user's session. The user may now point at objects with the laser pointer (changing the current interactor) and speak the new values. The two modalities are working together on a common problem and the user is freely switching between modalities using the relative strengths of each. The combination of these modalities is strictly opportunistic based on whatever interactive clients are available to the user.

Formally for user U to capture client C

$$C.task \leftarrow U.Session$$

Implementation requirements

To implement the concepts of Join and Capture we need to obtain the two part URL of the desired task and the identity of the user. We also need mechanisms for converting a user's identity into the URL for that user's session and the ability to transfer URLs among devices, clients and the user's session. In the following sections we will describe two implementations of these facilities.

JAVA RING

Our first implementation of Join and Capture uses a Java ring from Dallas Semiconductor as shown in Figure 1. The Java ring has a small Java virtual machine running in a microcontroller encased in the ring. In our system, each

Java ring is encoded with the URL of its owner's session object.

Figure 1 – Java ring for user identity

Each interactive client or service object has an associated iButton connector, which can accept the Java ring. When approaching a client or service, the user plugs in the Java ring. This makes a connection between the client/service and the user's session. The remaining question is whether to join or capture. Interactive services such as the thermostat can only be joined, so there is no question. All of our interactive clients initiate a platform-appropriate dialog for specifying join or capture. For example the speech client would immediately ask "do you wish to join or capture?" The user then responds with the desired choice. Because the Java rings have more memory than a single URL, some of our clients offer a dialog that manages a history of URLs and the titles of their interfaces. This provides the user with additional flexibility in working with multiple tasks. A user can walk around a new building and join several services. These are now remembered in the Java ring and can be used later when capturing interactive clients.

This implementation is not restricted to Java rings. In essence all that is needed is to have a tagging mechanism and a reading mechanism. For example every user could carry a unique RFID tag [WANT 99]. Each client or service would have a reader. Waving the tag next to the reader establishes the necessary task/user association. Conversely, the user could also carry a reader and have tags attached to clients and services. Smart cards and magnetic stripped cards can all be used in a similar fashion. Rekimoto's color codings [REKI 95] would work, but there are not enough possible encodings for such a solution to scale to large environments. Rekimoto's NaviCam could be converted to a portable client, which would automatically join any service that it detected.

PEOPLE WATCHER

Walking around a room and plugging rings into sockets or swiping cards through readers is still a somewhat cumbersome mechanism for selecting interactive behavior and assembling interactive resources. As an alternative we have developed the "people watcher" which is based on a set of cameras watching a room. In this scenario the user captures clients by walking up to them and joins services or clients by touching them.

Consider the scenario of Jim walking into a conference room where Jane is sitting working on a problem using her laptop. After a brief discussion they decide that they both could effectively work on the problem together. Jane steps to a projected whiteboard and the problem she was working on appears there. Jim then touches a spot on the projector labeled Join. The problem appears on his PDA where he can make modifications from his seat. Using his PDA client he detaches from the problem and searches the Internet for more relevant information. He then steps to the wall and his new information appears on the screen.

In the terms of our XWeb system the above scenario consists of Jane capturing the projector's client for her problem by stepping up to the wall. Jim joined the problem by touching the projector's Join label. Later, with new information, he captured the wall client by stepping to the wall himself, replacing Jane.

In our XWeb implementation we model all of these behaviors as Join and Capture of interactive services and clients. Our only sensors in this version are cameras mounted in the ceiling of a room. This is similar to the Follow-me project[HART 99] that uses ultrasonic beacons for the tracking of people and objects. The EasyLiving project [BRUM 00b] presents a similar scenario for home automation and uses cameras and a geometry model for matching up behaviors with user location. In our People Watcher we use much simpler image processing and the more powerful XWeb cross-modal interaction protocol along with the uniform concepts of Join and Capture.

For the above scenarios to work we need the following capabilities:

- The identity of each user (user session)
- To know when a particular user is standing near an interactive service or client.
- To know when a touchable spot has been touched.

There are two major tasks to be performed. The first is to set up a room so that it can use cameras to sense the user's location and behavior relative to services and clients in the room. The second is to perform the actual interaction using the room.

Setting up a People Watcher Room

To set up a room we need to define places that users can be standing or sitting that are associated with the various interactive clients and services. For example, we need to associate the place next to the whiteboard with the whiteboard client. We need to associate the "join" spot on the projector with the projector's URL. Our set-up consists of defining such "places" and "touches" for the cameras. Places are physical locations where the identity of a user can be recognized. Touches are physical locations where a user may perform an action by touching the spot. There is also a special place where users will be when they "register" into the room.

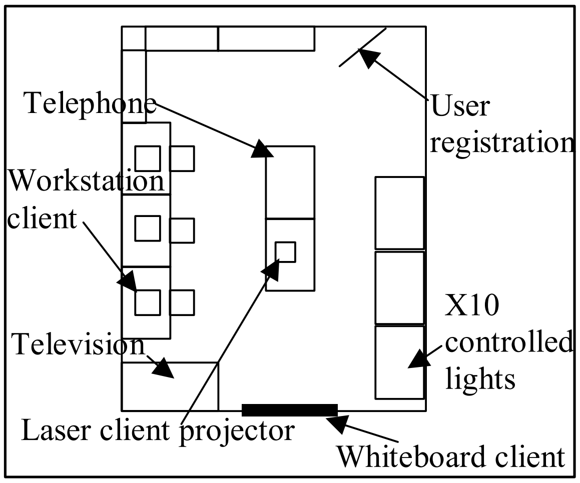

Figure 2 – Layout of a room

Consider the room layout shown in figure 2. There are several clients and several services. We have multiple cameras watching the room. We can uniquely identify regions in the room by rectangular hotspots in the views of two or more cameras, as shown in Figure 3. In setting up the room, we draw hotspots on the various camera images and associate client domain names or service URLs with these rectangles, as shown in Figure 4. Places are identified as either join, capture or identify a user. Touch hotspots have an associated place (where the user will be) and an associated service URL.

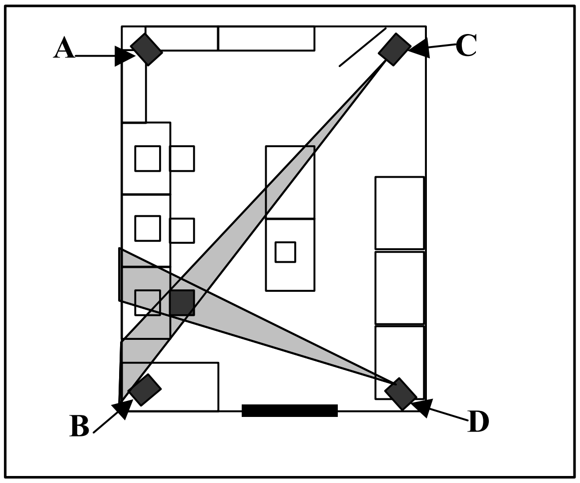

Figure 3 – Identifying a place

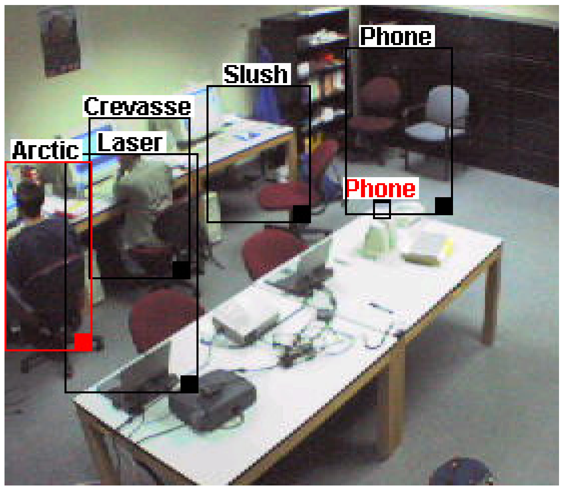

Figure 4 – Drawing hotspots

Each room has a registration area (an identified rectangle in one or more cameras). When the user steps into the registration area they must identify themselves with a Java ring or other identification. The cameras then sample the user's features (we use a simple color histogram) out of the registration area rectangles. These features are bound to the user's session URL discovered from the Java ring.

In defining places and touches we have avoided the concept of a global geometry model [HART 99, BRUM 00a]. We did this partly to avoid its complexity, partly to simplify our image processing and mostly because geometry is unnecessary for our purpose. We don't actually care where the user is. We only care about what client or service they are near. A global geometric model is one very flexible way to answer this question, but not the only way.

Interacting with the People Watcher

When a user steps into a place the cameras will notice the change in the associated hotspots, sample the image features and map those features to a particular user. If multiple cameras vote on the same user in that place then the join or capture operation associated with that place is done. If two or more cameras indicate that skin color appears in the same touch hotspot, then the associated place is sampled for user identity and a join operation is performed on the service associated with the touch.

Our image processing techniques are rather crude and subject to error. However, they do demonstrate a non-obtrusive interface to Join and Capture that allows users to assemble interactive resources and acquiring access to services simply by moving around a room and touching physical objects. The room setup is highly flexible. Any object in the room can become an identifier for a client or a service without physically modifying the room.

MANAGING MULTI-DEVICE INTERACTION

A key problem that was glossed over in the prior discussion is that multiple devices, possibly under the control of multiple users are asynchronously modifying data objects and session objects on a variety of servers. Because of the highly dynamic nature of the environment we propose, we cannot rely upon normal activity keeping everything straight. We must ensure that all clients are synchronized. We also have the problem that the mechanism must scale to the size of the Internet. Therefore there cannot be a global synchronization manager to which all clients register. Our problem is somewhat simplified in that each client is interacting with at most one server at a time. Therefore, we can ignore the issues associated with multi-server transactions.

XWeb uses a replicated client/server architecture. Clients maintain copies of that portion of the server tree that they are currently viewing. The server maintains the master copy. The key problems are to recognize when there are synchronization problems and to repair those problems. Our fundamental assumption is that the server is always right and the client interface must be updated to reflect what has happened at the server. The server never does any undo or repair. By holding the server changes as permanent, a client need not consider the state of any other client, unlike Grove [ELLI 89].

Our approach to management of change is optimistic serialization. Greenberg and Marwood have indicated problems both in the user interface and the implementation of this strategy [GREE 94]. However, we consider these interactive problems as less important than reliability and scalability of the solution. A key implementation problem in optimistic serialization is undo and repair of rejected changes. Every XWeb change record has sufficient information to undo every change. This is somewhat in contrast with serialization by operational transforms. Changes are serialized by the order in which they are received by each XWeb server. There is no order preservation or concept of transactions that involve multiple services. The key issues to be addressed in resolving conflicts is to determine when changes are out of order, to reorder changes whenever it will not affect the end result, and to undo changes that cannot be reordered.

The key to correct ordering is transaction IDs. We rejected any model for ordering that involved the internal timings of various clients as in [ELLI 89]. Transaction IDs are strings generated by a server. They are ordered in lexigraphical order and only their ordering is important to the algorithms. Servers are free to choose whatever encoding they wish. We consider it very important to the scalability of such systems that restrictions on server implementations be as few as possible.

Commutative changes

Testing for reorderability of changes is based on an IsCommutative predicate. IsCommutative must return true only if two changes can appear in any order without changing the result of applying both of them. Every client and every server must implement such a predicate, but it is not necessary that their implementations be equivalent. Again this is in keeping with our goal of minimizing compatibility requirements. This predicate need not exactly test the commutativity of two changes. Commutativity can be quite a complex issue. However, as long as IsCommutative only returns true when two changes are reorderable, the conflict resolution algorithm will perform correctly. The simplest implementation of IsCommutative is a constant false, which will always produce a consistent state across all clients but will cancel more changes than necessary. Our current implementation of IsCommutative returns true for changes that can be easily guaranteed not to modify the same pieces of data.

Server conflict resolution algorithm

The server will receive two kinds of events from clients. They are CHANGE requests and confirmations by clients that they have received change notifications. An XWeb server keeps a list of all clients that are subscribed and the transaction ID of the last change that each client has confirmed. The value LUN (Least Unconfirmed Notification) is the smallest ID from all clients. The server also remembers all changes whose IDs are later than LUN. These changes are used to determine whether a new change received from a client conflicts with already committed changes that the client did not know about.

When a server receives a change transaction from a client it checks to make sure that it can be reordered with all transactions that are later than the last one the client has confirmed. If it can be safely reordered (using IsCummutative) then the change is performed on the data, other subscribers are notified and it is added to the list of changes. If the new change is not commutative with changes that were committed after it's transaction ID, then it is discarded and its client is notified that the change was rejected.

When subscribing clients are notified of changes, they return a confirmation to the server. This confirmation contains the last transaction ID known to the client. Using this information the server can update LUN (least unconfirmed notification) and remove any earlier changes from its list of saved changes. Those earlier changes are now known to all subscribing clients and need not be considered in later client requests.

From the server's point of view it is checking to make sure that changes from each client are in order with any changes that a client does not yet know about or that their respective order does not matter. Any unreorderable changes are rejected and the client is notified. This is highly reliable and highly scalable if one assumes that there are many servers in the world each operating independently. Use of LUN and timeouts on subscriptions keep the server burden from becoming too large.

Client conflict resolution algorithm

The client algorithm is more complicated than the server algorithm because it must repair conflicts where the server need only reject them. The client layer must concern itself with three types of events: 1) Changes have been performed by the user interface but not yet sent to the server, 2) A change that has been sent to the server, but not yet confirmed by the server. 3) Change notifications from the server that were caused by other clients. One of the fundamental principles of our algorithms is that neither the server management nor the user interface can be blocked. The user must be able to move forward as if there was no server involved. The only perceived network delay should be when new data is being retrieved using the GET method.

In general the client algorithm works by keeping track of changes that the server does not yet know about and the any change that has been sent to the server, but not yet confirmed. When the client's user interface makes a change to the local copy of the data, a change record (with undo information) is added to the list of changes to be forwarded to server and the interface continues on its way.

When a client receives a response from the server to its change request it will either be an acceptance of the change or a rejection. If the change was accepted, it is forgotten. If the change was rejected, then the rejected change must be undone as well as any pending changes (not yet sent to the server) that are in conflict (using IsCommutative). Any changes in the pending queue that are not in conflict are retained. Whenever changes are undone, the normal Model-View-Controller mechanisms handle the user interface update.

A client also receives notification of changes that the server received from other clients. When one of these notifications is received, the client must check the change against all pending changes. Any pending changes that are in conflict must be undone. Undoing a pending change may also cause additional later pending changes to be undone. The notification change can then be performed. As long as two clients are not trying to manipulate the same data at the same time, our simple IsCommutative check for reordering will not flag any conflicts. If there is a conflict, however, the data will revert to whatever change reached the server first. This guarantees that all clients will settle to the same data, but may cause a user's changes to suddenly revert to another value. However, the user cannot distinguish such an undo from any other action performed on another client. Therefore we do not see this as a serious problem.

There is still a small issue of a change P that has already been sent to the server and is not yet confirmed by the server when the client receives notification of a change C by a different client. It is possible that the client's implementation of IsCommutative is more conservative than the server's implementation. In this case the client will see a conflict between P and C and will undo P. The server, however, has not yet considered P and may, by more careful inspection, determine that P and C are not conflict and therefore will accept P. To handle this case, when the client receives C and finds a conflict it will undo P and any other pending change that conflicts with either P or C, but it will save P. The client is predicting that P will be rejected. Later if P is rejected, it is discarded. However, if P is accepted, the client can take the saved copy of P and treat it like a new change notification from the server, undoing any pending changes that are in conflict and redoing the change P.

Summary of conflict resolution

By using the server's transaction IDs as our representation of time, we can make sure that every client is finalizing changes in the same order or in an order that will produce identical results. Because of the limited set of editing operations we can easily reason about all possible ordering conflicts among changes and provide for exact undoing. The key to the conflict detection is the IsCommutative test, which need not be identical among clients and servers as long as it never returns a false positive. This approach allows both servers and clients to never block and ensures that eventually the servers and all clients will settle on the same data values. This algorithm also imposes very minimal implementation consistencies. We believe that minimizing required consistencies is very important to scaling this infrastructure to the size of the Internet.

CONCLUSION

Nomadic computing requires that people be able to access information in a wide variety of physical situations. One approach to this is to require users to carry their computing with them. This paper describes an architecture whereby people can acquire interactive resources as they encounter them in their environment. We call this action a Capture.

Ubiquitous computing postulates that there be a wide variety of interactive services in the world and that we can computationally access them. Our architecture provides the concept of Join by which a user can acquire access to a service's interface.

By means of Join and Capture users can assemble a variety of multimodal resources to apply to a particular problem. The XWeb SUBSCRIBE mechanism provides the framework for all of these clients to operate together on a problem. For example, a user can combine speech clients with pointer-based clients for a more effective total interaction. Join and Capture provide a simple framework for users to exploit interactive resources and services as they encounter them in their environment.

We have also presented a robust algorithm for guaranteeing consistent data among clients manipulating to the same data. This algorithm can easily scale to many servers and many clients in a spirit similar to the architecture of the WWW.

REFERENCES

[BORR 00] Borriello, G. and Want, R. "Embedded Computation meets the World Wide Web," Communications of the ACM 43(5), (May 2000).

[BRUM 00a] Brumitt, B., Krumm, J., Meyers, B., and Shafer, S. "Ubiquitous Computing & The Role of Geometry," IEEE Personal Communications, (August 2000).

[BRUM 00b] Brumitt, B., Meyers, B., Krumm, J., Kern, A., and Shafer, S., "EasyLiving: Technologies for Intelligent Environments" Handheld and Ubiquitous Computing, (Sept 2000).

[COHE 97] Cohen, P. R., Johnston, M., McGee, D., Oviatt, S., Pittman, J., Smith, I., Chen, L., and Clow, J., "QuickSet: Multimodal Interaction for Distributed Applications," Proceedings of the fifth ACM international conference on Multimedia, (1997).

[ELLE 89] Ellis, C. A., and Gibbs, S. J., "Concurrency Control in Groupware Systems", Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data, 1989.

[GREE 94] Greenberg, S., and Marwood, D., "Real Time Groupware as a Distributed System: Cuncurrency Control and its Effect on the Interface," Proceedings of CSCW '94, (Oct 1994).

[HART 99] Harter, A., Hopper, A., Steggles, P., Ward, A., and Webster, P. "The Anatomy of a Context-Aware Application." Mobicom '99, ACM (1999)

[OLSE 00] Olsen, D. R., Jefferies, S., Nielsen, T., Moyes, W., Fredrickson, P., "Cross-modal Interaction using XWeb", ACM Symposium on User Interface Software and Technology UIST '00, (2000).

[OLSE 01] Olsen, D. R., Nielsen, T., "Laser Pointer Interaction", Human Factors in Computing Systems (CHI Letters, vol 3(1)), (2001).

[REKI 95] Rekimoto, J., and Nagao, K. "The World through the Computer: Computer Augmented Interaction with Real World Environments," ACM Symposium on User Interface Software and Technology (UIST 95), (1995).

[REKI 98] Rekimoto, J., "A Multiple Device Approach for Supporting Whiteboard-based Interactions", Human Factors in Computing Systems (CHI 98), ACM, (1998).

[REKI 00] Rekimoto, J., and Ayatsuka, Y., "CyberCode: Designing Augmented Reality Environments with Visual Tags," Designing Augmented Reality Environments (DARE 2000), ACM, (2000).

[WALD 99] Waldo, J. "The Jini Architecture for Network-centric Computing," Communications of the ACM (July 1999).

[WANT 99] Want, R., Fishkin, K. P., Gujar, A., and Harrison, B. L., "Bridging Physical and Virtual Worlds with Electronic Tags," Human Factors in Computing Systems (CHI 99),ACM (1999).

[WEIS 93] Weiser, M. "Some Computer Science Issues in Ubiquitous Computing", Communications of the ACM, Vol 36(7), (1993).