XWeb: An Architecture for Cross-modal Collaboration

Dan R. Olsen Jr., William A. Moyes, Sean S. Jefferies, S. Travis Nielsen
Computer Science Department, Brigham Young University, Provo, Utah, 84602, USA
{olsen, wmoyes, jefferie, nielsent}@cs.byu.edu

ABSTRACT
In this paper we describe the XWeb architecture for client/server interaction. We show how general interactive clients tuned to particular interactive devices can collaborate in a general way with information services. The independence of the implementations of interactive clients and servers is essential for pervasive and cross-modal collaboration. Communication about data changes and reasoning about change conflicts can provide such a collaborative substrate. An algorithm for conflict resolution is given. It is shown how the XWeb architecture naturally supports synchronous session management across interactive modalities.

Keywords
Collaboration architectures, World Wide Web, multimodal interaction, change-conflict resolution

INTRODUCTION
The advent of the Internet as a pervasive medium of communication has expanded the opportunities for collaboration among people across the barriers of time and space. However, the sheer size and variety of what is possible and available on the Internet causes problems for software architectures that support collaboration. What is needed is a pervasive collaborative architecture. For pervasive collaboration to succeed, all collaborators need to be using compatible software. Every new application and every new version of an application causes the collaborative fabric to break. The extent of this problem has been discussed elsewhere [10]. What is needed is a common format for collaborative communication around which a huge space of applications can be developed. This collaborative fabric must allow individual users their independence without breaking the collaboration. Requiring all users to synchronize on exactly the same software installation will not scale to communities the size of the Internet.

The pervasive collaboration problem is complicated by the fact that many work or play situations are not suited to standard desktop computing devices. Hands-free or eyes-free interactive situations would lead to speech or other audio interactions. Wearable computers can be more effective in field work situations [1]. Interacting on a wall display as an individual is different than interacting on a wall as a group, which is different than interacting through a remote control and a television. A truly effective solution to the pervasive collaboration problem must connect people working on a variety of interactive platforms.

The XWeb project described in this paper is focused on creating a pervasive collaboration substrate that can span disparate interactive modalities. We have taken as our model the client server principles of the World Wide Web while greatly expanding the interactive and collaborative capabilities. The key principles are

a) a standard communications protocol and medium for representation,
b) high flexibility in the kinds of information servers that can be provided, and
c) client software which can interact with any XWeb service.

Using the WWW model we separate client user interfaces from information services so that client software can be adapted to specific interactive platforms. So far we have implemented interactive clients for desktops, speech-only and pen-based wall displays. We are working on clients based on minimal button sets, multiscreen rooms with user tracking and wearable computers. The XWeb goal is that a user on any one of these interactive platforms can collaborate with any other user on any other client platform. The current XWeb implementation only has a limited set of interactive behaviors that we are working to expand. This paper is focused on the collaborative architecture used by all XWeb clients and servers rather than specific interactive techniques.

We will first present an overview of the XWeb architecture followed by three example applications that we have built. We will then show how the XWeb architecture addresses five key problems in collaborative software.

1) defining the user interface,
2) insulation of both client and server software from the collaboration facilities,
3) propagation of change,
4) resolution of change conflicts, and
5) session management

XWEB OVERVIEW

The WWW has been enormously successful in distributing information across the Internet. We believe that its key architectural advantage lies in the fact that individual users of the WWW can choose their client/browser software without any consideration of any other user. Having obtained and learned that software they can access and interact with any HTTP compliant server in the world. A single piece of client software unlocks a world of information resources. Conversely, information providers need only provide their information via an HTTP server and that information is available to a world of Internet users. The power lies in the fact that no user must consider any other user in their choice. Information providers who use HTTP/HTML as their interactive vehicle are no longer concerned with the software installations of their users. The WWW is built by a series of compatible but independent choices rather than a globally synchronized adoption plan.

This is in sharp contrast with current architectures for collaboration. There are a variety of individual collaborative applications, notably word processors [9], shared whiteboards [11], and outline editors [6]. Because they are single applications, each with unique collaborative behavior they do not have the pervasive usefulness of a web browser. Collaborative architectures such as GroupKit [13], Suite [3], Prospero [5], and JAMM[2] require that all users share compatible software versions. Software choice therefore must be planned and synchronized rather than dynamically grown as the WWW has done. No globally synchronized solution can scale to the size of the Internet.

HTML/HTTP, however, is an interactively impoverished mechanism for collaboration. The model is publish-mostly with interactive capabilities added on later in the process and hacked into special encodings of URLs. HTML is interactively equivalent to the old IBM 3270 terminals from the early 1970's. However, there is a massive movement to rearchitect corporate information interaction into HTML because of its pervasive usefulness. Users willingly tolerate limited interactivity in exchange for uniform universal access. The key lesson from the WWW is that collaborative environments the size of the Internet are far more valuable than carefully crafted, highly focused collaborative single applications.

The goal of the XWeb project is to retain the architectural lessons from the WWW while expanding the interactive and collaborative facilities. The basic architecture for XWeb is shown in Figure 1.

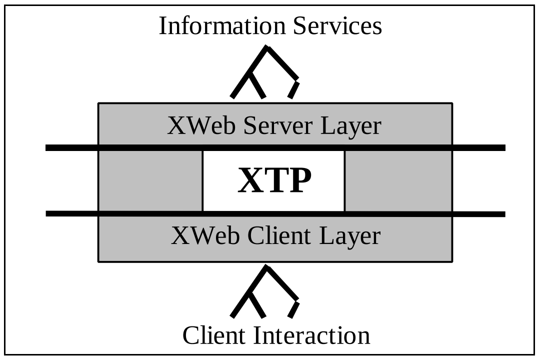

Figure 1 - XWeb Architecture

The model for XWeb, rather than document downloading, is the editing of XML trees. A file system for example can be presented through XWeb as an XML tree that clients may browse.

The connective tissue that holds XWeb together is the XWeb Transport Protocol (XTP). XTP is modeled after HTTP and retains the GET method for retrieving data from XWeb servers. In fact XWeb servers can respond to WWW browsers in delivering information. The protocol, however, has been expanded to include the method set

GET - to retrieve data
CHANGE - to make modifications to server data
FIND - to search a server's information
SUBSCRIBE/CLOSE - to request change notifications

The CHANGE method uses the XChange language, which is an XML representation of modifications to be made to the server's information tree. Changes include setting values or objects, deleting objects and moving objects around.

The FIND method for searching is based on XQL. It does not have a bearing on the rest of this paper.

The SUBSCRIBE method, along with CHANGE, is the heart of XWeb's collaborative capabilities. Using the SUBSCRIBE method, a client can specify a portion of a server's XML tree to be monitored. The client specifies the URL of a tree of its own to which change records will be sent whenever the server's tree is modified.

EXAMPLE APPLICATIONS

The first example is a simple tic-tac-toe (naughts and crosses) game for two players. The underlying model for this application is quite simple, consisting of nine values of null, X or O. There are a variety of interfaces to this application that can be used. On a desktop interface each cell could be a drop down menu of the three choices. On a pen-based interface the choices could be associated with Graffitti gestures of space, X and O. By giving each cell a name such as north, north-east, east, etc a speech interface is also possible. In XWeb such a game is simple to create and accessible from all interactive modalities.

A second example is a system we have constructed for extracting records of French Huguenots emigrating to the United States. A set of widely distributed volunteers view scanned images of original records and enter the personal information. Frequently they work alone, but many times they collaborate with others in deciphering handwriting or discussing how information should be represented. For our purposes the underlying model is a list of people with subitems for information such as name, age, gender, place of birth, etc. In addition, each person has several lists of relatives mentioned in the record along with their names, relationships and reason for mention. For each person there is an extensive tree of extracted information that is quite easily represented in XML. However, because of other constraints on the project the data is stored in an Access relational database and only served via XWeb as an XML tree.

A third example is home automation control of items such as lighting, sprinklers, heating, and entertainment. The possible settings for a home are easily represented as an XML tree. However, modifying the settings must also trigger the appropriate controllers. Access to the home automation can be through a variety of input devices such as television set-top boxes, cell phones outside the home, and voice control inside the home. People are frequently confused by the initial configuration of such services and must collaborate remotely with vendor support staff.

DEFINING THE USER INTERFACE

The details of the interactive clients and their interfaces are beyond the scope of this paper. However, information servers must provide not only the data but information on how to interact with the data and which information is to be made available. This information is provided in an XView descriptor that is an XML representation of how data is mapped to the standard widget set understood by the clients. An XView includes information such as icons, field names, speech synonyms, interface structure, layout, and help text. These resources are used by the clients in constructing appropriate interfaces to the data being served. An information service, however, only communicates with the clients in terms of GET and CHANGE methods. All interactive techniques are embedded in the clients and parameterized by the XView information.

When a client wishes to access a collaborative resource it provides a special two part URL. The first part is the URL for the data to be manipulated and the second is a URL for the XView descriptor that defines the interaction. This allows views to be applied to multiple data objects and a given data object to have multiple ways it can be accessed and manipulated.

ISOLATING THE COLLABORATION

To be successful, XWeb must provide a robust infrastructure for supporting collaboration that does not require extensive involvement from either user interface software or from new information services. XWeb provides XTP, the XWeb server layer, and the XWeb client layer as shown in Figure 1.

In the case of our tic-tac-toe example a simple XML file for the data and another for the XView can be placed in the directory of the server. The game requires nothing more than the changing of X's and O's. The Huguenot and automation problems are more complicated because SQL queries and X-10 home automation commands must be generated in response to changes.

We use Java throughout the XWeb implementation except where information services or interactive device drivers must be called. Central to our implementation is a Java interface called XMLObj, which provides methods for getting and setting attributes as well as adding, changing and removing child XMLObj objects. Using XMLObj, interactive clients view the information being edited as an abstract tree. By making changes to that tree the XWeb Client Layer converts the changes into CHANGE methods that are sent to the server. Clients only know about editing an XMLObj and receiving change notifications. They are insulated from the rest of the XWeb architecture. This is similar to the shared values mechanism in GroupKit[13].

On the server side, services implement the SrvObj interface. This provides basically the same methods as XMLObj. These methods are used by the server layer to formulate XML records to be sent in response to GET method requests. In addition, server objects provide the FIND implementation. For example, this allows our JDBC server implementation to translate XWeb FIND requests into SQL for more efficient processing of searches on the Huguenots. New SrvObj implementations are only required for new types of services. Only one implementation of the JDBC service is required to manage any number of database services.

Neither the services, nor the interactive clients must deal with anything other than their shared model of a modifiable and searchable tree. All networking and collaboration is masked by the XWeb layers.

PROPAGATION OF CHANGE

The key collaborative feature of the XWeb infrastructure is the propagation of changes. This is managed using the XTP CHANGE and SUBSCRIBE methods. The current implementation of XWeb assumes that served information is a tree. Based on this assumption any XWeb data item can be specified by the domain name of the site and a path name consisting of edge labels from the root of the site to the desired data item. This path name for any XWeb data item is key to the ability to reason about changes and to support robust collaborative behavior. For example it is quite simple to determine if two changes can possible impact the same data object.

A key assumption in our architecture is that all changes to the served information will come as CHANGE records submitted to the server. We enforce this constraint even on the services themselves. If a service wishes to change its information (possibly in response to some external event) then it will do so by processing a change through the server. This restriction provides the server layer with knowledge about all changes to the underlying information.

CHANGE records contain a sequence of primitive editing operations. The entire change is treated as atomic. There are operations to set values or objects, delete objects, insert into lists, move and copy. Any manipulation to a tree can be constructed from these operations. A key part of each such editing operation is the path name of the object to be modified. Because the set of editing operations is small it is possible to reason more exactly and reliably about changes and their conflicts than is possible in application-specific architectures. The combination of GET and CHANGE allows for interactive editing that is independent of a specific application or set of interactive techniques. This independence is key to achieving our cross-platform goals.

In addition to the actual changes themselves, the CHANGE message has two properties WHO and THRU which identify the user and the software/hardware platform requesting the change. This identity information is key to providing awareness in the interactive clients and for maintaining history information about the change patterns on a particular site. At present none of our interactive clients use this information. However, others have developed a variety of awareness techniques for which this information would be the foundation [4, 8].

Subscribe

The SUBSCRIBE method is the heart of our collaborative framework. In a SUBSCRIBE request, a client provides the path name for the root of a subtree of some information site (source tree) and the address and port to where the change notifications should be sent. The SUBSCRIBE request tells the server that any change to the indicated source tree is to be forwarded as a change to the client tree. The server can refuse the subscription request (because it imposes computational burdens on the server). The SUBSCRIBE method can also indicate the desired timeliness of the notifications ranging from instantaneous to weeks or months. At present our implementation only provides instantaneous notification.

The XWeb server layer maintains a list of all active subscriptions. Clients terminate their subscription gracefully by sending an CLOSE message or ungracefully by not responding to notifications. Our current implementation uses a simple list of subscribers. A large commercial-grade server could exploit the hierarchic nature of the XWeb name space to efficiently manage a large number of subscribers. For each subscription, the XWeb server layer stores the path name of the subscribed source tree, and the URL for the client tree.

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[7]. The user interface problems occur where multiple users make changes to the same data objects. The data appears to oscillate between the differing values. Although this is annoying, it does reflect reality. Similar conflicts occur in the physical domain. When two people attempt to manipulate the same object at the same time, confusion occurs, their hands get tangled, apologies are given and the situation is negotiated. We see similar social protocols being applied in the electronic domain. We therefore have chosen optimistic serialization as the mechanism that provides the fastest interactive response time.

The key implementation problem in optimistic serialization is undo and repair of rejected changes. When undo is based on arbitrary application semantics it is very complicated. In essence the problem is that of comparing the relative impact of all possible pairs of changes, and undoing any possible change. Given even a modest number of application specific commands the problem becomes intractable. With XWeb's application independent model for change, with its limited number of operators, the undo/repair problem is quite manageable.

Changes are serialized by the order in which they are received by the XWeb server layer. As each change is received, it is first sent to the appropriate SrvObj implementation to perform the change. If this is successful, then the server checks the change against each subscription path. Each subscriber then is forwarded a CHANGE message that is appropriately pruned to the desired subtree.

To receive subscription notifications, the XWeb client layer maintains a small server-like implementation of its own that can process CHANGE records on its local tree. These changes are then sent as notifications to the user interface implementation. This is similar to view notification in the Model-View-Controller architecture.

RESOLUTION OF CHANGE CONFLICT

A key problem in providing an application-independent collaborative infrastructure is ensuring that change conflicts are handled appropriately. This topic has been treated extensively in the literature. There are user interface level controls such as floor control, which inhibit more than one user from making a change. This is somewhat cumbersome to use and is very unlike normal person-to-person discussion. Floor-control is possible in XWeb, but we do not rely upon it. Locking mechanisms have also been proposed[15]. Mechanisms for automatic lock acquisition upon entering a view are possible. Such mechanisms, however, assume some "planning ahead" of the information usage. We prefer to use such mechanisms, not as our guarantee of change correctness but as a protocol for coordinating the activities of people. The highly tailorable mechanisms in Prospero[5] are noteworthy, but antithetical to XWeb's approach of ubiquitous, uniform behavior across a huge variety of information services, interactive clients, and applications. A key to the WWW success is the uniformity of user interface and infrastructure behavior. Tailoring of the conflict management to individual applications is inconsistent with such a world view.

Overview

XWeb uses a replicated client/server architecture. Clients maintain copies of that portion of the server tree that they are currently viewing. They obtain this copy in response to a SUBSCRIBE message. 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 [6].

The server serializes CHANGE transactions by the order in which they arrive at the server. Each processed transaction is assigned a transaction ID that is a string. The transaction ID can be in any format that satisfies the server needs as long as successive transactions are monotonically increasing using normal lexicographic ordering. The relationship between a client and a server can be characterized by the largest transaction ID that the client knows about. This is very important in deciding how much history both clients and servers must maintain to correctly support the conflict repair algorithm.

After a server has processed a CHANGE transaction, it assigns it a transaction ID and propagates the transaction to all relevant subscribers. The server expects a confirmation of the notification but does not wait for it. All management of the serialization and notification is handled in the XWeb server layer and hidden from any SrvObj implementations. This greatly simplifies the addition of new services and makes guaranteeing the correctness of conflict repair possible. If every new service was involved in conflict repair, the correctness would be a hopeless cause.

When a client's user interface makes modifications to the viewed data, a CHANGE record is sent to the server, but the client does not wait for confirmation. A client will, however, queue any further changes without sending them to the server until the pending change is confirmed. The user does not see this because the interaction continues as if all changes were confirmed. This optimistic approach greatly enhances the responsiveness of the user interface. When a client receives notification of changes that other clients have made, it may have unconfirmed changes of its own. Resolving pending changes with change notifications from the server is the key part of the algorithm.

A client must maintain sufficient information so that it can accurately undo any changes that have not yet been confirmed by the server. Because there are only five possible editing operations out of which all changes are composed, saving information for undo is quite simple. Undo information is only saved as long as changes are unconfirmed. The undo information can only grow significantly larger than the original downloaded information if the user interacts very rapidly and the server delays confirmation for a very, very long time. This management of the undo history and the client repair is hidden in the XWeb client layer, which simplifies the correctness of conflict repair.

We would like to avoid undo wherever possible in repairing conflict problems because retraction of changes can cause user confusion. To avoid undoing changes, we test to see if changes can be reordered without changing the end result. This reordering is based on the IsCommutative predicate. This predicate is true if two changes are guaranteed to be reorderable. It returns false otherwise.

Conflict Resolution Approach

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 cancel changes that cannot be reordered.

Representing the correct order

The key to correct ordering is transaction IDs. We rejected any model for ordering that involved the internal timings of various clients as in [6]. Attempting to correlate client clocks across unknown clients that are not in the control of the service is fruitless. The exact time of a CHANGE is not important for correctness, only the serialized order. Any algorithm that relies on clock synchronization between clients will not scale to an Internet-sized solution. By using lexicographical ordering on transaction IDs, all clients can determine relative ordering without concerning themselves with what information a server may encode in the ID. Each client and each server maintains the ID of the last transaction that they know the other has processed. This is sufficient for detecting and resolving ordering problems.

Commutative changes

Testing for reorderability of changes is based on our IsCommutative predicate. This predicate need not exactly test the commutativity of two changes. Commutativity can be quite a complex issue. Because of our choice of XQL as our model for hierarchically naming objects on a server, commutativity is not actually knowable in all cases. For example some nodes can be referenced both by name and by index. A change based on name references cannot always be tested correctly against a change based on indexed references. The IsCommutative predicate only returns true if the changes are known to be reorderable. If they are not reorderable or if the algorithm cannot determine, it returns false.

The simplest implementation of IsCommutative is a constant false, which is always correct. However, with this implementation no pair of changes can be reordered and many more changes are cancelled or undone than would be necessary. Correctness is preserved, but the user interface suffers.

Our current implementation of IsCommutative returns true for changes that can be easily guaranteed not to modify the same pieces of data. It is not completely exact. However, it does report commutativity for the vast majority of the cases that occur in practice. Its success is based on the fact that a given XView of a piece of data is probably shared by all clients currently active on that data. Such a view will use the same mechanism for referring to data objects, which will then result in exact comparisons in the IsCommutative test. All that is required for correctness is that the predicate never returns true when two changes cannot be reliably reordered. The implementation of IsCommutative need not even be the same for all clients or for clients and the server. This independence of implementation further enhances client independence.

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.

Server state information

There are several pieces of information that must be retained by the server in order to implement the conflict resolution algorithm. They are:

- The current server data.
- LST - last server transaction, which is the ID of the last transaction processed by the server.
- UT - unconfirmed transactions. This is a list of transactions for which at least one of the subscribers have not confirmed knowledge of the transaction.
- CLT - client last transaction. This is a vector of transaction IDs with one entry for each subscribing client. It contains the transaction ID for the last transaction for which the client is known to have received notification.
- LUN - least unconfirmed notification. This is the least transaction ID found in CLT.

LST and UT are used when a transaction is out of order and to test if it can be appropriately reordered. CLT and LUN are used to determine when UT can be pruned so that it is not necessary to remember the entire history of a server.

Change Transaction

When a server receives a change transaction T it will be accompanied by a transaction ID (T.LastID). This is the ID of the largest server transaction that the client knew about when it created T. Transaction T can only be valid if it can be reordered relative to any already processed transaction that the client did not know about. The set of possibly conflicting transactions are those in the UT list whose IDs are greater than T.LastID. The IsCommutative test will tell us when reordering is possible. The method for handling change transactions then is as follows.

```
DoChangeTransaction(T)
  { For all transactions U in the list UT
        such that T.LastID>U.ID
    { if not IsCommutative(U,T)
      { discard T
          notify the client that T is rejected
          return
      }
    }
    Apply transaction T to the server data
    T.ID = new transaction ID
    Send acceptance of T to the client
    Send change notifications to all relevant subscribers
    Update CLT and LUN to reflect the fact that T.Client
      knows about transaction T.LastID
    Add T to UT
    Prune from UT any transaction less than or equal to
      LUN
  }
```

The CLT vector tracks the last transaction that each client is known to be aware of. By taking LUN as the min of all of these we can then readily know which transactions can be pruned from LT. This algorithm needs some modification to allow for unresponsive clients to time out so that UT does not become unreasonably large. When a timed out client gets reactivated, it simply resubscribes to the most recent data from the server. The data so obtained will have the last transaction ID in it so that the client can know where it is starting from in the server's transaction history.

Note that the server does not block on any of the confirmations or notifications that it sends. The server proceeds asynchronously whether clients respond immediately or not.

Confirmation of subscription notifications

The second message that the server can receive is a confirmation from a client that a change notification has been received. The purpose of these confirmations is to update CLT, LUN and UT. When such a confirmation is received it has its transaction ID and the client it came from. The CLT entry for the client is updated to the new ID. LUN is recomputed as the min of all CLT and then UT has removed any transactions with IDs less than or equal to LUN.

Note that if the number of subscribers gets large, the computation of LUN may become time consuming. This becomes a simple tradeoff between the size of UT and the time to compute LUN. Any balancing of these tradeoffs will not effect the correctness of the algorithm provided that LUN is always less than or equal to the true min of CLT. If UT is sorted in transaction ID order then having UT grow long due to an out of date LUN is only a space problem, not a speed problem. This is true because the algorithm will only look at the most recent changes that are greater than T.LastID.

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. This additional complexity is one of the reasons that we implemented the XWeb client layer so that user interface software would be shielded from this complexity.

The client layer must concern itself with four types of change requests. 1) Changes that are being constructed by the user interface, but have not yet reached closure. 2) Changes that have been closed by the user interface but not yet sent to the server. 3) A change that has been sent to the server, but not yet confirmed by the server. 4) Change notifications received from the server 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 or FIND messages.

Client layer state information

The client layer maintains the following information.

- The current client data tree including all changes made so far by the user interface. This is the information that the user sees.
- LST - the last known server transaction.
- P - a pending change that has been sent to the server, but not yet confirmed.
- SP - a saved version of P that is usually empty. P is moved to SP if a change is received from the server which the client believes is not commutative with P.
- PCQ - the pending change queue. These are changes that have been closed by the user interface, but not yet sent to the server.
- OC - an open change record that has not yet been closed by the user interface.

One of the complications of this algorithm is the transaction P that has been sent, but not yet confirmed. In essence the management of this transaction is now delegated to the server for acceptance or rejection. However, the client must still consider this transaction when responding to other changes received from the server before P is confirmed.

The heart of the algorithm lies in the management of the transactions in PCQ. If a change arrives from the server or transaction P is rejected, the pending transactions must respond to this. In our algorithm, the only possible repairs are reordering or discarding (with undo) of a pending change. Any discarded changes may also conflict with changes that occurred after it. Thus the discarding and undoing may propagate back through PCQ.

The purpose of OC is to manage changes that the user has made, but are not yet ready to be sent to the server. Consider a dialog box with OK and Cancel buttons. The user may make changes to the information in this box but will not want them sent until OK is pressed. If Cancel is pressed these changes will be discarded and never sent. However, even if the changes are not yet closed by the user, they still must be considered when responding to changes received from the server by subscription. In the algorithms below we will consider OC as a single atomic change that is either accepted or rejected. In practice it might be better to treat OC as a series of smaller changes which can be considered individually. In terms of conflict resolution OC (either atomic or multipart) behaves exactly as if it were the last transaction or set of transactions in PCQ. The only difference is that OC is never sent to the server until it is closed by the user interface. For this reason we will for the most part omit OC from the algorithms below and treat it as the last part of PCQ to simplify the discussion.

Data change from the user interface

Whenever the user changes any part of the client data tree, that change is appended onto OC. One possible optimization is the compression of redundant changes. Suppose that a user changed a person's name to "Olson" and then changed it to "Olsen" before closing the change. OC can be compressed to include only the last value, eliminating the first change as ultimately irrelevant. There are a variety of such compressions such as insertions followed immediately by deletions of the same item. The client layer can handle these before they are ever forwarded from to the server. All user changes are accumulated in OC until closure. Many of our clients, however, close all changes immediately, which renders OC mostly meaningless.

User interface closure of a change

Whenever a change is closed by the user interface, OC is appended onto the end of PCQ and OC is set to empty. If P and SP are both empty, then there are no changes waiting for confirmation from the server and the oldest change in PCQ is removed, placed in P and sent to the server. When P is sent to the server, P.LastID is set to LST so that the server will know the server data context in which P was created.

Purging PCQ

When receiving notifications of new changes or confirmation of the client's own changes there are times when all non-reorderable changes must be purged from the PCQ. Since this algorithm is general to the remaining client events, we present it first. The PurgeTC algorithm accepts a transaction T and a transaction queue TQ. Its purpose is to return a new transaction queue that includes only those transactions that are reorderable with T. In addition, it must also remove any transactions that conflict with any older transactions that are removed. Of course the heart of the algorithm is the IsCommutative test.

```
PurgeTQ( T, TQ)
  { if (TQ is empty)
    {    return empty; }
    else
    {    F = TQ.FirstElement
         if  IsCommutative(T,F)
         {   return F + PurgeTQ(T,TQ.Remainder) }
         else
         {  NTQ = PurgeTQ(F,TQ.Remainder);
            return PurgeTQ(T, NTQ)
         }
    }
  }
```

Subscription change notification

Periodically the server will send a client notification of changes that have been made to subscribed parts of the server's tree. These are generally produced by changes requested by other clients. The key problem is to determine which of the pending changes known to the client are incompatible with this new change. Note that in the vast majority of the cases there either are no pending changes or the changes are not related to each other. When a change C is received the method DoChange is invoked.

```
DoChange(C)
{    Undo transactions in PCQ in last to first order
     if (P is not empty) Undo P
     Apply C to the client data
     Set LST to be C.ID
     if (P is not empty)
     {    if (IsCommutative(C,P))
          {    Apply P to the client data }
          else
          {    SP = P
               PCQ = PurgeTQ(P,PCQ)
               P = empty
          }
     }
     PCQ = PurgeTQ(C, PCQ)
     Apply transactions in PCQ to data in first to
               last order
}
```

The basic approach of this algorithm is to undo all pending changes in P and PCQ, purge any that are not commutative with C and then redo those that are left. In normal use, the size of P and PCQ is small and most commonly empty, therefore there is not much overhead involved in undoing and redoing.

If the new change C and the pending change P are not commutative, then P must be removed. P is special, however, because P has been delegated to the server, but has not yet been confirmed by the server. Because the client and the server need not have identical implementations of IsCommutative, it is possible that the server may determine that P is commutative with C even when the client does not believe it is so. If the client does not believe that C and P are commutitive, it purges P, but saves it in SP so that it can be reinstated if the server actually accepts it.

Server confirmation of a change

When a server confirms a CHANGE request it will either accept or reject it. A change may be rejected either because it is determined to be invalid by the SrvObj that implements that part of the tree or because it conflicts with a change that the client did not know about when the request was originally sent. The change being confirmed is either stored in P or in SP depending on whether the client believes that the change has been invalidated by a previous change notification from the server.

```
if the change C was accepted by the server
{    if (P is empty)
     { DoChange (SP) }
     LST = C.ID;
}
```

If P is empty, it means that it was rejected because the client assumed that a change from the server was in conflict with P. However, the server, whose opinion must hold at all times, has just confirmed P in this case. We find the change stored in SP and reapply it as if it was a change notification just received from the server. This will handle any conflicts with changes that have been entered since P was first sent to the server.

```
if the change C was rejected by the server
{   if (P is not empty)
    {   undo PCQ in last to first order
        undo P
        PCQ = PurgeTQ(P,PCQ)
        Reapply PCQ in first to last order
    }
}
```

If P is empty then the client has already handled the conflict and purged P from any conflicting changes. If P is empty and has been rejected then it and any conflicting changes in PCQ must be removed.

Whether the change C is accepted by the server or not, after the confirmation, the first change in PCQ must be moved to P and sent to the server using LST as its LastID.

Summary of conflict resolution

By allowing 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.

SESSION MANAGEMENT

The collaborative facilities that we have described so far will function regardless of whether participants know about each other or not. Each change notification received from the server carries with it the information about who initiated the change, which clients can use in their awareness presentations. All of our current clients automatically subscribe to any data segment that they are currently viewing. This means that if two people visit the same site at the same time they are automatically collaborating. This is somewhat like Team Rooms[14].

At times such casual connections are not sufficient. If there is an ongoing phone discussion about some data, it would be helpful for all participants to be synchronized to the same view of that data. We handle this via sessions. A particular XWeb session is defined by three things.

- The URL of the data being manipulated
- The URL of the XView that defines the user interface
- The path of the particular interactor or interactor fragment that has the current selected focus

All clients, whether they be visual or audio, operate in terms of a focus which is highlighted for the user. If the client is engaged with a session, then any changes to the focus or any selection of a link to other data or other views must be propagated to all participants in the session.

Sessions are simply XML <session> objects residing on any XWeb server. A data item with a <session> tag is special to XWeb clients. It contains the three times discussed above as the focus of the session. Whenever the XWeb client layer retrieves a session tag from a server, it will automatically.

- Subscribe to the <session> object on the server that it was retrieved from,
- Make the data, XView and focus from the <session> object the new focus of the client, and
- Subscribe to the data referenced by the <session>.

Whenever the user changes the client's focus in any way, it is processed as a change to the <session> object. These changes are forwarded to the server. The client can also receive change notifications via its subscription to the <session> object, which are then used to change the client's focus. Our use of semantic session focus is derived from Smart Telepointers [12]. This type of focus management will work with all interactive modalities including speech. Geometric telepointers, however, are quite limited in their applicability. The implementation of sessions was quite simple given the CHANGE/SUBSCRIBE mechanisms already built into XWeb. Creating a new session simply involves posting a new <session> object on some server. The user interface for session management is easily built using any XWeb client with an appropriate XView for editing the <session> objects.

SUMMARY

The goal of XWeb is to provide an interactive and collaborative architecture that can be accessed from any interactive platform. Interactive clients handle all user interface chores in a manner that is appropriate to the interactive devices that they have available. Interaction with the server is by means of CHANGE requests on the server's data rather than propagation of interactive events. Collaboration is supported by allowing clients to SUBSCRIBE to server data and receive change notifications whenever any other client makes a change. The subscription and change notification are all handled in standard software layers that are independent of information servers or interactive client implementations. We have developed an optimistic serialization algorithm for managing change conflicts and guaranteeing data consistency among clients and servers over time without causing any of them to block while waiting for changes to process. We have also developed a session management system for maintaining shared focus which is simply a special case of the CHANGE/SUBSCRIBE infrastructure. With the clients we have implemented so far we have demonstrated effective collaboration among desktop, speech, and pen-based platforms.

REFERENCES

[1] Bass, L., Kasabach, C., Martin, R., Siewiorek, D., Smailagic, A., Stivoric, J., "The Design of a Wearable Computer", Human Factors in Computing Systems (CHI '97), March 1997.

[2], Begole, J., Struble, C. A., Shaffer, C. A., and Smith, R. B., "Transparent Sharing of Java Applets: A Replicated Approach," Proceedings of the 10th Annual ACM Symposium on User Interface Software and Technology , (1997).

[3] Dewan, P., and Choudhard, R., "Flexible User Interface Coupling in a Collaborative System," Human Factors in Computing Systems, 1991.

[4] Dourish, P., and Bly, S., "Portholes: Supporting Awareness in a Distributed Work Group," Conference Proceedings on Human Factors in Computing Systems , (1992)

[5] Dourish, P. "Using Metalevel Techniques in a Flexible Toolkit for CSCW Applications," ACM Transactions on Computer-Human Interaction 5, 2 (June 1998).

[6] 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.

[7] 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).

[8] Gutwin, C., Roseman, M., and Greenberg, S., "A Usability Study of Awareness Widgets in a ahared Workspace Groupware system," Proceedings of the ACM 1996 Conference on Computer Supported Cooperative Work , 1996.

[9] Neuwirth, C. M., Kaufer, D. S., Chandhok, R and Morris, J. H., "Computer support for distributed collaborative writing: defining parameters of interaction"; Proceedings of the conference on Computer Supported Cooperative Work , 1994.

[10] Olsen, D. R., "Interacting in Chaos", interactions, September 1999.

[11] Pedersen, E. R., McCall, K, Moran, T. P., and Halasz, F. G., "Tivoli: an electronic whiteboard for informal workgroup meetings" Human Factors in Computing Systems (CHI 93) , 1993.

[12] Rodham, K. J., and Olsen, D. R., "Smart Telepointers: Maintaining Telepointer Consistency in the Presence of User Interface Customization; ACM Transactions on Graphic 13, 3 (Jul. 1994)

[13] Roseman, M. and Greenberg, S. "Building Real-Time Groupware with GroupKit, A Groupware Toolkit." ACM Transactions on Computer-Human Interaction, 3, 1 (March 1996).

[14] Roseman, M., and Greenberg, S., "TeamRooms: Network Places for Collaboration," Proceedings of the ACM 1996 Conference on Computer Supported Cooperative Work , (1996).

[15] Wiil, U. K., and Leggett, J. J., "Concurrency Control in Collaborative Hypertext Systems," Hypertext '93, (Nov 1993).