Gone But Not Forgotten: Designing for Disconnection in Synchronous Groupware

Carl Gutwin[1], T. C. Nicholas Graham[2], Chris Wolfe[2], Nelson Wong[1], and Brian de Alwis[1]

Computer Science, University of Saskatchewan
110 Science Place, Saskatoon, SK, S7N 5C9
[carl.gutwin, nelson.wong, brian.de.alwis]@usask.ca

School of Computing, Queen's University
Kingston, ON, K7L 3N6
[graham, wolfe]@cs.queensu.ca

ABSTRACT

Synchronous groupware depends on the assumption that people are fully connected to the others in the group, but there are many situations (network delay, network outage, or explicit departure) where users are *disconnected* for various periods. There is little research dealing with disconnection in synchronous groupware from a user and application perspective; as a result, most current groupware systems do not handle disconnection events well, and several user-level problems occur. To address this limitation, we developed the Disco framework, a model for handling several types of disconnection in synchronous groupware. The framework considers how disconnections are identified, what senders and receivers should do during an absence, and what should be done with accumulated data upon reconnection. We have implemented the framework in three applications that show the feasibility, generality, and functionality of our ideas. Our framework is the first to deal with a full range of disconnection issues for synchronous groupware, and shows how groupware can better support the realities of distributed collaboration.

Author Keywords

Groupware design, disconnection, network connectivity.

ACM Classification Keywords

H.5.3 [Information Interfaces and Presentation]: *CSCW*

General Terms

Design, Human Factors, Reliability

INTRODUCTION

A central assumption of synchronous groupware is that the members of the group are *temporally present* – that is, they are actively observing changes to the shared workspace and noticing new updates as they arrive. Many types of groupware have been successfully built on this model, such as shared editors, instant messaging systems, audio and video conferencing tools, and multi-player games.

These applications work (as groupware) by sending real-time communication messages, awareness events, and model updates. When the synchrony assumption holds, these systems only need to send the current state of the activity – they can assume that other participants will receive the new information soon after it is sent, and that no-one misses anything. However, there are many situations in group work where the synchrony assumption does not hold, and where people have become disconnected from synchronous interaction – because of network delay, network outages, or explicit departure. These episodes can cause several problems:

- interruptions in real-time streams (audio, video, or telepointers) lead to interpretation difficulties;
- network outages often force people into laborious re-connection procedures, and often cause loss of current state information such as an avatar's location;
- temporal absences cause people to miss real-time communication and awareness events, leading to loss of context or confusion about how changes occurred;
- if work continues on both sides of a disconnection, problems can occur when trying to merge different versions of the shared data upon reconnection.

Although researchers have investigated many issues within these areas (e.g., visualizations for change awareness [21], smoothing network interruptions in streaming media [14], and data consistency algorithms [13,15,20]), there is little work on underlying design principles that will allow synchronous groupware to deal comprehensively and appropriately with periods of asynchrony.

To address this shortcoming, we have designed an application-level framework for dealing with disconnection in synchronous groupware. The *Disco* framework is based on the three phases of the asynchronous period – detecting that a group member is disconnected, determining what to do during the absence, and dealing with accumulated data when synchronous interaction resumes. The framework divides the design space using three concepts: the endpoint role (sender or receiver), the receiver's connection state (connected or disconnected), and a set of adaptation mechanisms based on the length of the disconnection or the volume of data accumulated during the absence.

These divisions define categories of adaptive behaviour stating what senders and receivers should do during a disconnection. We have implemented the framework at the application level and have tested it on three different

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CSCW 2010, February 6–10, 2010, Savannah, Georgia, USA.
Copyright 2010 ACM 978-1-60558-795-0/10/02...$10.00.

systems (an instant messaging system, a shared graphical editor, and a real-time game). The example systems show that our framework allows a much larger range of responses to periods of asynchrony than what has been seen in previous groupware systems. The adaptive behavior specified in the framework can help to solve the problems of interpretation and missed information mentioned above.

We make three main contributions. First, we expand on previous ideas about asynchronous periods within synchronous activity, and show that several types of disconnection (interruption, network outage, and explicit departures) can be brought together under a single conceptual umbrella. Second, we identify and categorize both new and previously-studied techniques for dealing with disconnection (including systems, interface, and interactional aspects). Third, we provide an application-level reference implementation that demonstrates the functionality and feasibility of our design, and that shows the range of adaptive behaviours that are possible.

RELATED WORK

The Disco framework is based on previous work in several areas of CSCW, including three reviewed below: fault tolerance, asynchronous groupware, and change awareness.

Fault tolerant groupware

The process of returning a system to an operational state following partial failure is termed *failover* [14]. Returning the system to fully operational state is called *restoral,* while returning the system to an operational but degraded state is called *recovery*. In general, recovery is possible if the failed component of the system does not hold resources crucial to the system's continued operation.

Support for failover has been build into some groupware toolkits, such as Ensemble [22], which provides a fault-tolerant component-based infrastructure, or the YCab system [3], which addresses partial failure in mobile environments through redundancy. These approaches support recovery, but not repair in the case where no service is available to take over a failed function. Corona [10] uses distributed publish-and-subscribe and reliable group multicast communication so that clients can handle temporary failures, but has no provision for re-sending missed messages. The group multicast provides reliable message delivery, but has no failover provision.

Failover mechanisms have also been documented for a small number of groupware applications. Some systems, such as the WebArrow conferencing tool, support partial failover through off-the-shelf technology such as load balancers and redundant database clusters, combined with custom algorithms for attempting to recover from failures in clients or servers [14]. Navarre et al. have shown how user interfaces in airline cockpits can be reconfigured to adapt to hardware failure [16]. Other approaches are taken by games such as Age of Empires 2, which freeze gameplay when one peer becomes unresponsive; after a time threshold, the unresponsive player is removed, and the other players are permitted to continue [1]. In persistent-world games such as World of Warcraft, disconnected players remain in the world for a few minutes after disconnection before state information (such as location) is discarded.

While all of these approaches provide support for restoring a groupware session to at least partial operation following failure, little is done to address the human factors of the reconnection process. For example, WebArrow notifies disconnected participants that reconnection is being attempted, but after the connection is repaired, only limited information (i.e., chat messages, but not voice) are resent.

Handling reconnection is related to the latecomer problem, which deals with providing late entrants the correct session state [4, 12]. Late-entrant solutions archive activity so that it can be sent to newcomers. These approaches are restricted by the need to archive all session events, whereas in failure situations, only changes made since the disconnection need to be archived. Late-entrant solutions also provide no support for clients during a disconnection.

Asynchronous groupware

Canonical asynchronous work is activity where group members never expect to interact at the same time: e.g., emailing document revisions, or asynchronously editing a wiki. Systems to support this type of work can be designed very differently than synchronous groupware – with store-and-forward communication, little concern for network delay, and no need to send real-time awareness information.

One area where researchers have explicitly considered disconnection in the design of asynchronous groupware is in mobile systems, where unreliable networks and disconnections are common (e.g., [5,6,12]). A variety of techniques and architectures have been proposed to maintain data consistency through asynchronous interaction [20], detect and repair conflicts [5], and merge changes from multiple users. Many mobile systems employ simple push-based store-and-forward methods for sending data from a central server to mobile clients (e.g., [18]). Reconnection issues for these systems are often based on merging, difference calculation, conflict detection, and versioning. However, a few application-level strategies have also been proposed. For example, Dix [6] suggests that when data reflects an attribute of the world (e.g., a current temperature) "we may often assume that the most up-to-date information is correct" (p. 178), which is similar to our 'freshest data' strategy described below.

Other types of asynchronous interaction have also been studied, such as 'multi-synchronous' work (where multiple people edit a document simultaneously but independently). Techniques specific to these types of asynchronous work have been proposed – for example, Dewan [5] discusses techniques for providing real-time awareness information during simultaneous but independent editing sessions, in order to try and prevent later merge conflicts.

Some of the closest systems to our interests are those that explicitly support both asynchronous and synchronous interaction. Three types of systems have been seen in previous literature. First, groupware based on spatial metaphors provides a persistent environment that equally supports synchronous and asynchronous work. For example, TeamRooms [8] provides virtual rooms that persist when users leave: synchronous work is supported when multiple users are in a room at the same time, and asynchronous work is supported because people can leave artifacts for others. Changes are tracked through versioning.

Second, purely asynchronous systems (such as document repositories or wikis) can be adapted for occasional synchronous use. For example, the DOORS architecture [19] allows real-time work when two users happen to be logged into the system at the same time. This system recognizes that update requirements are different for synchronous and asynchronous work, and provides incremental updates during synchronous periods, and model updates when an asynchronous user connects. In addition, DOORS has a mechanism to convert the many small updates produced during synchronous work to a larger single update for asynchronous users.

Third, some synchronous systems provide limited support for asynchronous use. Prior work in this area has focused mostly on the latecomer problem (as described above), and on change awareness, reviewed below.

Change Awareness in Asynchronous Systems

At the user level, a major concern for asynchronous groupware is change awareness – that is, mechanisms for informing users about what changes have occurred in a document or a workspace since that user was last in the system. Techniques exist for showing changes both to shared artifacts and documents, and for showing the past actions and locations of user embodiments (the following list is adapted from a review by Tam and Greenberg [21]).

- *Lists of changes*. Command-based comparison tools such as 'diff' show a list of lines that are different, and how one version can be transformed into the other.
- *Parallel displays*. Some systems for comparing text documents show the different versions side by side (e.g., some document comparers show the original, the new version, and the changes, in three columns).
- *Annotations*. Some systems mark changes in margins or overviews, such as change bars and balloons in the document margin (e.g., Microsoft Word), or marks on an attribute-mapped scrollbar (e.g., in Eclipse).
- *Highlights*. Objects that have changed can be indicated with a visual highlight; e.g., many word processors mark changes from different users in different colours.
- *Playback*. Some visual workspaces show changes by allowing replay of the changes as they happened.
- *Overlay of intermediate states*. The progress of a change can be shown by overlaying all intermediate states in a visual workspace.
- *Motion traces* are visual representations of an embodiment's recent past. For example, motion lines can be added to telepointers to show recent movement [9].
- *Location histories* are summary representations of where people have been working. For example, 'heat maps' or colour-coding can show where people have spent the most time working in a document [11].

We now turn to the work that we have done to deal with disconnection in synchronous groupware, beginning with the types of disconnection handled by the framework.

TYPES OF DISCONNECTION

We identify three types of disconnection, differentiated by the intent of the user and by which network endpoints perceive the disconnection as having occurred.

Delay-based Interruptions

Interruptions (also called *jitter*) are short-term gaps in message delivery that are caused by a variety of problems that can occur between the creation and eventual delivery of a message (e.g., congestion in the wide-area network, a low send rate, or a high CPU load). In these situations, messages cannot be delivered and so they pile up, thus losing their intended temporal spacing (e.g., the regularly-spaced updates in a telepointer stream). When the problem clears, the messages are delivered all at once.

Although the underlying network connection is still valid during a delay-based interruption, the delivery pattern is exactly that of a short-term disconnection (i.e., no messages for a period of time, then delivery of all the messages as a group). However, interruptions are only noticed at the receiver, since the network connection is not affected.

Variance in delivery time is unavoidable, but most interruptions are too small in duration to be noticed by users. Therefore, we are primarily interested in gaps that are larger than 50-100ms, depending on the application.

Network Outages

In a network outage, a groupware node loses its network link to another node. Outages can occur for several reasons – network failure, mobile devices moving between zones, laptops going to sleep, unplugged cables, or inadvertent closing of the application – but outages are always unintentional, which differentiates them from departures (see below). Outages can be of varying length (from less than a second to many hours), since re-establishment of the connection depends on the availability of the network.

Outages are only reported back to the application when using connection-oriented protocols such as TCP/IP; although outages can also affect connectionless protocols, no exception will be raised when the outage occurs. However, many groupware applications (even many mobile applications such as IM systems) depend on the delivery guarantees of connection-oriented approaches. In our implementation of Disco, we ensure notification of outages even for connectionless protocols, by adding a connection-oriented 'heartbeat' to every point-to-point channel.

Explicit Departures

In a departure, the user explicitly logs out of the session or quits the application. Departures are different from outages in that they are intentional, and the other nodes in the groupware system receive notification that the user is leaving. Departures are also typically much longer than network outages, since the user is unlikely to log out if their intended absence is short.

Last, we note that these three types of disconnection do not describe occasional packet loss – instead, they deal with complete packet loss for a specified period of time. Intermittent and infrequent packet loss can be viewed as more of a quality of service issue than a disconnection issue, and is therefore out of the scope of our framework.

THE DISCO FRAMEWORK

The Disco framework is organized into three parts, reflecting the three phases of a disconnection: identifying that a disconnection has occurred, adapting system behaviour to deal with the asynchronous state, and handling accumulated messages upon a return to synchronous interaction. We first introduce three basic concepts used to divide the design space: the endpoint role, the receiver's connection state, and the adaptation mechanism.

Endpoint role (sender or receiver)

We model a groupware system as a collection of one-way point-to-point communication channels. The endpoint role indicates which end of the connection we are concerned with – the sender or the receiver. In any disconnection, we consider the receiver to be the absent node, and the sender to be the node that is still acting synchronously. This model can be used equally well with both client-server and peer-to-peer architectures.

Nodes are generally senders and receivers at the same time, and so may be acting in both roles during a disconnection. For example, if a node is disconnected but able to continue working on a local copy of the shared data, it can be both a sender (it wants to send its local changes to others but cannot), and a receiver (it cannot receive others' changes). We allow dual roles to proceed independently.

Connection state (connected or disconnected)

Participant nodes can be in one of two states in terms of their temporal presence: connected (implying that they are interacting synchronously) or disconnected (they are not interacting synchronously from the perspective of a sender).

Transition events indicate when nodes in the groupware system enter a new state. As described above, nodes will not be notified of the disconnection in all situations – in particular, senders are not notified of interruptions.

Adaptation mechanism

Disconnections are inherently variable (e.g., they are unpredictable in duration), so we need a way of changing behaviour to adapt to different situations. The two most obvious variables in a disconnection episode are the length of the absence, and the amount of accumulated data.

Duration of the disconnection

The amount of time that someone has been absent is a common-sense indicator for adapting behaviour for both senders and receivers. This mechanism is based on people's real-world behaviour: for example, a person would summarize a situation differently for a period of ten minutes, ten hours, or ten days. Absence time could be used in several ways (e.g., as a single continuous variable), but we adopt the idea of the logarithmic time scale that has previously been used to describe several aspects of human action (e.g., perception of system feedback).

Our time-scale mechanism is based on a loose 'powers of ten' principle (see example scenarios below). The divisions in the scale depend on the application's temporal granularity of interaction. For example, an IM client is unlikely to worry about millisecond-scale gaps, but this scale might be important for systems that send streaming media. In addition, some categories are not feasible for some disconnections – e.g., it is unlikely that outages can be repaired quickly enough for a sender to make use of a millisecond (or even a tenth-of-a-second) time scale.

Volume of the accumulated data

The number of accumulated events or messages can be a second measure for adapting behaviour. In our framework, senders store messages in different ways based on how many messages have arrived for the receiver. Again, we propose a powers-of-ten scale: for example, fewer than 10 messages, 10-100 messages, and so on. Message volume could be used separately or together with time (e.g., a new behaviour could be started if absence time is greater than 1 minute OR more than 100 messages have been received).

Maintaining immediacy with a rolling rejoin buffer

As the timescale increases, systems may wish to provide rejoining users with the most recent synchronous events (i.e., as if the disconnection had been short). This can provide rejoiners with rich context as they move back into synchronous interaction. Therefore, systems may maintain two separate repositories of information (one for the full duration of the absence, one to provide recent actions).

Framework Phase 1: identifying absence

The first stage of an asynchronous period is the determination that asynchrony is occurring. Each of the causes introduced above is characterized in a different way:

- *Interruption*. Interruptions are noticed at the receiver only, by comparing message timestamps with receipt times. Therefore, dealing with short network interruptions is a receiver-only behaviour (see Figure 1). The receiver only notices jitter after the disconnection is over – so the 'smoothing' state in Figure 1 is transitory.
- *Outages*. Network outages are noticed at both the sender and receiver for connection-oriented protocols (e.g., through the raising of a socket exception). Both roles will use these events to switch to different behaviours.
- *Departure*. Since a departure is an intentional exit, there is notification of the departure through the application.

Phase 2: adaptive behaviour during disconnection

For the sender, adaptive behaviour during the disconnection determines what to do with messages that accumulate for the receiver. The sender must store information that will be useful to the receiver, but not use too much memory for storing it. Based on the powers-of-ten mechanisms, the sender will re-process the stored messages at each time (or volume) boundary, using five general strategies:

- Filtering or elimination of messages;
- Transformation of messages to new representations;
- Joining of individual messages into a group;
- Summarization (joining and transforming to produce a summary representation);
- Sending the current data model instead of updates.

A second issue for the sender concerns how to represent the disconnected user in the interface, and what types of interactions to allow with the disconnected user. In most cases, a disconnection will result in a change to the visual representation of the user in the application interface (e.g., graying-out of the avatar as seen in some online games), and this representation may also change as the disconnection time increases. Disconnection also means that a user will not respond to interaction, and so the sender must determine how to inform other users about this state. For example, attempting to start a voice conversation with a disconnected user might result in a reminder that they are absent, and an option to send them a text message instead.

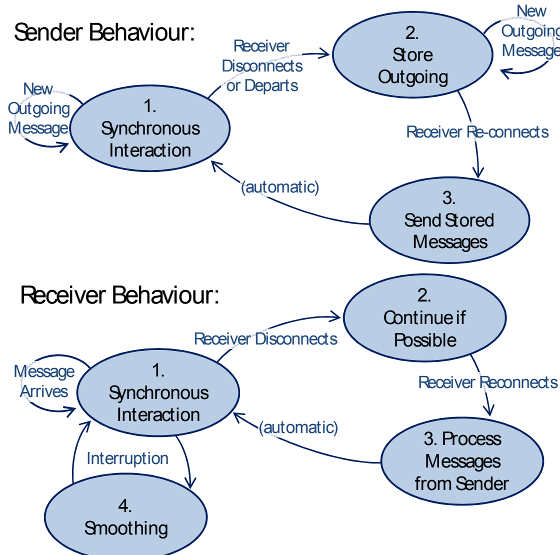

Figure 1. Logical behaviour of sender (top) and receiver (bottom) during different types of disconnection.

The receiver must also consider how to behave during a disconnection (assuming that the application allows it to continue working). In the early stages of the disconnection, the receiver has the option of attempting to hide the asynchrony by simulating the actions of other entities in the distributed system – this is possible for avatars (e.g., games often predict avatar motion [16]), and also possible (although more difficult) for telepointers. For longer disconnections, applications could switch entities over to AI control (i.e., a player's avatar becomes a robot); this would preserve continuity during the absence, but could lead to merge problems on reconnection.

Phase 3: re-establishing synchronous interaction

The sender's job in this phase is relatively straightforward: when the receiver reconnects, the sender sends the accumulated data that has been stored during the disconnection. The receiver, in contrast, has more decisions to make at this stage – when synchrony is restored, the receiver must determine how to use and represent the information about the past that has been received.

The receiver uses different strategies for model updates and awareness messages. Model updates are the simplest case: a full-model update involves replacing the receiver's current model with the new one, and incremental updates can be processed as they are received. Note that the framework does not consider the problem of merging and concurrency control, which can present substantial difficulties; we assume that these are handled by a separate module as has been described by other researchers (e.g., [13,15,20]).

Awareness messages require additional decisions. We identify six actions that can be taken with awareness data:

- *Playback*. The receiver can play out stream-based awareness information (e.g., telepointer motion) if the delay is not too large.
- *Fast playback*. For continuous streams, the receiver can play back the asynchronous events faster than normal, in order to catch up with the current information.
- *Freshest data*. In situations where the receiver only needs the current state (e.g., of a telepointer or avatar), the system can discard all but the most recent update.
- *Smoothing*. For short-term gaps, the receiver can use a jitter buffer to reorganize incoming data; at the cost of extra latency, streaming data is played out smoothly.
- *Short-term traces*. Information about the recent past can be converted to a trace – a visual representation of the past that is presented on the current frame. Examples include motion lines or multiple cursors [9].
- *Long-term traces*. Past information can be added to the summary representations that are used for asynchronous awareness (e.g., activity maps).

EXAMPLE APPLICATIONS

Here we present three application scenarios that illustrate the Disco framework. We look at three different types of application – an instant message system, a shared editor, and a real-time game) that demonstrate different requirements for sender and receiver behaviour, and different sensitivities to time scale (see Figure 2 and video figure). In each example, we indicate how the sender and receiver should behave in various disconnection circumstances – these behaviours are fully built in the applications using our reference architecture (see below).

Application 1: Real-time instant messaging

In this system, synchronous behaviour means sending each key press and caret-move event as they occur. The system

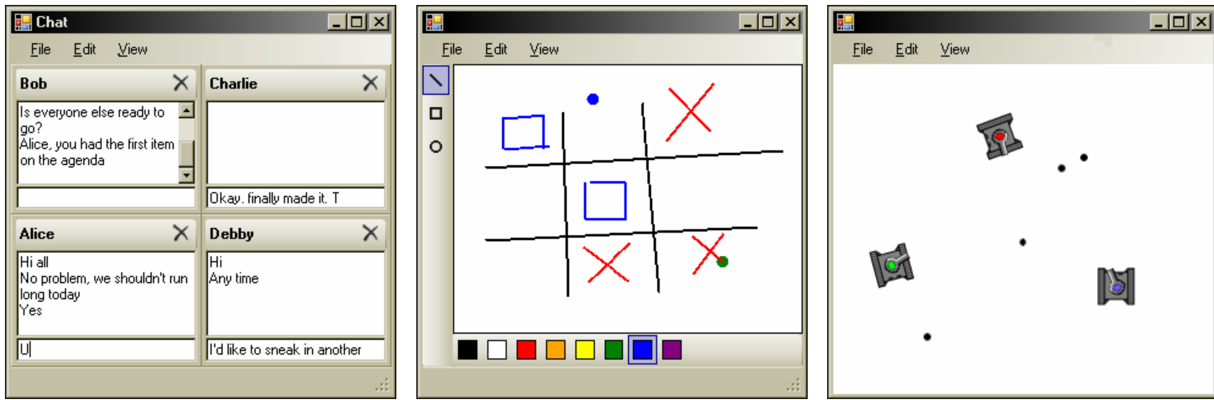

Figure 2. Example applications: Instant messaging (left), drawing editor (middle), multi-player tank game (right).

contains a different composition panel for each person, and a summary transcript that merges the messages once they are completed (not shown). In this example, we use the term ‘IM message’ to refer to the user’s text, and ‘update message’ to refer to the groupware system’s messages.

Sender: on receiver disconnect due to outage

When the sender determines that the receiver has become disconnected, it carries out the different storage behaviours shown in Table 1. The rationale for these behaviours is that we want to send as much information as possible during synchronous interaction; at small gaps (1-10 seconds), ephemeral awareness information (caret movement and individual keypresses) cannot be used by the receiver, so these are pruned from the queue; at absences above 10 seconds, the receiver will not need to see the temporal spacing of separate IM messages, and so we maintain only a single transcript; above 10 minutes, the receiver is unlikely to read all of the intervening IM messages, so we produce a summary of who has been talking; and above one hour, several conversations may have gone on, so we switch to a long-term summary of the discussion.

Table 1. Message storage behaviour for sender

At time:	*The sender switches to this storage behaviour:*
< 1 sec.	Store individual keypresses and caret moves. Store individual session messages.
1 – 10 seconds	Stop storing individual keypress messages. Discard caret-move messages. Join keypress updates into ‘current partial IM message’. Add new keypresses to partial IM message. Store completed individual IM messages. Store individual session messages.
10 sec. – 1 minute	Join existing individual IM messages into transcript. Add keypress updates to current partial IM message. Add new IM messages to existing transcript. Store individual session messages.
1 – 10 minutes	Stop storing session messages. Add new IM messages to existing transcript. If data volume large, compress transcript.
10 min. – 1 hour	Continue data storage as above to maintain full record. Create new medium-term summary representation of existing transcript, every 10 minutes. Maintain most recent message in rolling-rejoin buffer.
1 hour – 1 day	Continue data storage as above to maintain full record. Create new long-term summary representation of existing transcript, every hour.
> 1 day	Continue data storage as above to maintain full record. Stop maintaining separate recent-period record.

Receiver: on network outage

When the receiver disconnects due to a network outage, the only action is to enter an automatic reconnection loop.

Receiver: on reconnection after outage

When the receiver reconnects, it will immediately receive a set of data from the sender. This data has been processed by the sender, but there are still decisions for the receiver about how to use the information. Again, the receiver’s behaviour uses a powers-of-ten scale, as shown in Table 2.

Table 2. Receiver behaviour after network outage ends.

After time:	*The receiver does this:*
<1 sec.	Catch up using fast playback
1 sec. – 10 min.	Display transcript Display current partial message
10 min. - 1 hour	Display medium-term expandable summary Display messages from rolling-rejoin buffer
1 hour – 10 hrs.	Display long-term summary, expandable into transcript Display messages from rolling-rejoin buffer
> 10 hrs.	Display long-term summary, expandable into transcript Display messages from rolling-rejoin buffer

Application 2: Drawing editor with telepointers

This scenario involves a canonical shared drawing editor (Figure 2), where all lines are drawn in real time (i.e., each line segment is sent as a separate message), and each person is represented by a real-time telepointer. The application also shows a translucent ‘heat map’ representation of telepointer activity over the life of the document. As with the previous example, the tables below specify behaviour for the sender during and after the disconnection.

Table 3. Behaviour for sender on network outage.

At time:	*The sender switches to this storage behaviour:*
< 1 sec.	Store individual telepointer moves. Store individual line-segment messages.
1 – 10 seconds	Stop storing individual line-segment messages. Store individual completed-line messages. Join telepointer moves into aggregate position list. Join new line segments into ‘current line’ message.
10 sec. – 1 minute	Stop storing individual telepointer moves. Create heat-map representation for telepointers. Add telepointer moves to heat-map representation.
1 – 10 minutes	Stop storing individual line messages. Create multi-line collection. Add new completed line messages to line collection.
10 min. – 1 hour	Discard multi-line collection (full model update will be sent).

Unlike the chat system, the receiver in the drawing editor has the possibility of predicting during the disconnection period, although only for a short time since telepointer motion is not easy to simulate. Therefore, we determine two behaviours for this period, shown in Table 4.

Table 4. How the receiver behaves during disconnection.

After time:	*The receiver does this:*
< 100ms	Predict other telepointer positions.
>100ms	Mark other telepointers as "not updating." Allow user to continue as if single user.

When the receiver reconnects and receives the accumulated data, the following behaviours will be carried out.

Table 5. Receiver behaviour after network outage ends.

After time:	*The receiver does this:*
< 100ms	Add past telepointer locations to short-term trace Move telepointer to most recent location Play out line-segment updates normally Add telepointer data to heat map display.
100ms – 1 sec.	Add past telepointer locations to short-term trace Fast playout of line-segment updates. Add telepointer data to heat map display.
1 sec. – 10 sec.	Add last 500ms of telepointer positions to trace. Draw completed lines. Draw current partial line. Add telepointer data to heat map display.
10 sec. – 1 min.	Add last 1000ms of telepointer positions to trace. Draw completed lines. Draw current partial line. Add telepointer heat map data to heat map display.
1 min. – 10 min.	Draw completed lines. Draw current partial line. Add telepointer heat map data to heat map display.
> 10 min.	Replace current model and redraw all. Draw current partial line. Add telepointer heat map data to heat map display.

Application 3: Real-time multi-player game

Our third application is a simple 2D multi-player game. Each player controls a tank avatar in an overhead-view battleground, and can shoot at other players (Figure 2). There are a number of differences between the tank game and the other two applications. First, the tank game uses an explicit client-server model: the server maintains the current game state and determines critical events such as hits and misses. Second, since the tank avatars have inertia, the game uses prediction more extensively. Third, avatar position is a server-maintained value, so this location must be preserved for the receiver when it reconnects.

Behaviour for the sender during the disconnection, and for the receiver during and after the disconnection, is shown in the following tables.

Table 6. Sender behaviour during disconnection.

At time:	*The sender (server) switches to this behaviour:*
< 1 sec.	Store individual tank moves. Determine hits and misses as normal.
1 sec. - 10 sec.	Stop storing tank movement messages (positions are held in server's game-state structure) Predict receiver's tank position for other players
> 10 sec.	Freeze receiver's avatar Protect receiver's tank from hits (disconnection shield)

Table 7. Receiver behaviour during disconnection.

After time:	*The receiver (client) does this:*
< 10 sec.	Predict other tank positions.
> 10 sec.	Indicate disconnection to user

Table 8. Receiver behaviour on reconnection.

After time:	*The receiver (client) does this:*
< 1 sec.	Process tank move messages.
> 1 sec.	Replace game state with new full model.

IMPLEMENTATION ARCHITECTURE

Figure 3 shows the conceptual architecture of our implementation of the Disco framework. For simplicity, the architecture shows a single sender, which is responsible for conveying application events to a single receiver. In practice, a given node may act as both sender (of its own events) and receiver (of other nodes' events), and each sender may have multiple receivers.

The architecture and our example applications are developed in C# using the Fiaa toolkit [23] for connectivity and network transport.

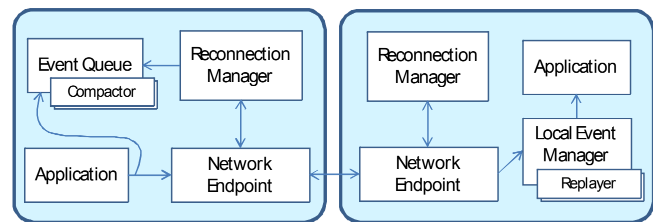

Figure 3. Architecture of Disco implementation.

This architecture has been used to implement the three applications discussed above. The main advantages of the architecture are:

- *Flexibility:* through the use of plug-ins, a wide range of behaviours can be implemented for sender and receiver;
- *Performance:* disconnection handling is asynchronous to the application, so connected participants do not pay a performance penalty for others' disconnection;
- *Reusability*: the architecture is generic (and parameterized via plug-ins), allowing easy re-use of its core code across highly varied applications.

In normal operation, the application generates a sequence of *events* which are processed locally and sent to the receiver. Events may affect the application state (e.g., adding new text into a chat session or adding a new line segment into a drawing), or may provide awareness (e.g., moving a telepointer.) The receiver interprets these events locally, updating its own application state and user interface.

Event Queuing

To prepare for the eventuality of disconnection, the sender queues all outbound events. Events are queued even when the receiver is connected. This means that if an outage occurs, no messages are lost before it is detected. This is particularly an issue with unreliable protocols, where message arrival is not acknowledged.

The *event queue* is a simple FIFO queue with no knowledge of event semantics. Application events are appended to the queue as well as being sent over the network to the receiver. Since the queue may rapidly grow to an unbounded size, it may be periodically *purged* or *compacted.*

Purging. The receiver periodically sends heartbeat messages to the sender, including the sequence number of the last received event. When the sender receives a heartbeat, it knows that messages up to the given sequence number have been received, and can remove those messages from the event queue. Under normal (connected) operation, this bounds the data that the queue must hold.

Compacting. The event queue may periodically employ application-specific techniques to reduce its size. These techniques implement behaviours such as those described in tables 1, 3, and 6. The system may select compaction techniques based on the size of the queue and the age of the data it contains. Different techniques may be used on different parts of the queue. For example, queue entries older than one second might be compacted differently from newer entries. Compaction can include arbitrary transformation of the contents of the event queue, for example, compressing the payload of an event, or replacing a sequence of incremental events with a single state update.

Compactors are implemented as plug-ins, each capable of implementing a single technique. Plug-ins are typically application-specific, and may embody knowledge of the semantics of events.

Disconnection

When a network outage occurs, the sender's and receiver's *reconnection managers* attempt to re-establish the connection. Outages can often be resolved quickly, but if reconnection is not successful, other techniques (e.g., exponential back-off, manual reconnection) must be used. In the case of departure, no reconnection is attempted.

During a disconnection, the event queue continues to maintain (and if necessary compact) the events that have been missed. Automatic event purging is not performed during disconnection, although compactor plug-ins may be able to purge events in some applications.

During disconnection, a *local event manager* may use application-specific prediction techniques to continue sending events to the application. For example, in the tank game, the local event manager generates events that continue play of the tank of a player who is temporarily disconnected. These prediction techniques are encapsulated in *replayer* plug-ins, analogous to the sender's compactors.

In the case of interruption, the local event manager can deal with late-arriving messages by applying application-specific replayers (e.g., to create a telepointer trace).

Reconnection

On reconnection, the sender transmits the contents of the event queue covering the missed events. Once these have been sent, the application is permitted to resume sending new events to the receiver.

At the receiver, the local event manager may transform the incoming event stream as described in tables 2, 4, 5, 7 and 8. For example, a replayer plug-in might speed up the playback of telepointer events, or might apply heat map data to the local heat map. Replayer plug-ins may communicate directly with the application.

Latecomers

The architecture can address issues not directly related to disconnection. For example, a latecomer joining a session can be treated as a receiver which has been disconnected since the beginning of the session. To accommodate latecomers, the event queue must contain enough information to recover the entire session state.

ASSESSING THE DISCO FRAMEWORK

Our three example applications – chat, draw, and tank game – have been implemented using the architecture described above. The applications all operate successfully both as ordinary groupware applications and as disconnection-tolerant systems, which illustrates the feasibility and functionality of the framework. No major problems were encountered during the development or run-time testing of the examples (see video figure).

Since this is the first instance of a comprehensive disconnection framework, our evaluation is primarily concerned with testing the functionality of the implementations and assessing the architecture at a conceptual level, rather than comparing to other existing systems. We recognize that previous work has developed individual techniques that may have functionality not seen in our framework; but we argue that ours is the only approach that deals with the full range of issues for disconnection in synchronous groupware.

The example applications make it clear that the framework is capable of solving many of the user-level problems described at the start of the paper. By explicitly including interruption, outage, and departure in the framework, Disco covers a much wider range of disconnection durations than have been considered before, and explicitly addresses user concerns such as interrupted streams, missed changes, and laborious reconnection and state loss. (We do not, however, deal with merge problems; we assume this can be handled by an existing algorithm, as discussed below). In the next sections, we look more closely at the framework's strengths and potential weaknesses in several areas – overhead, effort, generalizability, genericity, and extensibility.

Runtime Overhead

Our approach clearly imposes overhead on the operation of the sender. The sender must enqueue all events that it sends in case of disconnection, and to conserve space, the queue must be periodically compacted. There are two major features of the approach that mitigate this overhead. First, the operation of the event queue (and its compactors) is

asynchronous to the application itself; i.e., the application never needs to wait while the queue performs its tasks.

Second, the compactor plug-in architecture allows developers complete control over the trade-off between runtime overhead and smoothness of reconnection. Developers control the amount of data to retain, the complexity of the compaction algorithms, and the time scales at which the algorithms are run. If they are unhappy with the mechanism's default behaviour, developers can tune the approach to best suit their application's needs.

In addition to the overhead of storing data, our approach involves overhead in reconnection, which may lengthen the reconnection process. The architecture can help with this problem: since compaction is performed during the disconnection, the event data can be ready to send as soon as a network connection is reestablished. Additionally, if too much time is required to process events on the receiver, the developer can reduce or eliminate unnecessary events.

Programmer Effort

Providing more intuitive reconnection is a new task for groupware developers, increasing the complexity of their applications and the time required to produce them. We believe that to meet usability requirements of modern applications, this work is necessary. Much of the complexity of supporting better reconnection can be embedded into a toolkit. As shown in the architecture of Figure 3, the mechanisms for capturing and storing events, applying compaction algorithms, negotiating reconnection, replaying events, and detecting and responding to jitter are generic and can be re-used from one application to the next.

Developers must provide custom compaction and replayer plug-ins, but even many of these can be reused between applications. For example, tasks such as discarding old telepointer messages or playing back events at double speed can be supported by a library of common plug-ins. To support the interface to these plug-ins, the event queue and local event manager components have generic mechanisms for selecting which plug-in to use, based on the age of the data and the size of the queue.

Generality

A significant strength of our approach is that the handling of disconnection and reconnection is treated orthogonally to the groupware application itself. As we have discussed, this reduces the need for developers to solve the same common problems of detecting disconnection, reestablishing the connection, queuing, compacting and transmitting events, and replaying events on the receiver.

It is an open question whether this orthogonality leads to cases where desired behavior cannot be implemented. To address this question, we have implemented the three applications described above. These applications have different run-time characteristics and different approaches to dealing with reconnection. The approaches illustrate several examples of both application and awareness data that must be transmitted over the network, and several different kinds of compaction algorithms. This diversity lends confidence to the broad applicability of this architecture.

Ease of Creating New Compactors and Replayers

As discussed above, a library of compactors can handle many common cases; however, new compaction techniques will be required as new types of application arise, so it is important to determine the ease of building these plugins.

Compactors are designed around a simple pattern. They have access to the queued events, and can perform whatever application-dependent algorithm they wish upon these events. Similarly, replayers receive a stream of events, and interact with the application to enact them. The complexity of compactors and replayers is therefore in their own algorithms, allowing developers to determine a tradeoff between functionality and ease of development.

Limitations and Future Work

The current version of the framework has several known limitations that we plan to address in future work.

Interaction with Concurrency Control. If a disconnected node is permitted to modify the shared data, then the reconnection process must detect and resolve any conflicts between the operations performed while disconnected and those performed by other session participants. For example if a disconnected player is replaced by an AI player in the tank game, then on reconnection, the player's tank position must be reconciled with the AI tank position. In some applications, therefore, compaction and replay algorithms must take account of the need to resolve conflicts. For example, a compactor may produce model updates only if an algorithm is available to merge models [15]. Since by their nature updates made on a disconnected node are optimistic, replayers may need to perform corrections (e.g., using operational transform [20], or rollbacks [13]). We assume that concurrency control will be handled by a separate module, and we plan to test this integration to ensure that the two systems work as expected.

Streaming Media. We have discussed telepointers as an example of streaming media, but the more traditional sound and video have not been directly addressed in our research. However, both sound and video can be buffered in the event queue using standard techniques. An application may choose to limit the size of the buffer, based either on time or size cap, throwing away older data. On reconnection, the queue contents may be transmitted, and, for example, played back at double speed by the replayer.

Quality of Service. Disconnection may make it impossible for an application to achieve its quality of service goals. For example, feedthrough times are worsened by even short outages. High bandwidth may be required to transmit large state updates following reconnection. Message storage must work together with the sender's QoS system, an issue that we will consider in future work in this area.

Attention-based Disconnection. There are types of temporal absence that are not caused by disconnection – in particular, when a user's attention is elsewhere than the application, they are just as unavailable as they would be in a network outage. In future, we are interested in adding attention-based asynchrony to the framework – many of the mechanisms would remain the same, although identifying the start of an absence is more complicated than identifying network-based disconnection.

CONCLUSION

Disconnections are common in synchronous groupware, and can cause several problems for members of the group; however, few researchers have looked at disconnection issues from the perspective of applications and users. In this paper, we introduced a conceptual framework called *Disco* that deals with disconnection in synchronous groupware. The framework considers how disconnections are identified, what senders and receivers should do during the absent period, and what should be done with accumulated data upon a return to synchronous interaction. We demonstrated the framework by implementing its ideas in three different applications. Although there is future work to be done in order to extend the framework and integrate it with other systems-level functionality, Disco is the first approach to disconnection that deals comprehensively with interruption, outage, and departure as parts of the same phenomenon, and that considers data, interface, and interaction issues. The Disco framework is a further step towards groupware that support synchronous work without ignoring the realities of networked systems.

REFERENCES

1. Bettner, P., and Terrano, M. 1500 archers on a 28.8: Network programming in Age of Empires and beyond. *Proc. GDC 2001* (www.gamasutra.com/view/feature/3094/1500_archers_on_a_288_network_.php).
2. Bezerianos, A., Dragicevic, P., Balakrishnan, R. Mnemonic rendering: An image-based approach for exposing hidden changes in dynamic displays. *Proc. UIST 2006*, 159-168.
3. Buszko, D., Lee, W., and Helal, A. Decentralized ad-hoc groupware API and framework for mobile collaboration. *Proc. Group 2001*, 5-14.
4. Chung, G., Dewan, P., and Rajaram, S. Generic and composable latecomer accommodation service for centralized shared systems. *Proc. EHCI 1998*, 129-148.
5. Dewan, P., and Hegde, R. Semi-Synchronous Conflict Detection and Resolution in Asynchronous Software Development, *Proc. ECSCW 2007*, 159-178.
6. Dix, A. Cooperation without (reliable) communication: Interfaces for mobile applications. *Dist. Sys. Eng.*, 2, 1994, 171-181.
7. Edwards, K. Flexible conflict detection and management in collaborative applications. *Proc. UIST 2007*, 139-148.
8. Greenberg S. and Roseman, M. Using a Room Metaphor to Ease Transitions in Groupware. In *Sharing Expertise: Beyond Knowledge Management*. (M. Ackerman et al, eds). Cambridge, MA, MIT Press, 2003, 203-256.
9. Gutwin, C. and Penner, R. Improving interpretation of remote gestures with telepointer traces. *Proc. CSCW 2002*, 49-57.
10. Hall, R., Mathur, A., Jahanian, F., Prakash, A., and Rassmussen, C. Corona: a communication service for scalable, reliable group collaboration systems. *Proc. CSCW 1996*, 140-149.
11. Hill, W., Hollan, J.., Wroblewski, D., and McCandless, T. Edit wear and read wear. *Proc. CHI 1992*, 3-9.
12. Ionescu, M., and I. Marsic. Latecomer and Crash Recovery Support in Fault Tolerant Groupware. *IEEE Distributed Systems Online*, 2, 7, 2001.
13. Karsenty, A. and Beaudouin-Lafon, M., An algorithm for distributed groupware applications, *Proc. Distributed Computing Systems,* 195-202, 1993.
14. Long, B., Dingel, J., and Graham, T.C.N. Experience applying the SPIN model checker to an industrial telecommunications system. *Proc. ICSE 2008*, 693-702.
15. Munson, J. P. and Dewan, P. A flexible object merging framework. *Proc. CSCW 1994,* 231-242.
16. Navarre, D., Palanque, P., Basnyat, S., Usability Service Continuation through Reconfiguration of Input and Output Devices in Safety Critical Interactive Systems, *Proc. SAFECOMP 2008,* LNCS 5219, 373-386.
17. Pantel, L. and Wolf, L.C., On the suitability of dead reckoning schemes for games, *Proc. NetGames*, 79-84, 2002.
18. Pinelle, D., Dyck, J., and Gutwin, C. Aligning Work Practices and Mobile Technologies: Design for Loosely-Coupled Mobile Groups, *Proc. Mobile HCI 2003*.
19. Preguica, N., Martins, L., Domingos, H., and Duarte, S. Integrating Synchronous and Asynchronous Interactions in Groupware Applications, *Proc. CRIWG'05*, 809-104.
20. C. Sun and C. Ellis. Operational transformation in real-time group editors: issues, algorithms, and achievements. *Proc. CSCW 1998,* 59–68.
21. Tam, J., Greenberg, S. A Framework for Asynchronous Change Awareness in Collaborative Documents and Workspaces. *IJHCS*, 64, 7, 2006, 583-598.
22. Vogel, W. Object Oriented GroupWare using the Ensemble System, *Proc. OOGP*, 1997.
23. Wolfe, C., Graham, T.C.N., Phillips, W.G., and Roy, B., Fiia: User-Centered Development of Adaptive Groupware Systems, *Proc. EICS 2009*, 275-284.