DiscoTech: A Plug-In Toolkit to Improve Handling of Disconnection and Reconnection in Real-Time Groupware

Banani Roy[1], **T.C. Nicholas Graham**[1], **and Carl Gutwin**[2]

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

[2]Department of Computer Science, University of Saskatchewan
110 Science Place, Saskatoon, SK, S7N 5C9
carl.gutwin@usask.ca

ABSTRACT

Disconnection and reconnection are common problems for users of synchronous groupware, but these problems are not easy for developers to handle because of the wide range of scenarios and timeframes that must be considered. We have developed a new toolkit called DiscoTech that helps programmers deal with disconnection. The toolkit is based on five design dimensions that determine how stored information can be manipulated as the system waits for an absent user to rejoin, and how information should be replayed upon reconnection. DiscoTech provides a plug-in architecture to handle a wide variety of behaviours that developers may need during disconnection; these plug-ins range from fully generic tools to customized strategies with full knowledge of the groupware application. We present the design of the DiscoTech toolkit, show examples of its use, and provide evidence that it covers a broad range of situations, imposes minimal performance overhead, and is easy for programmers to learn. DiscoTech handles a wider range of issues than previous toolkits, without requiring undue effort, and provides a practical way to improve the real-world usability of synchronous groupware.

Author Keywords

Real-time groupware, toolkits, disconnection, reconnection.

ACM Classification Keywords

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

INTRODUCTION

Collaborators in synchronous groupware frequently become disconnected from the session (e.g., because of power failure, network outage, network latency, or explicit departure). Disconnections can range in duration from a few seconds (e.g., due to network congestion) to hours or days (e.g., stowing a laptop while flying). Disconnected participants lose track of the ongoing collaboration, and when they rejoin the session, they can have difficulty understanding both the state of the workspace and the current activities of other people. Previous work in CSCW has shown the kinds of problems that disconnection can cause, and has proposed several solutions for handling reconnection [12], late entrance (e.g., [4,14,16,18]), periods of asynchrony [22,23], or network faults [13].

CSCW 2012, February 11–15, 2012, Seattle, Washington.

However, existing techniques handle only a limited number of disconnection and reconnection situations, and there is little support for dealing with different lengths of disconnection, user-level concerns such as summarization, and application-level requirements for changes made to stored data during an absence. As a result, developers must have a deep understanding of the issues underlying disconnection, and must write a great deal of code to handle different scenarios. Recently, a general framework of disconnection was proposed [12], showing that there are several recurring patterns in the way that disconnection and reconnection can be handled, and suggesting that this service would be an ideal candidate for a toolkit approach.

We have developed such a toolkit – called *DiscoTech* – that simplifies developer tasks in handling disconnection and reconnection, based on an analysis of user-level questions that need to be answered on reconnection (what happened during the absence, what is the state of the workspace, and what is happening now). These questions are formalized in a design space with five dimensions that specify how messages can be processed in different disconnection scenarios. The different strategies on the five dimensions are implemented as plug-ins in DiscoTech's component architecture, which allows developers to handle a wider range of disconnection and reconnection requirements than have been seen in any previous toolkit. The architecture allows three types of plug-ins: *fully generic*, which address standard issues such as message format, sampling, and ordering in the data that is queued during a disconnection; *partially generic*, which are parameterized to allow tuned behaviour (e.g., to allow re-ordering with priority for certain message types); and *customized*, which allow application-dependent transformations of data based on the systems specific needs (e.g., changing movement data to a 'heat map' representation). Different plug-ins and parameters provide support for disconnections of different lengths and for different usage scenarios. The goal of DiscoTech is to make it simple for the groupware programmer to handle standard cases, but also to make it possible to handle more complex cases when required.

Our initial evaluations of DiscoTech show that it succeeds in three important design areas. First, we demonstrate the

coverage of the toolkit by showing how policies from previous work (including latecomer support) are handled by DiscoTech. Second, we show that the toolkit is easy for developers to learn and use by reporting on tutorial sessions with four programmers. Third, we provide empirical evidence that DiscoTech adds minimal overhead for connected clients, and that memory use is appropriately constrained when people are disconnected.

Our work makes two main contributions. The design space provides a set of concepts to show the range of solutions for handling disconnection, and identifies a number of specific strategies that can be implemented as plug-ins. The DiscoTech toolkit itself provides an architecture and model for using and composing individual plug-in solutions, and our evaluations show that the toolkit has broad coverage, is easy for developers to use, and performs well. DiscoTech provides a wider range of solutions to the problem of disconnection than exist in any previous toolkit, without causing undue programmer effort or performance penalties. The libraries for DiscoTech are freely available (hci.usask.ca/software/DiscoTech_Sourcecode.zip) and work with any .NET language.

RELATED WORK

Overview of the disconnection problem

Issues of robustness, fault tolerance, network disconnection, and late joiners have been investigated for some time (e.g., [1,3,5,10,15,28]), but the user view of groupware disconnection has not been widely considered. Gutwin et al. [12] recently provided an overview of the problem from this perspective, stating two main factors for handling disconnection: the time scale of the absence, and the phase of the disconnection (specifically identifying a disconnection, determining what to do during the absence, and determining what to do on reconnection.)

Time scale is important in these decisions because users want different kinds of information depending on how long they have been away; therefore, duration of absence can be used as the basis for adapting storage behaviour.

The high-level design presented by Gutwin et al. included this conceptual architecture and several example systems, but did not go into detail about the range of techniques that could be used in handling disconnection and reconnection, or about development infrastructures that could allow the disconnection problem to be dealt with in an efficient and general fashion by groupware programmers.

Techniques for handling aspects of disconnection

Strategies for dealing with disconnection issues have been explored in research on individual systems and toolkits. The range of techniques seen in previous literature was part of the motivation for a plug-in architecture for the DiscoTech toolkit, and many of the techniques described below have been used to inform the design of DiscoTech's specific plug-ins. We do not cover some basic issues, such as identification of disconnection, or reestablishing a network connection (although these are handled bythe toolkit). In addition, we consider fault tolerance for data (e.g., data replication [20]) and consistency maintenance for replicated models (e.g., [26]) to be outside the scope of DiscoTech. These issues are important but can be handled by other modules in the overall groupware architecture. Strategies for handling disconnection include:

- *Persistence*. Maintaining a persistent state for the shared environment allows that state to be sent to users after a disconnection. This strategy is used for place-based groupware (e.g., TeamRooms [25]) and for the virtual worlds used in games (e.g., World of Warcraft).
- *State recovery*. Even in groupware systems that are not place-based, the state of documents or workspaces can be stored at a central location and sent to a reconnecting user (e.g., the YCab framework [3]). In pure state-recovery systems, no event-based information (such as awareness events) is stored. In some systems (e.g., DOORS [23]), event-based information exists when users are connected synchronously, but events are not stored during periods of asynchrony.
- *Store-and-forward*. In this approach, events are queued at a central server and are sent to the receiver upon request (e.g., XTV [5] or Chung et al.'s latecomer service [4]). Unlike state-based systems, state is reconstructed at the receiver using forwarded events.
- *Replay*. On reconnection, queued events from store-and-forward systems are delivered to the receiver and can be replayed – both to reconstruct state, and to provide an understanding of what happened during the absence. For example, DreamObjects [18] allows playback of some or all of the queued events, and a system by Manohar and Prakash provided a 'video-player' interface that allowed playback to be paused or fast-forwarded [19].
- *Type-based selection.* A few systems select what messages to send after a reconnection, based on the message type. For example, WebArrow [17] only sends chat messages after reconnection, not voice.
- *Discarding old state-update events*. Some event-based systems recognize that certain events only update state variables (e.g., a temperature from a sensor), and discard all but the most recent of these events [7].
- *Reordering.* Some games priorize state updates for nearby avatars ahead of those for distant avatars [2], or move important messages (e.g., deaths and spawns) to the front of the queue.
- *Coalescing and chunking.* When messages are batched at the sender, some systems reduce space requirements by aggregating several messages into one (e.g., ReConMUC [1]) or converting incremental updates into a larger single updates (e.g., change multiple draw messages into a single polyline) [23].
- *Compression.* Batched updates can be compressed using libraries such as zlib (e.g., XTV [5], ReConMUC [1], or Chung's latecomer system [4]).
- *Continuation while disconnected*. When a connection is lost, some systems such as DISCIPLE [15] allow a

mobile client to work offline. These systems are similar in approach to groupware that supports both asynchronous and synchronous work (e.g., DOORS [23]), but applies the idea to network disconnection.

- *Content-based filtering*. The volume of messages stored for a particular receiver can in some cases be reduced by selecting only those messages that match certain patterns (e.g., keywords in an IRC client [1]).

Latecomer support

The existing CSCW toolkits that are closest to our work are those that deal with the 'latecomer' problem that has been considered since the earliest groupware systems (e.g., [4,18,19,27]). Toolkit approaches to the latecomer problem can generally be divided into those that provide state-based recovery (e.g., Habanero [16] or Suite [8]), those that provide replay-based recovery (e.g., Chung et al.'s accommodation service [4] or XTV [5]), and those that combine both approaches (e.g., DreamObjects [18]).

Although there are often different issues at play in supporting latecomers versus disconnected users, many of the ideas are similar – and in fact some researchers have noted that absences in the middle of a groupware session can also be thought of as 'latecomers' (e.g., [4]). However, no groupware toolkit has addressed these concerns in a way that provides both flexibility and simplicity for the application programmer. In order to fill this need, we designed the DiscoTech toolkit.

DESIGN DIMENSIONS FOR DISCONNECTION

The overall goal of handling disconnection is to help collaborators rejoin and continue the collaborative work session as smoothly and seamlessly as possible. This involves three main questions from the user's perspective:

- *What happened during the absence?* The returning user needs to 'catch up' on important events and changes that have occurred during the absence.
- *What is the state of the workspace?* The user needs to understand the current state of the collaboration artifacts, both in order to interpret changes and to understand the overall state of the collaboration.
- *What is happening now?* The user needs information about current activities in order to smoothly re-integrate into the group's activity. For example, the user needs the last few messages of a chat conversation in order to restart their participation in the conversation.

These needs involve competing goals: the user wants to get as complete an understanding as possible of the missed events, but reconnection data must be delivered as quickly as possible if the user is to rejoin the collaboration in a timely fashion. This implies, for example, that in many cases the events cannot simply be replayed in their original form (since this would take too much time); in addition, this means that the transmission time of stored messages after reconnection must be minimized.

From these user-level requirements, we identify five dimensions along which groupware messages can be manipulated or altered during or after a disconnection.

1. Message Format: changing the way messages are represented and packaged, primarily to reduce volume.

- *Compression*: using lossless compression techniques (such as Zip or GMC) to reduce message size; some lossy techniques also apply (e.g., re-encoding JPEGs).
- *Aggregation*: repackaging to incorporate several individual events per message. For example, sending multiple telepointer locations per message saves the overhead of identifying each event as a telepointer.
- *Chunking*: adding together several events to create a single larger event. For example, changing a set of character-by-character typing events into a single chat message event; or coalescing several pixel edits into a larger bitmap region event.

2. Sampling: selecting a subset of the stored messages based on a particular property. Sampling techniques imply that some messages are deleted from the queue.

- *Sampling by priority or type*: selecting messages based on an explicit priority designation in the message, or on an implicit designation based on message type. An example of the latter approach is to select messages that involve changes to the model and discard awareness messages (e.g., telepointer movement).
- *Sampling by time*: messages may be selected either by time period (e.g., randomly retain one message per second, to show the distribution of events), or by time cutoff (e.g., retain the last ten seconds of activity).
- *Sampling by count*: similar to time sampling, but using the number of messages rather than time as the governing structure. This approach might retain every Nth message, or just the most recent N messages.

3. Pacing: changing the temporal spacing of a stream of messages, primarily to reduce the time needed for the user to interpret events on reconnection.

- *Uniform speedup*: message timestamps can be uniformly altered to decrease playback time; for example, events could be replayed at double speed.
- *Non-uniform time change*: times can be altered to achieve specific purposes, such as removing gaps in the message stream, or reducing less important sequences.
- *Quick-as-possible playback*: in extreme cases, events can be played with no time delay – that is, messages are processed directly one after another.

4. Ordering: changes to the order of messages in the queue, without removing any messages. As with sampling, decisions are made based on properties of the message data.

- *Priority*: some messages may be considered as more important and can be moved to the front of the queue. Examples include important game events (e.g., hits), or messages that are important for causal processes.
- *Size*: if small messages can be processed faster, it may be useful to collect them at the front of the queue.

- *Recency*: in a standard queue, recent messages would arrive last after reconnection; in situations where messages do not have causal dependencies, however, recent messages could be delivered first. For example, a chat system might show the recent parts of an ongoing conversation, and then gradually fill in earlier parts.

5. Transformation: changing the content or representation of messages, either to reduce size or interpretation time. Transformations are often paired with visualizations that show the information in a way that is different to the original presentation.

- *Averaging*. Some messages (such as position or movement events) can be combined through averaging (e.g., showing the average location of an avatar over one-second intervals, rather than at each movement).
- *Summarization*. Messages can be summarized without changing the basic data representation. For example, text chat could be replaced with a shorter text summary.
- *Representation*. Messages can be processed to create a completely new representation. For example, the text of a chat conversation could be replaced with information about participants' activity level; similarly, telepointer movements could be replaced with a 'heat map' showing overall activity. Transformations can also convert event updates to state structures (e.g., converting movement messages to a 'game frame' containing every avatar location).

PLUG-IN ARCHITECTURE

Guided by this design space, we have created the DiscoTech architecture. DiscoTech's goals are:

- *Ease of Use:* Groupware developers should be able to use the architecture without having to substantially modify it or their application. The architecture should be available as a usable toolkit with a simple API.
- *Expressiveness:* The architecture should support a wide range of disconnection/reconnection behaviour, as captured in the design space above.
- *High Performance:* Applications using the architecture should not pay a major performance penalty, and any penalty should be predictable and quantifiable.

We have realized these goals through a plug-in architecture that has been implemented within the DiscoTech toolkit. DiscoTech is based on an earlier design [12], but is novel in that it deals with disconnection issues using plug-ins. Groupware developers can use DiscoTech as a networking API that provides built-in disconnection handling. Developers stream their application events via the DiscoTech API, and events are propagated via the network to other clients. In case of network outages, events are queued on a server, and sent to the disconnected client upon reconnection. Developers can easily select among available strategies in the DiscoTech API for dealing with disconnection, or can develop custom plug-ins.

Ease of Use through the DiscoTech toolkit

DiscoTech allows groupware developers to easily add disconnection handling to their application. Developers simply route their application's events through the toolkit. Similarly to other middleware solutions, DiscoTech broadcasts the application's events to all other clients. DiscoTech handles disconnection and reconnection events by constantly queuing application events in case of disconnection, and by automatically providing catch-up functionality when the client reconnects.

Figure 1 shows how applications interact with DiscoTech. The current implementation uses a client/server architecture in which the application sends events to DiscoTech via its API. DiscoTech then propagates the events to other clients (via its *Sender* module). Events received from other clients are queued on an event queue, and are forwarded to the application via a call-back. DiscoTech therefore propagates events for applications, providing the network functions usually provided by a middleware such as GT [6] or by raw socket communication. DiscoTech has no knowledge of event semantics; events are simply viewed as serializable objects that can be queued or sent over a network.

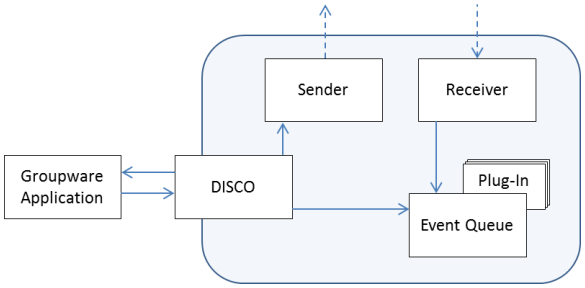

Figure 1: DiscoTech client

DiscoTech provides a server (Figure 2) whose primary function is to broadcast messages from each client to all of the other clients. Incoming events for client *i* are read by a Receiver component, which then enqueues the message on each client's Event Queue. A Sender process for each client waits for new messages to arrive on the Event Queue, and sends them to the client.

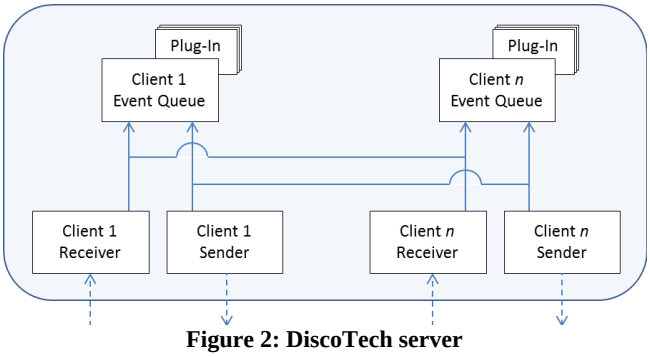

Figure 2: DiscoTech server

If a client disconnects, its sender/receiver pair attempts to reestablish connection. In case of a brief network disruption, the connection might be reestablished immediately. In a longer disconnection (e.g., a computer

being powered down for the weekend), the connection may take hours or days to reestablish. In the meantime, events are queued so that they can be sent to the client upon reconnection. During longer disconnections, the contents of the event queue must be modified in order to reduce space requirements and to avoid the need for lengthy data transmission following reconnection. The groupware developer can configure the techniques used to reduce the space requirements of the server-side event queue. Two techniques are used: (1) the data stored on the event queue is reduced through the approaches discussed in the design space (e.g., compression, aggregation, and sampling) and (2) messages can be compressed using standard techniques as part the network transmission process.

Following reconnection and receipt of data, the client controls how the data is sent to the application. Events can simply be forwarded to the application directly, or can be modified to help the user re-establish context. For example, the client's plug-ins may reorder events so that highest priority events are processed first, or may adjust playback to show missed activity at double speed.

Expressiveness through Plug-ins

Application programmers can use the DiscoTech toolkit as-is, selecting among a set of predefined compaction and replaying behaviours. This allows easy addition of disconnection handling – all that the developer needs to do is route the application's events through DiscoTech.

This approach limits disconnection handling, however, to behaviours that can be programmed without knowledge of the underlying semantics of the events. For example, the following generic behaviours are possible with the toolkit:

- Compression – using standard libraries such as zlib
- Truncation – the last *k* events (or *k* seconds of events)
- Sampling – discarding all but a subset of events
- Accelerated playback of events

However, many interesting behaviours require knowledge of the semantics of the events themselves, such as:

- Aggregation of multiple events to a single event conveying the same information
- Discarding events which are superseded by later events
- Transforming events into a summary representation

DiscoTech uses a plug-in mechanism to open the toolkit,, allowing programming of custom disconnection behaviours. Plug-ins can be attached to either the server or client-side event queues. Plug-ins are periodically invoked by the toolkit (as triggered by timer interrupts or by the queue reaching a threshold size), and may arbitrarily modify the contents of the queue. Plug-ins fall into three categories: *generic*, *partially generic* and *custom*.

Generic Plug-ins: Generic plug-ins, as described above, provide functionality that can be implemented with no knowledge of events' semantics. Events are considered to be generic objects, tagged with a time stamp specifying when the event originated at the source client. The toolkit provides a set of generic plug-ins, providing simple and useful disconnection behaviour. Generic plugins are parameterized by the type of message on which they should operate (e.g., awareness messages only), and by the timeframe in which they should operate.

Partially Generic Plug-ins: The toolkit provides plug-ins that must be parameterized by the application programmer to adapt them to the event types used by the application. For example, a *chunker* combines groups of events into coarser-grained events. This plug-in could transform individual keystroke events from a chat system into a single event, or could coalesce line-segment events into a poly-line for a drawing program. While the aggregation behaviour is generic, the specifics of how a set of events are transformed into a single event are application-specific. Users of this plug-in must parameterize it by providing the implementation of a *Chunk* method that takes a list of events and combines them into a single event. In DiscoTech, this plug-in is implemented as an abstract class, requiring the application programmer to create a concrete subclass that implements the *Chunk* function.

As a second example, the *AggregateToState* partially-generic plug-in allows a sequence of events to be collapsed into a single state update event, possibly involving a significant change in form. As we shall see, *AggregateToState* can be used to convert a sequence of line drawing commands to a single bitmap image, or a sequence of telepointer move events to a matrix representing a heatmap. This plug-in must be customized with the underlying data structure (e.g., the bitmap) and with a method that applies events to the data structure (e.g., drawing the line on the bitmap).

Partially generic plug-ins significantly extend the expressiveness of the toolkit. The cost to the application programmer is that he/she must understand the functionality of the plug-in well enough to provide the code customizing it for the application's event types.

Custom Plug-ins: Some functionality is so dependent on the underlying event types that no generic behaviour is possible. For example, a sequence of moves in a chess game might be summarized using standard chess notation, or a sequence of video frames might be converted through image analysis to a summary of the video's content.

These three plug-in types represent a progression from the simplicity of full genericity to the power of custom components. This allows developers to choose the degree to which they wish to engage with the toolkit – if standard behaviour is acceptable, there is little for developers to learn. If they wish to have sophisticated, custom behaviour, the toolkit makes it possible.

High Performance through Light-Weight Operation

While disconnection handling can greatly improve the usability of groupware applications, it must not unreasonably reduce application performance. DiscoTech is

designed to provide minimal overhead during normal connected state, and to provide developers with hooks to manage resource consumption during disconnected states.

Application events sent from one client to another pass through the DiscoTech server, which broadcasts them to other clients. This message broadcasting architecture is widely used in groupware implementation (e.g., GT [6], or many commercial games). DiscoTech adds the overhead of queuing events on the server rather than sending them directly. When a client is connected, however, events can be removed from the queue as soon as they are added. As shown below, the overhead of the queue is minimal compared to the cost of sending messages over the network.

When a client is disconnected, the server enqueues its messages until the client reconnects. This can lead to unbounded memory use, as there is no limit to the length of disconnection. Server-side plug-ins allow the developer to reduce queue size with a variety of pruning and compaction techniques. Through parameters on these plug-ins, the developer can control how aggressively these algorithms are applied – e.g., how often they should be invoked and how much data should be retained.

Following reconnection, the queued data is sent to the reconnected client. As the event queue grows, so does the bandwidth required to transmit these events. Bandwidth requirements are controlled through two mechanisms. First, as described above, developers can control the amount of data stored on the event queue. Second, the network module can minimize transmission time by delivering the queue as a whole, and by using compression to reduce size.

Later in the paper, we present evaluation results showing that DiscoTech indeed imposes minimal overhead during normal connected operation, and provides hooks for control over resource requirements during disconnection.

DETAILS OF PLUG-IN IMPLEMENTATION

The main novelty of DiscoTech is its plug-in approach to handling disconnection and reconnection. Plug-ins are classes that descend from the abstract base class *DTPlugin*, and are all designed around a similar pattern – plug-ins have access to the queued events, and can perform manipulations on that queue depending on the algorithm built into the plug-in. We have implemented plug-ins to correspond to all of the types detailed in the design space, including reformatters, resamplers, re-orderers, re-pacers, and transformers (see descriptions above).

A plug-in's 'main' method is called *ApplyPlugin.* When invoked by the toolkit, this method applies the plug-in's algorithms to the event queue. Plug-ins have two properties: *EventQueue*, which links to the event queue that is to be manipulated; and *Timeframe*, which specifies the time period for which the plug-in is to operate. In addition, certain plug-ins have additional properties (e.g., the *TimeBasedTruncator* plug-in takes an argument on creation to determine the age at which events will be deleted from the queue). Last, plug-ins all have a method that carries out the desired actions on the given queue (this method is specific to the plug-in type).

Application programmers can use the built-in generic plug-ins (e.g., the *Truncator*) without any customization. For the partially generic plug-ins, such as the *Chunker* plug-in (see above), the programmer needs to redefine the behaviour of the *Chunk()* method. For a custom plug-in, the programmer creates a new class that inherits the abstract base class *DTPlugin*. With custom classes, programmers can arbitrarily change the content of the event queue.

Multiple plug-ins can be applied to the event queue (e.g., to provide different treatment for old versus new events). Each plug-in must therefore be capable of recognizing event types generated by other plug-ins. For example, if chunking introduces a new event type into the event queue to represent the newly 'chunked' data, other plug-ins must be capable of dealing with that event type.

USING THE DISCOTECH TOOLKIT

DiscoTech is implemented in C#, and can be used in any .NET project by including DiscoTech's client-module or server-module libraries with the client or server application. There is a clean separation between the application and DiscoTech's components; the application only interacts with the toolkit's *Disco* component (which acts as a mediator between the application and DiscoTech). At the simplest level of use (i.e., generic plug-ins only), developers need only learn the DiscoTech APIs in order to pass events through the Disco instance; disconnection and reconnection behaviour is handled transparently. Programmers instantiate DiscoTech with details of the server IP address and port, and then define two event handlers to receive command and data messages.

The programmer also needs to specify the set of plug-ins they want to use: e.g., to use the 'HeatmapTransformer' plug-in between 60 seconds and 10 minutes of disconnection, the following lines are used:

```
DTPlugin hm = new HeatmapTransformer(60,600);
Disco.AddPlugin(hm);
```

For sending a message over the network, the programmer uses the following call:

```
Disco.SendMessage(message);
```

where 'message' is a string or an event object. The receiver will get the message via the event handler mentioned above.

EXAMPLES USING DISCOTECH

We show two examples – a chat system and a paper-reviewing application – that demonstrate a range of strategies for handling disconnection, and show that DiscoTech supports both low-cost generic behaviours as well as application-specific behaviour when needed.

Text Chat Application

In this simple chat system (Figure 3), users type messages and press Send to propagate the chat to all users. The application uses two event types: one to send the content of a chat message, and one to send an 'is typing' awareness

message. When a client is disconnected, these messages are queued on the server; for long disconnections, messages are transformed to take up less space according to the following rules:

- Events less than 10 seconds old are preserved as-is.
- Chat messages older than 10 seconds are chunked into a single message (using the partially-generic *Chunker* plug-in)
- Awareness events older than 10 seconds are removed from the queue (using the generic *Truncator* plug-in).
- Chat messages older than 10 minutes are replaced with a 'chat frequency' event that counts the number of chat messages from each user. This uses a custom plug-in requiring knowledge of the application's event types.

On reconnection, messages are replayed at the client according to the following rules:

- 'Chat frequency' events are passed to the client, which highlights users' icons based on number of messages (this is application-specific behaviour).
- Chunked chat messages are passed to the client so that they can be immediately shown in the chat pane. This is default behaviour requiring no plug-in.
- Newer chat messages are replayed at double speed (using the generic *Speedup* plug-in) until the playback catches up. The order and timing of the messages matches their original input, but the timeline is compressed. This allows users to see the rate at which messages were originally typed, but faster.

Figure 3: Chat application

Despite its simplicity, this example illustrates much of the power of the DiscoTech architecture. It is easy to configure the server to limit the amount of information that is stored, an approach that allows a client to be disconnected indefinitely. A custom plug-in is used to create the chat frequency event, but if developers prefer not to create this plug-in, the generic truncation plug-in could be used to discard old chat messages. This shows that developers can choose between simple behaviour with generic plug-ins, or extended behaviour with custom plug-ins.

The reconnection plug-ins are used to help users regain context. Here, highlighting user icons gives a general picture of who was active during the disconnection. Older chat messages are simply posted to the window, while newer messages are played back at the relative rate they were originally typed.

The plug-ins used in this application are parameterized by thresholds such as the age at which awareness messages should be discarded, or the age at which double-speed playback should be used on the client. These parameters are simple properties that are set through the DiscoTech API.

Collaborative Paper Reviewing System

Our second example is a system used by groups of authors to review and comment on a paper draft (Figure 4). The application allows real-time text highlighting, whiteboard-style annotations, and addition of typed comments, and also provides telepointers to convey gestures. The application also shows a summary of comments in a separate window.

Clients send a stream of events specifying the current position of each user's mouse pointer. Clients also send events capturing users' markup operations, such as adding a comment or an annotation.

Although this application is substantially different from the chat system, the programmer handles disconnection in a similar fashion. All editing operations must be preserved on the server event queue. If the disconnection is lengthy (>10 minutes), the partially generic *AggregateToState* plug-in is used to combine all edit events into a single event that conveys the current state of the annotations. The handling of telepointer events depends on the length of the absence:

- The last 60 seconds of telepointer events are stored as-is and replayed with the *Speedup* plug-in;
- For longer disconnections, telepointer events are transformed to a heatmap display (Figure 4) which shades the document according to the number of telepointer events in each area. A custom plugin (*HeatmapTransformer*) converts telepointer events into the heatmap data structure.

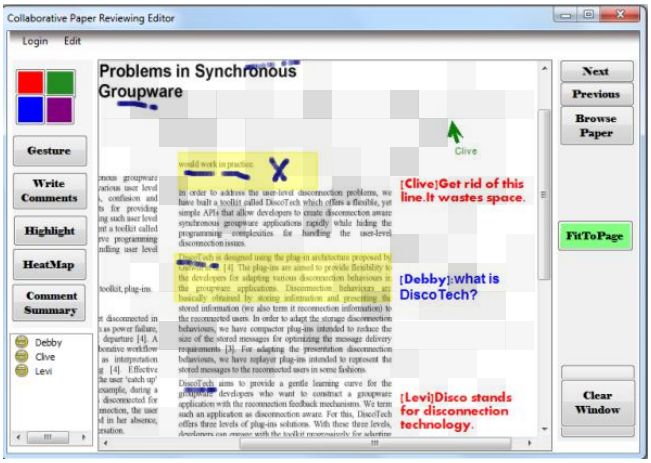

Figure 4: Reviewing application showing heatmap.

These examples show common ways in which disconnection behaviour is handled by DiscoTech. A small number of plug-ins allows a wide range of application behaviour across different application types. Plug-ins provide general solutions to common problems that recur across applications (e.g., truncation of awareness events, accelerated replay, or shifting to a state representation as the length of the disconnection increases).

EVALUATION OF DISCOTECH

We have performed three evaluations of the DiscoTech framework. As discussed earlier, our key claims are that DiscoTech is *expressive,* supporting a wide range of disconnection behaviours; that DiscoTech is *easy to use*, with a plug-in architecture that hides the disconnection handling infrastructure, but still provides customizability; and that DiscoTech is *performant*, demanding only modest overhead in feedthrough times and memory use.

Expressiveness

DiscoTech can reproduce a wide variety of policies and strategies seen in previous work. In the following list, we show how several earlier approaches, including latecomer support, are accomplished using DiscoTech.

- *Maintain model state*. We provide the partially-generic 'aggregate-to-state' plug-in, which can convert model updates to a single state representation. Although not a purely state-based approach, this plug-in allows developers to come arbitrarily close to this behaviour.
- *Store-and-forward*. DiscoTech provides this policy natively through its queues, although not all updates will be forwarded (i.e., if pruned by another plug-in).
- *Event replay*. This is possible with the generic 'Speedup' plug-in. One earlier system provided a 'VCR-style' interface for replay [4]; we do not supply this specific interface, but the Speedup plug-in could easily support a replayer with 'pause' and 'fast-forward' controls.
- *Selection filtering*. DiscoTech supports filtering with generic plugins (which are parameterized by message type); more complex filtering (e.g., by message content [1], and the "freshest data" policy [7]) can be achieved through custom plug-ins.
- *Reordering*. This policy is part of the DiscoTech design framework, but must be implemented as a custom plug-in.
- *Coalescing, chunking, and compression*. We handle these strategies through partially-generic plug-ins such as the 'Chunker' described above. In addition, the full queue is compressed before delivery (upon reconnection).
- *Working while disconnected*. Although DiscoTech's conceptual architecture supports this policy, we currently uses a client-server architecture, so working while offline is not possible; however, a peer-to-peer version of DiscoTech (in development) will support this approach.
- *Latecomer support*. Support for latecomers is built into DiscoTech; we implement this by creating an extra client at system startup that is considered to be always disconnected. When a late entrant joins, we copy the extra client's queue to the new entrant and send it to them. Note that whatever policies are in place for disconnected clients will also affect the late entrant – e.g., they may receive a summary representation of chat rather than all of the text. This is different from schemes that guarantee the sending of a full model, but our approach provides more flexibility for the application programmer – we believe that what is sent to a late entrant should be an application-level decision, which can be achieved with DiscoTech.

Ease of Use

To obtain early feedback on ease of use, we solicited opinions of four software developers not associated with this project. Two developers were undergraduate research interns, and two were graduate students. All were familiar with the concepts of groupware and with development using C#, but none had programmed disconnection handling in a groupware application. Each was asked to carry out a one-hour tutorial which showed how to convert a single-user drawing application into a multi-user version including disconnection handling. We observed each developer as they worked through the tutorial, and then conducted a semi-structured interview after the session.

All four developers were able to complete the tutorial in one hour, correctly adding multi-user functionality to the application and experimenting with a variety of disconnection behaviors by swapping plug-ins. All reported that they found the toolkit's API straightforward, and that DiscoTech would be a suitable candidate for future projects. One participant commented that "The API is really generic… Since there is only one method call and a callback, it is very straightforward." Others stated "I would use DiscoTech because I don't want to reinvent the wheel." and "I would use DiscoTech. It does [it] all for me." While this was far from a comprehensive study, it does provide encouraging feedback that people with strong programming background and familiarity with groupware can learn the DiscoTech concepts quickly, and see value in it for programming disconnection behavior.

Performance

In order to characterize the overhead of using DiscoTech, we tested its effect on an application's feedthrough time and memory requirements. In both cases, we used the canonical groupware example of a shared drawing program.

To test *feedthrough time*, we ran a drawing program under two conditions: with and without disconnection handling enabled. In the "no disconnection handling" condition, DiscoTech's server-side queue was removed, and the server simply forwarded incoming events to all clients, thus simulating a traditional message broadcasting architecture. In both conditions, a "sender" client was automated to inject a new random drawing command every 2,000 ms, over a total of 200 seconds. Between one and four additional "receiver" clients received these drawing events. Feedthrough time was measured as the time from the event being sent by the sender client to the time that the message was received by a receiver client. A local area network was used between the client and server computers with ping times not exceeding 1 ms.

The results of this experiment are shown in Figure 5, which shows that feedthrough time increases with number of clients, but that the impact of DiscoTech's message queuing on feedthrough time is negligible.

To characterize DiscoTech's *overhead when a client is disconnected,* we tested the same drawing program, with

one "sender" client, one connected "receiver" client, and one disconnected "receiver" client. In this scenario, the server must use its event queue to store information allowing the disconnected client to be brought up to date following reconnection.

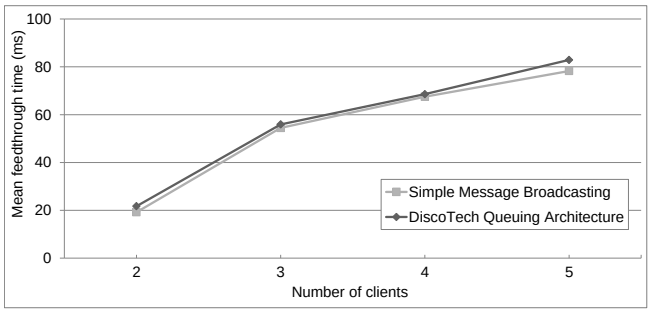

Figure 5: DiscoTech feedthrough time

The application transmitted two forms of events: telepointer movement events, and drawing commands. We measured server-side storage use under two conditions. The "without compaction" condition measured the space required to store the telepointer and drawing messages as-is, with no attempt to compress them to a smaller size. The "with compaction" condition used partially generic plug-ins to reduce storage size: (1) Drawing events: any drawing events occurring over the last ten minutes are represented in the queue as sent by the client; older drawing events are applied to a special bitmap message (using an *AggregateToState* compactor). (2) Telepointer events: the last minute of telepointer events are stored as sent by the client; events between one and ten minutes old are averaged to provide at most one event per five seconds (using an *Averager* compactor), and events older than ten minutes are converted to a heatmap (using *AggregateToState*).

Each condition was run for 12 hours (720 minutes). The results are summarized in Figure 6. In the "without compaction" condition, memory use climbed steadily over time. This is as expected, as events are continuously added to the event queue. Interestingly, even after 12 hours of disconnection, only 15 MB was required to store data from the drawing program. This hints that even with no compaction, small-group shared workspace groupware applications such as a drawing program can sustain lengthy disconnections with acceptable overhead.

In the "with compaction" condition, memory usage reaches a steady state after approximately one hour, at approximately 0.4 MB. This indicates that the aggregation to state strategy works well, where individual events are retained to allow progressive update (e.g., through double-speed replaying), while a full state snapshot allows aggregation of large numbers of events. The limitation of space to 0.4 MB also allows considerably faster reconnection than under the "without compaction" condition, as less data needs to be transmitted to the client.

This shows that with the use of parameterized plug-ins, DiscoTech can have modest and bounded storage requirements when handling disconnection. These results depend very much on the kind of application being tested. Our drawing surface was limited to 500 x 500 pixels; a larger drawing surface would have required more storage space. Similarly, trying to save the full contents of a video chat on a server would require considerably more storage. Nevertheless, for small-group shared-workspace groupware applications, this test indicates that storage requirements are well within the capacities of modern hardware.

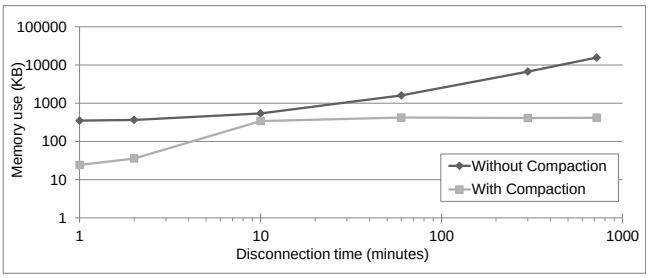

Figure 6: Memory use during disconnection (log-log plot)

CONCLUSIONS AND FUTURE WORK

Handling user-level disconnection and reconnection issues in synchronous groupware is difficult for developers. To simplify this support, we developed DiscoTech, a toolkit that covers a wider range of policies than any previous infrastructure, but that is easy to understand and imposes only minimal performance costs. DiscoTech's main novelty is that it addresses disconnection problems with a plug-in architecture, which provides both simple use in basic cases, as well as full customizability for experts. DiscoTech is based on a new representation of the design space that characterizes the types of behaviour that applications might need to exhibit as a consequence of disconnection. Our initial evaluations of DiscoTech indicate that it can cover a wide range of policies introduced in previous work, that it is easy for developers to learn, and that it does not impose major performance penalties during both connected and disconnected operation.

Our next steps with DiscoTech fall into three main areas. First, we will carry out further work on mechanisms for composing and timing the plugins. Currently the programmer must decide when different plugins will operate, and there is no system support for helping people schedule plug-ins and determine how different plug-ins will work together. In order to work on data that has already been manipulated by another plugin, all data must be tagged with a flag indicating whether the effects of the previous plugin can be reversed. We plan to explore other ways of dealing with this issue, such as with a timeline manager that can schedule and apply the plugins.

Second, we will test peer-to-peer versions of DiscoTech. The current toolkit places most of the responsibility for handling disconnection at the server – particularly during long disconnections. While this is a reasonable division

(since servers are less likely to disappear than clients), we are interested in a more distributed or peer-to-peer version of the architecture that uses strategies from fault-tolerant computing, and that will allow clients to continue work while disconnected (e.g., in multi-player games).

Third, we will carry out additional tests to further evaluate DiscoTech's ease of use and performance in realistic situations. We have made the toolkit freely available (hci.usask.ca/software/DiscoTech_Sourcecode.zip) in order to gather feedback from the CSCW development community, and we plan to build several further groupware applications in order to test DiscoTech's performance with short-term outages (e.g., caused by network jitter), long-term absences, and custom plug-in scenarios. We are also interested to see how frequently the different plug-in types are used in practice, and whether a small number of plug-ins can cover a large majority of use cases for the toolkit.

ACKNOWLEDGEMENTS

This research is supported by NSERC through the Strategic Grants program and the GRAND NCE.

REFERENCES

1. Alves, P. and Ferreira, P., ReConMUC: Adaptable consistency requirements for efficient large-scale multi-user chat. *Proc. CSCW 2011*, 553-562.
2. Brockington, M., Level-Of-Detail AI for a Large Role-Playing Game, *AI Game Programming Wisdom*, S. Rabin ed., Charles River, 2002, 419-435.
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. *EHCI 1998*, 129-147.
5. Chung, G., Jeffay, K., and Abdel-Wahab, H., Dynamic Participation in Computer-based Conferencing System, *Journal of Com. Comm*. 17, 1 (1994), 7-16.
6. de Alwis, B., Gutwin, C. and Greenberg, S., GT/SD: performance and simplicity in a groupware toolkit. *EICS 2009*, 265-274.
7. Dix, A., Cooperation without (reliable) communication: Interfaces for mobile applications. *Dist. Sys. Eng*. 2 (1995), 171-181.
8. Dewan, P. and Choudhary, R. Primitives for Programming Multi-User Interfaces, *UIST 1991*, 69-78.
9. Edwards, W., Flexible conflict detection and management in collaborative applications. *Proc. UIST 2007*, 139-148.
10. Fraga, J., Siqueira, F., Favarim, F. An adaptive fault-tolerant component model. *WORDS 2003*, 179-186.
11. Graham, T.C.N., Kazman, R. and Walmsley, C. Agility Experimentation: Practical Techniques for Resolving Architectural Tradeoffs. *Proc. ICSE 2007*, 519-528.
12. Gutwin, C., Graham, T.C.N., Wolfe, C., Wong, N. and de Alwis, D. Gone but not Forgotten: Designing for Disconnection in Synchronous Groupware. *Proc. CSCW 2010*, 179-188.
13. 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.
14. Illmann, T., Thol, R. and Weber, M. Transparent Latecomer Support for Web-Based Collaborative Learning Environments. *Proc. CSCL 2002*, 540-541.
15. Ionescu, M., Krebs, A. and Marsic, I. Dynamic Content and Offline Collaboration in Synchronous Groupware. *Proc. CTS 2002*
16. Ionescu, M. and Marsic, I. Latecomer and Crash Recovery Support in Fault Tolerant Groupware. *IEEE Distributed Systems Online* 2, 7, 2001.
17. 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.
18. Lukosch, S. Transparent Latecomer Support for Synchronous Groupware. *Proc. CRIWG 2003*, 26-41.
19. Manohar, N.R. and Prakash, A. The Session Capture and Replay Paradigm for Asynchronous Collaboration. *Proc. ECSCW 1995*, 149–164.
20. Narasimhan, P., Moser, L. E. and Melliar-Smith, P. M. Eternal: a component-based framework for transparent fault-tolerant CORBA. *Softw., Pract. and Exper*. 32, 8 (2002), 771-788.
21. Navarre, D., Palanque, P. and Basnyat, S. Usability Service Continuation through Reconfiguration of Input and Output Devices in Safety Critical Interactive Systems. *Proc. SAFECOMP 2008*, 373-386.
22. Pinelle, D., Dyck, J. and Gutwin, C. Aligning Work Practices and Mobile Technologies: Design for Loosely-Coupled Mobile Groups. *Mobile HCI 2003*, 177-192.
23. Preguica, N., Martins, L., Domingos, H. and Duarte, S. Integrating Synchronous and Asynchronous Interactions in Groupware Applications. *CRIWG 2005*, 89-104.
24. Roseman, M. & Greenberg, S. Building Real-time Groupware with GroupKit, a Groupware Toolkit. *ACM ToCHI*, 3,1(1996), 66-106.
25. Roseman, M. and Greenberg, S. TeamRooms: Network Places for Collaboration. *Proc. CSCW 1996*, 325-333.
26. Sun, C. and Ellis, C. Operational transformation in real-time group editors: issues, algorithms, and achievements. *Proc. CSCW 1998*, 59–68.
27. Vogel, J., Mauve, M., Geyer, W., Hilt, V., Kuhmünch, C. A generic late-join service for distributed interactive media. *Proc. ACM Multimedia 2000*, 259-267.
28. Vogel, W. Object Oriented Groupware using the Ensemble System. *Proc. ECSCW OOGP* 1997, 23-26.