It's About Time: Confronting Latency in the Development of Groupware Systems

Cheryl Savery and T.C. Nicholas Graham

School of Computing
Queen's University
Kingston, Canada K7L 3N6

{savery | graham}@cs.queensu.ca

ABSTRACT

The presence of network latency leads to usability problems in distributed groupware applications. Example problems include difficulty synchronizing tightly-coupled collaboration, jarring changes in the user interface following the repair of conflicting operations, and confusion when participants discuss state that appears differently to each of them. Techniques exist that can help mitigate the effects of latency, both in the user interface and the groupware application. However, as these techniques necessitate the manipulation of state over time, the effort required to implement them can be significant. In this paper, we present *timelines*, a programming model allowing the explicit treatment of time in groupware applications. The model has been implemented as part of the *Janus* toolkit.

General Terms

Languages, Human Factors, Design

Author Keywords

Groupware toolkits, temporally-aware groupware

ACM Classification Keywords

H.5.3 Information Interfaces and Presentation: Group and Organization Interfaces—*Computer-supported cooperative work*; D.2.2 Software Engineering: Design Tools and Techniques—*user interfaces*

INTRODUCTION

A fundamental challenge in implementing distributed groupware is overcoming *network latency*, the time required to communicate information over networks. Latency leads to usability problems such as difficulty coordinating tightly-coupled activities [5], jarring corrections when two people perform conflicting actions [11], and confusion when different users' views are poorly synchronized [26].

CSCW 2011, March 19–23, 2011, Hangzhou, China.

There are two broad strategies for mitigating the usability problems resulting from network latency. Problems can be addressed in the user interface, by improving users' awareness of how latency is affecting their interaction; e.g., telepointer trails give temporal context to remote participants' actions [13]. Problems can also be addressed "behind the scenes" through custom consistency maintenance algorithms that reduce the impact of latency for a particular style of groupware (such as first-person shooter games [3] or real-time strategy games [4].) We term groupware applications that use either of these approaches to explicitly recognize time in their design as *temporally-aware*.

Most existing groupware toolkits provide shared state abstractions that attempt to hide the presence of latency. These toolkits provide no mechanism for detecting and managing divergence between the local and remote state, and therefore make it difficult to implement temporally-aware groupware.

In this paper, we present *timelines*, a novel programming model for shared-state distributed groupware. Timelines expose the temporal dimension of shared state, allowing programmers to manipulate past and future state values. Timeline variables can be shared between remote participants in a groupware session. In so doing, they allow programmers to modify the rate at which time flows, and to create divergent timelines for different participants. These provide the necessary constructs for creating temporally-aware groupware. Timelines have been implemented within the *Janus* toolkit, and have been used by the authors and by other developers to create a range of groupware applications.

We illustrate the advantages of timelines by showing how they can be used to implement several classes of temporally-aware groupware. These include:

- Helping participants manage the effects of latency by showing the temporal context for the actions of others;
- Helping participants coordinate their actions by better synchronizing events on participant clients;
- Reducing jarring effects through smooth corrections;
- Reducing confusion by basing application decisions on the affected participants' frame of reference, and
- Reducing loss of context following short-term disconnection from a session through progressive catch-up.

The paper is organized as follows. We first present a novel classification of ways in which groupware can be made temporally-aware. We then discuss how existing programming tools and frameworks treat the notion of time, motivating the need for our *timelines model*. Finally, we present the model, and then discuss its implementation in the *Janus* toolkit.

CONFRONTING LATENCY: MAKING GROUPWARE TEMPORALLY AWARE

Most groupware applications strive to hide from their users that there is a time delay between performing an action and having others see it. These time delays do exist, however, and necessarily lead to temporary divergence of state between the clients of different users. Examples range from multiplayer games, where other players' positions may be incorrectly reported [1], to shared editors such as Google Docs, where users' edits take time to propagate through the network. Attempting to hide latency can negatively affect the usability of the application, for example leading players to shoot at the wrong location in a game, or leading to confusion when discussing recent edits in a text document.

Another approach is to confront latency in the programming of groupware applications, either through novel ways of presenting information that help users compensate for the negative effects of latency, or through algorithms that reduce the negative consequences of temporal divergence at the application level.

In this section, we present five broad ways in which confronting latency can help application usability. Existing programming tools provide no support for implementing such time-sensitive semantics, which may discourage some programmers from using them.

Giving temporal context to actions of other users

When interacting with others at a distance, it can be difficult to maintain the context of their interactions. This is particularly true when multiple collaboration modalities are used, and are poorly synchronized due to network latency.

For example, if a collaborator is pointing using a telepointer while talking over voice over IP, her voice may not be synchronized to the pointing gesture, leading to confusion. This confusion can be helped with *telepointer trails* (figure 1), where not just the current telepointer position is shown, but also its recent positions. As Gutwin et al. have shown, these telepointer trails can help remote users maintain context [13].

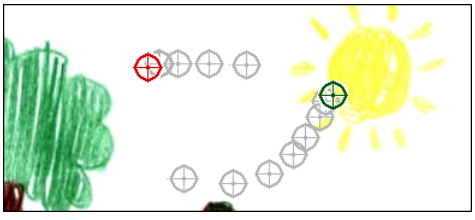


Figure 1. Telepointer trails give temporal context to the actions of remote participants

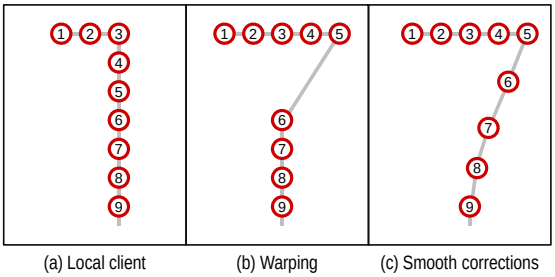


Figure 2. Smoothing corrections can reduce jarring changes to the user interface. Given avatar movement on a local client (a), corrections on a remote host can lead to large warps (b), or can be smoothed (c).

The technical challenge in programming such mechanisms that give temporal context to the actions of others is that it is insufficient to maintain just the current system state. Past state must be retained as well, and mechanisms must be created allowing the application to efficiently access earlier states.

Improving coordination

Collaborators can better coordinate their activities if they see the same actions at the same time. Various algorithms have been developed that manipulate time to improve the consistency of experience of different groupware participants. Notable among these is local lag [18], which delays the application of local inputs to allow them time to propagate around the network.

Programming local lag requires a mechanism for delaying inputs for the appropriate amount of time and a means for estimating message delivery time between the different nodes. It also requires a policy for handling messages that take longer than the lag constant to arrive.

Smooth corrections

When participants in groupware sessions perform conflicting actions, the presence of network latency means that there may be a noticeable delay before the conflict is detected. Conflict can arise when two people perform competing actions (e.g., two people making different edits to the same text at the same time), or as a consequence of errors in predictive consistency maintenance algorithms (e.g., dead reckoning may incorrectly cause a game player's view of another player's position to diverge from the real position [1].)

When conflicts are detected, they must be resolved so that all participants' views return to a consistent state. The simplest (and most frequently used) solution to this is to compute what that consistent state should be (e.g., through operational transform [24]), and to set that state on all participants' clients. The problem with this approach is that the user may be presented with jarring changes to her display with no obvious explanation of why they have occurred.

An alternative solution is to allow the transition to a consistent state to be carried out progressively, allowing the user to see what changes are being carried out [23]. For example, in the online game World of Warcraft, when the local repre-

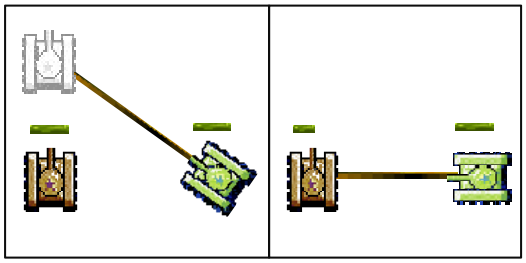
sentation of another player's position is corrected, the other player's avatar runs quickly to the correct position, rather than simply warping. This takes longer to return the client view to a consistent state, but is less jarring. This idea is shown in figure 2: figure 2(a) shows the actual path taken by an avatar. Figure 2(b) shows how the path would appear to a remote participant, assuming that dead reckoning is used to predict the avatar's location. Note how the avatar "overshoots", then jarringly warps to the correct location (at position 6) when the next positional update is received. Finally, figure 2(c) shows the path when using smooth corrections: at time 5, the error is detected, and the avatar moves at increased speed until it attains the correct location at time 9. Algorithms such as this require controlled deviation in the experience of different users, as implemented by divergent states and differing rate of time flow.

Application decisions based on other participants' frame of reference

A core problem in the development of distributed groupware is that no one node can have complete knowledge of the state of any other node. When application-level decisions are required, the state of other nodes must either be guessed (possibly leading to error), or complex (and slow) protocols must be used to ensure their states are synchronized.

A key example of this problem is in multiplayer shooter games, where one player attempts to aim and shoot at another. The shooting player aims at the target player's avatar as represented on his own computer. The target player may, however, have moved in the meantime, and that movement may not yet have propagated to the shooter's client. The decision as to whether the shooter has hit the target therefore requires knowledge of both the shooter and target client states. This problem is illustrated in figure 3(a), where a shooter needs to understand that the position of the target's avatar is incorrect, and guess where it is on the target player's client.

The Half-Life series of games bases hit decisions on the shooter's viewpoint, not the target's [3]. A server determines whether the shooter was correctly aiming at the target at the time of shooting, based on the information available to the shooter at that time. This allows direct aiming, as shown in figure 3(b).



(a) Leading target (b) Aiming at target

Figure 3. a Shooting often requires players to "lead", guessing where the target actually is (a). The Half-Life algorithm allows players to shoot at the target where they see it (b).

Implementing this Half-Life algorithm is complicated by the server's needs to be able to reconstruct the state of the shooter's client at the time the shot was made.

Allow catch-up following short-term disconnections

Short-term network congestion is frequent, leading to brief periods where a groupware participant sees no updates from other participants. The usual solution to these short-term disconnections is to simply apply all remote updates once they arrive (after the congestion clears). Similarly to the problem of abrupt corrections discussed above, this may lead to jarring changes in the user interface, and a loss of context.

For example, in a drawing application, a user would lose the timing and ordering of the drawing operations performed by other participants during the disconnection. A better approach might be to play the missed operations at double speed, preserving order and timing [15]. Such algorithms require the program to buffer incoming messages, and manipulate the flow of time as they are replayed.

Summing up: temporally aware groupware

Together, these examples show how groupware applications can be improved by confronting latency. Some of our examples enhance participants' user interfaces (temporal context, smooth updates), while others use time behind the scenes to improve participants' quality of experience (improved coordination, basing application decisions on other participants' frame of reference.)

These examples break the paradigm of distributed model-view-controller which underlies many groupware development tools (e.g., GT/SD [7], Clock [10] and Fiia [25]), because the distributed models are explicitly allowed to diverge (for disconnection recovery or smooth corrections), because actions may be deferred to the future (for consistency maintenance algorithms such as used in Half-Life), and because access to past or even future states may be required (for basing application decisions on others' frame of reference.)

Programming support that allows the manipulation of time is required. We now review existing programming models that explicitly recognize time, following which we introduce our own timelines model and its implementation in the *Janus* toolkit.

CURRENT APPROACHES FOR MANAGING TIME

The preceding examples lead us to a list of requirements for programming with time:

1. Data must have a temporal dimension, allowing a variable's values to be accessible at any point in the past, present or future;
2. Data from multiple points in time must be accessible at the same time; and
3. A mechanism must exist to control interpolation and extrapolation between/from values at different times.

Existing groupware toolkits satisfy none of these temporal requirements, but instead attempt to preserve the illusion that

shared state is the same on all clients [7, 10, 22, 25]. Most toolkits support the development of groupware through abstractions supporting shared state and mechanisms for distributed message passing. The shared state abstractions provide no means for determining in what ways one participant's view differs from another, or for accessing past or future values of the state.

We therefore look to other programming environments where time has been explicitly incorporated into the language. Dataflow programming languages (such as Lucid [2]) represent variables and expressions as an infinite series of data objects as opposed to single values. In dataflow languages, variables move sequentially from one state to the next; however, there is no mechanism for accessing an arbitrary state in the past or future. Constraint imperative languages extend data flow languages to express temporal constraints in user interfaces [9], but again do not permit manipulation of past or future states.

The field of animation has a long history of managing variables that evolve over time. Myers et al. have shown how constraints can be used to create animated interfaces [20]. However this work provides no notion of attributes existing as a continuous series of values and it does not support extrapolation beyond the ending values.

A variety of commercial toolkits are available which manage some notion of time. Quicktime (http://bit.ly/cH6hVk) provides extensive support for time based media. Toolkits such as Adobe Flash (www.adobe.com), Core Animation (http://bit.ly/a3B1z3) and Windows Presentation Foundation (WPF) (http://bit.ly/kFqqE) provide explicit access to time to help create animations. They allow attributes of an object (such as position or colour) to be set at two points in time. It is then possible to access interpolated values at any point in time between these start and end points. However, the programmer is limited to accessing data from one point in the animation at a time, and there is no notion of sharing these animations between participants connected by a network.

Spatio-temporal databases [21] capture spatial and temporal aspects of data and deal with the position and/or geometry of objects changing over time. Spatio-temporal databases support queries about time, temporal properties, and temporal relationships allowing data to be accessible at any point in time. As well, data from multiple points in time may be accessed within the same query.

Process historians such as OSISoft's PI System (osisoft.com) and AspenTech's InfoPlus.21 (aspentech.com) are used in the process control industry to store time series data and events. The APIs for these systems implement many of the principals required by temporally-aware groupware including the ability to set and get values for any arbitrary time and automatically interpolate values between time intervals. Data can be accessed using either absolute or relative time.

These tools and programming languages introduce a variety of concepts for manipulating data which changes over time. Our work extends the temporal components found in these environments by applying them to shared data in a groupware toolkit. Specifically, our *timelines model* combines the ability of Flash and WPF to index variables by time, and the ability of spatio-temporal databases and process historians to set and query data at arbitrary times in the past and future. In the following section, we will demonstrate how timelines help programmers add temporal-awareness to their groupware applications, addressing usability problems of latency.

EMBRACING TIME: THE TIMELINES MODEL

The *timelines model* helps programmers to create temporally-aware groupware. Rather than hiding the consequences of network latency, timelines allow states to be indexed by time, and allow temporally-based control of how remote updates are applied locally. These features of the model simplify the programming of temporally-aware groupware applications as characterized in the last section.

In the timelines model, shared state variables are not represented as instantaneous values in time, but as values indexable by time. Variables represent all the values they have held in the past and all values they will hold in the future. As we shall see, this is a simple yet powerful way for programming temporally-aware groupware. For example, the telepointer trails of figure 1 can be implemented simply by accessing (and drawing) the previous positions of the telepointer from its "position" timeline.

Timelines are composed of:

- Get/set operations that access the timeline's value at a given time;
- Interpolation and extrapolation functions that estimate values for times when no value is known;
- A remote update function that processes timeline updates from remote peers.

Figure 4 shows how these elements are combined. In the timeline, v_1, v_2 and v_3 represent values for times t_1, t_2 and

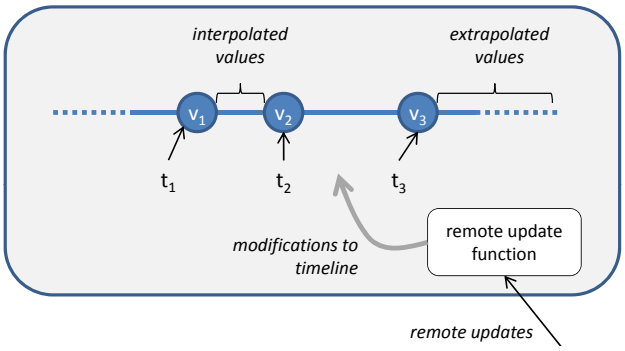


Figure 4. Elements of the timelines model: a timeline includes a set of past and future known values (v_1, v_2, ...), along with the times at which they hold (t_1, t_2, ...). Values between these times are computed using an interpolation function, and values after these times are estimated using an extrapoloation function. A remote update function determines how updates received from peers are inserted into the timeline.

t_3 respectively. These are *known values*, meaning that they have been explicitly set in the timeline.

While timelines may contain objects of arbitrary complexity, we introduce the model by examining a timeline for a simple integer value. Consider $health$, an integer timeline representing the health points of an avatar in a multiplayer game. Storing values in the timeline is as simple as assigning those values at a specific time. Thus,

```
health(-100) = 200;
```

assigns the known value of 200 to the integer timeline at 100 ms in the past, and

```
health(0) = 230;
```

assigns the known value of 230 to the timeline at the current time. (Time references are expressed in milliseconds, may be positive or negative, and are relative to the current time.)

When the value for a given time is not known, it is interpolated or extrapolated from its neighbouring known values. The value of $health(-50)$ would be derived as 215 using linear interpolation between the two known values of 200 and 230, and the value of $health(100)$ would be extrapolated to be 260. Thus, although our timeline currently contains only two known values, any value from the past or future can be determined through interpolation or extrapolation. As more known values are added to the timeline, the interpolated and extrapolated values may change.

As we shall see, in our *Janus* toolkit, all timelines derive from a *Timeline* base class. The constructor for a timeline object requires one parameter, the string name of the timeline. If *IntegerTimeline* is an implementation of a timeline containing integer values, then

```
IntegerTimeLine health
   = new IntegerTimeLine("player1Health");
```

creates a local instance of the "player1Health" timeline.

If two clients both create instances of the "player1Health" timeline, *Janus* synchronizes both instances. The timeline's remote update function specifies how updates arriving over the network are to be applied. Whenever a timeline is updated by assigning a new value at a given time, an update message is sent to any synchronized instances on other clients. By default, the remote update function adds the incoming update into the timeline at the correct location. As we shall see, this behaviour can be overridden in useful ways.

To illustrate the model, we show how it can be used to implement the examples of temporally-aware groupware that we have introduced earlier.

Providing temporal context: Telepointer Trails

As shown in figure 1, telepointer trails can help retain context when another participant is using her pointer to gesture. Assume we use a timeline *tp* to represent the position of another participant's telepointer, and assume that a *DrawTelepointer* operation draws a telepointer at a given location in a given colour. Then a code sketch for drawing a telepointer with its trails is simply:

```
for( int time = -300; time < 0; time += 50 )
{
    DrawTelepointer( tp(time), gray );
}
DrawTelepointer( tp(0), blue );
```

Here, the "trail" is represented as a set of gray telepointer symbols, placed at past positions. These positions are sampled over the last 300 ms at 50 ms increments. The current telepointer is drawn in blue. Times are relative to the current time, allowing this code to work whenever it is executed.

Improving coordination: Local Lag

Another example is the local lag algorithm, used to increase consistency in participants' experience. As described earlier, a constant delay (e.g., 100 ms) is applied to all input operations. When the operation is applied on remote sites, the delay must compensate for the time to deliver the operation so that the operation is applied at the same time on all clients. Local lag algorithms must have some means of dealing with messages which take longer than the constant delay to arrive, and therefore cannot be applied on time. In the timelines model, this behaviour is easily specified.

Consider that "Alice" and "Bob" control avatars in a virtual world. Alice's avatar's position is represented in a timeline of positions called *alicePos*. Alice's client uses local lag to set positions in response to her movement commands. Assuming that the lag constant is *LAG*, and Alice moves to a new position $\langle x, y \rangle$, then the operation on Alice's client to process the movement is simply:

```
alicePos(LAG) = (x, y);
```

For example, if $LAG = 100$, then Alice's position has been set to $\langle x, y \rangle$ 100 ms in the future.

On both Bob and Alice's clients, Alice's position is drawn as:

```
DrawAvatar( alicePos(0) );
```

That is, the avatar is drawn at its position at the current time. (A typical approach would be to put this *DrawAvatar* command into a loop executing at 60 frames per second.)

This very simple code has a range of interesting effects, as illustrated by figure 5. On Bob's client, messages indicating Alice's movements are automatically inserted into the timeline when they arrive. If the real time of the movement is t (i.e., t is the time that Alice set the position plus *LAG*), then the new position $\langle x, y \rangle$ is inserted at time t.

If the message took less than *LAG* ms to arrive (figure 5(a); hopefully the normal case), then on Bob's remote client, the new position appears in the future, allowing the present position to be interpolated (using previously recorded positions). Therefore, the local lag functionality supports smooth movements on remote clients without annoying corrections.

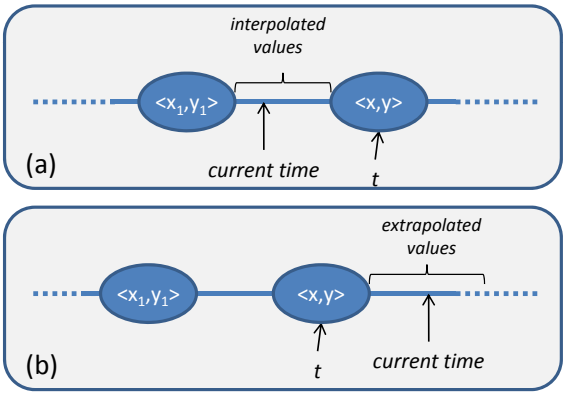


Figure 5. Local lag in the timelines model. Messages may arrive (a) as future states, enabling the use of interpolation, or (b) as past states, requiring the use of extrapolation.

Alternatively, if the message took more than *LAG* ms to arrive (figure 5(b)), then the positional update appears in the past, and current positions are extrapolated from this (and possibly other) past values.

This simple example illustrates the power of the timelines model. Very little code is required to implement local lag, since the problems of synchronizing lag between different clients and of dealing with messages which take longer than the lag constant to arrive are handled automatically.

The model offers significant flexibility. By overriding the default interpolation/extrapolation functions, a wide range of behaviour can be specified. And as we shall see in the next two examples, custom handling of remote messages allows clients to diverge their timelines from that of other clients, allowing easy expression of complex temporal behaviour.

Smooth Corrections: World of Warcraft position fixups

Smooth positional corrections [23] can be easily specified using timelines. The simplest way of handling position corrections is to immediately move (or "warp") the avatar to the new position. This can have a jarring effect as an avatar suddenly jumps across the screen. An alternative solution, is to have the avatar move quickly to the new position.

We implement smooth corrections by overriding the position timeline's default remote update function. As we have seen, remote updates are normally handled by adding the incoming value to the timeline at the appropriate time. This approach replicates the timeline on all clients that have access to it. For smooth corrections, however, we purposely wish the timelines to diverge – when a local client receives a correction, the local timeline is modified to gradually move towards a consistent state.

Figure 6 shows this approach. Assume the avatar of a remote player has been extrapolated, based on earlier known positions, to be at a location *currentPos* at the current time. The client receives a message indicating that the avatar was actually at a position *fixupPos* at some earlier time *fixupTime*. Extrapolating from this position and time, we deduce that the avatar should in fact currently be at position *correctCurrentPos*.

The simple solution to this error would be to update the current position to *correctCurrentPos*, and this is in fact what the timelines model does by default: when a value is set in the past, all future known values are considered to be invalidated and are removed from the timeline. The current position would therefore be extrapolated from the fixup position.

Instead, we decide to move the avatar to the correct position over the next 2,000 ms. The avatar's target position (*targetPos*) is determined by extrapolating 2,000 ms into the future from the *correctCurrentPos*. The avatar will move quickly over the next two seconds to that position.

This is accomplished as follows using the timelines model. First, we need to save the current position before the update is applied, i.e., the avatar position at time 0. This is necessary because inserting a new past value into the timeline will alter the current position.

```
currentPos = avatarPos(0);
```

Next, we place the fixup position (the position contained in the latest update message) into the timeline at the correct time. This allows us to estimate a target position we want to arrive at 2,000 ms in the future:

```
avatarPos(fixupTime) = fixupPos;
targetPos = avatarPos(2000);
```

Finally, we insert both the current position, and the target position into the timeline. (The assignment of the target position is necessary as its extrapolated value changes following the update of the current position.)

```
avatarPos(0) = currentPos;
avatarPos(2000) = targetPos;
```

This example illustrates how one form of smooth correction can easily be implemented via timelines. The key concept is that by having the ability to modify a timeline, programmers are able to explicitly control the divergence of timelines be-

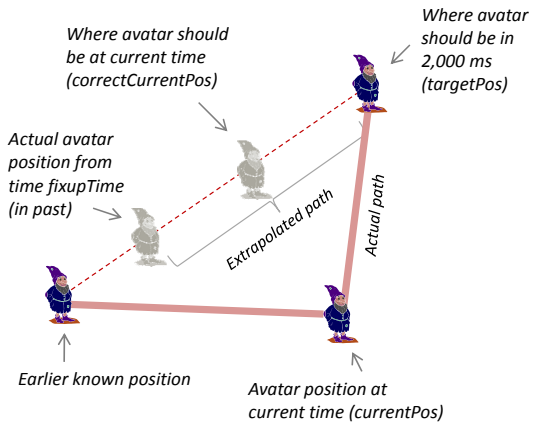


Figure 6. A timelines implementation of smooth corrections in the style of World of Warcraft positional updates.

tween different clients. The remote client, where the avatar is being controlled, sees neither the error nor the correction.

Application decisions based on other participants' frame of reference: Half-Life Targeting

The examples of temporally-aware groupware that we have explored so far directly modify the user interface, and are therefore prominently visible to the user. Another use of temporal awareness is to improve how application-level decisions are made, by allowing the application to consider that different participants' clients have different states. As we have discussed, one example of this is the aiming mechanism in the Half-Life series of games, where a server arbitrates hit decisions based on the state of the shooter's client at the time the shot was made. This requires the server to be capable of unwinding time to determine the position of the target avatar on the shooter's client.

In Half-Life, this problem is made more difficult through the use of lag compensation, which allows each player to sees his own avatar in real-time and a lagged version of all remote players [3]. This "remote lag" algorithm applies a constant lag to the actions of other players, removing the effects of variance in packet delivery time, and allowing the use of interpolation rather than error-prone dead reckoning when displaying the positions of enemy players.

Figure 7 shows how timelines are used to solve this targeting problem. Two clients (one for the shooter and one for the target) and a server share timelines representing the current and past states of the shooter and the target. When players perform input actions, the appropriate timelines are updated at time 0. To implement remote lag, remote players are drawn at time $-$*REMOTE_LAG*, ensuring that they are rendered a constant time in the past. Timeline updates are automatically propagated between the server and clients. Assuming a message takes L ms to travel from the shooter client to the server, the hit decision is based on the state of the shooter at time $-L$ (when the message was sent) versus the state of the target at time $-L -$ *REMOTE_LAG* (where the shooter believed the target was at time $-L$).

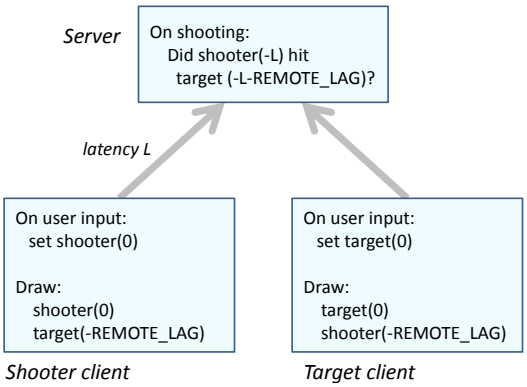


Figure 7. A timelines approach to implementing Half-Life's hit determination code

More precisely, two timelines are used: *shooterTimeline* and *targetTimeline*. These timelines specify the player's position, heading and whether the player is currently shooting. Periodically, the server checks the *shooter* timeline to see whether a shot has been fired. The time of the last known shooter status is queried and saved to t. The state of the shooter at that time is retrieved, and used to determine whether the shooter was firing his weapon at that time:

```
t = LastKnown(shooterTimeline);
shooter = shooterTimeline(t);
if( shooter.IsShooting ) ...
```

If the player was shooting, the server then determines the state of the target player at the time of firing, as viewed by the shooter. This is done by subtracting the amount of remote lag from the time the shot was fired, and retrieving the target state at that time:

```
target = targetTimeline(t - REMOTE_LAG);
```

Finally a *TargetHit* function uses the avatars' positions and shooter's direction to determine whether the target avatar was hit. If the target was hit, its health points are decremented.

```
if (TargetHit(shooter.Position, shooter.Heading,
                  target.Position))
{
    targetHealth(0) = targetHealth(0) - 1;
}
```

Thus, the server needs to take account of the shooter's frame of reference when a shot was made to determine whether the shot hit the target. The timelines model's ease of accessing past states makes this straightforward.

Short-term disconnections: telepointer updates

As our final example, we consider the problem of updating a participant's state following a short-term disconnection. This example shows how the rate at which time flows can be modified.

In this example, we wish to replay all the actions that occurred while the client was disconnected. The actions are replayed at a faster pace allowing the client to see all the changes that occurred while she was disconnected, and to be quickly brought up to the current state.

This scenario is again straight-forward to program using timelines. Consider the following example in which a remote telepointer is being displayed. When the client reconnects, a variable *drawLag* is initialized to the duration of the disconnection in milliseconds. (This disconnection time could easily be determined based on the time of the last update received.)

```
drawLag = DisconnectionTime;
```

A second variable *elapsedTime* contains the time in milliseconds between successive executions of a *DrawTelepointer* function. It is assumed that some form of delay exists between successive executions of the *DrawTelepointer* function. (*DrawTelepointer* might be repeatedly invoked through a timer event.) On the first execution of *DrawTelepointer*, the telepointer is drawn at its position as of the disconnec-

tion's start. With each successive execution, *drawLag* is decremented by an amount equal to the *elapsedTime*. This has the effect of replaying the telepointer positions at twice the original speed.

```
DrawTelepointer(int elapsedTime)
{
    Draw( pointerTimeline(-drawLag) );

    if (drawLag > 0)
    {
        drawLag -= elapsedTime;
    }
}
```

Summing Up: the Timelines Model

This section has introduced the timelines model, and shown how it allows explicit manipulation of time in groupware applications. The model's ability to access and modify past and future states, to change the rate of time, and to create divergent timelines for different participants provides the necessary infrastructure for the creation of a broad range of temporally-aware groupware applications. We now discuss how timelines can be supported in a practical toolkit.

IMPLEMENTING TIMELINES: THE JANUS TOOLKIT

In order to support experimentation with the timelines model, we have implemented it within the *Janus* toolkit. The toolkit is named after the eponymous Roman god, who as with users of timelines, had the ability to see into the past and future.

Janus is written in C#, although it could easily be ported to any other object oriented language. It has been used by the authors and by other developers to create several groupware applications based on Windows Forms and Microsoft's XNA game development library.

Within *Janus*, timelines are implemented as objects descended from the *Timeline* class. Each timeline has a base type (the type of the timeline state) and methods implementing interpolation, extrapolation and remote update handling. The default values of these methods can be overridden to create arbitrary timeline types. Timeline objects also provide *Get* and *Set* methods for retrieving/modifying the timeline's state, as well as methods for accessing the previous/next "known" value (i.e., value that is defined in the timeline, rather than interpolated/extrapolated.)

Each timeline object has a string identifier. If two clients create timelines with the same identifier, those timelines are synchronized. Whenever a timeline is modified, an update is sent over the network to its remote peers. When a remote update is received, it is applied via the remote update handling function. By default, this function simply inserts the new state into the timeline at the correct time; as we have seen, overriding this function can allow easy programming of interesting behaviours. To support synchronization, *Janus* uses a global clock, which we implemented using the Berkeley algorithm, whose accuracy is in the small tens of milliseconds [12].

Known timeline states (i.e., those that have been set with the *Set* method) are simply organized into a doubly-linked list, where each state is tagged by its time. The shared state object may be as simple as a single integer, for example representing the health points of an entity in a game, or it may be any arbitrarily complex object containing multiple properties such as the shape, rotation and colour of a drawing program entity. Although the time values stored in the linked list are real times (measured in ms since the epoch), the programmer always accesses states using relative time, where zero (0) means now, +10 represents 10 ms in the future, and -10 represents 10 ms in the past.

The *Get* method is used to retrieve the state of the timeline at a given time (past, present or future). The Get method uses the timeline's interpolation and extrapolation functions as necessary to provide values at times when none is known.

Distributed Architecture

From the developer's point of view, *Janus* has a peer-to-peer architecture. That is, updates are automatically routed between peers that share the same timeline, and all data is fully replicated. If a server is required (as with our Half-Life example of figure 7), one of the peers can be allocated a server role.

In the current implementation of *Janus*, we have developed a centralized message router to implement this peer-to-peer communication. The router is based on a distributed publish and subscribe architecture [8]. Timeline synchronization is implemented by peers subscribing to updates for the timeline in question. Updates are multiplexed so that one socket can be used to synchronize any number of timelines.

Two notable optimizations are possible with this architecture. First, updates are sent to peers whenever a timeline is modified. For objects with large state, this can consume significant bandwidth. In some cases, updates are predictable with high accuracy – e.g., changes to the position of a drawing program element that a user is dragging, or changes to the scroll position when a user is navigating a document. In our implementation of the Set method, we keep track of which messages have been sent to remote clients. Then, prior to sending an update to the timeline server, we calculate whether or not the remote client would be able to accurately extrapolate the new position if the message was not sent. If the remote client would be able to extrapolate the value of the update, then the update is stored locally but is not sent over the network. A second reduction in bandwidth requirements can be achieved by observing that often only part of the shared state has been changed. Instead of sending the complete state of changed values, deltas can be sent specifying what has changed.

The amount of memory used by our implementation represents an area for future optimization. Currently all values that are set are stored in the timeline, possibly requiring large memory. *Janus* currently truncates history to limit storage requirements. We plan to adapt algorithms for compacting groupware history developed for the latecomer problem [6], and mechanisms for compressing groupware messages [14].

Other areas for optimization include the networking infrastructure used by the toolkit. For example, recent techniques such as lazy scheduling [16] could be used to improve the efficiency of message delivery.

EXPERIENCE

Despite its status as a research prototype, *Janus* has been used by a growing number of developers to create temporally-aware groupware applications. The authors have used the toolkit to develop the cooperative Snagger game. With the toolkit, we were able to implement and experiment with a variety of consistency maintenance algorithms including local lag, dead reckoning, and the Half-Life algorithm.

The toolkit has been used by other developers to create several other games including three simple exercise games, a ubiquitous game based on spatial audio played on handheld devices, and a multiplayer physics-based platformer game. Because the latter game supports large numbers of players within a vast persistent world, the networking code needed to incorporate sophisticated interest management. These games were developed by five developers, none of whom were authors of *Janus* itself. All were students, ranging from undergraduate to Ph.D. level, and most had only passing experience with distributed systems programming. Despite this, they all reported finding it relatively straightforward to implement networking using the *Janus* toolkit. For the two player exercise games and the spatial audio game, the time to incorporate networking was measured in hours. Although the multiplayer game involving interest management was more complex, it was still just a matter of days to implement multiplayer support.

Our experience indicates that timelines can make it easy to implement basic networking in groupware applications. As we have seen, timelines also provide the opportunity to implement more sophisticated algorithms. Because timelines form a sufficiently different way of thinking about shared data, we have found that some instruction is required to illustrate the many ways in which they can be used.

Although our experience has largely focused on the development of multiplayer games, timelines can readily be applied to other groupware applications such as shared editors and drawing tools or chat applications where temporal awareness could lead to improved user experiences.

We have found timelines to be a powerful tool for adding temporal awareness to groupware applications. However, not all communication in groupware can be naturally modelled via shared data. For example, in the targeting example of figure 3, shooting actions are more naturally modelled via events than state changes. We are currently investigating approaches for integrating events with timelines.

DISCUSSION

We have illustrated that the development of temporally-aware groupware requires programmers to explicitly manipulate time, and have discussed that existing groupware toolkits do not provide the necessary mechanisms to do so easily. We have shown through examples that the timelines model (and its implementation in the *Janus* toolkit) can ease the development of temporally-aware groupware by allowing developers to access and modify past and future states, create divergent timelines, and change the rate at which time flows.

Many of our examples are drawn from multiplayer games. This is because games are a highly popular form of distributed groupware, and a great deal of innovation in user interfaces has come from well-funded gaming companies. While the benefits of temporal awareness can be applied to all forms of distributed, real-time groupware, outside the gaming sphere, most work has been done in research labs where the complexity of creating such applications has limited their exploration. We hope that once the *Janus* toolkit has been released to the public, it will permit wider exploration of temporally-aware groupware.

Interesting technical questions remain with the implementation of timelines. As discussed earlier, we plan to explore how timelines can be represented efficiently (using compaction algorithms), and how update messages can be compressed. A further interesting question is how accurately the toolkit can handle time. Our current implementation of the global clock can lead to divergence of tens of milliseconds between different nodes in the network, and the Windows CPU scheduling algorithm (based on multi-level priority queues) can lead to milliseconds of error when attempting to schedule an activity for a particular time. Characterizing the scope of these errors is an interesting question, as is the degree to which they can be mitigated by improved algorithms.

A significant issue to be addressed in future work is how traditional concurrency control algorithms can be meshed with the timelines model. Initial explorations indicate that the concepts fit well. A simple "lock" timeline whose state specifies the owner of a lock at a given time could support pessimistic concurrency control. For optimistic approaches (such as operational transform [24]), the concurrency control algorithm would be built into the remote update function, mediating when and how updates are applied to the timeline. Rollback algorithms [17] are easily implemented using timelines' ability to access previous states. Techniques that would allow the merging of data from multiple sources [19] might be possible by customizing the remote update function. Further work is required to rigorously examine how well such existing consistency maintenance algorithms can be adapted to the timelines approach.

We also plan to explore how richer interpolation and extrapolation functions can be used in timelines. In some applications, extrapolation could be based on user models or deeper application knowledge. In games, extrapolation of the behaviour of other players could be based on pathfinding algorithms, sophisticated artificial intelligence, or could be linked to a physics engine.

CONCLUSION

In this paper we have presented *timelines*, a programming model easing the development of temporally-aware group-

ware. We have argued that the presence of network latency requires better handling of time in groupware applications. Timelines makes the treatment of time an integral part of the programming model, allowing manipulation of past and future values, allowing changes in the rate at which time passes, and allowing control over how state diverges over time for different participants in a groupware session.

Timelines have been implemented within the *Janus* toolkit, and were used to implement all examples of temporally-aware groupware presented in the paper.

ACKNOWLEDGMENTS

We gratefully acknowledge the funding of the NSERC Strategic Project Grant on Technology for Rich Group Interaction in Networked Games and the GRAND Network of Centres of Excellence.

REFERENCES

1. S. Aggarwal, H. Banavar, and A. Khandelwal. Accuracy in dead-reckoning based distributed multi-player games. In *Proc. NetGames '04*, pages 161–165, 2004.
2. A. E. Ashcroft and W. W. Wadge. Lucid, a nonprocedural language with iteration. *CACM*, 20(7):519–526, July 1977.
3. Y. W. Bernier. Latency compensating methods in client/server in-game protocol design and optimization. In *Game Developers Conference*, 2001.
4. P. Bettner and M. Terrano. 1500 archers on a 28.8: Network programming in Age of Empires and beyond. In *Game Developers Conference*, 2001.
5. N. Bouillot. Fast event ordering and perceptive consistency in time sensitive distributed multiplayer games. In *CGAMES2005*, pages 146–152, 2005.
6. G. Chung, P. Dewan, and S. Rajaram. Generic and composable latecomer accommodation service for centralized shared systems. In *EHCI*, pages 129–147, 1998.
7. B. de Alwis, C. Gutwin, and S. Greenberg. GT/SD: Performance and simplicity in a groupware toolkit. *EICS*, pages 265–274, 2009.
8. P. Eugster, P. Felber, R. Guerraoui, and A. Kermarrec. The many faces of publish/subscribe. *ACM computing Surveys*, 35(2):114–131, 2003.
9. B. Freeman-Benson and A. Borning. The design and implementation of Kaleidoscope'90, A constraint imperative programming language. *Proc. IEEE Intl. Conf. on Computer Languages*, pages 174–180, 1992.
10. T. C. N. Graham and T. Urnes. Linguistic support for the evolutionary design of software architectures. In *ICSE 18*, pages 418–427, 1996.
11. S. Greenberg and D. Marwood. Real time groupware as a distributed system: Concurrency control and its effect on the interface. In *CSCW*, pages 207–217, 1994.
12. R. Gusella and S. Zatti. The accuracy of the clock synchronization achieved by TEMPO in Berkeley UNIX 4.3 BSD. *IEEE T-SE*, 15(7):853, 1989.
13. C. Gutwin. Traces: Visualizing the immediate past to support group interaction. In *Graphics Interface*, pages 43–50, 2002.
14. C. Gutwin, C. Fedak, M. Watson, J. Dyck, and T. Bell. Improving network efficiency in real-time groupware with general message compression. In *CSCW*, page 128. ACM, 2006.
15. C. Gutwin, T. C. N. Graham, C. Wolfe, N. Wong, and B. de Alwis. Gone but not forgotten: Designing for disconnection in synchronous groupware. In *CSCW*, pages 179–188, 2010.
16. S. Junuzovic and P. Dewan. Lazy scheduling of processing and transmission tasks in collaborative systems. In *Group'09*, pages 159–168. ACM Press, 2009.
17. A. Karsenty and M. Beaudouin-Lafon. An algorithm for distributed groupware applications. In *Proc. ICDCS*, pages 195–202, 1993.
18. M. Mauve, J. Vogel, V. Hilt, and W. Effelsberg. Local-lag and timewarp: Providing consistency in replicated continuous interactive media. *IEEE Transactions on Multimedia*, 6(1):47–57, 2004.
19. J. Munson and P. Dewan. Sync: A Java framework for mobile collaborative applications. *IEEE Computer*, 30(6):59–66, June 1997.
20. B. Myers, R. Miller, R. McDaniel, and A. Ferrency. Easily adding animations to interfaces using constraints. In *UIST '96*, pages 119–128, 1996.
21. N. Pelekis, B. Theodoulidis, I. Kopanakis, and Y. Theodoridis. Literature review of spatio-temporal database models. *Knowledge Engineering Review*, pages 235–274, 2004.
22. M. Roseman and S. Greenberg. Building real-time groupware with group-kit, a groupware toolkit. *ACM TOCHI*, 3(1):66–106, 1996.
23. J. Smed and H. Hakonen. *Algorithms and Networking for Computer Games*. Wiley, 2006. ISBN: 9780470018125.
24. C. Sun and C. Ellis. Operational transformation in real-time group editors: Issues, algorithms, and achievements. In *CSCW*, pages 59–68. ACM, 1998.
25. C. Wolfe, T.C.N. Graham, W.G. Phillips, and B. Roy. Fiia: User-centered development of adaptive groupware systems. *EICS*, pages 275–284, 2009.
26. J. Wu and T. C. N. Graham. Toward quality-centered design of groupware architectures. In *Engineering Interactive Systems*, volume 4940 of *Lecture Notes in Computer Science*, pages 339–355, 2008.