Multimedia Systems (2013) 19:271–287
DOI 10.1007/s00530-012-0271-3

REGULAR PAPER

Timelines: simplifying the programming of lag compensation for the next generation of networked games

Cheryl Savery · T. C. Nicholas Graham

Published online: 13 June 2012

Abstract Lag compensation algorithms used in networked games require programmers to manage the complexities of dealing with both time and shared state. This can make implementing lag compensation techniques challenging. The difficulties in expressing these algorithms limit experimentation with different algorithms and inhibit programmers from exploring the space of the algorithms and testing their effects. The solution is to have a programming model that is better able to deal with time. In this paper, we present such a programming model, *timelines*. Timelines dramatically reduce the time and effort required to implement lag compensation techniques by allowing for the explicit treatment of time. The timelines model has been implemented as part of the Janus toolkit.

Keywords Lag compensation · Consistency maintenance · Networked games

1 Introduction

Networked digital games afford an unprecedented level of interaction at a distance. Games permit people to closely coordinate the activities of small groups in real-time, to engage in large combats involving hundreds of people, and to view and react to the actions of opponents in real-time. New gaming technologies, however, are increasingly exposing the limits of interaction over the network. For example, motion-based input technologies such as Microsoft's Kinect open the opportunity for true real-time combat, where players react to other players' movements as they occur. Hardware advances have allowed games to include rich physics, enabling highly accurate sports and driving simulations, as well as allowing the game world itself to be physically modified as a consequence of player actions. While examples exist of games fully taking advantage of these techniques, the presence of network latency limits the ability to achieve such interactions over the network.

Specifically, true real-time combat is inhibited by the fact that it takes time to transmit each player's actions over the network. By the time a player sees another player starting an action, the action may have been completed, making it impossible to react. Similarly, it is difficult to synchronize physical actions (e.g., two players trying to kick the same ball) in the presence of latency. Current games make compromises to accommodate latency. For example, World of Warcraft's combat is based on special actions rather than true real-time movement. The execution of these actions is delayed, allowing time for them to be transmitted over the network. Games often compromise by making physics purely cosmetic. For example, a physics simulation might be used to show how the shards of a broken window fly through the air, but these positions are not used to compute damage to players.

Some of the limitations of latency are fundamental, but progress can be made through the deployment of novel lag compensation algorithms. For example, Bernier's Half-Life algorithm [3] helps with the problem of real-time aiming, and Sharkey, Ryan and Robert's local perception filters show great promise in the distributed physics problem [32]. A significant barrier to the invention, evaluation and deployment of such algorithms, however, is their complexity.

C. Savery (✉) · T. C. N. Graham
School of Computing, Queen's University,
Kingston, ON, Canada
e-mail: savery@cs.queensu.ca

T. C. N. Graham
e-mail: graham@cs.queensu.ca

It is hard to understand the consequences of new algorithms' tradeoffs on player experience; it is hard to compare alternative algorithms, and it is hard to communicate to game developers how the algorithms work so that they can be implemented in new contexts.

In this paper, we argue that much of the complexity of lag compensation algorithms comes from the fact that they deal with time, requiring programmers to consider not just what value shared data has, but when that value was held. Current programming languages do not provide support for dealing with time. We remedy this lack by presenting our timelines programming model for game networking. Timelines is a novel programming model for shared state in multiplayer games that exposes the temporal dimension of shared data. This allows programmers to access and manipulate past and future state values. Timeline variables can be shared between remote players, allowing programmers to modify the rate at which time flows and to create divergent timelines for different players. By making time an integral part of the programming model, timelines simplify the expression of lag compensation algorithms. This paves the way for the exploration of new ideas. Programmers can easily compare algorithms, test combinations of algorithms and develop new algorithms potentially enabling new styles of multiplayer games.

Our timelines model has been implemented within the Janus toolkit, and used by the authors and by other developers to experiment with lag compensation techniques and to create a range of multiplayer games. To show the power of timelines, we demonstrate how a range of existing algorithms can be expressed. To truly show the power of the approach, we use it to extend the local perception filters [32] algorithm, combining it with smooth corrections [34] to provide a novel solution to the distributed physics problem.

The paper is organized as follows. We begin by arguing why time plays such an important role in lag compensation, thus motivating the need for a programming model that explicitly deals with time. Next, we present our timelines model, showing how it facilitates the implementation of a range of lag compensation algorithms, including our novel extension to local perception filters. Finally we discuss the model's implementation in the Janus toolkit, and report our experience using the toolkit.

2 Why time matters

Lag compensation techniques are used in games to reduce surprising behavior due to the presence of network latency. Surprises might include not hitting an enemy who is clearly within the player's cross-hairs, being damaged in a collision when no obstacle can be seen, or seeing another player suddenly warp from one location to another. This surprising behavior results from differences in players' views of shared state due to the time to transmit state changes over the network.

Lag compensation algorithms attempt to reduce surprise by mitigating the effects of network lag. As we shall discuss in detail, there are three fundamental approaches to compensating for lag: dead reckoning [1], delaying local inputs [9], and remote lag [3]. These techniques are notoriously difficult to program, and their effects are difficult to analyze. In this section, we argue that this difficulty arises from the fact that the programmer needs to deal with time as well as with shared state data. For example, a lag compensation algorithm must account for when a shot was fired as well as where it was aimed to prevent situations where dead men are able to keep shooting [24]. Here we describe four key problems that highlight the importance of time when implementing shared data in networked games.

2.1 Stale message problem

Because it takes time to transmit messages over a network, state updates are necessarily out of date (or "stale") on their arrival. Clients must make adjustments to account for the untimeliness of the information; for example, a message indicating that a remote avatar has moved is in reality informing the local client where the remote avatar was at some point in the past.

Due to variance in network delivery times, it is difficult to estimate exactly when messages were sent (as required to assess exactly how stale the information is.) Solving this properly requires messages to be timestamped, and for the difference in clocks between the local and remote clients to be known [7]. In sum, the stale message problem requires programmers to treat messages not just as values, but as values that describe state at an earlier point in time.

2.2 Stale state problem

Some state within a game may be updated frequently; e.g., in a game executing at 60 frames/second, a local avatar's position may be updated every 17 ms. To conserve bandwidth, games typically transmit state updates less frequently, e.g., every 50 ms [3]. Updates may be further deferred if the local client can determine that the remote clients have enough information to accurately predict the updated state. Clients may therefore need to render several frames before new state information arrives, and thus must be able to estimate the present state of the remote client based on the last state information available. This requires clients to keep track of the age of local representations of remote state, and have mechanisms for compensating for state updates that have been suppressed by the remote client.

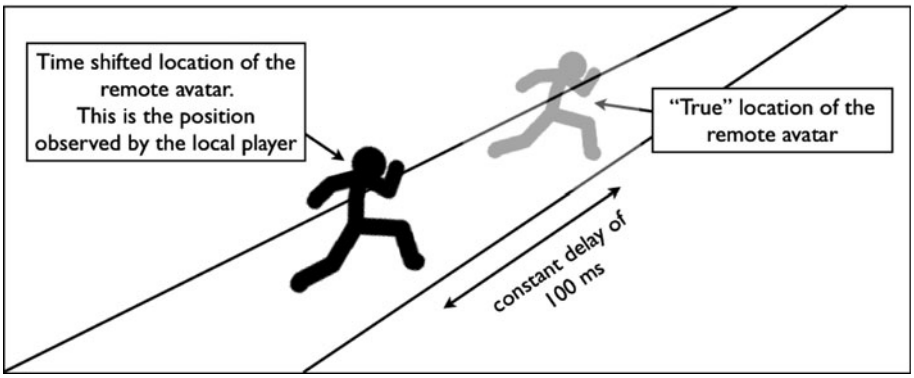

Fig. 1 Using remote lag, each player has his own view of the game world. The view of remote avatars is delayed by a fixed time which is greater than the network latency. This allows players to aim where they see other players' avatars instead of attempting to aim at the actual current position of the avatar

2.3 Frame of reference problem

Some games deliberately choose to use a different temporal frame of reference on each client. For example, in the Half-Life series first-person shooter games, a constant time delay is applied to the movements of other players' avatars [3]. This increases the predictability of the movement of remote avatars, at the cost of increasing the divergence of state between different players [31]. Thus, as shown in Fig. 1, instead of there being a single global game state, each player has his own personal view of the game world. Hits are then based on what a player actually sees on his screen instead of being based on the "true" location of the other player's avatar. This allows players to aim and shoot directly at a target avatar without accounting for incorrect positions caused by network delays. To provide consistency, and to prevent cheating, a server must be able to take into account what each player sees and reconstruct that view. This requires the server to unwind time and reconstruct the state of both the target and shooter clients at the point in time when a shot was fired.

2.4 Multiple-times at-once problem

Some lag compensation techniques require that the client be able to simultaneously access shared state in the past, the present and the future. For example, this problem arises in smooth correction algorithms, as shown in Fig. 2. When clients use prediction (such as dead reckoning) to extrapolate the position of a remote entity, that prediction will be incorrect whenever the entity has changed speed or direction. Once the error is detected, instead of immediately warping the remote entity to the correct position, the smooth corrections algorithm progressively moves the avatar to the new position. This provides a less jarring means of repairing this incorrect state. To implement smooth corrections, the programmer must access the previous state to know where the entity was, access the current state to know where the entity should be and calculate a future state to be able to smoothly merge the two states over time.

As we see from these examples, working with time is a fundamental component of lag compensation algorithms. The examples show how it is necessary for clients to interpret state and state updates relative to times in the past, and necessary for servers to reconstruct the differing temporal frames of reference of different clients, and sometimes necessary for clients to simultaneously access past, present and future values of the same state.

Traditional programming models treat state as having only a single value, describing the current time. This is the root cause of the difficulty of expressing lag compensation algorithms, which fundamentally must deal with different temporal frames of reference. We will show

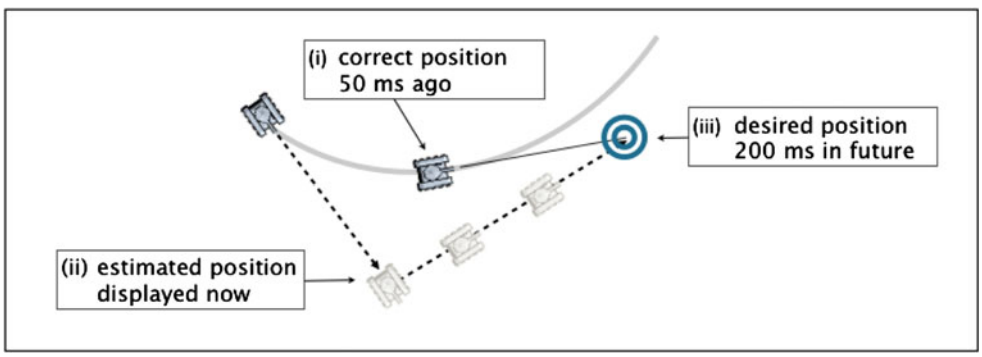

Fig. 2 Smooth corrections require access to (*i*) where the avatar was in the past (as reported by newly arrived message), (*ii*) where the avatar is now, and (*iii*) where we want the avatar to be in the future

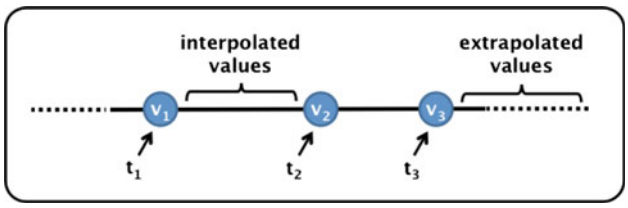

Fig. 3 Elements of the timelines model: a timeline includes a set of past and future values (v_1, v_2,…v_3), along with the times at which they hold (t_1, t_2,…t_3). Values between these times are computed using an interpolation function, and values after these times are estimated using an extrapolation function

that a programming model that explicitly incorporates time dramatically simplifies the implementation of these algorithms.

3 The timelines model

In distributed systems, shared data is typically stored as an instantaneous value representing the data's current state [8]. As we have discussed, this leads to difficulties when programming with time, as required in lag compensation algorithms. In contrast, our timelines model represents shared state variables 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. Timelines provide:

- Get/set operations that access the timeline's value at a given time; and
- Interpolation and extrapolation functions that estimate values for times when no value is known.

As we shall see, this simple model of shared data allows the simple expression of a wide variety of lag compensation algorithms.

Figure 3 shows how the elements of the Timelines model are combined. In the example timeline, *v*1, *v*2 and *v*3 represent values for times *t*1, *t*2 and *t*3, respectively. These are known values, meaning that they have been explicitly set in the timeline.

To illustrate this, consider *fuel*, an integer timeline representing the amount of fuel in a spaceship. Storing values in the *fuel* timeline is as simple as assigning those values at a specific time. Thus,

```
fuel(−100) = 230;
```

assigns 230 to the integer timeline at 100 ms in the past, and

```
fuel(0) = 200;
```

assigns 200 to the *fuel* timeline at the current time. (Time references are expressed in milliseconds, may be positive or negative, and are relative to the current time.)

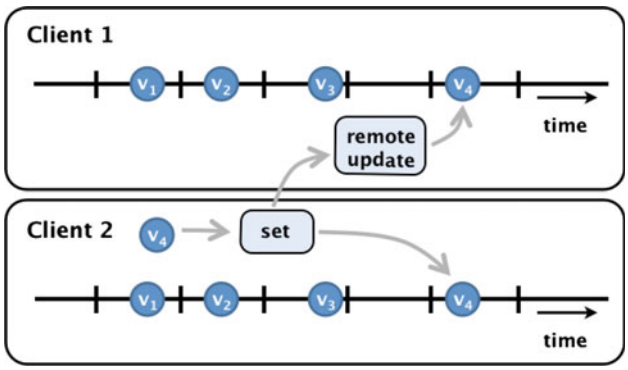

Fig. 4 Timelines are fully replicated. When a client sets a new value into the local timeline, the remote update function inserts the value into the same timeline on all other clients

When the value for a given time is not known, it is interpolated or extrapolated from its neighboring known values. The value of *fuel*(−50) is derived as 215 using linear interpolation between the two known values of 200 and 230, and the value of *fuel*(100) is extrapolated to be 170. 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.

The timelines model is fully replicated, with each client storing a local timeline object for each portion of the shared state in which it is interested. If two clients create instances of the same timeline, the timelines are automatically synchronized. Figure 4 shows two clients each with a copy of the same timeline. When Client 2 inserts a new value into the local timeline, the value is propagated over the network and a remote update function on Client 1 is invoked. The remote update function specifies how updates arriving over the network are to be applied. By default, the remote update function inserts the value into Client 1's timeline. However, as we will see, it is possible to override this default behavior to implement techniques such as smooth corrections.

To illustrate the model, we now discuss several lag compensation techniques used in multiplayer games and show how timelines can be used to implement them.

4 Using timelines

In this section, we illustrate the use of timelines to express a canonical set of lag compensation algorithms. Through these examples, we demonstrate the features of the model, as well as illustrate its expressiveness.

Multiplayer games use a variety of techniques for lag compensation. These techniques have generally been proposed to enhance user experience or to combat cheating

[13]. Lag compensation techniques used in games make tradeoffs between the degree of consistency provided and the responsiveness to player commands [31]. Based on these tradeoffs, we can divide the mechanisms used for lag compensation into three broad categories: predictive techniques, delayed input techniques and time-offsetting techniques. The algorithms used within each category are described below:

- *Predictive techniques* estimate the current value of remote state from state that is available locally. For example, the dead reckoning [21] technique is widely used to estimate the current positions of other players' avatars based on earlier position and velocity information that has been sent over the network. This helps to solve the "stale state" problem described in Sect. 2, by providing a means of estimating what has happened on the remote client between state updates. Predictive algorithms can be used to address the "stale update" problem. When a state update arrives over the network, prediction can be used to estimate the state's current value, as opposed to the value at the time the message was sent. To achieve this, state update messages must be timestamped, and the clocks on the local and remote clients must be synchronized [1]. Finally, if smooth corrections [34] are used to repair incorrect predictions, the programmer must be able to access states from the past, present and future, as shown in Fig. 2.
- *Delayed input techniques* such as bucket synchronization [4] and local lag [25] defer local actions to allow simultaneous execution by all clients. Programming local lag requires mechanisms for delaying inputs and 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. Bucket synchronization requires a mechanism for pausing all clients at exactly the same point of execution, applying all pending messages in the same order, and resuming execution.
- *Time-offsetting techniques* such as remote lag [3] and local perception filters [32] insert a delay in the application of remote state updates. For example, as shown in Fig. 1, a remote player's avatar's position is lagged by a constant number of milliseconds. This approach trades off the immediacy of positional updates versus reducing the jitter caused by variance in time to deliver updates over the network.

All of these techniques involve the manipulation of time—either predicting the future, scheduling actions for future execution, or providing divergent timelines for different players. In this section, we now show how the timelines model simplifies the implementation of algorithms from each of these styles of techniques.

4.1 Predictive techniques

Predictive techniques involve two components. First, extrapolation functions are used to estimate the current state of a remote entity based on previous known states. For example, dead reckoning is widely used as a mechanism for estimating the position of remote avatars [21]. Second, a mechanism is required to correct the state of an entity when the prediction is proven wrong. The following sections describe both of these components in more detail and show how using timelines facilitates their implementation.

4.1.1 Example: dead reckoning

Dead reckoning is commonly used in distributed interactive simulations to reduce the number of network messages required to convey positional updates. Dead reckoning is based on the assumption that entities rarely change direction or speed, and that therefore previous movement is an accurate predictor of current movement. The IEEE distributed interactive simulation standard defines a protocol for dead reckoning [20, 21] whereby an extrapolation equation is used to estimate the position of an entity. Instead of transmitting an update packet following each movement of the entity, updates are only transmitted when an error threshold is exceeded. When dead reckoning is used in multiplayer games, the client controlling an entity transmits position and velocity data to remote clients, which then use this information to estimate current or future positions of the entity.

Dead reckoning algorithms, however, frequently fail to account for delays caused by network latency. Typically, when a client receives a positional update, that position is set as the entity location at the current local time, not at the time the message was sent. Aggarwal et al. [1] have shown that treating remote updates as past events (i.e., accounting for network latency) improves the accuracy of dead reckoning. However, this approach requires the programming complexities of timestamping positional update messages and synchronizing the clocks on the local and remote clients.

Dead reckoning with lag awareness is built into the timelines model, requiring no extra programming by the game developer. The model's extrapolation function provides dead reckoning as the default behavior. When updates are received from a remote client, they are automatically inserted into the local timeline at the time they were sent, not the time they were received.

The following example shows how easily dead reckoning with lag awareness can be implemented using timelines. Consider that "Alice" and "Bob" each control avatars in a game. Alice's avatar's position is represented in a timeline of positions called *alicePos*. We assume the

traditional game architecture where inputs are polled and the frame is rendered asynchronously. Thus, when Alice moves to a new position (x, y), the operation on Alice's client to process the movement is simply:

```
alicePos(0) = (x, y);
```

This operation inserts the value (x, y) into the *alicePos* timeline at time 0, the current time. Then, on both Bob and Alice's clients, when the frame is rendered, Alice's position is drawn as:

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

That is, the avatar is drawn at its position at the current time.

On Bob's client, as messages indicating Alice's movements arrive, they are automatically inserted into the timeline taking into account the network latency. For example, if the message containing Alice's position required 60 ms to travel over the network, then the position is automatically inserted in the *alicePos* timeline on Bob's computer at time -60.

When Bob's client uses the value of *alicePos(0)* to render Alice's position, it extrapolates from her last known position. This example shows how the timelines model provides lag-aware dead reckoning as default behavior, requiring no special programming.

4.1.2 Example: smooth corrections

With dead reckoning, when a new positional update arrives that is significantly different from the current predicted location, the simplest solution is to immediately move the entity to the new location. This results in jerky animation and can be visually jarring for the player. Convergence techniques [34] may be used to correct these errors in a smooth, less surprising manner. We instead move the entity progressively to its correct position. For example, we might aim to have the entity in the correct position 200 ms in the future. It is not sufficient to move the entity to its current position 200 ms in the future; instead, we must estimate where it will be in 200 ms, and progressively move it to that location.

The first step in the algorithm is to select the location and time of the entity's corrected position (e.g., the new location 200 ms in the future.) The entity then moves at increased speed until it reaches this correct location, as shown in Fig. 5a. An even better correction can be accomplished by following a curved path to the new location, as shown in Fig. 5b. This progressive correction can be difficult to program, as we need to simultaneously deal with entity's current position, its correct (but stale) position, and its future correct position. Furthermore, the avatar's position must be updated over time until the correct position is attained.

Smooth positional corrections can be easily specified using timelines. We do this by overriding the position timeline's default remote update function of Fig. 4. Normally, remote updates are 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 toward a consistent state.

Figure 6 shows this approach. We use a timeline to represent the positions of a remote player's avatar. The current position of the avatar has been extrapolated, based on these known positions. We identify this extrapolated location as *currentPos*. Then, as seen in Fig. 6, 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 is to update the current position to *correctCurrentPos*, and this is in fact what the timelines model does by default.

Instead, we decide to smoothly move the avatar to the correct position over the next 200 ms. The avatar's target position (*targetPos*) is determined by extrapolating 200 ms into the future. The avatar will move quickly over the next 200 ms to that position.

We will now describe in more detail, how this is accomplished 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 200 ms in the future.

```
avatarPos(fixupTime) = fixupPos;
targetPos = avatarPos(200);
```

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(200) = targetPos;
```

The correction may be either a straight path toward the target position as shown in Fig. 5a, or if a nonlinear interpolation function is used, the correction will follow a curved path as in Fig. 5b.

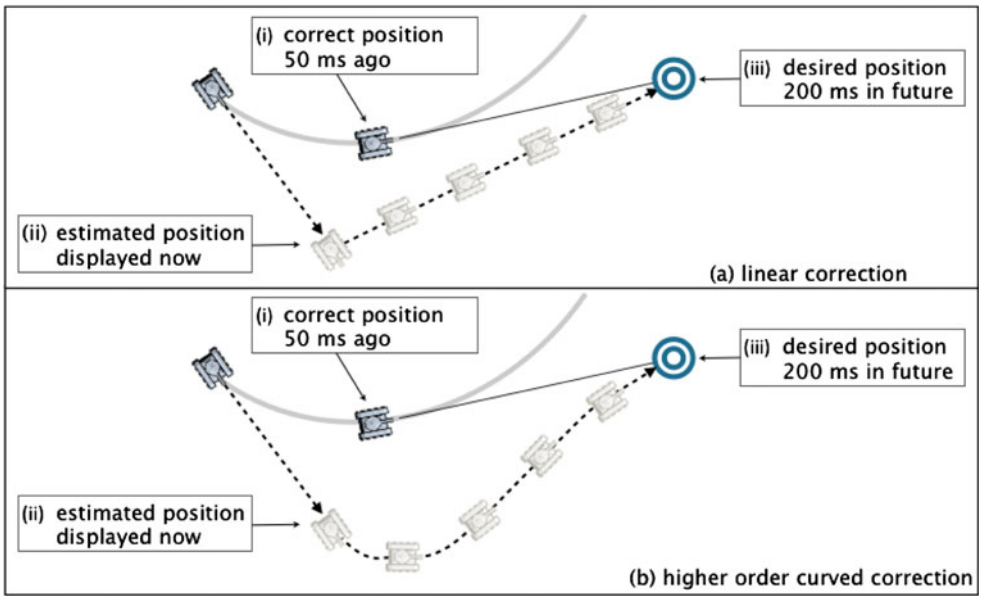

Fig. 5 Corrections to incorrect states predicted by dead reckoning may be corrected by **a** following a straight path to a future point predicted by dead reckoning, or **b** following a curved path

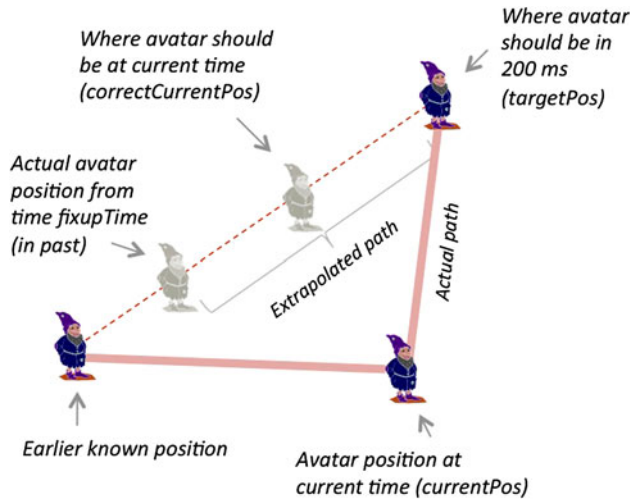

Fig. 6 A timelines implementation of smooth corrections

This example illustrates how smooth corrections can be easily implemented using timelines. By default, the timeline remote update function inserts values from remote clients into the local copy of the timeline. This function would contain just one step:

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

With the simple addition of four lines of code, the remote update function becomes:

```
currentPos = avatarPos (0);
avatarPos (fixupTime) = fixupPos;
targetPos = avatarPos (200);
avatarPos (0) = currentPos;
avatarPos (200) = targetPos;
```

and we have implemented smooth corrections. Also, depending upon the form of interpolation and extrapolation functions used (linear or higher order) the correction may follow either a straight line or a curved path.

The key concept illustrated in this example is that by having the ability to modify a timeline, programmers are able to explicitly control the divergence of timelines between different clients. The remote client, where the avatar is being controlled, sees neither the error nor the correction.

4.2 Example: delayed input techniques

While dead reckoning is a predictive technique that can lead to inconsistencies in the game state, delayed input techniques such as local lag and bucket synchronization take the opposite approach. With these techniques, the goal is to reduce or eliminate inconsistencies by delaying local actions.

Players can often better coordinate their actions if they see the same changes to game state at the same time. Various algorithms have been developed that manipulate time to help synchronize players' actions. Notable among these is local lag [25]. The key idea behind local lag is to delay the execution of local commands long enough that the commands have time to propagate to all remote sites and can then be executed simultaneously at all locations. Programming local lag is surprisingly tricky, requiring mechanisms for delaying inputs and 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.

Conversely, local lag is simple to implement using timelines. Consider again "Alice" and "Bob" from our

previous example for dead reckoning. As before, Alice's position is stored in the timeline *alicePos*. In this example, however, Alice's client uses local lag to set positions in response to her movement commands. We will assume that the local lag constant is *DELAY*. That is, if Alice presses a key to move her avatar, there will be a delay of *DELAY* ms before she observes the movement associated with that key press. Bob should also observe the same movement *DELAY* ms after Alice pressed the key.

Thus, if Alice moves to a new position (x, y), the operation on Alice's client to process the movement is simply:

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

That is, the position (x, y) is stored in the *alicePos* timeline *DELAY* ms in the future. For example, if $DELAY = 100$, then Alice's position is set to (x, y), 100 ms in the future.

As before, Alice's position is drawn on both clients as:

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

That is, the avatar is drawn at its position at the current time.

This very simple code has a range of interesting effects. On Bob's client, messages indicating Alice's movements are automatically inserted into the timeline when they arrive. If the message took less than *DELAY* to arrive (hopefully the normal case), then on Bob's remote client, the new position is inserted into the *alicePos* timeline in the future. For example, if $DELAY = 100$ and the message took 60 ms to arrive, the message is inserted into the *alicePos* timeline on Bob's computer at $t = 40$. This allows the present position of Alice's avatar to be interpolated (using previously recorded positions). Therefore, the local lag functionality supports smooth movements on remote clients without annoying corrections.

Alternatively, if the message took more than *DELAY* to arrive, say 130 ms, then the positional update is inserted into the *alicePos* timeline in the past ($t = -30$), and current positions are extrapolated from this (and possibly other) past values.

This simple example illustrates the power of the timelines model. Merely changing the time at which Alice's position is set in the timeline allows a game developer to switch from dead reckoning to local lag. Also, 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, requiring no code from the game programmer.

4.3 Time-offsetting techniques

Time-offsetting techniques render game entities at different times on different clients, typically displaying a delayed (or "time-offsetted") version of remote players and objects. This approach is useful when designers believe it is better to provide an accurate representation of the timing of other players' activities, even if that representation is delayed.

4.3.1 Example: remote lag

One time-offsetting technique is the aiming mechanism used in the Half-Life series of games [3]. In shooting games, an authoritative central server is usually used to arbitrate when a shooting player has hit another target player. The simplest means to implement this is to have a single canonical game state and after a shot is fired, the server determines the location of the target player and whether or not a hit occurred. This can make aiming difficult because the shooter must predict where the other player's avatar is going and aim ahead of it in order to get a hit (Fig. 7a).

In the Half-Life series of games, each client applies a constant lag to the actions of other players and the server arbitrates hit decisions based on the state of the shooter's client at the time the shot was made (Fig. 7b). This means that the shooter can aim directly at the target player. However, implementing this mechanism can be complex as it requires the server to be capable of unwinding time in order to determine the position of the target avatar on the shooter's client at the time the shot was fired. Since the lag is applied only to remote avatars, each client's frame of reference is different.

Figure 8 shows how timelines are used to solve this targeting problem. First, we will look at how timelines are used to easily render a current version of the local avatar and a time-delayed version of the remote avatar. We assume that we have two timelines *avatar1* and *avatar2*. These timelines will contain the position of the avatar, the direction it is aiming and an indication of whether or not the avatar is shooting. Without loss of generality, we assume that *avatar1* is currently shooting at *avatar2*. These

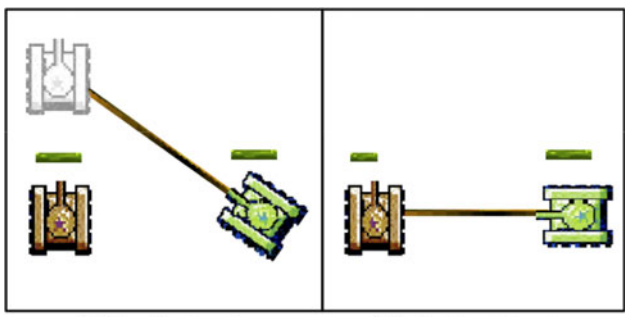

Fig. 7 Shooting often requires players to "lead", guessing where the target actually is (**a**). The Half-Life algorithm allows players to shoot at the target where they see it (**b**)

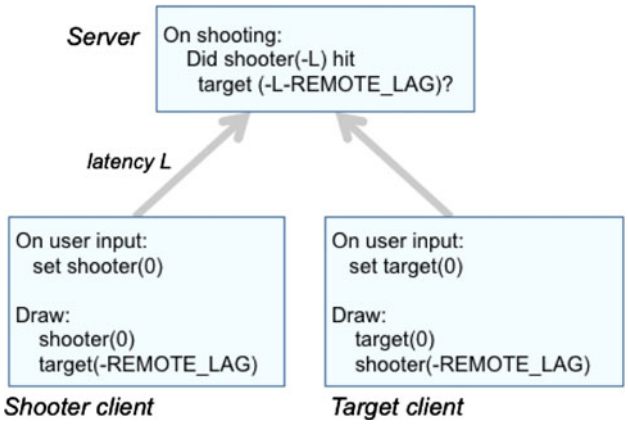

Fig. 8 A timelines approach to implementing Half-Life's hit determination code

two timelines are shared by two clients and a central server. When players perform input actions, the appropriate timelines are updated at time 0. Thus, if player 1's avatar has moved to position (*x, y*), is aiming in direction (*aim_x, aim_y*) and is shooting, then the avatar state is set as follows:

```
avatar1(0) = (x, y, aim_x, aim_y, TRUE);
```

When rendering the avatars, we want to display the local avatar at the current time, but display a delayed version of the remote player's avatar. Thus, assuming that the amount of delay to be applied to the remote avatar is *REMOTE_LAG*, then, on player 1's client, the avatars are drawn as:

```
DrawAvatar(avatar1(0));
DrawAvatar(avatar2(−REMOTE_LAG));
```

Now we will look at how timelines can be used to allow the server to determine whether or not player 1 was aiming at player 2 when he fired. First, we will need to know the length of time required for a message to travel from the client to the server. If we assume that a message takes L ms to arrive at the server. Then, the server must make the hit decision based on the state of the shooter (*avatar1*) at time $-L$ (when the message was sent) versus the state of the target (*avatar2*) at time *–L—REMOTE_LAG* (where the shooter believed the target was at time $-L$).

Periodically, the server checks the *avatar1* 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 = LastKnownTime(avatar1);
shooter = avatar1(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 = avatar2(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 or not the shot hit the target. To do this, the server must reconstruct the state of the shooter's client at the time of the shot. This requires the server to access the state of the shooter at the time the shot was fired and to access the state of the target at the point in time prior to the shot corresponding to what the shooter saw. The timeline model's ease of accessing these past states makes this straightforward. A past state can be retrieved from a timeline by simply specifying the time at which the data is needed. Thus, for the server to determine if a hit occurred all that is required is to know the time at which the shot was fired and the time delay associated with the target.

4.3.2 *Example: local perception filters*

Local perception filters [32] is another example of a lag compensation algorithm based on time offsets. The key idea is to continually adjust the amount of delay applied to non-player controlled entities depending upon their position relative to the local player's avatar. Despite the approach's great promise, we are aware of no games using local perception filters. We conjecture that this may be due to the difficulty of implementing it using standard programming tools, making it difficult to for developers to quickly evaluate how well the approach works in their game. As we shall show, the timelines model makes the implementation of local perception filters tractable, opening its application to distributed game physics.

We first describe the local perception filters algorithm, and then show how it can be implemented using timelines. We begin by looking at a single object whose motion is determined by a physics engine, for example in the simple soccer game as shown in Fig. 9. Through this example we highlight some of the shortcomings of other more

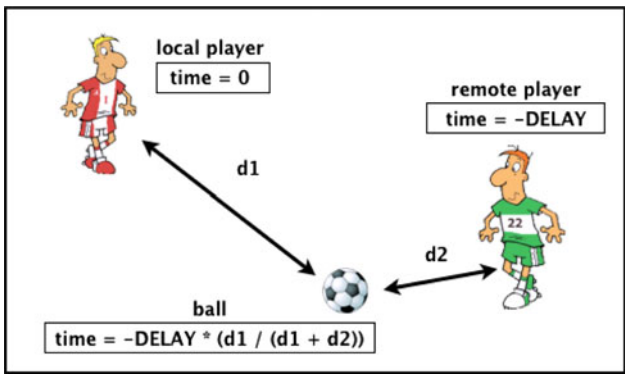

Fig. 9 When using local perception filters, the time used to retrieve the ball's position changes relative to the local and the remote players

commonly used lag compensation techniques. In the soccer game, two players kick a ball about in a 2-D world. The players are free to move their avatars around the world, and a physics simulation is used to determine the position of the ball. Ideally, both players would see the ball in the same position, and the ball would react instantly when it is kicked. However, due to network latency, this is not possible. Delayed input techniques such as local lag would provide the desired consistent view, but the local player would experience reduced responsiveness as local lag causes a delay between kicking the ball and seeing it move. Predictive techniques provide fast response times, but extrapolated positions are often inaccurate, particularly when the avatar changes direction or the ball is kicked. In the presence of physics, such inaccuracies can be highly visible.

Remote lag (as described above) applies a delay to the remote player's avatar and renders the local avatar in real-time. This provides immediate responsiveness for the local player. Since the position of the remote avatar is time-delayed, positional updates are treated as future values, allowing positions to be interpolated on the local client, providing smoother animation. Because the ball travels between the two players' avatars, when the remote avatar kicks the ball, the ball's motion must be delayed by the same amount of time as the remote avatar. Otherwise, the interactions between the player and the ball would appear unrealistic. However, when the ball is near the local player's avatar, it must move in real-time, or again the interactions would appear unrealistic. Local perception filters balance this by adjusting the ball's delay depending upon its position relative to the two avatars. For example, if the remote player's avatar is delayed by 100 ms, the ball is also delayed by 100 ms when it is next to the remote player's avatar, by 50 ms when it is halfway between the two avatars and by 0 ms when it is near the local player's avatar. This provides the local player with fast response times when he kicks the ball. It also allows him to view realistic interactions when the remote avatar kicks the ball.

To implement this example, a developer, therefore, needs to continually adjust the delay associated with the ball as it moves relative to the positions of the avatars. As an additional complication, the physics simulation for the ball must be carried out on the client whose avatar is closest to the ball. The delay in rendering the ball on the other client provides sufficient time for the ball's position to be transmitted over the network. In sum, to implement this simple game using local perception filters, the programmer must maintain different timeframes for both players, to adjust these timeframes dynamically, and to implement a distributed physics simulation with dynamic change of simulation host.

Timelines make this implementation tractable because they allow the programmer to easily access the state of an object at any point in time. Specifically, we use timelines to:

- pass the right to calculate the position of the ball using the physics simulation;
- determine the time delay for the ball; and
- determine the time delay for the other player's avatar.

In our example, we assume a client–server architecture with two clients each running a physics simulation to determine the position of the ball. The server uses the ball and player position information to arbitrate which player is controlling the updates to the ball position.

To illustrate how timelines can be used to implement local perception filters, we require three timelines: *player1Pos*, *player2Pos* and *ballPos* representing the positions of the three game entities. We also require one additional timeline, *ballControl*, that indicates which player is controlling the physics simulation for the ball and updating the *ballPos* timeline. The *ballControl* timeline contains discrete values, and thus uses stepping interpolation/extrapolation.

The server receives updates for the player and ball positions from each client and then, based on the distances of each player from the ball, determines which player should be updating the ball position. The server then sets this information in the *ballControl* timeline.

```
ballControl(0) = Player1;
```

The following steps then occur on each player's computer. Here we describe the steps assuming that player 1 is the local player. Then, on player 1's computer, the client checks the value in the *ballControl* timeline. If player 1 is controlling the ball, his client calculates the new ball position and sets that value in the *ballPos* time-line. If player 2 is controlling the ball, player 1's client does nothing as it will be able to get values from the timeline that are set by player 2's client. Thus, assuming *getUpdatedBallPos* is a method that updates the ball position and then returns that updated position, the code for adding new values to the timeline is:

```
if (ballControl(0) == myPlayerNumber)
{
    ballPos(0) = getUpdatedBallPos();
}
```

Next, to draw the ball in the correct position, the players must determine the delay to apply to the ball.

Assuming again that player 1 is the local player, then player 2's avatar is drawn at *DELAY* ms in the past. His avatar's position will be:

```
player2Pos (−DELAY);
```

and the local player's avatar (player 1) is drawn at:

```
player1Pos(0);
```

To estimate the ball's position, we use the position where it was last drawn as a first approximation. We will assume this value has been stored in a variable called *prevBall*. Then, assuming that the lag applied to the ball is linear depending upon the distance between it and the two players' avatars, we can calculate the *ballDelay* as follows:

```
d1 = length (player1Pos (0) − prevBall);
d2 = length (player2Pos (−DELAY) − prevBall);
ballDelay = DELAY ∗ d1/(d1 + d2);
```

Then the ball position to be rendered is:

```
ballPos(−ballDelay);
```

This example illustrates how timelines' ability to access entity positions at any point in time has allowed us to easily change the timeframe of the ball each time it is rendered. It also shows how it is possible to allow multiple clients to coordinate updates on shared state data. Given the ease with which local perception filters can be implemented with timelines, it becomes practical to assess their suitability for games under development.

4.4 Using timelines: summing up

This section has argued that lag compensation algorithms fall into the three classes of predictive, delayed input and time-offsetting techniques. We have shown that timelines can be used to express representative algorithms drawn from all three of these techniques. We have shown that complex algorithms can be expressed with very little code, making it tractable for developers to experiment with complex and novel lag compensation schemes.

5 Implementing timelines

The previous section showed how timelines' explicit treatment of time provides the necessary infrastructure for implementing many of the lag compensation techniques used in multiplayer games. We now discuss how timelines are implemented within our Janus toolkit. Janus provides a low-overhead implementation of timelines with a simple API. The toolkit is named after Janus, the Roman god of gates, doorways, beginnings and endings. Janus also had the ability to see into both the past and the future, just as users of the Janus toolkit are able to access previous and future versions of the game's state.

The Janus toolkit is written in C# and is compatible with any .NET language. It is built on top of the Lidgren Networking Library (http://code.google.com/p/lidgren-network-gen3), which provides reliable UDP messaging.

5.1 Object model

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.

Each timeline object has a string identifier. If two clients create timelines with the same identifier, those timelines are automatically synchronized. As shown in Fig. 4, whenever a new value is added to a timeline, an update is sent over the network to the client's remote peers. When the remote update is received, it is applied to the timeline via its remote update handling function. By default, this function simply inserts the new state into the timeline at the correct time and removes any later values in the timeline. As we have seen, overriding this function can allow easy programming of interesting behaviors, such as the smooth corrections of Sect. 4.1.2.

Known timeline states (i.e., those that have been inserted into the timeline 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, or it may be any arbitrarily complex object containing multiple properties. We have created a variety of standard timeline objects including an integer, a floating point number, 2-D and 3-D position vectors and more complex objects combining position, heading and velocity. Users of the toolkit can either use these existing timeline objects or create a new type of timeline object.

Although the time values stored in the linked list are real times (measured in milliseconds 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.

5.2 Interpolation and extrapolation

A timeline's Get method is used to retrieve its state 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. Default implementations of linear and stepwise interpolation/extrapolation are provided, and developers can create additional functions to provide domain-specific behaviors. For example, with the dead reckoning, the extrapolation function may be either first order (based on position and velocity) or second order (based on position, velocity and change in either speed or direction). The function may also be based upon only the most recent update, or it may be based upon two or more previous states. Our timelines implementation supports the use of a variety of extrapolation (and interpolation) functions, and thus all these options are possible merely by selecting a different extrapolation function. Which form of extrapolation function is most suitable depends upon both the type of game and the type of motion [27]. With timelines, the choice of function can be modified at runtime, thus facilitating adaptive techniques such as the use of position-based history [33] where the extrapolation function changes based on the motion of the entity.

5.3 Distribution and networking

From the developer's point of view, Janus has a peer-to-peer architecture. That is, updates are automatically broadcast to all peers that share the same timeline, and all data is fully replicated. If a server is required (as with our targeting and shooting example of Fig. 8), 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 [11]. As indicated previously, a string identifier is associated with each instance of a timeline object. When a client creates a timeline object with a given identifier, the identifier is passed to the message router and the client automatically subscribes to updates for that object. When a client stores a new value in a timeline object using its Set method, the value and the time associated with it are sent to the message router which forwards the data to all other clients who have subscribed to that timeline object.

By default, the Janus Toolkit does little to minimize the number and size of messages passed between clients. However, several features are available that dramatically reduce bandwidth requirements of applications using the toolkit. First, the programmer can set a minimum time interval between updates. Thus, not all changes to the local timeline are propagated over the network. For example if the local client updates positions every 20 ms and the minimum time interval for sending updates is set to 60 ms, then only one-third of the updates are sent. Second, each client continually keeps track of which values have actually been sent over the network. Then, prior to sending an update, the client performs a check to determine whether or not remote clients can accurately predict the new updated state. Only if the remote client is unable to predict the new state within a set error threshold will the new state be transmitted. By setting the size of the error threshold, the programmer can control the fraction of messages that are sent. Auto-adaptive dead reckoning schemes [5] can also be implemented by adjusting the error threshold depending upon the game situation or factors such as network congestion and bandwidth availability.

Finally, the programmer is able to optimize the size and format of messages. By default, Janus uses object serialization to convert objects to a byte array for transmission over the network. This has the advantage of making it simple to create new timeline classes without the need to worry about how the data is transmitted. However, the built in object serialization can generate unnecessarily large messages. Programmers can override the default serialization methods, and are thus able to optimally format the messages passed.

To support synchronization, Janus uses a global clock. Determining and maintaining a global clock on all clients can be a daunting task due to clock drift, network latency and jitter. We have implemented our global clock using the Berkeley algorithm [16]. In our implementation, the message router periodically sends timing messages to all clients. The message router analyzes the timing messages returned from the client, discarding any outliers and then sends updates to the global time back to the client.

5.4 Alternative architectures

The timeline model does not impose any specific architecture on the game. In the Janus toolkit, we chose to implement a message broadcasting underlay with a peer-to-peer overlay. The decision to use a central message router was made purely for simplicity of implementation, and could be replaced by true peer-to-peer message broadcasting techniques. This implementation easily supports a variety of overlay architectures. As shown in Fig. 10, the message communication architecture is determined by the topology of clients' subscriptions to timeline objects. Figure 10a shows a peer-to-peer overlay where each client (C1 and C2) subscribes to updates for all timelines (TL1 and TL2). In the client–server model shown in Fig. 10b, each client shares a timeline only with the server. That is, client C1 updates timeline TL1 and the message router only

propagates those updates to the server S1. Similarly, client C2 updates timeline TL2 and the message router only propagates those updates to the server S1. Timelines TL3 and TL4 are both updated by the server and the updates for these timelines are forwarded to both clients. In the hybrid model in Fig. 10c, timelines TL1 and TL2 are shared only between one client and the server, while timelines TL3 and TL4 are shared between the two clients and timeline TL5 is shared by all.

6 Background and related work

We have shown that by making time an integral part of the programming model, timelines have simplified the implementation of lag compensation techniques used in multiplayer games. To place this work in context, we now look at the support provided by existing game networking libraries for time-based programming, and then review other programming environments which incorporate time.

6.1 Networking support for multiplayer games

Existing game networking toolkits provide only limited support for manipulating time, contributing to the difficulty of implementing many lag compensation algorithms. ZoidCom (http://zoidcom.com) includes a special replicator that implements client side prediction, dead reckoning/extrapolation, interpolation, movement correction and local overrides. As we have seen, interpolation and extrapolation are important to programming lag compensation algorithms, but are not sufficient to fully solve the problems discussed in Sect. 2, particularly the frame of reference or multiple-times at-once problems.

OpenTNL (http://opentnl.org) also includes mechanisms for interpolation and extrapolation of object positions. NetZ (http://quazal.com) provides two models for shared objects: attribute propagation which uses data extrapolation to reduce bandwidth requirements, and step-by-step synchronization. Other game networking libraries such as ClanLib (http://clanlib.org), Nevrax/NEL (http://opennel.org), OpenSkies (http://openskies.net), RakNet (http://www.raknet.net) and ReplicaNet (http://replicanet.com) provide basic networking services and include object replication, NAT punchthrough, message reliability and techniques for reducing bandwidth requirements. However, they provide little or no support for interpolation and extrapolation of shared data objects.

As game networking libraries provide limited support for programming with time, we look to other programming environments that have integrated time into the programming model.

6.2 Other programming environments for managing time

A number of programming languages explicitly incorporate time. 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 dataflow languages to express temporal constraints in user interfaces [14], 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. [26] have shown how constraints can be used to create animated interfaces. 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 languages 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 (http://www.adobe.com),

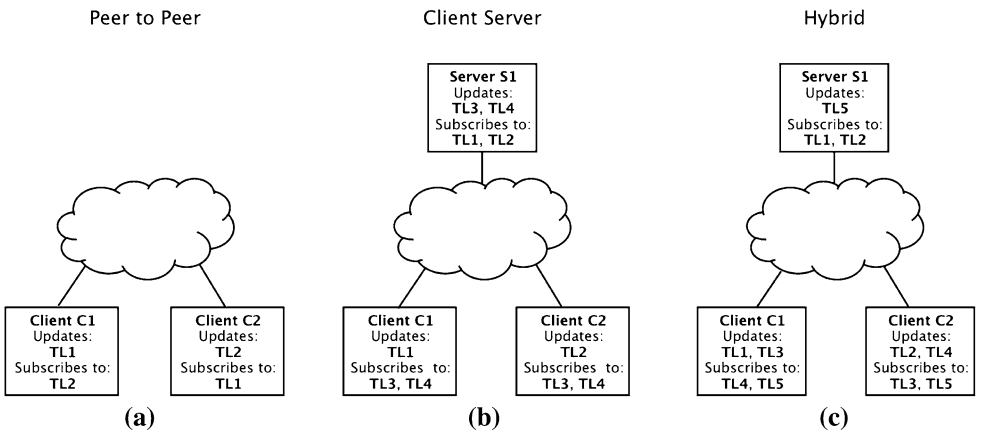

Fig. 10 The Janus toolkit allows for a variety of overlay architectures including peer-to-peer, client–server and a hybrid model depending upon the subscription list for each timeline

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 color) 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 [28] 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. We were able to draw from some of these concepts. However, the concept of embedding a database is impractical for real-time games where local replication and immediate access is required.

Calculating changes in shared state over time is a fundamental concept in distributed simulation [15]. Either a time stepped or an event-driven execution model may be used. Standards such as the IEEE standard for High Level Architecture [19] provide a protocol for object model interoperability which includes a time dimension and specifications for extrapolation (dead reckoning).

Timelines are perhaps closest to the programming abstractions offered by process historians such as OSISoft's PI System (http://osisoft.com) and AspenTech's InfoPlus.21 (http://aspentech.com). These 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 for programming with state and time 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. Process historians are tuned for the very different domain of process control, and are not designed for use in distributed systems.

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 networking 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, and applies these concepts to shared state data of the form used by networked games.

To the best of our knowledge, our timelines model and its implementation in the Janus toolkit is unique. It is the first programming model for shared state data that integrates time and state. By making time an explicit dimension of shared data objects, the timelines model makes it considerably easier to express a wide range of lag compensation algorithms used in multiplayer games.

7 Discussion

We now describe our experience using timelines, and the strengths and limitations of this programming model.

7.1 Experience

Despite its status as a research prototype, the Janus toolkit has been used by the authors and other developers to experiment with a variety of lag compensation algorithms and to create several multiplayer games based on Microsoft's XNA game development library. The games include the Balloon Burst, Truck Pull and Pedal Race exergames [35], the Eliminate 3-D first-person shooter, the Speed Demons racing game, the Growl Patrol ubiquitous game [23], and the Liberi persistent world building game [18]. The toolkit has also been used to implement the OrMiS tabletop military simulation tool [29].

These games were developed by ten 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 straightforward to implement networking using the Janus toolkit. For Balloon Burst, Truck Pull, Pedal Race, Growl Patrol and OrMiS, the time to incorporate networking was measured in hours. Liberi had difficult performance requirements due to its basis in a large, fully deformable world. For Liberi, Janus was used to implement interest management and distributed physics algorithms. Despite this, it was still just a matter of a few weeks to implement multiplayer support.

The developers using the Janus toolkit were primarily interested in creating simple networked games to be played over a local area network. They have mainly relied on prediction or local lag or a combination of these two techniques. The one exception was a fourth year undergraduate student who implemented three games for the purpose of evaluating the effect of different lag compensation techniques on player experience and performance. With each of these three games it was possible to switch between local lag, remote lag and prediction during game play. Also, the smooth correction technique was implemented in each game.

Although smooth corrections are relatively simple to implement with the Janus toolkit, most developers chose to not incorporate this technique in their games. One possible reason is that the games were only played over a local area network and thus jarring corrections were not an issue.

Incorporating smooth corrections as the default behavior for the remote update function for the standard timeline classes provided with Janus would increase the use of this technique even for the most novice developers.

With Janus, we have largely focused on the development of multiplayer games. However, timelines can readily be applied to distributed simulation and to groupware applications such as shared editors and drawing tools or chat applications [30].

Our experience indicates that timelines can make it easy to implement basic networking in multiplayer games. As we have seen, timelines also make it tractable to implement sophisticated algorithms. For example, we have experimented with the combination of the local perception filters [32] algorithm with smooth corrections [34] to provide a novel solution to the distributed physics problem. Thus far, we have created a credible simulation involving up to four players interacting with tens of objects over networks with up to 100 ms of latency. We are continuing to work on increasing the number of players and objects.

From this experience, we anticipate that the largest benefit of timelines is that they enable developers to quickly assess the benefits of different algorithms, to create new algorithms, and to combine existing algorithms in novel ways. While in theory such work is possible with traditional techniques, it is not always practical to do so given the hectic timeframes of commercial game development.

7.2 Player experience

Advanced lag compensation techniques show great promise for improving user experience while playing games. For example, in the Half-Life series of games [3], remote lag improves both player experience and performance by allowing players to aim directly at their targets, and the bucket synchronization algorithm was key to the implementation of Age of Empires [4]. However, it is not always clear which techniques provide the best fit for which game situations. As shown by work by Stuckel and Gutwin [36], Pantel and Wolf [27], Zhao et al. [37] and ourselves [31], the algorithms must be tested and then evaluated to determine their effect. To do this, the developer must select an algorithm, implement it and then evaluate its impact on player performance and experience in a range of game situations. If the technique is too difficult or cumbersome to implement, this creates a significant barrier to doing such evaluations. For example, local perception filters [32] first appeared in the literature in 1998, but (prior to the efforts reported in this paper) have never been implemented in any multiplayer game. Timelines make the implementation of these techniques tractable and thus can allow developers to experiment with different techniques and tailor the techniques used to specific situations within a game.

7.3 Strengths and limitations

The power of the timelines model lies in its explicit treatment of time. Automatic interpolation and extrapolation mechanisms allow programmers to easily access shared state data from any point in the past or future. This technique is powerful for manipulating shared data, although it is not without limitations. In our example of implementing local perception filters, we have shown how two clients can coordinate updating a single timeline representing the ball position. In general, however, the current timeline implementation does not support multiple clients updating the same timeline, as the updates from one client by default can overwrite updates made by the other client. Overriding the default remote update function can solve this synchronization issue; however, to-date this has been left to the developer using the toolkit to provide the implementation. In future versions of the Janus toolkit, we will provide a variety of options for remote update handling that will support synchronization techniques such as time warp [22], optimistic synchronization protocols [12], conflict merging and/or operational transform [10].

By default, timelines require the entire shared object to be sent over the network for each update. This makes them unsuitable for large data structures. We have begun to explore how timelines can be made more efficient by sending only changes to the shared state, as opposed to sending the entire object. Using customized serialization methods we are able to transmit only the portions of the data structure that have changed. Further work is required to generalize this solution for all timelines objects.

We have shown how the timelines model makes it easy to access shared state data at any point in time. However, the same is not true for command type data, such as “shoot” or “crouch”. There is no method to interpolate or extrapolate these types of actions and thus each command must be accessed individually. We have experimented with various options for integrating commands into our timelines model. Some options include, allowing the programmer to access a list of commands that occurred over a time range, or using an event-driven model for commands, possibly delaying events based on the command timestamp.

Our implementation of the global clock has been used successfully to synchronize clients on a variety of computers. We have found that over the local area network we are able to synchronize the clocks generally within a few milliseconds. Testing over wide area networks indicates that clock synchronization within the small tens of milliseconds is achievable. A more sophisticated algorithm may be required under conditions involving higher network latency and jitter. Also, we have not yet implemented safeguards to ensure that changes to the clock occur gradually and that the clock cannot move backward.

We have not seen a need for such safeguards in our studies to-date, but this may require consideration in the future.

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 history developed for solving the groupware latecomer problem [6], and mechanisms for compressing messages [17].

Timelines provide a novel programming model. For programmers familiar with message passing techniques, the shift to thinking about a shared state model indexable by time can be significant, perhaps analogous to the shift from procedural to object oriented programming. We have found that developers who dive into the model without carefully studying its documentation and examples make the mistake of trying to treat it like a message passing system. As with all novel programming models, developers need to adjust to the model's way of thinking.

8 Conclusion

In this paper we have presented the timelines programming model for lag compensation in networked games. Timelines facilitate the implementation of a variety of lag compensation techniques by making the treatment of time an integral part of the programming model. Timelines allow programmers to manipulate past and future values and control how state diverges over time for different players. Timelines have been implemented within the Janus toolkit, and were used to implement all examples presented in the paper. The Janus toolkit and documentation are available for download at http://equis.cs.queensu.ca/~equis/Janus.

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. Aggarwal, S., Banavar, H., Khandelwal, A.: Accuracy in dead-reckoning based distributed multi-player games. In: SIGCOMM'04 Workshops, pp. 161–165. ACM Press, New York (2004)
2. Ashcroft, A.E., Wadge, W.W.: Lucid, a nonprocedural language with iteration. CACM **20**(7), 519–526 (1977)
3. Bernier, Y.W.: Latency compensating methods in client/server in-game protocol design and optimization. In: GDC (2001)
4. Bettner, P., Terrano, M.: 1500 archers on a 28.8: network programming in age of Empires and beyond. In: GDC (2001)
5. Cai, W., Lee, F., Chen, L.: An auto-adaptive dead reckoning algorithm for distributed interactive simulation. In: Proceedings of the Thirteenth Workshop on Parallel and Distributed Simulation, pp. 82–89 (1999)
6. Chung, G., Dewan, P., Rajaram, S.: Generic and composable latecomer accommodation service for centralized shared systems. In: Proceedings of the IFIP 2.7 Working Conference on Engineering for Human-Computer Interaction, EHCI'98, Herkalion, Crete, 14–18 September 1998
7. Cristian, F.: Probabilistic clock synchronization. Distrib. Comput. **3**, 146–158 (1989). doi:10.1007/BF01784024
8. de Alwis, B., Gutwin, C., Greenberg S.: GT/SD: performance and simplicity in a groupware toolkit. In: Proceedings of the 1st ACM SIGCHI symposium on Engineering interactive computing systems, EICS'09, pp. 265–274. ACM, New York (2009)
9. Diot, C., Gauthier, L.: A distributed architecture for multiplayer interactive applications on the internet. In: Network, pp. 6–15. 1999
10. Ellis, C.A., Gibbs, S.J.: Concurrency control in groupware systems. In: SIGMOD '89, pp. 399–407 (1989)
11. Eugster, P., Felber, P., Guerraoui, R., Kermarrec, A.: The many faces of publish/subscribe. ACM Comput. Surv. **35**(2), 114–131 (2003)
12. Ferretti, S.: A Synchronization protocol for supporting peer-to-peer multiplayer online games in overlay networks. In: DEBS '08, pp. 83–94 (2008)
13. Ferretti, S.: Cheating detection through game time modeling: A better way to avoid time cheats in P2P MOGs? Multimedia Tools Appl. **37**(3), 339–363 (2008)
14. Freeman-Benson, B., Borning, A.: The design and implementation of Kaleidoscope'90, a constraint imperative programming language. In: Proc. IEEE Intl. Conf. on Computer Languages, pp. 174–180 (1992)
15. Fujimoto, R.M.: Parallel and Distributed Simulation Systems. Wiley, New York (2000). ISBN 0-471-18383-0
16. Gusella, R., Zatti, S.: The accuracy of the clock synchronization achieved by TEMPO in Berkeley UNIX 4.3 BSD. IEEE T-SE **15**(7), 853 (1989)
17. Gutwin, C., Fedak, C., Watson, M., Dyck, J., Bell, T.: Improving network efficiency in real-time groupware with general message compression. In: CSCW, p. 128. ACM (2006)
18. Hernandez, H., Graham, T.C.N., Fehlings, D., Switzer, L., Ye, Z., Bellay, Q., Hamza, A., Savery, C., Stach, T.: Design of an exergaming station for children with cerebral palsy. In: CHI, pp. 2619–2628 (2012)
19. IEEE standard for modeling and simulation (M&S) high level architecture (HLA)–object model template (OMT) specification. In: Proceedings of IEEE Standard 1516.2-2000 (2000)
20. IEEE standards for distributed interactive simulation—Application protocols. In: Proceedings of IEEE Standard 1278–1993 (1993)
21. IEEE standard for distributed interactive simulation—Application protocols. In: Proceedings of IEEE Standard 1278.1-1995 (revision of IEEE Std 1278–1993)
22. Jefferson, D.: Virtual time. ACM Transact. Program. Lang. Syst. **7**(3), 404–425 (1985)
23. Kurczak, J., Graham, T.C.N., Joly, C., Mandryk, R.: Hearing is believing: Evaluating ambient audio for location-based games. In: ACE 2011, pp. 32:1–32:8 (2011)
24. Mauve, M.: How to Keep a Dead Man from Shooting. In: Interactive Distributed Multimedia Systems and Telecommunication Services, pp. 199–204. LNCS 1905, Heidelberg (2000)

25. Mauve, M., Vogel, J., Hilt, V., Effelsberg, W.: Local-lag and timewarp: providing consistency in replicated continuous interactive media. IEEE Transact. Multimedia **6**(1), 47–57 (2004)
26. Myers, B., Miller, R., McDaniel, R., Ferrency, A.: Easily adding animations to interfaces using constraints. In: UIST '96, pp. 119–128 (1996)
27. Pantel, L., Wolf, L.C.: On the suitability of dead reckoning schemes for games. In: NetGames, pp. 79–84. ACM (2002)
28. Pelekis, N., Theodoulidis, B., Kopanakis, I., Theodoridis, Y.: Literature review of spatio-temporal database models. Knowl. Eng. Rev. **19**, 235–274 (2004)
29. Graham, T.C.N., Bellay, Q., Schumann, I., Sepasi, A.: Towards game orchestration: Tangible manipulation of in-game experiences. In: TEI 2012. ACM Press (2012)
30. Savery, C., Graham, T.C.N.: It's about time: Confronting latency in the development of groupware systems. In: Proceedings of the ACM 2011 conference on Computer supported cooperative work, CSCW '11, pp. 177–186. ACM, New York (2011)
31. Savery, C., Graham, T.C.N., Gutwin, C.: Human factors of consistency maintenance in multiplayer computer games. In: GROUP, pp. 187–196 (2010)
32. Sharkey, P.M., Ryan, M.D., Roberts, D.J.: A local perception filter for distributed virtual environments. In: Proceedings of the Virtual Reality Annual International Symposium, pp. 242–249. IEEE Computer Society Press (1998)
33. Singhal, S.: Effective remote modeling in large-scale distributed simulation and visualization environments. PhD thesis, Standford University (1996)
34. Smed, J., Hakonen, H.: Algorithms and Networking for Computer Games. Wiley, New York (2006). ISBN: 9780470018125
35. Stach, T., Graham, T.C.N.: Exploring haptic feedback in exergames. In: Human-Computer Interaction INTERACT 2011, pp. 18–35 (2011)
36. Stuckel, D., Gutwin, C.: The effects of local lag on tightly-coupled interaction in distributed groupware. In: CSCW, pp. 447–456. ACM (2008)
37. Zhao, S., Li, D., Gu, H., Shao, B., Gu, N.: An approach to sharing legacy tv/arcade games for real-time collaboration. In: Proc. ICDCS, pp. 165–172. IEEE (2009)