The Human Factors of Consistency Maintenance in Multiplayer Computer Games

Cheryl Savery[1], T.C. Nicholas Graham[1] and Carl Gutwin[2]

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

[2]Department of Computer Science
University of Saskatchewan
Saskatoon, Canada S7N 5C9
gutwin@cs.usask.ca

ABSTRACT

Consistency maintenance (CM) techniques are a crucial part of many distributed systems, and are particularly important in networked games. In this paper we describe a framework of the human factors of CM, to help designers of networked games make better decisions about its use. The framework shows that there is wide variance in the CM requirements of different game situations, identifies the types of requirements that can be considered, and analyses the effects of several consistency schemes on user experience factors. To further explore these issues, we carried out a simulation study that compared four CM algorithms. The experiment confirms many of the predictions of the framework, and reveals additional subtleties of the algorithms. Our work is the first to look comprehensively at the tradeoffs and costs of CM, and our results are a strong starting point that will help designers improve on the user's quality of experience in distributed shared environments.

Categories and Subject Descriptors

H.5.3 [**Information Interfaces and Presentation**]: Group and Organization Interfaces—Computer-supported cooperative work.

General Terms

Human factors, Algorithms

Keywords

Consistency maintenance, game development, game usability

1. INTRODUCTION

Consistency maintenance (CM) is the process of establishing and preserving a shared and equivalent state across two or more nodes in a distributed system [26]. Consistent representations of a shared environment are a critical requirement for many situations in distributed groupware and networked games: in a shared editor, discussing a text document requires that people see the same representation of the text; in a multi-player game, all nodes must agree on issues such as whether an enemy was hit by the player's attack, whether the player collided with an obstacle or not, and whether a trade was successfully completed. For these and many other situations, CM is a necessary part of the software design for a distributed environment.

The need for consistency in these and other situations, however, does not mean that CM should be generically applied across all aspects of a distributed application. The main reasons for being selective are that not all situations in a distributed interaction have the same consistency requirements as the examples above, and that CM techniques are costly, both in terms of development complexity and in terms of reducing the users' quality of experience. The second of these problems – the effects of CM on user experience – is the more important of the two, since it is possible for CM to cause as many new problems for user interaction as it solves. For example, CM techniques may add latency to the visual representation of another player's actions, causing problems for communication and coordination; they may incorporate correction actions which can be visually jarring and difficult for people to understand; and they can prevent access to objects in the environment, which can cause frustration and confusion.

These costs imply that designers should apply CM judiciously, carefully choosing CM techniques to match to the user experience requirements of a particular situation. To do this, designers need to know both the consistency requirements for the scenarios that will occur in their applications, and the effects of different CM schemes on different aspects of user experience. However, there is currently very little information that can guide design decisions in this area.

In this paper, we provide this information: we present a three-part framework that explores the human factors of CM for multiplayer games. The first part of the framework recognizes that different scenarios within a game can have very different consistency requirements, and specifies the requirements for several canonical game situations. Second, the framework discusses how the requirements can be met, details specific metrics for measuring user experience, and outlines the main characteristics of several different CM approaches. Third, we identify four specific game CM algorithms and test them in a simulation study. The study puts the relationship between CM approaches and user experience on an empirical footing, and confirms the analyses made possible by the framework – for example, that injecting local lag delay on processing user inputs reduces a game's responsiveness, and that using client-side predictive algorithms improves animation smoothness. The study also reveals interesting additional behaviours of the CM algorithms: for example, that variance in latency has a bigger negative effect on consistency than the latency itself, and that when using local lag

GROUP2010, November 7–10, 2010, Sanibel Island, Florida, USA.

techniques, the reduction in responsiveness and propagation delay can actually be smaller than the added lag.

Our framework and evaluation make three important contributions to the design of multi-user games and distributed collaborative systems: first, we establish that CM schemes must be carefully matched both to the consistency requirements and the user-experience requirements of particular interactive scenarios; second, we provide a framework that specifies requirements, aspects of user experience, and the characteristics of several main CM approaches; and third, we provide the first empirical study comparing the effects of different CM algorithms on user-experience metrics. The understanding and knowledge provided by this work can provide much-needed guidance to designers who must satisfy both the needs of the distributed system and the needs of the people who use it.

2. RELATED WORK

Although there has been relatively little research into the human factors of consistency management schemes, many other aspects of user experience in distributed groupware and multiplayer games have been studied in previous literature. Researchers have looked at issues such as helping people maintain awareness of others (e.g., [18]), how multi-player game usability can be assessed (e.g., [22]), determiners of social presence in distributed work (e.g., [20]), and the effects of latency and other network problems on communication and coordination (e.g., [2, 10]). In addition, there has been a great deal of work on specific concurrency and CM algorithms (e.g., [26, 19]); further details of some of these techniques are reviewed later in the paper. Both streams of this work are vital to groupware design, and form a foundation for our investigation of experiential issues in the domain of consistency.

The intersection of these areas has not received a great deal of attention from CSCW or the game research community, but prior work in has considered three substantial issues in the human factors of CM. First, an early analysis by Greenberg and Marwood [9] looked at consistency requirements for real-time distributed groupware, and introduced the idea that not all actions in a shared editor necessarily need to be governed by CM. We build directly on this idea, and expand on it in the framework described below. Second, a few researchers have looked at how specific algorithms can be used to deal with particular user experience problems: for example, preventing the "dead men shooting" problem using Time Warp [15], or using local lag to support situations that require tightly-coupled coordination [6, 25]. Third, research has looked at the effects of prediction on player interpretation and the costs of correcting erroneous predictions (e.g., [21]).

These few previous studies and analyses, however, do not provide a comprehensive understanding of when CM is needed in groupware or networked games, how designers can determine requirements for different interaction scenarios, what are the user-experience factors that designers need to preserve, or how various CM techniques can affect those factors. In the next sections, we provide a framework that provides initial understanding in this area, particularly for designers of multi-player games.

3. CONSISTENCY REQUIREMENTS

A consistency requirement for a particular situation implies that there are non-deterministic changes being made to some state that is both distributed and shared. To understand the consistency requirements for a particular interactive scenario, therefore, it is important to understand three aspects of multi-player games: the types of entities that can be in the shared environment, the types of interactions that players can have with those entities, and the types of consistency that could be required for an interaction.

3.1 Types of Shared Entities

At some level all entities in a multi-player game are equal – that is, they are all just entries in a database or objects in a model. However, different types of entities often have different consistency requirements, and so it is useful to group entities into broad categories. We have identified eight main groups that have proven to be useful in thinking about consistency.

- *Local avatar.* The local player's avatar is a special entity, since it is the player's embodiment in the world. Since the local avatar is the agent through which the player acts in the game, requirements for consistency will often be tighter than for representations of others' actions. For example, it is important to show fast and accurate feedback as a player moves in the world.
- *Other players' avatars.* The avatars belonging to other players may have different consistency requirements depending upon their relationship with the local player. Opponents will typically have stricter requirements than those of players cooperating with the local player. For example, in a shooter game there would be a low tolerance for inconsistencies in the target player's position.
- *Player variables.* Variables such as lives, health, and money are often critical to the objectives and outcome of the game, and so have stricter consistency requirements than other entities. For example, the simple understanding of whether a player is alive or not is of critical importance; failure of CM in this regard can have dramatic effects on the game [15].
- *Inventory.* There are often a large number of objects attached to a particular player, such as weapons or treasure obtained from previous interactions. These items are similar in many respects to those existing in the game world, but may have different requirements because their use is often restricted to a single player.
- *Objects in the world.* Games provide a wide variety of resources in the shared environment, from game pieces in chess, to moveable obstacles in a racing game, to treasure in an online RPG. There are a wide variety of possible consistency requirements for these entities, given their wide range of possible uses, and requirements are determined by their intended use (see below).
- *Terrain.* The world itself is comprised of shared entities (trees, rocks, ground, buildings, etc.) that can be modified in some games, which may require that the changes be propagated to other nodes. Examples include one player destroying a bridge or digging a hole that others can fall into.
- *Attack objects.* Objects used in attacks – such as bullets, spells, or punching fists – are generally different from other world objects, since they move (often quickly) and have important effects on other entities. Consistency requirements are often different (and stricter) for these objects. We note that in some games, bullets and fists are purely cosmetic since the hit decision is based on avatar positions at the start of the action; however, some games are moving towards modeling these attack objects (e.g., using the *Wii MotionPlus* for real-time control), and some games allow players to make use of ordinary objects in attacks (as with Half-Life's gravity gun).
- *Chat.* Written conversations are an important part of many games, and these statements, while not part of the game world itself, are also a set of data that must be shared across multiple nodes. Since chat is typically not editable, it has reduced consistency requirements compared with a traditional shared editor; but there could be requirements related to ordering of messages in the transcript.

3.2 Types of Interaction with Entities

The ways in which entities can interact is a main determiner of consistency requirements – not in terms of the actual actions that occur, but in terms of the dependencies and effects of these actions on entities that are important to the game. We have identified five factors that are common in many games; these allow designers to reduce the number of situations where CM needs to be imposed.

- *Is interaction possible with the entity.* The most basic question in determining consistency requirements is whether anything in the game can affect or change the entity. If not, there is no need to track or maintain the shared state of the object: for example, objects such as clouds in the sky of the game world may be completely independent from any actions, and so inconsistencies are unimportant.
- *Is interaction possible by multiple people at once.* It is also important to consider whether entities can be manipulated by just one person (e.g., an item in a player's inventory) or by multiple players (e.g., an item lying on the ground); interactions that can involve multiple people have stricter consistency requirements (e.g., ensuring that only one person can pick up the object).
- *Ability to affect critical game variables.* The degree to which interaction with an entity can affect important game variables (such as health or life) determines the need and type of CM. For example, the action of defusing a bomb can have a major impact on player life and health, and so must be more consistently represented across the nodes of those affected. This factor makes important distinctions between situations that on the surface look similar. For example, consistency requirements for interactions with other players are not all equal, since interactions with teammates have less effect on critical variables such as health and life (assuming no friendly fire kills) compared to interactions with (i.e., attacks on) enemies.
- *Timescale of future interactions.* The amount of time that will elapse before another person or object could possibly interact with the entity is a limiting factor on consistency requirements. For example, if a player wants to pick up an object from the environment, and no other avatar is near enough to cause a conflict, there is no requirement to impose CM on the interaction. Note that this limitation is not the same as simple proximity, since some types of interactions have effects at a distance (e.g., beam weapons can shoot instantaneously across infinite distance).
- *Probability of future interaction.* Consistency requirements are also affected by the likelihood that an interaction with an entity will occur in a certain time period. In situations where interactions are unlikely, the requirements for consistency are reduced. For example, if a player modifies the game world's terrain, but does so in a location that is rarely travelled, there is less of a need to guarantee that the modifications are copied immediately to each node. Reducing consistency based on probabilities can lead to problems – e.g., if another player does take the rare path, a problem could result – but using likelihood as the basis for applying CM techniques is reasonable when computational and network resources are constrained.

3.3 Types of Consistency Requirements

Consistency between two versions of a model can diverge in three main ways – in magnitude, in time, and in rate of change – and consistency requirements for a particular situation essentially translate to the allowable tolerance for each type of divergence. Specifically, requirements can be stated in terms of:

- *State divergence*, which specifies, for a given instant in time, how much two players' view of a model differ. For example, a situation where a group of player avatars moves from one place to another can tolerate a higher degree of state divergence (i.e., players will not be affected by positional inaccuracies as the group moves) than a first-person shooter game where players target each other with sniper rifles (and where divergence in positions can lead to incorrect kills).
- *Propagation delay*, which indicates how long it takes the game to bring all clients to a consistent state after a change on one client. Requirements for propagation delay indicate the amount of time that a situation can tolerate an inconsistency. In a first-person shooter, high propagation delay can lead to "dead men shooting" due to time to deliver a death decision. For game situations such as an avatar changing socks or transmitting a chat message, players can accept longer propagation delay.
- *Timeline divergence*, which specifies the extent to which two players' experiences of a phenomenon differ. For example, one player might view the casting of a "fireball" spell as three seconds of casting animation, followed by an explosion effect, followed by the deduction of hit points from the target. Another player on a slower connection might see a compressed timeline, where the casting animation lasts only one second. While both players see the same final effect, the timeline has diverged. If the length of the casting animation gives the player the opportunity to take counter-measures (such as trying to interrupt the caster), then such timeline divergence can have significant gameplay consequences. Another example of timeline divergence could be one player's view of another player's movement, where the exact path may diverge, possibly involving jarring corrections.

Some CM algorithms may also result in "collateral damage", where meeting a requirement on one of the three attributes listed above may degrade other aspects of player experience. For example, the popular local lag algorithm [25] reduces a game's responsiveness, while locking or serialization algorithms may reduce the smoothness of animation. Consistency requirements must therefore be considered in the context of broader player experience.

Poor choices about CM techniques can have significant impact on game usability. For example, a study of problems reported in 108 game reviews showed that 29 were reported as being insufficiently responsive to user input, and 42 reported unpredictable response to user's actions (e.g., as a result of poor hit detection) [22]. While these did not necessarily result from consistency maintenance problems, this data shows that the sorts of problems that arise from poor CM are considered to be serious flaws in games.

Finally, we note that there are interface and game-design strategies that can reduce these requirements, which we do not detail here. For example, one technique that reduces consistency requirements is the idea of object pointing, in which a player selects another player as the target of an action (e.g., a spell) rather than moving a cross-hair target; this selection-based targeting dramatically increases the tolerance for state divergence.

3.4 Consistency Requirements of Canonical Game Situations

Here we characterize several canonical game situations in terms of the concepts introduced above, in order to show how this part of the framework can be used to analyze and discuss the requirements of different game scenarios.

- *FPS death decisions.* First-person-shooter games have strong consistency requirements for making decisions about whether one player has killed another. This decision must be the same on all clients, and has a low tolerance for propagation delay, since the decision has important consequences for some players' fu-

ture actions (i.e., games do not want situations where dead men are allowed to keep shooting).

- *Trade.* Many games allow players to buy, sell, or barter items; these transactions must be consistent across all clients, but in most cases there is a larger tolerance for propagation delay (i.e., it is not a problem if the transaction is not propagated to all clients for a few seconds).
- *Feedback for a moving player.* Responsiveness is a critical part of a user's experience in a networked game, and so providing fast feedback about a local player's movements in the world is critical.
- *Other players' positions.* As introduced above, there are substantial differences in consistency requirements for representing others' positions, depending on their relationship to the local player. If the other player is an enemy in a shooter game, then there is low tolerance for state divergence (so that aiming and shooting can be carried out successfully). If the other player is not likely to be targeted (e.g., is on the local player's team) positional consistency is less important.
- *Clothing choice.* An avatar's clothes are generally not objects that other entities in the world can interact with, and so the consistency requirements for showing a change of clothing are low (but not non-existent, since the change should eventually be propagated to all clients). There are nonetheless game situations where these types of decorative changes could be important for the game (e.g., waving a red glove could be the signal to start a coordinated attack); in these special cases, the tighter consistency requirements will have to be recognized by the designer.
- *Physics effects.* Many games have physics engines that create realistic effects for in-game events (e.g., falling rocks, spinning cars, shattering glass or explosions). However, the particles and objects used in these visual effects generally do not interact with players (i.e., a shard of glass from a shattering window in F.E.A.R. will not harm or kill a player). As a result, the particles' locations need not be consistent across different clients.
- *Avatar animations.* Avatar-based games provide standard animations for actions such as walking, waving, or pointing. For example, a walking animation may use a looping track that plays the next frame every iteration of the frame loop. Since the positions of hands and feet often do not affect critical game variables (e.g., the hit-box for an avatar might be a rectangular box that ignores arms and legs, and so the position of limbs does not affect gameplay), there is high tolerance for state divergence in these models (e.g., one client might have an avatar in the middle of a stride animation, whereas another client has it at the end, but the location of the avatar is the same in both cases).

4. MEETING CM REQUIREMENTS

The previous sections have outlined the kinds of problems that arise due to difficulties in maintaining consistency, and have illustrated that different gameplay situations may have very different consistency requirements. In this section, we first introduce the technical side of the CM problem, indicating why no single algorithm is sufficient for balancing all CM requirements. We then introduce operational metrics that can be used to characterize the effects of specific CM algorithms on user experience, and finally introduce the core elements of CM algorithms typically used in games.

4.1 The CM Problem: Game Architectures

The difficulty of achieving consistency in multiplayer games is influenced by how games are architected. Games are almost universally built as client-server applications with partial replication. The problem is then to maintain consistency between the partial replicas - in the terms introduced above, state divergence is a consequence of inconsistent replicas, and propagation delay results from the time required to send data over the network.

Games are typically based on a *frame loop*, consisting of the three steps of polling input devices, updating game state, and rendering the frame. This loop is executed as quickly as possible: for example, to achieve a frame rate of 60 Hz, less than 17 ms is available for each iteration. This means that any complex calculations (e.g., complex simulations of physics or artificial intelligence) cannot be performed inside the frame loop.

In most games, the client implements the frame loop and the server implements the game logic and maintains the canonical game state. Since any network communication with the server takes milliseconds to resolve, server requests cannot be performed within the frame loop. Therefore, any shared data required to compute the next frame must be available on the client (via partial replication) and any communication with the server must be carried out asynchronously.

A primary reason why this architecture is widely used (over, for example, peer-to-peer architectures) is security. Game players have proven willing to devote extraordinary efforts to cheating [11], to the point of cracking encrypted client-server protocols and hand-modifying the compiled binary of the client. This means that clients cannot make decisions that affect the game state. For example, in a first-person shooter game, the client is not allowed to decide whether a shot has hit an opponent. Similarly, the client is not allowed to store "secret" information that the player should not have access to, such as the location of objects or players behind a wall. These restrictions naturally lead to an architecture where the client implements the user interface and performs actions that do not affect game state, while the server makes game-critical decisions (or audits decisions made by the client).

There are three main ways in which this architecture affects consistency maintenance:

- *Execution speed:* the required speed of execution of the frame loop implies that some data must be replicated to the client. This leads to state divergence when replicas fail to have the same value.
- *Update frequency:* updates are transmitted between client and server asynchronously, normally at a rate far slower than the frame loop's frequency. In general, this leads to higher propagation delay. For game objects changing their state in real time, this may lead to stuttering animation. Game architectures frequently use predictive algorithms to attempt to guess the current state of remote game objects (described further below).
- *Security:* Clients may not store any "secret" game-affecting data. This restricts the use of potentially helpful CM algorithms; e.g., the client cannot pre-fetch "secret" data that might be useful in the near future, and cannot reduce propagation delay by simulating the server's AI algorithms or carrying out game-affecting physics calculations.

4.2 Metrics

The architecture described above both causes consistency problems and constrains the space of solutions. Consistency problems can be addressed through CM algorithms (as will be discussed in section 4.3), and through game design decisions that reduce the need for consistency.

Algorithms necessarily trade off qualities of the player's experience. For example, the *local lag* algorithm [25] reduces state divergence at the cost of reduced local responsiveness. Predictive algorithms such as dead reckoning [21] improve smoothness of an-

imation and may reduce state divergence, but at the cost of timeline divergence when the predictive algorithm fails (e.g., exhibited as jarring warps of player positions.) It is important to note that to fix one set of usability problems due to poor consistency, we frequently need to make other aspects of the player's experience worse.

To allow us to more precisely assess how the choice of CM algorithm can affect user experience, we identify five performance metrics. The metrics are based on two factors: they correspond to commonly-seen measures of the performance of groupware systems and games, and they can be observed and computed in real games as they execute. Underlying the choice of these metrics is the hypothesis that each value, if allowed to become large, will in some circumstances negatively affect the player's quality of experience in the game. These metrics allow us (and others) to develop clear guidelines about the consequences of a particular CM choice. The metrics are:

- *Magnitude of state divergence* captures the degree of consistency between two players' views of part of the game state at a given time. For example, the state divergence of two players' views of a single avatar could be measured as the distance between the avatar's position on each client. An appropriate distance metric is required to measure how far the states diverge; for example, Euclidean distance might be used for positional information.
- *Magnitude of propagation delay* is the time from a player performing an action to other players' seeing the propagation of that action. In a first-person shooter, this could be the time between one player pushing "W" and others seeing that player's avatar move.
- *Corrections* counts the number and magnitude of modifications required to repair an incorrectly divergent state. This metric is associated with the *timeline divergence* requirement, as it measures the work needed to regain consistency in divergent timelines. For example, when dead reckoning [21] is used to estimate the position of moving objects, a correction (or *warp*) may be required to move the object to its correct location. As with state divergence, a distance metric is required to measure the magnitude of a correction.
- *Response time* represents the delay from a player performing an action to his seeing the result of that action. In a first-person shooter game, this might be the time between pushing the "W" button to seeing the avatar move forward.
- *Animation delay* measures the delay between on-screen updates of a particular game object's state. E.g., this might represent the time between positional updates of another player's avatar.

These measures are not independent of one another. For example, high animation delay increases response time [27], poor feedthrough time may lead to increased state divergence, and high state divergence may in turn lead to more severe corrections. Our simulation experiment (reported below) helps tease out the nature of these subtle relationships.

Ideally, each of these metrics has small values. The question of how small they have to be depends on the requirements of that interactive scenario, but there are general results that indicate when problems may arise. According to Schneiderman, in highly interactive tasks, response times become noticeable once they exceed 50–150 ms, while with less interactive tasks, delays of up to one second are tolerable [23]. According to Jay et al., propagation delay is noticeable starting at 50 ms, and differences in propagation delay become evident in 50 ms increments [12]. Based on analysis with the model human processor, Card et al. suggest that animation rates must be in the range of 10–20 frames per second (or 50–100 ms animation delay) for animation to be smooth [5]. Far higher animation rates are required for fast-moving images, unless additional effects are used (e.g., motion blur). Requirements for magnitude of state divergence and corrections are specific to the task being performed, as motivated by our examples from section 3.

4.3 Conceptual CM Approaches

Consistency maintenance in games draws on concepts from distributed databases and groupware, but also introduces several new ideas. Consistency approaches used in traditional groupware fall into four broad categories: locking, optimistic locking and transformation, serialization, and "social protocols".

- *Locking.* Locking and transaction-based approaches require a node to obtain a "lock" before carrying out changes to shared context. To obtain transaction-level serializability, algorithms such as two-phase locking are required [8]. Locking is used in game situations where transactional atomicity is required, for example, in a player trade consisting of a withdrawal and deposit operation pair.
- *Optimistic locking.* Optimistic approaches allow operations to be enacted locally on the assumption that conflicts between concurrent operations are rare. Optimistic algorithms are widely used in games to reduce response time: for example, player movement in first-person shooter games is generally performed optimistically. Optimistic algorithms differ in how they respond to conflicts: *rollback schemes* (such as TimeWarp [16] and ORESTE [13]) undo conflicting operations to an earlier correct state and then reapply operations in a canonical order; *operational transform* avoids rollbacks by transforming incoming operations so that the state converges [26]. Both approaches can lead to jarring corrections, and are inappropriate in some cases (e.g., where a death decision might be reversed retroactively.) Operational transform is rarely (if ever) used in games, as it favours fully-replicated architectures, and because the situations where optimistic algorithms are used in games rarely require OT's complex generality.
- *Message-level serialization.* A third approach commonly used in groupware is serialization [1, 7], in which messages are sent to a centralized serializer which broadcasts the messages to all clients in a consistent order. Serialization results in poor response time, as operations cannot be enacted locally until they have travelled to the server and back again. Serialization is also a poor choice in cases where the timing of messages is as important as the order [3, 28]. Serialization is widely used in online games, however, to determine a canonical order of actions for example, to determine which player shot first in a gunfight.
- *Social protocols.* The final traditional approach is to use no algorithm at all, relying instead on people's awareness of each others' activities and their willingness to avoid conflicts [9]. Social protocols are rarely (if ever) used for CM in games, however, because games are often competitive, and because the risk of cheating is ubiquitous [11].

In addition to these approaches, games have developed additional techniques that are less commonly found in traditional groupware: prediction, local lag, and remote lag.

- *Prediction.* Predictive strategies attempt to determine the position of an object in the game world, based on a model of the object's movement. A commonly-used method for prediction is dead-reckoning, where the recent locations of the object are used to determine the likely next location. Dead-reckoning assumes that objects will continue to move in their previous heading and

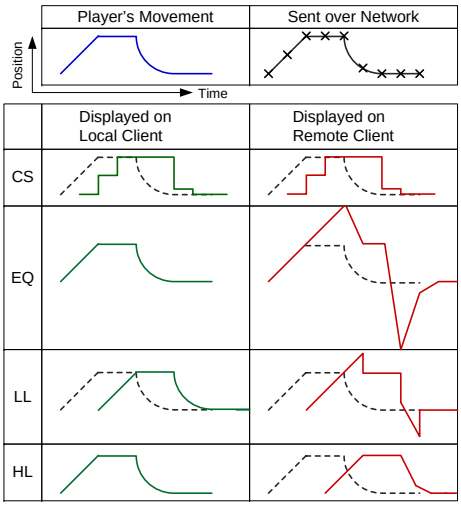

Figure 1: The four CM algorithms: CS=Centralized Serialization; EQ=EverQuest; LL=Local Lag; HL=Half-Life. Assuming that player position is a 1D value changing over time, latency is a constant 50 ms, LL adds 100 ms of lag time, and the HL algorithm adds 150 ms of lag on the remote client, shows how each algorithm would represent a given positional trace on the player's computer and on a remote player's computer.

velocity, which is often true for objects with inertia. Predictive algorithms can reduce state divergence (when predictions are correct), or increase it when incorrect. In addition, prediction can improve animation smoothness, but can lead to jarring corrections when errors are made.

- *Local lag.* State divergence is caused by the time it takes for updates to be sent across the network. This delay cannot be reduced, but the divergence can be eliminated by *adding* delay to the local input. For example, if a user presses the "W" key to move forward, they will not see their avatar move until after the delay period. Local lag makes it possible to synchronize the local and remote views of the world. When using local lag, developers must choose between a lag constant that balances the penalty to responsiveness versus the possibility of messages arriving late (i.e., after the lag period has expired.) A variant on local lag is *bucket synchronization* [3], which is used in games such as the popular Age of Empires series. In bucket synchronization, inputs performed in a given frame are collected and applied at the beginning of a subsequent frame.
- *Remote lag.* This strategy is a buffering approach designed to allow more updates to arrive at the remote client before they are used. This approach is widely used in streaming-media applications such as voice over IP, but is also used in a few multiplayer games. By allowing two positional updates to arrive before the first position is displayed, the client can accurately calculate a smooth animation (using interpolation between the two updates). This reduces jarring corrections, but at the cost of increasing state divergence.

5. SPECIFIC CM ALGORITHMS

The specific algorithms used in games are often a combination of the conceptual approaches introduced above. Here we identify four typical algorithms that we will use in the study described below. Three of the algorithms are derived from actual games, and one is a simple baseline approach. Figure 1 gives a simplified view of the operation of each of the algorithms.

In section 6, we evaluate these algorithms in the context of player movement, a canonical game situation that was described earlier in the framework. In this scenario, the CM problem is that of ensuring that the positions of different players' avatars are consistent on all clients. This is a central problem, since player position is crucial to navigation, aiming, and coordination of player actions. It is also a technically demanding problem – the designers of the game *Lineage 2* claim that player movement accounts for upwards of 70% of all network traffic in online role playing games [14]. Player movement is a strong example of a situation where traditional CM algorithms are not applicable.

5.1 Centralized Serialization

Our first algorithm follows the serialization approach described above. By itself, this approach is rarely used in production games, but we include it here to provide a baseline comparison. In this algorithm, movement updates are sent to the server, where they are validated and then broadcast to all clients. On each client, the player's position is reported as the last positional update received from the server. As can be seen in Figure 1, this algorithm exhibits a low update rate, due to the step function representing the last value received from the server. Centralized serialization might work reasonably over a local-area network, but gives unacceptable responsiveness and animation delay in a wide-area context.

5.2 Centralized Serialization + Optimistic Updates + Prediction: EverQuest Algorithm

This algorithm, used by the popular massively-multiplayer online game *EverQuest* (as well as many other games), introduces optimistic local updates and prediction using dead reckoning [21]. The EverQuest algorithm has not been published, but has been reverse-engineered by the open-source *ShowEQ* project [24].

In this algorithm, updates to the player's own position are optimistically committed locally, allowing smooth and timely local feedback (see Figure 1); these optimistic updates can be overruled through post-hoc server validation. On the remote client, position updates are received from the server, and therefore appear later than on the local client (i.e., increased state divergence). To avoid the step-function effect of centralized serialization, dead-reckoning is used to predict the position of remote avatars by assuming that they will continue to move in their previous heading and velocity. Figure 1 shows that although dead-reckoning provides smoother animation updates, it can lead to jarring corrections when an avatar abruptly changes direction (i.e., the spikes in the figure).

5.3 Local Lag + Prediction

This algorithm is one of a family of variants of local lag, and addresses the large state divergence and corrections of the EverQuest algorithm. Local input is delayed so that, despite delay from network latency, the local and remote clients see the same action at the same time. In addition, the remote client uses prediction to smooth avatar movement between updates. Figure 1 shows how local and remote movement are synchronized, lagging behind actual movement. Due to variance in packet delivery times, messages will sometimes fail to arrive within the bound of message delivery times. In such cases, standard repairs can be applied; for example, Mauve shows how TimeWarp can be combined with local lag [16].

Local lag (and its bucket synchronization variant) are a good choice for real-time strategy games where a delay in applying inputs is not easily noticed; it is a poor choice for first-person shooters where poor response time is more visible.

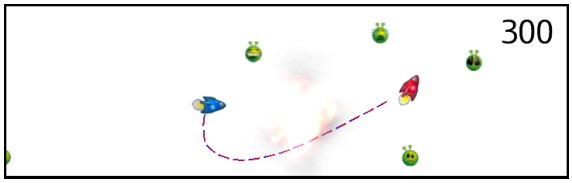

Figure 2: In *Snagger*, two players each control a ship, and cooperatively drag a net to catch aliens. Snagger was used to generate example input data used to compare the four CM algorithms of section 5.

5.4 Remote Lag + Interpolation: Half-Life Algorithm

The algorithm used in the *Half-Life* series of first-person shooter games addresses the problem of timeline divergence of predictive algorithms, at the cost of state divergence [2]. The key elements of this approach are *remote lag* and *interpolation*.

Local inputs are applied immediately to optimize responsiveness. Remote inputs are delayed long enough for *two* positional updates to arrive. Prediction is not required, since the algorithm interpolates between the two pending updates. Players thus view remote avatar movement exactly as it occurred (within the limits of interpolation), but offset in time.

To avoid the problems of increased state divergence, the server resolves hit decisions based on the shooting player's view of the remote avatar. This means that a player may be killed despite having just moved behind cover (if another player saw them clearly on their local machine).

We now turn to the study that we carried out to test these four CM algorithms in actual game situations, and how they affected our five metrics of user experience.

6. SIMULATION EXPERIMENT

In order to test these different CM approaches in terms of the user-experience metrics defined in the framework (section 4.2), we carried out a simulation experiment. The experiment helps illustrate that user experience tradeoffs between CM algorithms can be measured empirically through our metrics, and it serves as a first step towards our goal of cataloguing such tradeoffs to help game designers. As we shall see, our experiment confirmed expected properties of the algorithms, as well as revealing some surprising properties.

6.1 Experimental Design and Methods

The experiment consisted of ten simulation trials of each of the four algorithms under three different latency conditions. The algorithms introduced above were fully implemented, and were tested using recorded game traces and recorded latency values. The five metrics of section 4.2 were measured under each condition.

6.1.1 Apparatus: Snagger Game

To collect game data for the simulations, we used *Snagger*, a simple network game shown in figure 2. Snagger is an example of a game requiring close coordination between two players. The game consists of two clients and a server which passes messages between the two clients. Each player has a spaceship avatar, which is moved forward using the up arrow key and rotated using the left and right arrow keys. The objective of the game is for the players to move their ships around the screen and capture aliens in a net that is strung between the two ships. To capture an alien, the players must manoeuvre their ships so that an alien is inside the loop formed by the net and then press the space bar. Snagger was developed using C# and XNA Studio 3.1.

To provide consistency between trials with the four algorithms, we instrumented Snagger to run using user input read from a data file. The data file contains a sequence of keyboard states, which Snagger reads as if the inputs were coming from a real keyboard. Prior to the experiment, input data was collected from live play of the game (by players unassociated with this paper), over a local area network. During the game play, their inputs were captured by recording which keys were being pressed each time the input was polled. Ten sets of data were collected, each consisting of 1,500 inputs. Each data set corresponded to approximately 25 seconds of play.

6.1.2 Apparatus: Simulated Network Latency

The experiment simulated three network conditions: zero latency, low latency and high latency. The latency conditions attempted to approximate the conditions one might experience when playing a game on real game servers, where zero latency is similar to LAN play, low latency is similar to a server on one's own continent, and high latency corresponds to a server on another continent. (Although the same input was played back for all simulated network latencies, it is quite likely that players would adjust their behaviour as the latency increases, and that the inputs may not correspond exactly to what one would experience in a high-delay situation).

The protocol used by most network games (including Age of Empires, EverQuest and Half-Life) is UDP [17]. In the experiment, we simulated UDP's latency and packet loss rate. CAIDA, the Cooperative Association for Internet Data Analysis, has reported that "ping" round trip time data matches well with the round trip time for UDP [4]. Based on this, we used "ping" round trip time data to approximate UDP latency. The servers yahoo.com and yahoo.cn were selected as representative of a server on the same continent and of a server on another continent (based on our North American location.) "Ping" data was recorded for a three hour period for each server. Analysis of the data for round trip times showed that it was distributed according to a Poisson distribution with median values of 33 ms and 323 ms for yahoo.com and yahoo.cn respectively.

The "ping" file was split into 10 parts, each long enough to be usable for one simulation trial. In each trial, the system loaded files for the game data and network latency; as each message was sent from the client to the server and back to the other client, the message was delayed by a time equal to the next round trip time from the file.

6.1.3 The Algorithms

The four algorithms were implemented as described in section 5. In the Half-Life game, a message containing the player's position is sent to the server every 50 ms [2]. For consistency, we adopted the same frequency for the other algorithms.

The local lag and Half-Life algorithms required us to choose values for their remote and local lag parameters.

The choice of a local lag constant should strike a balance between the negative effects of the lag itself on responsiveness of the user interface versus the state divergence and corrections resulting from late messages [16]. We chose a lag constant of 100 ms, based on Stuckel and Gutwin's findings that this is the largest tolerable lag in highly interactive tasks [25].

For Half-Life, the remote lag value should ensure that at least one future position is available at all times, allowing the position of the remote avatar to always be interpolated. In practice, it is impossible to set an upper bound on the time it may take for a message to arrive. We therefore set remote lag equal to twice the

Test Case			Results				
Alg.	Lag (Local/ Remote)	Median Network Latency	State Diverg. (%)	Prop. Delay	Corr./ sec. (%)	Resp. Time	Anim. Delay
CS	0/0	0	0.14	29	6.4	18	49
EQ	0/0	0	0.08	29	0.4	1.5	17
LL	100/100	0	0.04	78	0.0	90	17
HL	0/100	0	0.68	78	0.0	1.5	17
CS	0/0	33	0.33	63	6.5	58	50
EQ	0/0	33	0.10	64	0.7	1.5	17
LL	100/100	33	0.06	80	0.1	88	17
HL	0/166	33	1.09	145	0.0	1.5	17
CS	0/0	323	2.10	346	8.3	342	66
EQ	0/0	323	0.50	345	3.8	1.5	17
LL	100/100	323	0.30	343	2.7	94	17
HL	0/746	323	4.55	725	0.0	1.5	17

Figure 3: Results summary. CS=Centralized Serialization; EQ=EverQuest; LL=Local Lag; HL=Half-Life. Times in ms. Within each latency condition, all differences in the table are significant at the $\alpha = 0.05$ level, except propagation delay for the CS and EQ algorithms under the medium and high latency conditions.

median network latency (as described above), plus (to allow for occasional lost messages) twice the time between message sends (2 * 50 ms in our case). The remote player lags used in the experiment were therefore set at 100, 166 and 746 ms for zero, low and high latency respectively.

For the EverQuest and Centralized Serialization algorithms, no additional lag was used.

6.1.4 Metrics

The five metrics were calculated as follows:

Response time is calculated by storing the time each input is polled and then determining the first time at which that input has an impact on the displayed position of the local avatar. Our measure does not include operating system time (the time that elapses between when a key is pressed and when the input is polled) or the time between invocation of screen update operations and the appearance of new information on the physical display.

Propagation delay is calculated by storing the time associated with each input by the remote player and then determining the first time each input has an impact on the displayed position of the remote avatar on the local computer. As with response time, it does not include operating system time.

Corrections are calculated each time the local player receives a message containing an updated position for the remote player's avatar. The estimated position of the remote player's avatar is calculated just before the new position message is processed. A second position is then calculated just after the new position message has been processed. The distance warped is calculated as the distance between these two positions and is expressed as a percentage of the screen size.

State divergence is calculated as the difference between the positions of the remote avatar on the local and remote client. State divergence is also expressed as a percentage of the screen size.

Animation delay is the time between changes in the position of the remote avatar. Whenever a screen update is invoked, if the position of the remote avatar has changed, then the elapsed time since the last position update is recorded.

6.1.5 Experimental Procedure

Each of the four algorithms was tested under each of the three latency conditions, for a total of 12 conditions. Each condition was tested ten times, and each condition used the same ten input game-data files and latency files, as described above.

For each trial, two instances of the Snagger client and the Snagger server were run on a single computer (2.66 GHz Intel i5 processor, 3 GB RAM). Each trial used different input and latency files. The computer was rebooted between each trial. Values for the metrics over each trial were determined and their values recorded.

6.2 Results

The results from the experimental runs are summarized in figure 3. All the metrics have been designed such that a small value is good and a larger value indicates poorer performance.

As shown in Figure 3, centralized serialization has the highest animation delay, since it does not extrapolate or interpolate avatar positions between updates coming from the network. The EverQuest and Half-Life algorithms had the lowest response times, since they perform immediate optimistic updates. Local lag has the highest response times (because of the injected local lag), and the lowest state divergence (as updates are synchronized on all clients.) Half-Life has the lowest corrections value (as it uses interpolation rather than predictive extrapolation); it also has the highest state divergence, due to the injection of remote lag without corresponding local lag.

The results allow us to summarize the strengths and weakness of the four algorithms as applied to the avatar movement problem:

- *Centralized serialization* has, as expected, no obvious benefits, performing poorly under all metrics. The performance of this algorithm clearly shows the hazards of applying generic CM strategies for specific game situations.
- The *EverQuest* algorithm provides a balance between all of the metrics - one of the main advantages of this algorithm is that it has no significant weaknesses. It is an appropriate choice for games (such as EverQuest itself) where modest state divergence and occasional corrections are considered acceptable. It would not be appropriate, however, for first-person shooter games.
- *Local lag* is a good choice in situations where it is important to have low state divergence and few corrections, and where a commensurate penalty in response time is acceptable. For example, local lag is a good match with the indirect control of real-time strategy games (where target positions of game entities are specified via "click to move" input).
- The *Half-Life* algorithm sacrifices state divergence for corrections. This algorithm is a good choice in cases where fast response time is required, and where it is important that all players see the same thing (although displaced in time.)

6.3 Observations and Analysis

The experiment provides several lessons for developers of consistency maintenance algorithms.

Combining different approaches can improve user experience. The example of the centralized serialized algorithm shows that using a single CM strategy can lead to flaws in the representation of motion (e.g., a very low update rate for avatar animation). However, these weaknesses can be reduced by adding prediction to the scheme (as is done in the EverQuest algorithm). Prediction is potentially a valuable adjunct to several other types of CM techniques, but we note that prediction approaches only work well when players move in a predictable fashion – for example, prediction is used for EverQuest, in which player movement is primarily used to

travel from one location to another. Prediction is not used in most shooter games, however, because players generally try to move in ways that make them difficult to hit, which is also much less predictable – the result would be large corrections in player locations, something that would not be acceptable in this type of game.

All approaches break down at a certain level of latency. The experiment shows that all the algorithms have at least one major weakness when round-trip latency grows above about a third of a second. These results reinforce that CM techniques do not compensate for latency, they only try to protect certain user or system requirements from latency's effects. However, game design can make up for some of these limitations, such as by increasing the time scale of interaction, by reducing the precision of the weapons (no sniper rifles, only shotguns), or by slowing down movement.

Variance in latency can be worse than latency itself. As latency increases, so does the variance in message delivery times. This results in negative consequences for each of the algorithms. Under centralized serialization, all clients are sent the same updates at the same time; however, messages arrive at the clients at different times, causing significant state divergence. Variance in latency increases the difficulty of setting local and remote lag parameters (these tradeoffs are discussed below.) A further effect is that animation delay under centralized serialization increases with latency. This is surprising, as messages are sent at a constant 50 ms rate, and so one might expect (as seen in the zero latency case) that animation delay would be close to 50 ms. However, due to variance in latency, it is possible for two position updates to arrive within the same interval between frame loop updates, causing the first update to be effectively lost. As the variance in latency increases, this occurs more frequently, leading to increased animation delay.

Propagation delay comes from more sources than network latency. When building games as distributed systems, it is easy to focus on network latency as the primary source of propagation delay. However, as seen from the results for centralized serialization and EverQuest in the zero latency case, other sources can result in significant delay. These include (1) message send delay: since positions are sent periodically (e.g., every 50 ms) and not as they occur, there can be a delay between a change being made and a change message being sent; (2) frame-loop delay: if the frame loop executes once per 17 ms, there is a delay of up to 17 ms from a message's arrival to its being processed by the game; and (3) operating system delay: such as the time for messages to be processed through the TCP/IP stack, and time for the network reading thread to be scheduled.

Adjusting algorithm parameters can enable helpful tradeoffs. Our experiment used particular fixed values for each algorithm's parameters, but adjustments to these parameters can change the tradeoffs inherent in each approach. For example, the local lag algorithm in our experiment added a constant 100 ms lag. However, any amount of lag can be used in this scheme, which gives the designer a tradeoff between state divergence and local responsiveness. More local lag reduces divergence, and less improves responsiveness. It is therefore possible to use local lag to reduce some of the divergence, without sacrificing all of the local player's ability to control their character. The appropriate settings for each algorithm depend (once again) on the human factors of the interaction: for example, adding 50ms of local lag might not be noticed by players, but might bring the state divergence within the tolerance that is needed for a game situation. It is interesting to note than when interpolation is used, response time and propagation delay can actually be *less than* the injected lag. This is because the effect of a queued positional update influences interpolation.

7. DISCUSSION

Consistency maintenance is a filter through which all player-to-player interactions are mediated. The framework we have presented in this paper helps to capture the costs that different CM choices impose on the player's experience. We now reflect on lessons for game developers from these results, and present avenues for future research.

7.1 Lessons for Game Designers

Although this work only provides a small part of a comprehensive treatment of how CM can affect user experience in games, there are still a number of lessons that we can suggest to game designers. The specific strengths and weaknesses of the four algorithms that we tested have already been summarized, and designers can use these guidelines when designing for avatar movement. The conceptual framework and our experiences using it also suggest four general lessons for game designers:

- First, game designers must recognize that CM techniques have real effects on a user's quality of experience, that different game situations have different consistency requirements, and that analysing these requirements is a critical part of designing the game *experience*, not just part of designing the distributed system.
- Second, game designers should understand that there are a wide variety of CM techniques and approaches, and that the requirements of many situations can often be satisfied with lightweight techniques or small changes to the design of the game. The key principle is that the choice of CM technique (or techniques) should be appropriately matched to both the consistency requirements and the user-experience requirements of the situation.
- Third, games can be designed around the limitations of CM algorithms. The disadvantages of a CM algorithm can in many cases be minimized through careful design of the interaction in the game. For example, the Half-Life algorithm is tuned for realistic and smooth movement of remote avatars, rather than for minimizing spatial divergence: this means that other players will move naturally with few corrections, but will appear behind their true position. This strategy can cause severe problems for hit detection, but the game designers avoid the problems by calculating hits based on what the shooter can see, not on the target avatars' true positions This game-design strategy preserves the shooter's user experience, but allows targets to be killed even when they were not really in the line of fire. One reason for favouring shooters over targets is that shooters have unequivocal feedback about aiming (i.e., the crosshair), and there is therefore much more frustration for shots that should hit but don't, than there is for shots that do hit but shouldn't have.
- Fourth, designers should be aware that there is still much to know in this area, and that there are still few guidelines for scenarios other than avatar movement. For these situations, designers must still carry out their own investigations (and adaptations) based on their understanding of the different CM approaches, and test the effects in their own iterative playtesting.

7.2 Future Work

There are a number of areas where further work is needed in understanding the human factors of consistency maintenance. First, our work here has examined four CM algorithms in one typical game situation - but much more is needed before designers will have a full set of guidelines that they can draw on when developing games. Therefore, we plan to expand our investigation to cover additional game scenarios, different CM approaches derived

from other existing games and whether a single game could take advantage of multiple CM strategies based on different observable network properties or the intensity of interaction at a given point in time.

Second, we plan to replicate the results seen here with other games, to deepen our understanding of the material that is already in the framework, and to help determine if the guidelines are general across different genres.

Third, we plan to refine and expand the conceptual understanding of CM requirements and user experience metrics, through additional studies and surveys of game players, and additional analysis of game mechanics, types, and genres. We believe that there is room in the framework for additional kinds of requirements (e.g., synchronization between action and speech) and additional metrics for assessing algorithms.

Fourth, we will continue experiments in the style of Stuckel [25] to help quantify metric values which represent tipping points in user experience.

8. CONCLUSION

Designers have had little guidance in choosing appropriate consistency maintenance techniques for distributed multi-player games. To help address this problem, we presented a framework of the human factors of CM: we identified requirements for distributed interaction, established metrics that can be used to quantify user experience effects, and analysed the effects of several CM approaches on these metrics. We conducted a study to test the predictions made by the framework, and gathered empirical evidence about the tradeoffs inherent in four real-world CM techniques. The framework and the empirical evidence for the tradeoffs is a strong start towards the goal of providing a comprehensive understanding of different CM techniques for a wide variety of game situations. Much more remains to be done, but our work shows that CM schemes can be successfully analysed and tested, and shows that we can establish guidelines that will help designers make informed decisions when consistency requirements and user-experience requirements collide.

ACKNOWLEDGMENTS

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

9. REFERENCES

[1] J. Begole, C. Struble, C. Shaffer, and R. Smith. Transparent sharing of Java applets: A replicated approach. In *UIST*, pages 55–64. ACM, 1997.

[2] Y. W. Bernier. Latency compensating methods in client/server in-game protocol design and optimization. In *Game Developers Conference*, 2001.

[3] P. Bettner and M. Terrano. 1500 archers on a 28.8: Network programming in Age of Empires and beyond. In *Game Developers Conference*, 2001.

[4] CAIDA. Packet matching for NeTraMet distributions. `http://bit.ly/9evQAj`.

[5] S. Card, T. Moran, and A. Newell. The model human processor – An engineering model of human performance. In *Handbook of perception and human performance.*, volume 2, pages 45–49. Wiley, 1986.

[6] L. Chen, G. Chen, H. Chen, J. M. S. Benford, and Z. Pan. An HCI method to improve human performance reduced by local-lag mechanism. *IwC*, 19(2):215–224, 2007.

[7] G. Chung and P. Dewan. A mechanism for supporting client migration in a shared window system. In *UIST*. ACM, 1996.

[8] K. Eswaran, J. Gray, R. Lorie, and I. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):633, 1976.

[9] 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. ACM, 1994.

[10] C. Gutwin. The effects of network delays on group work in real-time groupware. In *ECSCW*, pages 299–318, 2001.

[11] G. Hoglund and G. McGraw. *Exploiting online games: cheating massively distributed systems*. Addison-Wesley, 2007.

[12] C. Jay, M. Glencross, and R. Hubbold. Modeling the effects of delayed haptic and visual feedback in a collaborative virtual environment. *TOCHI*, 14(2):8, 2007.

[13] A. Karsenty and M. Beaudouin-Lafon. An algorithm for distributed groupware applications. In *Proc. ICDCS*, pages 195–202, 1993.

[14] J. Lee. Considerations for movement and physics in MMP games. In *Massively Multiplayer Game Development*, pages 275–289. Charles River Media, 2003.

[15] M. Mauve. How to Keep a Dead Man from Shooting. In *Interactive Distributed Multimedia Systems and Telecomm. Services*, pages 199–204. LNCS 1905, 2000.

[16] 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.

[17] S. McCreary and K. Claffy. Trends in wide area IP traffic patterns. Technical report, CAIDA, 2000.

[18] R. J. Moore, E. C. H. Gathman, N. Ducheneaut, and E. Nickell. Coordinating joint activity in avatar-mediated interaction. In *CHI*, pages 21–30. ACM, 2007.

[19] J. Munson and P. Dewan. A flexible object merging framework. In *CSCW*, pages 231–242. ACM, 1994.

[20] K. Nowak and F. Biocca. The effect of the agency and anthropomorphism of users' sense of telepresence, copresence, and social presence in virtual environments. *Presence: Teleoper. Virtual Environ*, 12(5):481–494, 2003.

[21] L. Pantel and L. C. Wolf. On the suitability of dead reckoning schemes for games. In *NetGames*, pages 79–84. ACM, 2002.

[22] D. Pinelle, N. Wong, and T. Stach. Heuristic evaluation for games: usability principles for video game design. In *CHI*, pages 1453–1462. ACM, 2008.

[23] B. Shneiderman. *Designing the user interface: strategies for effective human-computer interaction*. Addison-Wesley, 1997.

[24] ShowEQ project. `http://www.showeq.net`.

[25] D. Stuckel and C. Gutwin. The effects of local lag on tightly-coupled interaction in distributed groupware. In *CSCW*, pages 447–456. ACM, 2008.

[26] C. Sun and D. Chen. Consistency maintenance in real-time collaborative graphics editing systems. *TOCHI*, 9(1):1–41, 2002.

[27] C. Ware and R. Balakrishnan. Reaching for objects in VR displays: lag and frame rate. *TOCHI*, 1(4):331–356, 1994.

[28] S. Zhao, D. Li, H. Gu, B. Shao, and N. Gu. An Approach to Sharing Legacy TV/Arcade Games for Real-Time Collaboration. In *Proc. ICDCS*, pages 165–172. IEEE, 2009.