Plug-Replaceable Consistency Maintenance for Multiplayer Games

Robert D.S. Fletcher, T.C. Nicholas Graham and Christopher Wolfe
School of Computing
Queen's University
Kingston, ON, Canada, K7L 3N6
{fletcher,graham,wolfe}@cs.queensu.ca

ABSTRACT

Consistency maintenance of replicated data in multiplayer games is a challenging issue due to the performance constraints of real-time interactive applications. We present an approach which separates game logic from consistency maintenance code through the use of reusable, plug-replaceable concurrency control and consistency maintenance (CCCM) modules. Using plug-replaceable consistency maintenance strategies also permits rapid comparisons of multiple approaches, which facilitates experimentation. We conduct a case study to illustrate how multiple consistency maintenance strategies can be applied without changing the original game code.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques—*modules and interfaces, object-oriented design methods.*

General Terms

Measurement, Performance, Design, Experimentation

Keywords

Consistency maintenance, multiplayer game, workspace model

1. INTRODUCTION

One of the biggest challenges in developing online games is maintaining consistency of different players' views of the shared world. In server-based games we use the term *fidelity* to represent the degree to which a player's view of the world corresponds to the canonical view of the server.

Games' requirements for fidelity depend on the kind of activity that the player is carrying out. High fidelity operations (such as trading) require correct results no matter what the cost to the game's responsiveness. Alternatively, compromises to fidelity may be acceptable in order to improve the game's responsiveness, such as in the case of non-player character movement. The effects of fidelity errors can range from annoyance and inconvenience all the way to game-breaking bugs.

Server-based games rely on replicating some or all of the game's state on both the client and server. *Consistency maintenance* algorithms ensure that changes to one copy of the data are reflected in the other replicas. Consistency maintenance algorithms make different tradeoffs between fidelity and responsiveness. Games therefore may employ multiple consistency maintenance algorithms addressing the different fidelity requirements of the games' subsystems.

Consistency maintenance algorithms are often complex, and therefore hard to program correctly. In fact, errors in consistency maintenance code have been cited as the primary cause of item duplication (or "dupe") errors in massively multiplayer online games [1]. In this paper, we present a novel technique for consistency maintenance based on the Workspace Architectural Model.

This model provides three advantages over hand-coding of replica consistency maintenance. First, the programming of consistency maintenance is separated from the game application itself, simplifying the main application code. Second, multiple plug-replaceable consistency maintenance schemes can be used in the same application, facilitating experimentation. Finally, consistency maintenance algorithms can be collected and reused in future applications.

In order to demonstrate our approach, we have constructed a consistency maintenance testbed and used it to evaluate the tradeoffs between three consistency maintenance algorithms.

2. CONSISTENCY MAINTENANCE IN GAMES

Multiplayer games are distributed systems consisting of both replicated and centralized data. In multiplayer games using the client-server model, the server maintains the canonical state of the game world. When clients connect to the server, parts of the game state are copied to the client from the server, allowing updates to be presented to the player more quickly. Where possible, the client works with its local copy of the game state, and changes made to its version are applied to the canonical state on the server. The server's copy of game data is always considered definitive. There are many approaches to consistency maintenance, each providing varying degrees of fidelity. It is important to consider the interactions that game data supports when choosing an algorithm to maintain its consistency.

2.1 Aspects of the Player Experience

Consistency maintenance algorithms embody tradeoffs between different performance attributes. Important among these are fidelity, feedback time, degree of warping, and animation rate. Different types of game data require different tradeoffs, so a single game might best use a variety of consistency maintenance schemes.

- *Fidelity.* The *fidelity* property captures the degree to which the client's representation of the game world matches the canonical world state. Game actions such as trading or purchasing goods as well as decisions with respect to character death must correctly reflect server state. Yet, maintaining a faithful representation requires high interactivity with the server which is often prohibitively expensive. Fidelity can be sacrificed in favour of reducing network traffic and improving the more time-dependent properties of feedback time and animation rate.

- *Feedback Rate.* Feedback time is the delay between a player performing an action and seeing the results of that action. For any real-time interactive application (such as a game), minimizing feedback time is a primary concern. Feedback time can be aided by optimistic consistency maintenance strategies such as operational transform [9] and rollback schemes [5].

- *Degree of Warping.* Clients commonly use *dead reckoning* [6] to update the positions of mobile entities based on their last known position and velocity. The snapping of an entity to a new position following server update is often referred to as *warping.* The degree of warping can be measured by counting the frequency and magnitude of required corrections.

- *Animation Rate.* The animation rate refers to the frequency with which the positions of mobile entities are updated on the player's display. The animation rate of necessity cannot exceed the game's frame rate, but may be considerably slower. A high animation rate makes movement appear smooth to the player.

2.2 Related Work

There are many different approaches to consistency maintenance for distributed game objects. From the players' perspective, it is important that the gameplay does not suffer because of the implementation of these routines. This imposes time constraints on the algorithms, particularly on feedback. Schneiderman characterizes acceptable feedback time for highly interactive tasks as being between 50-150 ms [8].

An examination of the design decisions required during the development of a consistency maintenance scheme was documented during the development of the game Age of Empires [2].

Consistency maintenance algorithms suitable for use in games [4] fall into three basic categories: *pessimistic*, *optimistic*, and *predictive* algorithms. Pessimistic algorithms [2, 10] ensure that any reference to a replica is always consistent with the canonical server version. Pessimistic algorithms are ideal when complete fidelity is required but are not appropriate for highly interactive tasks as server interactions are expensive operations.

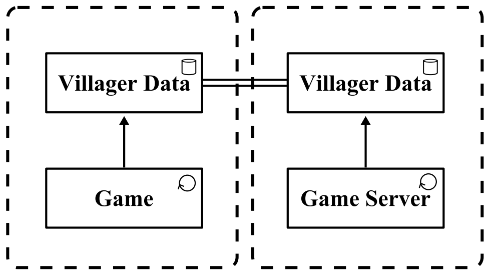

Figure 1: Conceptual architecture.

Optimistic algorithms allow for local replicas to be updated immediately. The replica then reports changes applied to to the canonical state. Should state update conflict, the algorithm is responsible for resolving the problem and returning all replicas to the canonical state [3, 4, 9, 5]. Predictive algorithms are optimistic algorithms capable of extrapolating future state changes.

3. THE WORKSPACE APPROACH TO CONSISTENCY MAINTENANCE

Our approach to supporting plug-replaceable consistency maintenance is based on the Workspace Architectural Model [7]. The workspace model provides high-level facilities supporting the development of distributed, multi-user applications. While a comprehensive discussion of the features of this model is out of the scope of this paper, there are two features that are particularly useful for implementing concurrency control in games: support for *conceptual modeling of software architectures*, and the ability to express *synchronization* of data at the architectural level.

A conceptual architecture expresses the structure of an application in terms of *components* and *connectors* between the components. The conceptual architecture does not specify how the components are allocated to computational nodes, or any of the details of networking, consistency maintenance or concurrency control.

Components, represented by boxes, are the smallest architectural unit of the workspace model. The boxes which contain circular arrows are *actor* components. Actors are capable of initiating activity, and therefore maintain an executing thread. *Stores*, indicated by a cylinder, are passive data components which cannot initiate activity or call other components. Stores may be synchronized with other stores of the same type. The application programmer can assume that all synchronized stores will be observationally equivalent throughout execution.

Architectural components are grouped into *workspaces*, shown with a dashed line. Workspaces are used simply to show that components belong together, not that they are implemented on the same node; a single workspace may involve multiple nodes.

3.1 Refinement to an Implementation

Programmers create applications at the level of the conceptual architectures such as that shown in Figure 1. At runtime, the fiia.net toolkit [7] manages the implementation of this architecture as a distributed system through the use of a refinery. Since there can be multiple different legal refinements for each conceptual architecture, the application programmer can add hints to influence the refinement process.

Valid implementations that the toolkit might choose include replicating the data and maintaining two copies (one local to the client and the other to the server) or maintaining a single copy, forcing either the client or the server to access the component remotely. In the replication case, the toolkit is obliged to maintain the consistency of the two copies of the data.

When synchronization between two stores is implemented via replication, the refinery inserts special Concurrency Control, Consistency Maintenance (CCCM) components into the implementation architecture. These components behave as intermediaries between the replicated object and all incoming and outgoing connectors, ensuring that messages are ordered or transformed to guarantee observational equivalence between the components. CCCM components can embody any consistency maintenance scheme. The implementation level architecture of Figure 2 shows how CCCM components are inserted into the case study architecture. Creating CCCM components involves writing code to examine incoming messages and decide if any need to be modified, created, deleted, or simply relayed.

4. CASE STUDY

We now demonstrate the effectiveness and ease of use of plug-replaceable consistency maintenance schemes via a case study. The hypotheses of the case study are that games developed using the workspace architectural model do not have to contain code to deal with replica consistency, and that the workspace model makes it easy to deploy, modify and experiment with multiple consistency maintenance schemes within the same program.

In the case study, a non-player character (called a *villager*) traverses a set of waypoints on the server. Every 500 ms, a message is sent to the client informing it of the villager's location. The client's job is to produce a view of the current position of the villager providing high fidelity, low warping, and smooth animation.

4.1 Conceptual Architecture

Figure 1 shows the conceptual architecture of our case study. In it we have two workspaces indicating the components for a client and server respectively. The *game client* and *game server* actors contain the main game-loop objects. The outgoing arrows represent *call connectors*, specifying that the components may make synchronous call. The two synchronized *Villager Data* stores contain all of the information about the villager, the mobile entity (mob) in our game.

We use architectural attributes to request a replicate implementation of the *Villager Data*, and have experimented with three different CCCM components. The game has been instrumented to allow us to measure different aspects of the player experience.

To simplify instrumentation, the application is run on a single computer. We emulate latency by delaying the transmission of data across synchronization connectors. Delay is based on a normal distribution. Varying the mean and standard deviation of this distribution has allowed us to experiment with different network delays.

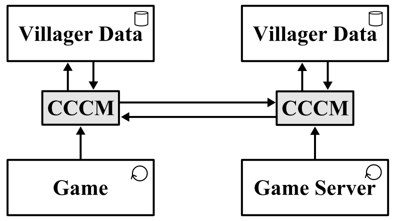

Figure 2: Implementation architecture.

4.2 Experimental CCCM Modules

We experimented with three different consistency maintenance schemes, each implemented via a different CCCM component. The *Non-Predictive CCCM* simply applies positional updates from the server as they become available with no prediction. The *Simple Predictive CCCM* uses dead reckoning to smooth the motion of the villager between server updates. The client continuously updates its copy of the villager's position based on the its last known position and velocity. Finally, the *Optimized Predictive CCCM* takes a more sophisticated approach to dead reckoning by considering the timestamps of server updates. The client CCCM uses an estimated latency value (based on latency observed to-date) to determine if its current predicted position could be reached from the last received server update.

4.3 Controlled Variables

The simulation was run nine times total, three times for every CCCM module, once for each latency condition. A normal distribution of latency was used each time, with different characteristics. A *no latency* condition ($\mu = 0$ $\sigma = 0$), a *low latency* condition ($\mu = 100$ $\sigma = 10$), and a *medium latency* condition ($\mu = 500$ $\sigma = 100$) are used.

We monitored the client-side villager position in order to determine the degree of warping, the animation rate, and the client's fidelity. The rate of animation is computed by measuring the average movement of the client mob (lower numbers are more smooth.) The degree of warping measures the average number of warps/second. A warp occurs when a server update is received whose villager position differs from the client position by a value above some threshold (5 world units.) Fidelity is the average distance between the client and canonical villager positions (lower numbers represent higher fidelity).

4.4 Results

The results of our experiment are summarized in Table 1. The values shown represent a 95% confidence interval around the observed mean.

The non-predictive algorithm performed poorly with respect to all aspects of the player experience. The fidelity degraded proportionally to the increase in latency. The rate of warping was consistent with the rate of server updates.

The predictive algorithms provided significantly better fidelity, the optimized predictive producing slightly better fidelity scores. The degree of warping was similar between both predictive algorithms, however the simpler algorithm showed a smaller degree of warping. The smoothness of an-

Table 1: Results from case study.

Non-Predictive CCCM				
Lat.	Warps (Hz)	Warp Magnitude	Fidelity	Animation Rate
None	2.0	49.79 ± 0.66	29.43 ± 0.88	46.06 ± 1.65
Low	2.0	50.03 ± 0.80	39.17 ± 0.89	46.31 ± 1.70
Med	2.0	50.62 ± 0.77	75.25 ± 1.22	46.90 ± 1.69
Predictive CCCM				
Lat.	Warps (Hz)	Warp Magnitude	Fidelity	Animation Rate
None	1.4	8.95 ± 0.54	1.62 ± 0.19	9.35 ± 0.15
Low	1.2	10.96 ± 0.91	7.79 ± 0.30	9.57 ± 0.17
Med	1.7	15.13 ± 1.17	48.16 ± 0.70	10.27 ± 0.24
Optimized Predictive CCCM				
Lat.	Warps (Hz)	Warp Magnitude	Fidelity	Animation Rate
None	1.4	8.63 ± 0.52	1.26 ± 0.14	9.28 ± 0.14
Low	1.3	12.90 ± 1.24	5.70 ± 0.37	9.79 ± 0.21
Med	1.6	18.77 ± 2.02	27.48 ± 0.94	10.84 ± 0.37

imation was consistent across the predictive algorithms.

4.5 Discussion

We hypothesized that replacing consistency maintenance schemes would be simple. This turned out to be the case. Between each test, changes to the code were limited to the architectural attribute guiding the refinement of the synchronization connector. No changes at all were required in the mob movement method.

Implementing the rest of the game was also straightforward, as it did not require any consistency maintenance code. The villager class consisted only of vectors representing velocity and position as well as a method to update position based on a time delta.

The CCCM's themselves needed to be programmed to give implementations for the consistency maintenance schemes. CCCM's encapsulate a single consistency maintenance algorithm, and are programmed without knowledge of their context of application. This means that once implemented, CCCM's can be re-used in different applications without modification. We envision that a library of interesting CCCM algorithms can be created, appropriate to different consistency requirements in games.

5. CONCLUSION

In this paper, we have shown how consistency maintenance schemes for multiplayer games can be implemented in a plug-replaceable manner. Based on the workspace architectural model, the approach allows developers to write game code independently of consistency maintenance concerns, simplifying game programming. We have shown that plug-replaceability allows easy experimentation with different consistency maintenance schemes. We have illustrated the approach with a simple case study, experimentally showing how different consistency maintenance schemes provide differing player experiences. The case study demonstrates the simplicity of coding applications when consistency maintenance is factored into plug-replaceable CCCM components. This work is ongoing.

6. ACKNOWLEDGEMENTS

We gratefully acknowledge the funding of the Natural Science and Engineering Research Council of Canada and the NECTAR CSCW research network. This work would not have been possible without the availability of Greg Phillips' fiia implementation of the workspace architectural model. We wish to thank David Smith and the rest of our colleagues in the EQUIS lab for their help with formulating this paper. We would also like to thank Emily Adkins-Taylor for her valuable revisions and support.

7. REFERENCES

[1] J. Beardsley. Seamless servers: The case for and against. In T. Alexander, editor, *Massively Multiplayer Game Development*, pages 213–227. Charles River Media, Inc., Hingham, MA, 2003.

[2] P. Bettner and M. Terrano. 1500 archers on a 28.8: Network programming in age of empires and beyond, March 2001. Available at `http://www.gamasutra.com/features/20010322/terrano_02.htm`.

[3] W. K. Edwards and E. D. Mynatt. Timewarp: techniques for autonomous collaboration. In *CHI '97: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 218–225, New York, NY, USA, 1997. ACM Press.

[4] S. Greenberg and D. Marwood. Real time groupware as a distributed system: concurrency control and its effect on the interface. In *In Proc. of CSCW '94*, pages 207–217, New York, NY, USA, 1994. ACM Press.

[5] C.-L. Ignat and M. C. Norrie. Grouping in collaborative graphical editors. In *In Proc. of CSCW '04*, pages 447–456, New York, NY, USA, 2004. ACM Press.

[6] L. Pantel and L. C. Wolf. On the suitability of dead reckoning schemes for games. In *NetGames '02: Proceedings of the 1st workshop on Network and system support for games*, pages 79–84, New York, NY, USA, 2002. ACM Press.

[7] G. Phillips, T. C. N. Graham, and C. Wolfe. A calculus for the refinement and evolution of multi-user mobile applications. In *Design, Specification and Verification of Interactive Systems (DSV-IS 2005)*, pages 137–148. Springer LNCS, 2005.

[8] B. Schneiderman. *Designing the User Interface : Strategies for Effective Human-Computer Interaction.* Addison-Wesley Computer and Engineering Publishing Group, Reading, UK, 1997.

[9] G. Shelley and M. Katchabaw. Patterns of optimism for reducing the effects of latency in networked multiplayer games. In *Proc. of FuturePlay 2005*, 2005.

[10] T. N. Wright, T. C. N. Graham, and T. Urnes. Specifying temporal behaviour in software architectures for groupware systems. In *Proceedings of Design, Specification and Verification of Interactive Systems (DSV-IS'2000)*, pages 1–18. Springer LNCS, 2000.