A Protocol for Distributed Collision Detection

Tom Ching Ling Chen
School of Computer Science
McGill University
Montreal, Quebec, Canada H3A 2A7
ching.chen@mail.mcgill.ca

Clark Verbrugge
School of Computer Science
McGill University
Montreal, Quebec, Canada H3A 2A7
clump@cs.mcgill.ca

***Abstract*—Scalability of multiplayer games can be improved by client-side processing of game actions. Consistency becomes a concern, however, in the case of unpredictable but important events such as object interactions. We propose here a new *motion-lock* protocol for distributed collision detection and resolution. The motion-lock protocol improves performance of motion prediction by giving stations time to communicate and agree on the detected collisions. This reduces the divergence of distributed object states and post-collision trajectories. Offline and online simulation results show the motion-lock protocol results in qualitative and quantitative improvements to consistency, with negligible network impact and a minimal sacrifice in the responsiveness of player controls. Our design can be used to hide latency and reduce server load in current multiplayer online games, improving scalability and furthering fully distributed designs.**

I. Introduction

To reduce bandwidth and mask network latency, many multiplayer games make use of dead-reckoning algorithms to predict the motion of game objects. Adaptive designs can be quite successful, predicting behaviour with low error for many complex, if locally smooth motions [1]. Dead-reckoning is least successful, however, in the presence of unpredictable, dynamic interactions such as collisions. Errors in dead-reckoned motion can affect the time, location, and even detection of collisions, easily resulting in large visual errors as game state deviates and is eventually synchronized. Figure 1 shows examples where errors introduced by dead-reckoning cause a collision to be missed or detected erroneously (respectively). The potential for this behaviour is an important aesthetic concern in client/server architectures, and has a major impact on overall game consistency in peer-to-peer contexts.

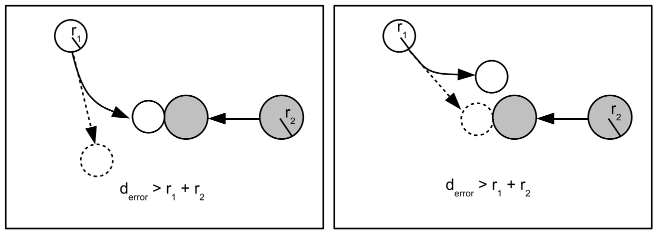

Fig. 1. Missed and false collisions when error in the location of one object is greater than the sum of the radius of the objects.

Here we investigate a new protocol for improving collision consistency between pairs of distributed objects. Our *motion-lock* protocol works by observing object behaviour and preventing unpredictable local object movements in the presence of potential collisions. By matching limitations on object activity to network delay, the prediction of future collisions can be greatly improved, in optimal cases resulting in perfect collision synchrony. This simple design has few drawbacks, adding only minimal additional network cost and no discernible user impact.

We evaluate our design experimentally, measuring and comparing behaviour to an industry-standard dead-reckoning design. Our design shows significant benefit to game consistency, halving the inconsistency times over more straightforward approaches on average, and noticeably improving visual appearance.

A. Contributions

Our work makes the following specific contributions:

- We present a new protocol for improving collision consistency in a distributed, peer-to-peer environment. Our approach has low network and user impact, but provides significant benefits to collision consistency.
- Our protocol is examined in both offline and online simulation, and we experimentally evaluate behaviour under our approach in comparison to other designs. Our motion-lock protocol greatly reduces inconsistency time and improves visual appearance.

In the next section we give background and related work on distributed motion prediction and collision detection. Section III describes our main protocol design, and Section IV gives experimental data comparing our design with others. We conclude and discuss future work in Section V.

II. Background and Related Work

Modern multiplayer games are based on various designs, extending from basic client/server to more distributed implementations. In either context we assume game object data is either managed locally, or *replica*ted from a remote *master*.

Within such a distributed context, inconsistencies can occur as network messages are lost or delayed. Several designs have been proposed that improve state consistency. Optimistic approaches, such as *TimeWarp* [2], rewind and recalculate state when new information is discovered; this helps correctness [3], although it is not always suitable for real-time games. A more practical technique is *Local lag,* which introduces a delay in local event processing, providing time for the corresponding network messages to reach their destinations [2]. A similar approach is used in bucket synchronization, which also delays events, assigning them to discretized-time buckets for current or future processing [4]. This allows stations to move at variable frame-rates and further accommodates network latency. Our design for motion-lock is based on similar delay principles, although we apply it to predicted rather than known events, and only to collision events rather than all events.

In more distributed contexts such as DIS [5] (and in actual games), latency and packet loss are compensated by *dead reckoning,* extrapolating the future state of an object based on past data and an underlying motion model. The same design of course also works to limit bandwidth, only sending updates when a master believes a replica has a sufficient deviation. The *Position History-Based Dead Reckoning* (PHBDR) protocol [1] builds on this design, further reducing bandwidth and improving remote tracking.

Many improvements to position estimation have been proposed. Tumanov *et al.* use a *Kalman filter* to predict the future state of the master on the sender side [6]. The predicted master state is then sent to replicas such that it will arrive in time without the need for extrapolation at the receiver side. Other work has focused on developing heuristics that can be applied to the basic dead-reckoning approach. Cai *et al.* adaptively change the error threshold of the dead reckoning algorithm depending the distance between the objects. When two objects enter each other's area of interest such that the distance between them is small, the error threshold is reduced so that the update frequency is increased to reduce prediction error [7]. Similarly, the *pre-reckoning* algorithm overrides the error threshold and sends updates immediately to the replica if the motion of the master shows certain behaviours, such as starting to move, coming to a stop, and making a sharp turn [8]. Kenny *et al.* calculate the deviation error of the replica and send back the error to the master, so that the master can change the error threshold accordingly. This creates a close-loop control system to adjust the update frequency [9].

Some research into distributed collision detection has also been performed, although primarily in terms of improving position estimates at collision points. Ohlenburg adaptively increases the rate of updates from master to replicas when objects are close to each other and may potentially collide [10]. The results show great improvement in object collisions, but also show that by increasing the update rate the bandwidth usage increases dramatically. For more predictable, non-player controlled objects, the Deterministic Object Position Estimator uses an object's past trajectory to predict the motion of the object and the collision point and time [11]. The estimator gives accurate results for objects that follows predictable trajectories, but of course does not apply to player-controlled objects or other objects with less predictable movements. Our technique offers a heuristic solution to collision handling that allows game peers to detect and resolve collisions as locally as possible, based on the general principle of avoiding server or other indirection. For distributed hybrid and P2P games this has been an effective technique, previously applied to reducing the communication cost of interest management [12] as well as the cost of position updates [13].

We perform part of our experimental testing within *NetZ,* a distributed game middle-ware for multiplayer games [14]. NetZ uses a master duplica (replica) approach, and a publish-subscribe model to automate the object replication process. NetZ implements a number of features to help synchronize game objects and maintain consistency, including PHBDR, local-lag, and bucket-synchronization [15].

III. Motion-Lock Protocol

Our work is intended to improve dead-reckoning, allowing either predictive or distributed resolution of object collisions. We assume a core scenario consisting of two objects undergoing potential collision, with one or both objects not under local control. Network latency and jitter mean the positions of the master and duplica objects can deviate, and so in the absence of central authority multiple stations need to come to agreement on the existence of a collision, and if so on the resulting state.

Performance and real-time requirements further complicate this scenario. Consistency can be provided by, for instance, each station simply informing others when collisions are detected. Network latency, however, means distant stations display collisions late, resulting in visually confusing deviations and subsequent corrections to the (post-collision) state.

Below we present our motion-lock protocol solution. We first discuss the basic protocol, and then present two extensions, addressing multi-object collision scenarios and improving post-collision behaviour.

A. Motion-Lock

A naive solution to distributing collision resolution suffers from network latency, with distant stations being forced to make use of late data. Our motion-lock protocol improves behaviour by making predictions of collisions, allowing sufficient time for objects to receive notifications. To ensure predicted behaviour is not invalidated by local client (player) actions or other events, we briefly lock the motion of objects in such situations. The resulting guaranteed and future-scheduled collision can be transmitted to other stations ahead of time, improving game consistency.

Predicting collisions is straightforward, and we use a simple linear model leaving more complex modeling for future work. To determine a potential collision, at every frame the future trajectories of objects are estimated using linear extrapolation. Estimated trajectories are checked for intersection, and if a pair of objects will collide at a future time the collision point and time is determined.

To prevent external events acting on locally mastered objects from altering predictions, object motions are locked for the duration of the protocol, ensuring regular and predictable movement. Naturally, if object motions are locked for too long, players may feel a loss of control over game objects. We thus define a maximum locking threshold T_{lock}; when a collision is predicted for two objects, their motions are locked only when the time remaining before the collision is no more than T_{lock}. Following known thresholds in user tolerance of latency, and matching our own experience, a threshold of around 100ms is maximal [16]. A practical value for T_{lock} depends of course on the specific game and expected network latency.

Once the objects' motions are locked they are committed to the collision, and details of the scheduled collision are sent to all relevant stations. Under optimal circumstances all stations will receive notifications prior to the collision; more generally the protocol will reduce the duration of local inconsistency by up to T_{lock}.

B. Spatial-temporal Bucket Synchronization

Our basic protocol considers only pair-wise collisions, as is common in practice. In dense situations, however, an object may experience multiple collisions in short periods, and our collision predictions can be negated or altered by other nearby collisions inducing new interactions. Best accuracy would be achieved by discarding commitments and revising predictions, but this has large costs in reversing previous notifications.

A more efficient design is achieved by prioritizing the first collision, and ignoring any later detected collisions with a locked object. Visually, however, the objects will appear to penetrate each other as collisions are ignored. We thus augment our basic protocol with *spatial-temporal synchronization* for improving the appearance and consistency of multi-object collision scenarios.

This technique works by grouping overlapping collisions and resolving them in detection order. When an object X is found to collide with an already locked object Y, if the projected collision time is prior to Y's existing, committed collision time, the collision of X and Y is delayed accordingly. X locks its motion and becomes committed to the new collision time. The end result is that all three objects resolve their collisions at the same time. This design extends naturally to larger groups of colliding objects.

Note that all the objects newly joined to the collision must have their original collision time within the T_{lock} threshold of the first collision. This means that all overlapping collisions are both spatially and temporally close to the first pair of colliding objects. Since humans do not perceive collision points accurately [17], the small difference in collision point implied by reordering collisions is not observable, and outweighs the very noticeable visual confusion of inter-penetrated objects.

C. Post-Collision Trajectory Agreement

Identification of a collision event in time is not entirely sufficient for good visual consistency. Whether due to interpolation error or local events, stations may resolve a collision with objects in slightly different states, and small differences in object position or orientation can result in large deviations in post-collision trajectories. This can be corrected by communicating full state information, either pre- or post-collision, although doing so can result in visual discrepancies.

Our design improves appearance by instead only partially correcting object state. As well as collision time we thus also send a post-collision direction vector, and use this to ensure objects end up heading in the same direction. After the collision objects on both the detected and any informed stations will then travel in at least parallel if not identical directions. Later position corrections may still be required, but with similar trajectories the differences tend to be smaller and later position corrections less noticeable.

It is possible of course for two stations to both detect the same collision and send conflicting final trajectories to each other. Further agreement protocols could be added to our protocol to handle this situation; in our implementation work we give the local trajectory priority. A more detailed investigation of this problem is left for future work.

IV. Experimental Analysis

To evaluate the effectiveness of the motion-lock protocol, we implemented our design in an offline simulator and in an online multiplayer middle-ware used in the gaming industry. The offline simulator allows us to easily test the protocol in different network conditions, while the online middleware allows us to test the protocol in a real network. For comparison, in both offline and online systems we also set up control and naive-send protocols. We experimentally compare the duration of inconsistency due to collisions, as well as the deviation in post-collision trajectories.

Our offline simulator has adjustable network latency and packet-loss rates. The simulator is written in python and modeled using the Discrete Event System Specification (DEVS) formalism; further details can be found in [18]. The station module consists of a simple 2D physics simulation that contains circular objects moving and colliding with each other. A position history-based dead reckoning protocol with a polynomial motion model is implemented in the simulator to synchronize the motion states of the master and the replica. The machine used for the offline simulator is an Intel Core 2 2GHz machine with 2GB of memory and runs on 32 bit Windows Vista.

We base our online investigation on Quazal's *NetZ* middleware [14]. NetZ is a framework for managing distributed game states, and has been used by gaming companies to provide online multiplayer functionality. NetZ includes a number of techniques, such as local-lag and PHBDR, to help reduce the effect of network latency. We implemented the protocols within NetZ, building a test suite from a supplied 3D physics simulation involving simple spherical objects. Tests were run on two machines connected through residential ISPs to the internet, the machine above with a 6Mbps upload/800Kbps download connection and an Intel Core 2 Quad 2.5GHz machine with 4GB of memory running 64-bit Windows 7 and using a 3Mbps/512kbps connection.

A. Offline Experiment Setup

For our offline experiments we implemented control and naive-send protocols to compare with motion-lock. The control only has the underlying PHBDR protocol to synchronize the motion states. No collision agreement mechanism is implemented, and if a collision is missed it will only be corrected in terms of position updates from later messages. This represents a worst case for distributed collision handling. The naive-send protocol sends out a message whenever a collision is detected. All participating stations will eventually be consistent (assuming the message is actually received), but the interval in which a given collision is not represented at both stations is bounded only by message latency. We compare with the naive-send protocol in order to evaluate the effect of reducing the collision inconsistency interval in motion-lock.

All protocols behave well if movement of all objects is highly predictable, such as with purely linear movement. We thus set up scenarios of greater complexity, interacting combinations of circular and linear movement. All scenarios involve two stations, A and B, such that A contains master M_a and replica R_b and B contains master M_b and replica R_a.

- CIRCULAR-LINEAR-COLLIDE (CLC) In this scenario, M_b is moving in straight line while M_a on A is moving in a circle to simulate less predictable movement. The extrapolated states of R_a is thus inaccurate. The objects will collide at one point; however, R_a's deviation may cause it to miss the collision with M_b. This scenario is to evaluate how the protocols deal with missed collisions.
- CIRCULAR-LINEAR-PASS (CLP) Motion of the objects is similar to CLC except the objects do not collide and instead miss each other by at least 1 object radius. Again, however, the state of R_a is inaccurate and in this case may cause false collisions.
- CIRCULAR-CIRCULAR-COLLIDE (CCC) Here both objects are moving in circles and the objects will collide at one point. This is to evaluate how the protocols perform when the states of both replicas are inaccurate.
- CIRCULAR-CIRCULAR-PASS (CCP) This scenario is similar to the CCC scenario except the objects do not collide, missing each other by at least an object radius.

Three different network conditions are considered for each scenario. *1*) The good network condition has 50ms latency and 10% packet-loss rate, *2*) the moderate network condition has 100ms latency and 20% packet-loss rate, and *3*) the congested network condition has 150ms latency and 40% packet-loss rate. Each combination of scenarios and conditions is run 50 times with different random seeds to generate 50 collisions. If a collision is detected, the run continues for 500ms more before terminating. If there is no collision, the run terminates after 3 seconds. The time of collision and deviation errors are recorded. The difference between the collision times of the two stations is the collision inconsistency interval. The post-collision deviation error is calculated by summing the deviation error from the time of collision to the end of the 500ms interval, sampled at every 20ms.

B. Results

Qualitatively, the results show significant visual improvement, with the benefit magnified in cases of greater latency and packet loss. For the control protocol large errors are generated when the stations fail to both detect or miss a collision, as one master reacts to the collision while the other ignores it. This is improved by naive-send, although the delay in notification still causes noticeable gaps in the position of objects at the time of collision, as objects continue to move. Motion-lock further reduces the gap, resulting in objects showing more natural collisions. We note that setting T_{lock} appropriately here is important as well; if set too large objects can appear to pull toward each other due to their motion being locked into the movement dictated by the prediction model.

Quantitatively, we measured the time of collision on each station to determine the duration of the inconsistency interval caused by the collision, as well as the positions of the objects after collisions to determine the post-collision deviation. Below we present and discuss numerical results from our offline simulation, followed by our online results where we extend the problem to a multi-object collision scenario.

1) Collision Inconsistency Interval: To determine how consistent stations are in terms of displaying collisions, we compute for each of our scenarious a collision inconsistency interval. In each case we recorded the time of collision detected on each station, calculating the relative differences to determine the interval. Here we compare the collision inconsistency intervals of only the naive-send and motion-lock protocols and not the control, since without an agreement protocol in the control the interval grows arbitrarily large if one station misses a collision.

Figures 2 and 3 show the collision inconsistency interval in our offline simulation for CLC, CLP, CCC, and CCP scenarios respectively. Naive-send, as expected suffers from the fact that if both stations do not simultaneously detect the collision the communication has the network latency as a lower bound. The motion-lock protocol generally and sometimes dramatically reduces the collision inconsistency interval, and even in high congestion scenarios the motion-lock protocol keeps the inconsistency time well below network latency.

2) Post-Collision Trajectory Deviation: The collision consistency interval measures consistency between stations. The visual disruption to a specific station, however, is better measured in terms of the difference in position between master and replica, since that bounds the amount of required visual correction. To compare the protocols we thus measured the deviation errors of the replicas for 400ms after each collision.

Figure 4 shows that the motion-lock protocol has significantly smaller deviation errors than other protocols. The improvement is naturally reduced in the presence of fast and hard-to-predict motion, where the accuracy of collision prediction has a strong impact on the ability to detect collisions early. In Figure 5 we can see that our simple linear prediction model is unable to improve performance beyond that of naive-send for the CCC and CCP tests.

C. Online Experiment

In our online experiment we evaluate behaviour of our protocol within NetZ, and applied to more complex, multi-object

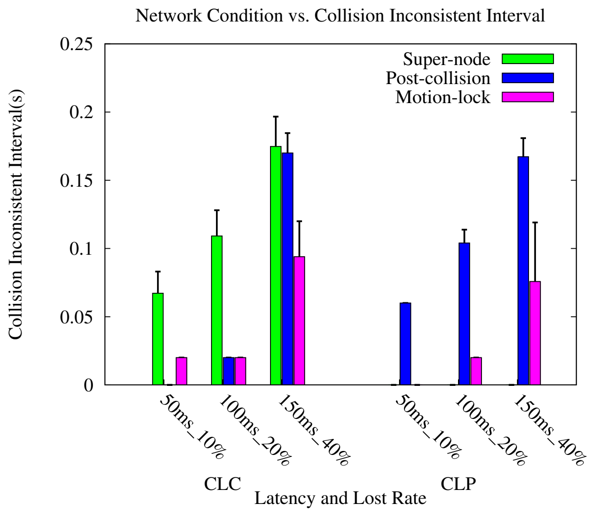

Fig. 2. Collision inconsistency interval for CLC and CLP Scenarios

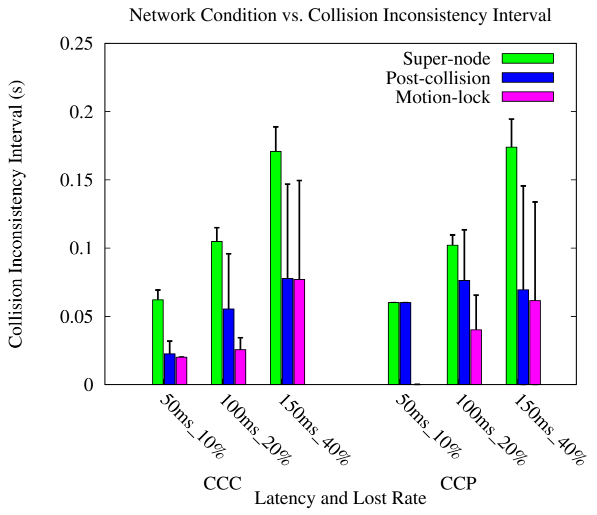

Fig. 3. Collision inconsistency interval for CCC and CCP Scenarios

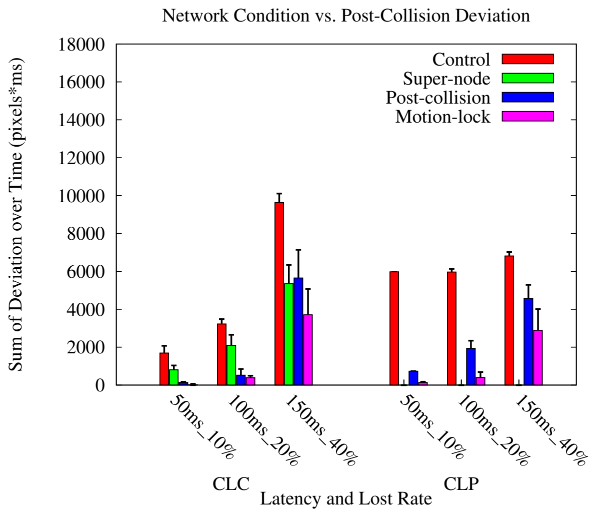

Fig. 4. Post-Collision Trajectory Deviation for CLC and CLP Scenarios

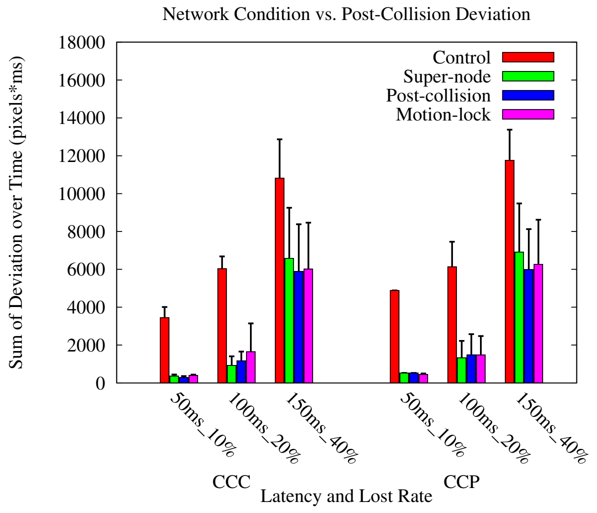

Fig. 5. Post-Collision Trajectory Deviation for CCC and CCP Scenarios

collision scenarios. This allows us to test the effectiveness of the Spatial-temporal bucket synchronization algorithm, and under real network conditions. As well as the control, naive-send, and motion-lock protocols, we thus also have data for two different versions of the motion-lock protocol, with and without the spatial-temporal bucket synchronization.

As a multi-object collision scenario we set up two stations connected through the internet. Station A contains one master and 7 replicas, while station B contains 7 masters and 1 replica. The objects are repeatedly and synchronously given forces the cause them to all collide at (approximately) the same point and time. After each collision, a 3 second period is given for the objects to finish bouncing, and the process is repeated. For each protocol the scenario runs for 20 minutes.

For deviation error analysis, object positions are recorded every 20ms between frames. Replica deviation errors are calculated by processing the locally stored data after the test have finished. Since stations run at slightly different frame rates, we interpolate master positions at intermediate times required for matching replica actions. The scenario can also create many collisions, and so instead of isolating each collision and measure the post-collision deviation, we sum the deviation for each object from the beginning to the end of each 20min run. To estimate the impact on user responsiveness, we recorded the proprtion of direction change commands discarded due to motion-lock.

1) Results: Two types of collisions can be observed in our scenario: actual multi-objects collisions involving more than one object at a point and time, and consecutive collisions, where one object experiences a rapid series of collisions with several others. The latter in particular amplifies error for the the control protocol, and complex collisions show significant visual errors, with objects undergoing corrections of up to several object diameters. This problem is reduced, but still present in the case of naive-send; long collision inconsistency intervals allow object states to diverge noticeably.

TABLE I
MULTI-OBJECT SCENARIO PERFORMANCE IN A REAL NETWORK (20 MINS)

Protocol	Deviation (pixels)			Incons. (ms)		Net (kBPS)	
	Sum	Avg.	Max	Avg.	Max	Sent	Rec.
Control	30740	0.51	42.57	∞	∞	8.14	27.27
Naive	19342	0.32	12.82	8.52	428	8.08	28.70
M-Lock	16148	0.27	13.30	7.62	39	8.09	29.18
w/ S-T Sync	17538	0.29	13.08	8.00	53	8.36	29.60

In the motion-lock protocol without the spatial-temporal bucket synchronization, collisions between a locked object and an unlocked object are ignored. Thus, although it provides an obvious visual improvement over naive-send, we still observed many object penetrations. With the addition of spatial-temporal bucket synchronization, no penetrations are observed. This represents a further significant improvement, although the benefit is slightly mitigated by more actual correction jumps, induced by the manipulation of collision time inherent in spatial-temporal bucket synchronization.

Experimental data showing total, average and maximum deviation error and average and maximum inconsistency is shown in Table I. From this we can see that motion-lock improves both inconsistency time and distance error. Numerically, spatial-temporal bucket synchronization shows some degradation over the base motion-lock, correlating with the increased number of corrections it requires.

Network performance is given for station A, having 1 master interact with 7 replicas, and shown in the right side of Table I. Neither the naive-send nor the motion-lock protocols require significantly more network bandwidth than the control. Unlike state updates, data required for collision agreement are only sent around the collision time, and thus form a small proportion of overall bandwidth costs.

We also considered the impact of locking motion on user control. For the two motion-lock protocols, around 3% to 4% of the commands to change the objects' motion are ignored due to motion-locking. Further, real-game and real-player testing is required of course, but the ratio is low, and could be reduced further at a cost of reduced collision consistency.

V. CONCLUSIONS & FUTURE WORK

In multiplayer games ensuring stations agree on the existence and result of each collision is important for game consistency, and critical to game immersion and ensuring fairness. We have presented a design that improves the scalability of collision detection, providing an efficient and practical means for client stations to resolve collisions independent of any central authority. This applies to P2P designs, but also as an optimistic approach for increasing visual responsiveness in client/server architectures. Our technique is validated in simulation under a variety of conditions, as well as through testing with actual game network middle-ware in a real network.

There are a number of variations and possible improvements to the basic design we presented here. Specific game contexts, for instance, will strongly affect how well motion-lock performs—in very sensitive collision contexts, such as steering a fast object through a dense field of obstacles, even the small reduction in user-control we observe may be excessive. Further validation is required, and improvements may be possible by specifying or identifying specific motions and game environments, locking some object motions but not others. Future work is also required to address cheating concerns; our protocol is designed for simplicity and efficiency more than security, and practical game contexts would require the protocol be hidden or well protected to prevent abuse.

ACKNOWLEDGMENTS

This work is supported by the Natural Science and Engineering Research Council of Canada. We thank Quazal for the use of their NetZ middleware.

REFERENCES

[1] S. K. Singhal and D. R. Cheriton, "Using a position history-based protocol for distributed object visualization," Stanford University, Stanford, CA, USA, Tech. Rep. STAN-CS-TR-94-1505, 1994.

[2] M. Mauve, J. Vogel, V. Hilt, and W. Effelsberg, "Local-lag and timewarp: providing consistency for replicated continuous applications," *IEEE Transactions on Multimedia*, vol. 6, no. 1, pp. 47–57, Feb. 2004.

[3] J. Müller, A. Gössling, and S. Gorlatch, "On correctness of scalable multi-server state replication in online games," in *NetGames '06*. ACM, 2006, p. 21.

[4] C. Diot and L. Gautier, "A distributed architecture for multiplayer interactive applications on the internet," *Network, IEEE*, vol. 13, no. 4, pp. 6–15, Jul/Aug 1999.

[5] I. S. Board, *IEEE standard for distributed interactive simulation - application protocols*. IEEE, 1995, IEEE Std 1278.1-1995.

[6] A. Tumanov, R. Allison, and W. Stuerzlinger, "Variability-aware latency amelioration in distributed environments," in *Virtual Reality Conference, 2007. VR '07. IEEE*, March 2007, pp. 123–130.

[7] W. Cai, F. B. S. Lee, and L. Chen, "An auto-adaptive dead reckoning algorithm for distributed interactive simulation," in *PADS '99: Proceedings of the thirteenth workshop on Parallel and distributed simulation*. Washington, DC, USA: IEEE Computer Society, 1999, pp. 82–89.

[8] T. Duncan and D. Gracanin, "Pre-reckoning algorithm for distributed virtual environments," in *Proceedings of the 2003 Winter Simulation Conference*, vol. 2, Dec. 2003, pp. 1086–1093 vol.2.

[9] A. Kenny, S. Mcloone, and T. Ward, "Controlling entity state updates to maintain remote consistency within a distributed interactive application," *ACM Trans. Internet Technol.*, vol. 9, no. 4, pp. 1–25, 2009.

[10] J. Ohlenburg, "Improving collision detection in distributed virtual environments by adaptive collision prediction tracking," in *VR '04: Proceedings of the IEEE Virtual Reality 2004*. Washington, DC, USA: IEEE Computer Society, 2004, p. 83.

[11] A. Krumm-Heller and S. Taylor, "Using determinism to improve the accuracy of dead reckoning algorithms," in *in Proc. of Simulation Technologies and Training Conference*, 2000.

[12] S. Krause, "A case for mutual notification: a survey of P2P protocols for massively multiplayer online games," in *NetGames '08*. ACM, 2008, pp. 28–33.

[13] J. Jardine and D. Zappala, "A hybrid architecture for massively multiplayer online games," in *NetGames '08*. ACM, 2008, pp. 60–65.

[14] Quazal, *Quazal Net-Z^{TM} 2008 Technical Overview*, March 2008.

[15] ——, *Combating Latency - Game Coherence Over the Internet, Network Problems - Quazal's Solutions*, January 2007.

[16] L. Pantel and L. C. Wolf, "On the impact of delay on real-time multiplayer games," in *NOSSDAV '02: Proceedings of the 12th international workshop on Network and operating systems support for digital audio and video*. New York, NY, USA: ACM, 2002, pp. 23–29.

[17] C. O'Sullivan and J. Dingliana, "Collisions and perception," *ACM Trans. Graph.*, vol. 20, no. 3, pp. 151–168, 2001.

[18] C. L. T. Chen, "Distributed collision detection and resolution," Master's thesis, McGill University, 2010.