Offloading AI for Peer-to-Peer Games with Dead Reckoning

Jiaqiang Bai, Daryl Seah, James Yong and Ben Leong
National University of Singapore

Abstract

In this paper, we study the performance of an offloaded AI agent with increasing network latencies and demonstrate that *dead reckoning* is effective in mitigating the observed degradation. Dead reckoning refers to a class of algorithms typically employed to predict the state of objects in existing games to mitigate the effects of game lag and improve player experience. For a deployed real-time tank game, we found that increasing latencies will cause gradual degradation to the performance of an AI agent and the performance is severely degraded when latencies reach about 300 ms. We show that a simple implementation of dead reckoning is able to delay the onset of performance degradation for round-trip latencies up to 250 ms and is relatively robust to network jitter and packet loss. Since the observed average latency within the continental North America is approximately 55 ms and inter-continental latencies are in the vicinity of 250 ms, our results demonstrate that it is feasible to offload AI to client machines. Most importantly, our method is practical because it does not require much additional code and it allows offloaded AI agents to be developed in a network-oblivious manner similar to what is presently done for server-based AI.

1 Introduction

There have been many proposals for peer-to-peer (p2p) games [9, 2, 3, 4]. While the traditional server-client architecture is expected to continue to be the dominant player for "heavy-weight" games like the World of Warcraft, multiplayer versions of the genre typically referred to as "casual games" are becoming increasingly commonplace. A p2p architecture is an attractive option for such games because by electing one of the client peers to act as the server, instead of relying on a traditional standalone server, we can reduce the need for centrally-provisioned hardware, and thereby reduce costs.

Our work is motivated by our experience in developing and deploying Facebook Tankie [13], a p2p multiplayer third-person shooter tank game on Facebook. The game is implemented as a *pseudo-server-based* architecture [4], where one node is elected to manage the synchronization of game state and a backup node takes over if the pseudo-server node should fail. Our experience shows that this approach works well in practice.

We observed however that in scenarios which involved a large number of AI agents, the pseudo-server node tends to get overloaded and suffer performance degradation. We therefore decided to explore the possibility of the offloading of AI to peer nodes, in order to make the pseudo-server process more *light-weight*. Because a light-weight pseudo-server maintains less game state, it is easier and faster to recover when it fails. With the availability of additional computational resources from peers, it is conceivable that we can support more complex AI behaviours [6].

The offloading of an AI agent to peer nodes is unfortunately not without complications. There are two key challenges: (i) the state of the game world as perceived by the client is slightly behind the actual state of the simulation at the server; and (ii) there is a delay between the time when a command is issued by the AI agent running on the client and the time when the command is executed on the server. If the person writing the AI agent has to take into account varying network latencies, his job will be significantly more complex, perhaps even intractable. Ideally, we would like to be able to offload an AI agent that was written to run on the server to a client with no or minimal modification.

In this paper, we study the performance of an AI agent as network latencies increase and evaluate the effectiveness of *dead reckoning* in mitigating the observed degradation. Our goal is to allow AI agents to be developed in a network-oblivious manner, similar to what is presently done for server-based AI, and also to understand the limits of performance that can be achieved for offloaded AI. Our key idea is to employ dead reckoning to predict the state of the simulation when a command is expected to be executed, and to present this state of the game world to the AI agent. The AI agent will then make its decisions based on this augmented state instead of the local state perceived by the client.

In our experiments, we verified that large network latencies can indeed wreak havoc when commands are issued by an AI agent across the network. In the context of Facebook Tankie, we found that increasing latencies will cause gradual degradation to the performance of the AI agent and the performance is severely degraded when latencies reach about 300 ms. We showed that a simple implementation of dead reckoning can delay the on-

set of performance degradation for network latencies up to 250 ms. From the data obtained in our Facebook deployment, we found that latencies within the continental North America are about 55 ms on average and inter-continental latencies are about 250 ms on average. Hence, our results suggest that it is practical to offload AI to client machines, at least within a continent, and possibly Internet-wide with further optimization.

In summary, our paper has two key contributions: (i) to the best of our knowledge, ours is the first experimental study of how network latencies can affect AI agents that are completely offloaded from a server; and (ii) we show that a simple implementation of dead reckoning can effectively delay the onset of performance degradation. But more importantly, not only is our method effective, it is simple and practical. Our implementation of dead reckoning involves less than 200 lines of Java code and it allowed an AI agent that was developed to be run on a server to be offloaded with minimal modifications.

The remainder of this paper is organized as follows: in Section 2, we provide a review of existing and related work. In Section 3, we describe our experimental methodology. Our evaluation results are presented in Section 4. Finally, we discuss our work in Section 5 and conclude in Section 6.

2 Related Work

To the best of our knowledge, the offloading of AI to clients was first proposed by Douceur et al. [6]. The context of their problem was however quite different. Their goal was to partially offload some AI computations in order to support increasingly complex AI in the context of massively multiplayer online games (MMOGs). Douceur et al. proposed the splitting of AI into two components and offloading the more computationally expensive component to clients. They also showed that their approach is able to tolerate latencies of up to 1 second. In our work, we offload an AI agent completely from the pseudo-server so as to reduce its complexity and simplify the recovery process in the event of node failures. Given the context of our work, Douceur et al.'s solution is not feasible. The splitting of an AI into two parts will make the recovery process for a p2p setup even more complicated. From our experience with game AI, we believe that it might not always be feasible in the general case to split AI into two components in a straightforward way. In particular, we could not come up with a natural way to split the AI agent for our game.

2.1 Dead reckoning

Dead reckoning refers to a class of algorithms that are typically employed to predict the state of objects and mitigate the effects of network latencies for clients in a networked game to improve player experience. With this technique, update packets are issued to notify clients of a change in game state only when a prescribed error threshold has been exceeded. The client then attempts to predict the current state of the game objects between status update messages. The prediction of future state based on past states received from the server typically occurs at every simulation step. The predicted state is updated with the last received state from the server when the two differs by more than a pre-determined threshold. When this happens, players may observe the "teleportation" of mobile objects [14].

Numerous improvements to the standard algorithm have been proposed [10, 8]. These improvements focus on prediction modeling, adjusting the threshold that triggers the generation of entity state update packets, and achieving smoother rendering. Machine learning has also been successfully applied to dead reckoning [11].

2.2 Effect of Network Latency on Games

The effect of network latency on offloaded AI is a relatively unexplored area. The effect of latency on human players is however a well-studied subject. For human players, the threshold of tolerance in general depends on the genre of the game and the methods employed to minimize the effects of latency. First-Person Shooter (FPS) games have a network latency tolerance in the 100 to 150 ms range [5]; third-person games like RPG and sports games can tolerate up to 500 ms of network latency [5]; RTS games can tolerate slightly higher network latencies of about 500 to 1,000 ms [1]; finally, MMORPGs are most forgiving and can tolerate up to 1,250 ms of network latency [7]. Facebook Tankie is a third-person shooter game that falls in the region between a FPS game and a RTS game.

Unlike humans, an AI agent does not get upset about lag and we are concerned mainly with the empirical degradation arising from network latencies for offloaded AI. We found that the degradation of AI performance is significantly more pronounced than what would be suggested by the above figures.

3 Methodology

In this section, we describe our network model, our implementation of dead reckoning and our experimental setup.

3.1 Simulation and Network Model

A networked game is essentially a simulation that collates and processes the inputs from a group of players. The game that we used for our experiments is Facebook Tankie [13], a multi-scenario third-person shooter tank game that was written in Java and deployed on Facebook. The game consists of objects like tanks, mines, obstacles and AI agents. It is built over *Hydra*, a p2p architecture for networked games, that uses an augmented server-client programming model [4]. Like other games,

it runs on an event-driven simulator that processes incoming messages (commands) in batches according to a discrete timestamp, that we call a *tick*.

Our network model is as follows: when a command is issued at time t, the command will reach the server at time $t + t_c$, where t_c is the network delay from the client to the server. The command is executed at the tick immediately following $t + t_c$ at the server and the update arising is sent to the client. Suppose the update arrives at time $t + t_s$, we refer to the quantity t_s as the *command response time*. The command response time is slightly longer than the round trip time (RTT) because there is a queuing delay at the server before a command is executed.

In the game, each player controls a tank from a top-down view and can execute commands like move forward/backward, turn, rotate tank's turret and fire projectiles. In our experiments, each tank has 100 hitpoints. When a projectile hits a tank, the tank suffers 10 points of damage. The tank is destroyed when its hit points reaches 0. While our Facebook game supports a variety of game scenario and victory conditions, we worked with a stripped-down AI agent that simply attempts to eliminate opposing tanks in our experiments.

3.2 Implementation of Dead Reckoning

In this section, we describe how we implemented dead reckoning to predict the state of the game world at the point where an issued command is expected to be executed. The only mobile objects in Tankie are the tanks and projectiles. Since projectiles move significantly faster than tanks and it is hard to write an AI agent that can avoid projectiles, we focus on the prediction of the positions of tank objects and ignore the projectiles. Our technique is however general enough to be applied to any mobile object.

First, we estimate the command response time t_s. This is a quantity that can change over time due to network conditions, so we keep a running estimate with a exponentially-weighted moving average (EWMA), i.e.

$$\hat{t}_s = \alpha \hat{t}'_s + (1 - \alpha)t \quad (1)$$

where $\hat{t}'_s$ is the current estimate and t is the instantaneous value. In our experiments, we set $\alpha = 0.5$. From the estimate $\hat{t}_s$, we can predict the number of game steps s for which we need to predict the states in advance for enemy tanks by dividing the estimated command response time $\hat{t}_s$ by p, the tick period.

Suppose a client observes the state W_c for the game world. We note that W_c is slightly outdated since there is a delay before the state of the game world at the server W_s is received at the client as W_c as illustrated in Figure 1. If a command is issued by the client when the game world is in state W_c, the command will reach the server and be executed when the state is W'_s. Our goal is therefore to compute an estimate of W'_s, which we refer to as W'_c, and pass this view of the game world to the AI agent for it to make its decision on the next move.

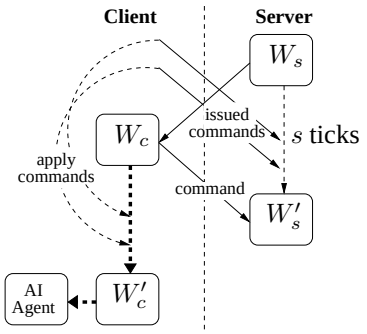

Figure 1: Implementation of dead reckoning.

We know that W_c is equivalent to W_s and W'_s is W_s some ticks later. It is not possible for the client to determine the number of ticks that would have elapsed between W'_s and W_s exactly and this number might vary when there are fluctuations in network latencies or queueing delay. We hence approximate this delay with s, the number of game steps that corresponds to the estimated command latency $\hat{t}_s$. To estimate W'_s from W_s, we observe that in addition to the evolution of the game world, there are also commands that would have arrived at the server and executed between W_s and W'_s.

In W_s, there is a tank object that is controlled by the AI agent and other objects that are either static or controlled by other players or agents. We know exactly what commands were issued in the past by the AI agent, but we do not know what commands were issued by other players or agents. Thus, we record the commands that were issued by the AI agent for the past s ticks and assume that a command issued by the client would take $\frac{s}{2}$ ticks to reach the server. W'_c is therefore estimated by simulating W_c for s ticks and applying the appropriate commands issued by the AI agent at their estimated arrival times between W_s and W'_s.

It is possible that commands might be received by the server from other agents between W'_s and W_s, but we do not take this into account because the client does not have access to this information. In addition, the commands issued by a client can sometimes be dropped because of packet losses in the network and there is also some uncertainty in the times when issued commands reach the server. We hence update our local estimates as and when state updates on the controlled tank object is received from the server.

It is important to note that when predicting the world state, we can safely ignore any states over which the client has authoritative control. For example, in Tankie, the client has authoritative control over when a tank can

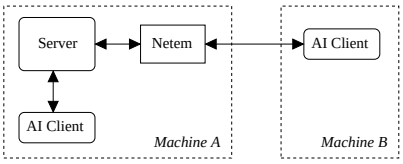

Figure 2: Experimental setup.

fire and over the direction of the turret. Such states are not predicted and we simply use the latest available value maintained at the client. Typical FPS games like Counter-Strike or Half-Life also allow clients to have authoritative control over some game states but retain control over whether shots hit at the server. This approach of allowing clients to have authoritative control over some game states is reasonable only for low latencies because a lagging client's avatar will otherwise appear to teleport on other players' screens. Unfairness can also arise when a client has such control if the latency to the server is high.

3.3 Experiment Setup

We ran our lab experiments on two Intel Core2 Duo machines running Linux kernel 2.6.24.4. One machine has Netem [12] installed to control the upstream and downstream network latency for connected machines. Netem helps the testing of protocols by emulating network variables like delay, loss, duplication and re-ordering.

In each experiment, two tanks controlled by identical AI agents were deployed against each other. One agent was run on the same machine as the server with no network latency while the other AI agent was run remotely on another machine connected through Netem configured with specific network latencies as shown in Figure 2. The AI agent attempts to encircle the enemy tank and fire at it. Because both tanks are in constant motion, they have to constantly adjust their turrets in order to score hits. We adopted this strategy because network latencies will have little or no effect on the stationary objects since no updates from the server would be required. By using identical AI agents, we ensure fairness and that neither tank would have an advantage arising from having a better AI agent. Furthermore, the layout of the game world and the spawn points for the tanks are symmetric to eliminate any bias due to strategic positioning. The size of the game world was small enough for both AI agents to always see each other.

An experiment consists of a series of 100 games. Each game ends when one tank successfully destroys its opponent. For each game, we record statistics like the winning agent, the ratio of successful hits versus total shots fired, the number of remaining hit points for the winning agent and the time taken for each game to complete. We repeated the experiments at various network latencies from 0 to 375 ms with a tick period of 50 ms.

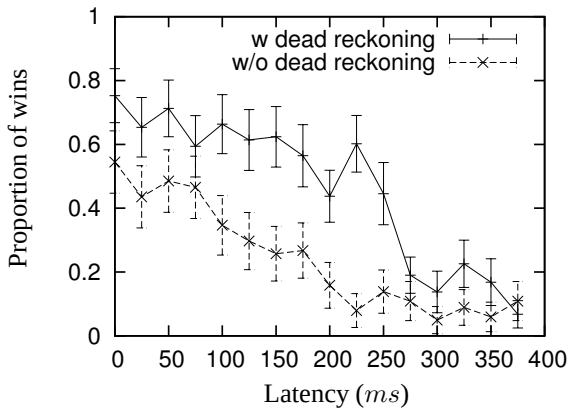

Figure 3: Plot of proportion of wins by the remote AI agent against network latency.

4 Results

In this section, we present the results for our experiments. We will refer to the AI agent that runs on the machine with the server as the *local agent* and the other agent as the *remote agent*. Error bars in the graphs indicate the 95% confidence intervals.

4.1 Proportion of wins

First, we measured the performance of the remote agent in terms of the proportion of wins. This is shown in Figure 3. When the latency is zero and dead reckoning is not enabled, the local and the remote agents are effectively equivalent and the proportion of wins by the remote agent is approximately 0.5, as expected.

The proportion of wins decreases rapidly after the latency reached approximately 100 ms and the remote AI agent is rendered completely ineffective by about 200 ms. In Figure 3, we observe that with dead reckoning, the performance of the AI agent starts degrading (i.e. the proportion of wins falls below 50%) only after about 250 ms. Thereafter, the performance of the remote AI agent degrades sharply.

Separately, we performed a simple test and had some human players play against the local AI agent on the remote client, and varied the latency. The human players concluded that the latencies were tolerable up till about 500 ms. Thereafter, the game became unresponsive and frustrating to play. This demonstrates that human players have a higher tolerance for network latency than AI agents. Note that the AI agent we employed was relatively good and it was difficult for an average human player to beat the AI when there is no network latency.

One interesting observation is that with dead reckoning, the performance of the remote agent surpasses that of the local agent even when the network latency is zero. While this may not seem intuitive, there is a good rea-

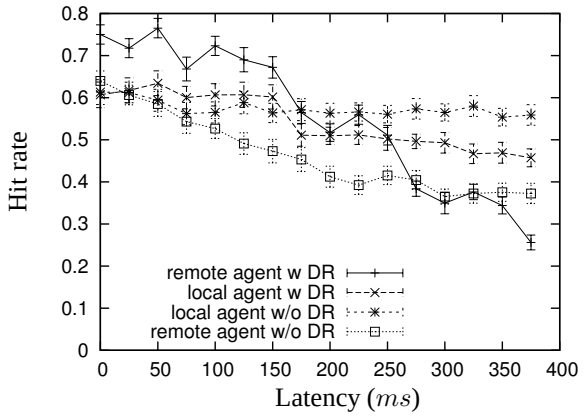

Figure 4: Plot of hit rate against network latency.

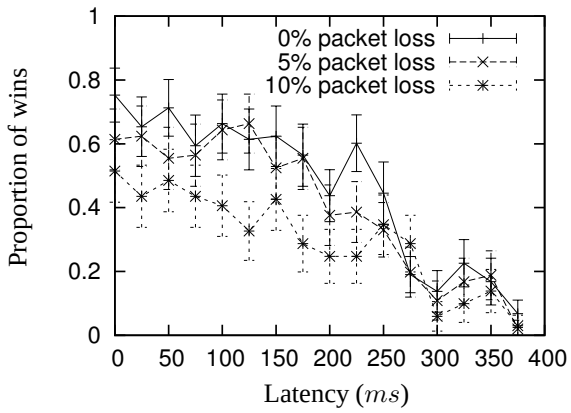

Figure 5: Plot of proportion of wins by the remote AI agent with dead reckoning against network latency with packet loss.

son. Because the server and the local agent are run as separate processes on the same machine, the simulation on the server is not synchronized with the local agent and a command issued by the local agent is buffered before execution. While the local agent makes a decision based on the current tick, the command is effectively only executed one tick later. It was interesting to see that dead reckoning naturally corrected for this phenomenon.

4.2 Firing accuracy

We found that the proportion of wins is a relatively coarse measure since two agents could conceivably be evenly matched until the last shot. Hence, we also compared the accuracy of the local and remote agents as shown in Figure 4. Hit rate refers to the ratio of hits scored on the opponent tank over the total number of shots fired. We see that the hit rate of the local AI agent is approximately 50 to 60% and relatively constant, as expected. It is not 100% because both tanks are in constant motion, resulting in occasional misses. We observe that the degradation of performance for the remote agent in terms of its firing accuracy is somewhat more gradual than what is suggested by the proportion of wins. We also see that dead reckoning seems to be able to improve the accuracy of the remote agent by about 10 to 15% when latency is below 200 ms.

The accuracy of the local AI agent was also observed to degrade at higher network latencies. We found that this was because at such latencies, the tank controlled by the remote AI agent tended to drift away, leading to more misses by the local-agent-controlled tank. This is a minor artifact of the experimental setup.

4.3 Packet Loss & Network Jitter

It is natural to expect the remote agent to perform worse when there are packet losses. When we first plotted the hit rate for the tank agents with emulated packet loss, we were surprised that we did not observe any significant differences in hit rates. We later discovered that packet losses simply caused the firing commands from the remote agent to be lost. These did not affect accuracy since it would be as if the shots were never fired at all. It turns out that the loss of state update packets from the server did not have a significant effect because these are sent at a much higher rate than firing commands.

As shown in Figure 5, with dead reckoning, the AI agent can tolerate up to about 5% packet loss. However, at 10% packet loss, the effectiveness of dead reckoning is reduced and the proportion of wins by the remote agent falls below 50%. Packet losses caused the remote-agent-controlled tank to fire fewer shots, which gave the local-agent-controlled tank an edge. The main lesson here is that the effect of packet losses depends on the implementation of the game. Because the observed packet loss rates in our Facebook deployment are typically below 5%, our results suggest that dead reckoning would be sufficiently robust in a practical deployment.

We also ran experiments at different levels of network jitter ranging from 0 to 10% and found that the AI agent with dead reckoning is able to tolerate network jitter up to 10% with little noticeable effect. To some extent, this is not surprising. The tick interval is 50 ms while the maximum network jitter in the experiment (at 370 ms delay) is about 37 ms. This means that most packets will still reach their destination within their expected arrival tick.

5 Discussion & Future Work

One drawback of offloading AI using dead reckoning is that it requires more memory and processing power to maintain additional copies of the game world and to compute the predictions of future state. We believe that this is not likely a problem since client machines tend to have more than enough processing capabilities to handle the

additional computations needed.

Of greater interest is perhaps the additional bandwidth that would be required for the offloaded AI. From our experience, we have seen that the increased throughput from the client hosting the offloaded AI to the pseudo-server machine is not a concern because the bottleneck lies instead in the outgoing bandwidth from the pseudo-server. Also, since clients already obtain updates of game state from the pseudo-server for the human player, the same state can be shared with an offloaded AI agent and so no additional outgoing bandwidth from the pseudo-server is required.

One possible criticism of dead reckoning is that code must be specially written to compute the predictions and this code is specific to a game. It would be much more convenient to have a technique that is oblivious to the game. Unfortunately, we have not been able to come up with a good approach to do so. There are however two observations in favour of dead reckoning: (i) many games already implement dead reckoning in order to improve the user experience of the human players; and (ii) we have found that a relatively simple implementation of dead reckoning is already effective. In particular, our implementation of dead reckoning for Tankie involved only 200 additional lines of Java code.

A final, but significant drawback to offloading AI to clients is that it exposes the AI to abuse by hackers since the AI code is now accessible on the client machines. Douceur et al. suggested running a deterministic AI agent and have multiple clients executing the same AI code [6]. The results returned by each agent are then compared and the decision is taken by quorum. This is an approach that we might explore as future work.

We recognize that our evaluation results are not entirely conclusive since we have only evaluated an AI agent for Tankie. While we cannot make any general claims, our results demonstrate that it is likely feasible to offload AI to clients for some games when latencies are within a reasonable limit. Furthermore, our predictions do not take into account the likely commands issued by the opposing AI agents. We believe that a more sophisticated dead reckoning algorithm [10, 8] will likely yield better results.

Our preliminary attempts to offload AI in Facebook Tankie have been promising. It is however somewhat complicated to set up controlled experiments in a deployed game with human players. A complete evaluation of the effectiveness of dead reckoning for offloaded AI in our real-world deployment remains work in progress.

6 Conclusion

We have shown that network latencies can significantly degrade the performance of an offloaded AI agent. Also, the extent of degradation is significantly more pronounced than what one would expect from previous studies of the latencies that can be tolerated by human players [1, 7, 5]. We demonstrated that a simple implementation of dead reckoning is effective at delaying the onset of performance degradation for round-trip latencies up to 250 ms. This observation is significant because it implies that the offloading of AI agents is feasible and useful, given that observed latencies for Facebook Tankie are within the 50 to 250 ms range. We believe that our approach is practical because it does not introduce significant complexity in the programming of the AI agents. Our work advances the current state of the art for the implementation and performance of AI in a peer-to-peer setting.

References

[1] BETTNER, P., AND TERRANO, M. 1500 archers on a 28.8: Network programming in age of empires and beyond. In *Proceedings of the 2001 Game Developers' Conference* (March 2001).

[2] BHARAMBE, A., AGRAWAL, M., AND SESHAN, S. Mercury: Supporting scalable multi-attribute range queries. In *Proceedings of SIGCOMM 2004* (August 2004).

[3] BHARAMBE, A., PANG, J., AND SESHAN, S. Colyseus: A distributed architecture for multiplayer games. In *Proceedings of NSDI 2006* (2006).

[4] CHAN, L., YONG, J., BAI, J., LEONG, B., AND TAN, R. Hydra - a massively-multiplayer peer-to-peer architecture for the game developer. In *Proceedings of NetGames '07* (September 2007).

[5] CLAYPOOL, M., AND CLAYPOOL, K. Latency and player actions in online games. *Commun. ACM 49*, 11 (2006), 40–45.

[6] DOUCEUR, J. R., LORCH, J. R., UYEDA, F., AND WOOD, R. C. Enhancing game-server AI with distributed client computation. In *Proceedings of NOSSDAV '07* (2007).

[7] FRITSCH, T., RITTER, H., AND SCHILLER, J. The effect of latency and network limitations on MMORPGs: a field study of everquest2. In *Proceedings of NetGames '05* (New York, NY, USA, 2005), ACM, pp. 1–9.

[8] HANAWA, D., AND YONEKURA, T. A proposal of dead reckoning protocol in distributed virtual environment based on the taylor expansion. In *Proceedings of CW '06* (2006).

[9] KNUTSSON, B., LU, H., XU, W., AND HOPKINS, B. Peer-to-peer support for massively multiplayer games. In *Proceedings of IEEE INFOCOM'04* (March 2004).

[10] KRUMM-HELLER, A., AND TAYLOR, S. Using determinism to improve the accuracy of dead-reckoning algorithms. In *Proceedings of SimTecT '00* (2000).

[11] MCCOY, A., WARD, T., MCLOONE, S., AND DELANEY, D. Multistep-ahead neural-network predictors for network traffic reduction in distributed interactive applications. In *Proceedings of TOMACS '07* (2007).

[12] NETEM. http://www.linuxfoundation.org/en/net:netem.

[13] YONG, J., BAI, J., SEAH, D., RAZEEN, A., NGUYEN, H., LIEW, J., KOH, Z. H., AND LEONG, B. Facebook Tankie, 2008. http://apps.facebook.com/tankgame/.

[14] ZHANG, X., GRACANIN, D., AND DUNCAN, T. P. Evaluation of a pre-reckoning algorithm for distributed virtual environments. In *Proceedings of ICPADS '04* (Washington, DC, USA, 2004), IEEE Computer Society, p. 445.