A Testbed for P2P Gaming using Time Warp

Stefan Tolic
University of Vienna, Dept. for Distributed and Multimedia Systems
Lenaug. 2/8
Vienna, Austria
getraktna@gmail.com

Helmut Hlavacs
University of Vienna, Dept. for Distributed and Multimedia Systems
Lenaug. 2/8
Vienna, Austria
helmut.hlavacs@univie.ac.at

ABSTRACT

Peer-to-peer based gaming is a new paradigm for distributed multiplayer online gaming that has attracted attention in the last years. It is known that P2P based topologies offer good scaling properties and mitigate unfairness otherwise observed for peers being far away and thus having large network lags. However, removing inconsistencies for high paced action games like FPS or tank battle games requires the implementation of a Time Warp-like mechanism, which itself may hinder gameplay for high lags. In this paper we present a tank battle game named Panzer Battalion. Created from scratch, this game follows the P2P approach and implements Time Warp for removing inconsistencies. Panzer Battalion is meant as a testbed for creating rollbacks and understanding, how Time Warp rollbacks depend on network lag, and how gameplay is altered by them.

Categories and Subject Descriptors

I.2.1 [**Applications and Expert systems**]: Games; C.2.1 [**Network Architecture and Design**]: Distributed Networks

General Terms

Keywords

Networked gaming, distributed simulation, Time Warp

1. INTRODUCTION

Multiplayer online games nowadays connect players from all over the world. Often players from different continents play together the same game. Due to the spacial distance, network packet delivery might require a substantial amount of time when sent from one player to another or to a central server. Depending on the *game type*, this inherent *network lag* (aka network latency) may then cause unfair conditions [2, 6], or different views and game states which might even contradict each other [27]. Contradicting game states are also called *inconsistencies.* However, in case of low interactivity, for instance for RTS or MMORPG games, network lag does not necessarily have a strong effect on the outcome [4, 7].

In this paper we introduce a testbed for observing the occurrence of inconsistencies and their removal when applying Time Warp, which is described below. The testbed is a fast paced tank battle game that was developed solely for this purpose.

2. RELATED WORK

Basically there are three types of topologies for multiplayer online games [5]. The simplest topology uses one central server that solely simulates the game environment and stores all game states.

Though being simplistic, this approach offers several drawbacks. First, a centralized environment is likely to run into performance problems, and scaling to hundreds or even thousands of clients taking part in the same game is difficult. Second and even more important, some players being far away from the server will observe a high lag, and usually their game input will arrive at the server much later than the one of their opponents. Especially for highly interactive games this means a considerable disadvantage, rendering the resulting game to be unfair, even though schemes for selecting the optimal hoster exist [8].

A more sophisticated approach would use $N > 1$ *mirrored game servers* in order to divide the load implied by all clients amongst the servers, and to move the replicated and locationally dispersed servers as close to the clients as possible [14, 3, 26]. Of course, being fixed in nature, servers remain in their locations and cannot dynamically move closer to their clients.

Finally, no dedicated server provider may exist and hosting the game is solely left to the clients. Especially in the recent years this new *peer-to-peer* paradigm has been researched extensively [25]. In a P2P topology, $N > 1$ clients, in this case called *peers*, participate in a game and no dedicated server exists. Scalability is provided by a principle called *locality of interest* [20] or *area of interest* [24, 9]. Players are located in a certain area in the virtual world and the player's peer only needs to communicate with peers having players in the same area [13]. Splitting the virtual world in this way results in so-called *zones* [10]. A general problem of P2P based systems is the difficult prevention of cheating [12].

Both mirrored servers and P2P approach, however, require that $N > 1$ computers store parts of the game state in parallel. One way of doing this is to define that each zone is simulated by one server or peer only, acting as the central server of this zone. This again yields the problem of lag differences for different clients of the same zone. Another approach is to fully *replicate* the game state amongst N server/peers. For the P2P topology this means that parts or all peers of a zone hold a full copy of the whole game state concerning their zone, and game state updates must be communicated to all other peers in the same zone. Since state updates are delayed by the network, inconsistencies may occur and may give rise to incorrect decisions [27].

In this paper we assume a P2P architecture with full replication for each zone, i.e., each peer of a zone simulates the whole zone itself. We focus on fighting games like FPS or tank battle games, with high interactivity and actions that cause instant game state changes (like shooting). In such fighting games, network lags may for instance lead to the effect that on one peer a playing character is shot (because the message making this player move was delayed by the network), while on another peer the player is moved in time and hence is not shot [21].

Basically there are three techniques to fight or at least mitigate the occurrence of inconsistencies. First, a technique called *Dead Reckoning* applies prediction of object movements, based on last known position and speed [22, 1]. In the above example, the predictor would estimate the network lag between the two peers and would estimate the position of each player at a given time. Dead Reckoning based on synchronized clocks is then called Globally Synchronized Dead Reckoning [1]. Although Dead Reckoning is able to reduce the number of occurring inconsistencies, it is not able to remove them in case they actually occur. Since prediction is never 100% accurate, and relies on suitable models, inconsistencies still may occur and lead to unwanted paradoxes.

Second, a technique called *Local Lag* puts each incoming packet into a queue and delays it for a fixed pre-defined time [17]. This way, packets arriving out-of-order may be put in-order in the local buffer, and inconsistencies may be removed. Local Lag only works efficiently in case the local delay is larger than the largest network lag. Since packets are always delayed, this technique is not usable for fast paced action games with high interactivity, e.g., FPS games, which are considered to be unplayable for latencies higher than 100 ms. However, Local Lag can actually be combined with Dead Reckoning to achieve better results [28].

Finally, the only option to actually remove inconsistencies in case they occur is given by a technique called Time Warp [16]. Time Warp is a technique well known in the area of parallel and distributed discrete event simulation (PDES) [11]. In Time Warp, the virtual world is split between logical processes (LPs), a generalization of peers simulating a distributed game. Each LP periodically stores a snapshot of its entire (game) state. In case a peer observes an inconsistency (because it received a *straggler* message from another peer), it performs a *rollback* and restores the game state that was observed right before the inconsistency time. In a way, the time is rolled back into the past. A peer rolling back may also decide that it sent messages to other peers in the past that were based on wrong assumptions, and thus caused inconsistencies there. In such a case the peer then sends so-called *null-messages* (void-messages), causing rollbacks on other peers. This may result in waves of null-messages flooding the whole P2P system, and undoing lots of work that was previously computed. Since peers by default simulate independently from each other, assuming that no inconsistencies occur, Time Warp is also often called to be an *optimistic* approach. In contrast, since Local Lag assumes that always inconsistencies occur and thus delays every single message, such approaches are also known as *pessimistic* techniques.

In our work we focus on Time Warp like systems, and ask how rollbacks and null-messages depend on network lag. Time Warp has been studied extensively in PDES, and is usually applied to a fully simulated system without real user interaction. In the gaming domain, in [15] the authors built a testbed based on Quake III, which was adapted to run Time Warp. The authors used bots only, and relate the influence of network lag and the usage of different Local Lag settings onto bot performance and Time Warp frequencies. In [23] the authors adapted a well known tank battle game called BZFlag[1] in order to evaluate a newly proposed hybrid architecture between client-server and P2P, using a central arbiter. They actually carried out experiments with real users, as we did, but did not focus on Time Warp. In [19] the authors adapted a simple third person capture-the-flag game, and a Quake 2 clone to combine Local Lag and Time Warp in a mirrored server setting.

The contribution of our work is as follows. We developed a tank battle game called "Panzer Battalion" from scratch, implementing high-quality graphics and control in order to provide a realistic game feeling. The game is meant to be a testbed for evaluating how rollbacks are created, depending on network lag, number of players, and other parameters. We explicitly focus on running experiments with real users, here also deriving their subjective opinion on how rollbacks influence the gaming experience.

3. PANZER BATTALION

The game "Panzer Battalion" was written for assessing Time Warp for P2P type decentralized, fast paced games. It is a surreal third person shooter, written in C++ under Linux. Players control a tank, collect power-ups and fight other players. The implementation relies on OpenGL for rendering, SDL for threads, window management, input handling and sound, DevIL[2] for image loading, arabica[3] for XML parsing, and BSD sockets for network communication. The graphics engine makes use of depth-fail stencil shadows, cartoon shading, normal mapping and other various shader effects. Running it requires the availability of an NVIDIA GForce 6xxx or another graphics card which supports Fragment Shader 3.0 or higher.

Figure 1 shows a screen shot of "Panzer Battalion". Players steer their own tank and roam through a plane. The tanks

[1]http://bzflag.org/
[2]http://openil.sourceforge.net/
[3]http://www.jezuk.co.uk/cgi-bin/view/arabica

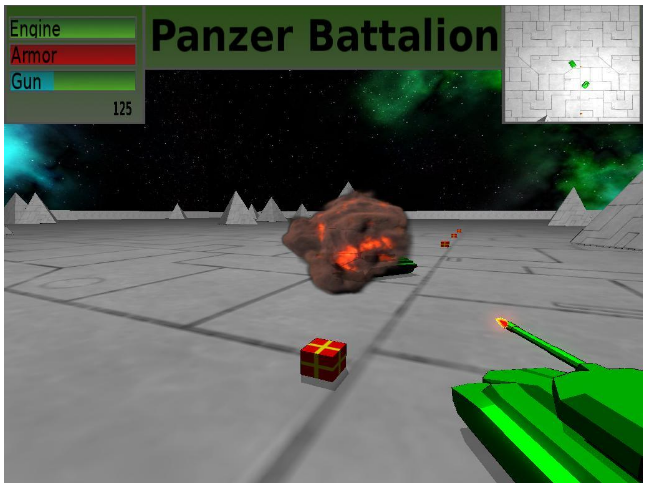

Figure 1: "Panzer Battalion" screenshot.

can shoot at each other, and in contrast to BZFlags, shots hit instantaneously, i.e., the velocity of a shell is infinite. Additionally, the plane is full of obstacles that block the view, and players can hide behind them. Figure 1 also shows some power-ups that can be collected in order to up-level the power of each tank. Power-ups include armor strengthening, increasing the gun fire rate, the gun's effectiveness, the engine speed, and additional shields. On the other side there are also booby-traps destroying a tank's armor. Indeed each power-up looks the same, and consuming them thus introduces a momentum of surprise.

Tank movement, graphics, and the game physics have been designed to be as realistic as possible in order to maximize the gaming fun. We think that real experiments with real human subjects need a game that maximizes the gaming experience.

3.1 Message Handling

"Panzer Battalion" implements a fully decentralized network topology based on the P2P paradigm. Each peer actually simulates its own version of the entire virtual game world. This means that the entire game logic is being interpreted on every instance of the game, for the whole game. Another important point is that every instance communicates with every other instance directly instead of using a central server. Since each peer computes its own game version, no player is having disadvantages because of high network lags. On the other hand, as was pointed out previously, inconsistencies may occur and may cause different, contradicting game states on different peers. The advantages of such a system are, for one, a complete removal of a single point of failure, meaning there is no game instance without which the game would fail. Additionally, as instances interact directly, we avoid the situation that certain players "far away" (large round trip time) from the server cannot play the game due to high lag. In "Panzer Battalion" however, while the interaction with players "near" to an instance would be smooth, the interaction with others will suffer sufficient lag. Because of this a situation might occur where different instances of the game would have different game states. In order to further discuss the problem in the next section the design of "Panzer Battalion" is described.

3.2 Game Engine Design

The entire game logic is based on two message queues: the *inner* and *outer* queue (see Figure 2).

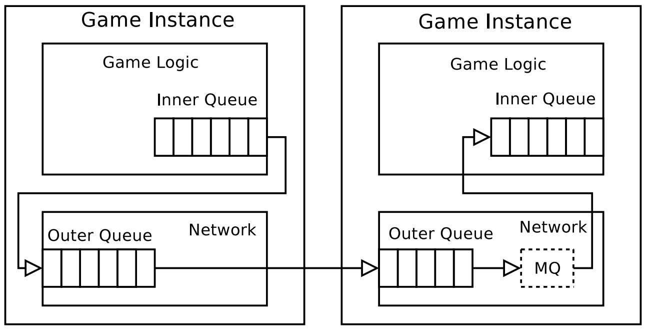

Figure 2: Flow of messages through the inner and outer queues.

The inner queue is responsible for all the events relevant to the game play itself. For example a message containing the data for the move of a particular object is processed in the inner queue. The outer message queue handles all network based tasks, such as connecting, transmitting messages, clock synchronization etc. The meaning of "MQ" is to emulate network latency (see Section 3.4). Every message has a time stamp, and messages are ordered in ascending order in the inner queue. The only messages that are sent over the network are those containing the action of the *tank* controlled by the player playing that particular instance of the game. In case a tank does something, for instance move or shoot, a message is put into the inner queue of the tank's game instance, and in case it is relevant to others, it is put into the outer queue as well (and subsequently sent to every other game instance). Upon receiving a message from another peer, the message is first put into the outer queue. From there it is passed on to the inner queue, where it is finally executed. The game logic, the rendering and input handling is asynchronous with the inner queue and synchronous with the outer queue. Table 1 shows the messages that are sent between the different game instances (peers).

Message	**Interpretation**
Move to location	A tank appears a certain location
Change angles	Viewing direction of a tank
Shoot	A tank shoots
Dead	A tank broadcasts that it has died
Damage	A tank receives damage
Respawn	A dead tank comes back to life

Table 1: Messages sent between peers.

In order to be able to determine inconsistencies, we synchronize the computer clocks of the peers as accurately as possible. This is done by using a protocol similar to the network time protocol (NTP) [18]. Before sending a message, the system time of a peer is determined by calling SDL GetTime(). This time stamp is then added to the message and sent to other peers.

3.3 States, Inconsistencies and Rollbacks

We define an inconsistency as an occurrence of the following situation: Assume that $N > 1$ peers send a number of messages to each other, and that each message determines an event that is applied a certain time stamp. Further assume that time stamps are all different due to high resolution clocks. Due to this time stamp, all messages can be ordered. At one particular peer the messages arrive at its inner queue and are executed. Due to network latencies, this execution of messages might not be done in the same order, as imposed by the message time stamps. This out-of-order execution leads then to a game state S_1. Now assume that executing the messages in-order would actually lead to a game state S_2. We say that an inconsistency has occurred if $S_1 \neq S_2$.

In order to remove inconsistencies, we implemented Time Warp. "Panzer Battalion" records its entire game state every 100 ms. Figure 3 a) shows the inner queue of any game instance. The execution pointer points to the next message to be executed. The messages left to this pointer have already been executed, the messages to the right are yet to be executed. At periodic time instances, the states are saved.

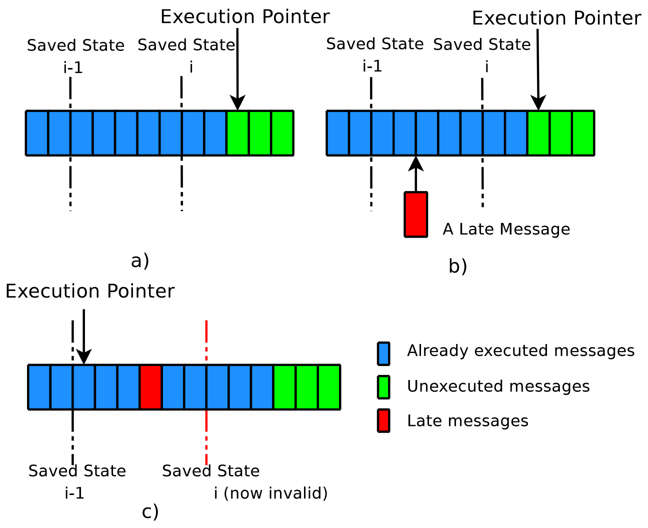

Figure 3: A late message arrives and triggers a rollback.

Figure 3 b) shows the reception of a late message in the inner queue. At this time point it is not known whether this message would actually cause an inconsistency. The game instance then decides to rollback to the saved state before this straggler, in this case state S_{i-1} (see Figure 3 c)). The execution pointer is set to the message right after S_{i-1} and all messages after S_{i-1} are actually re-executed. An example for a rollback and its possible consequences are shown in Figures 4 and 5. First a tank A shoots another tank B, destroying it as a result. However, shortly after a rollback is carried out since the game instance of tank A did not get a shoot message from B in time, stating that actually B shot first. Upon arrival of this straggler, A is forced to rollback, and tank A is destroyed by B's shot.

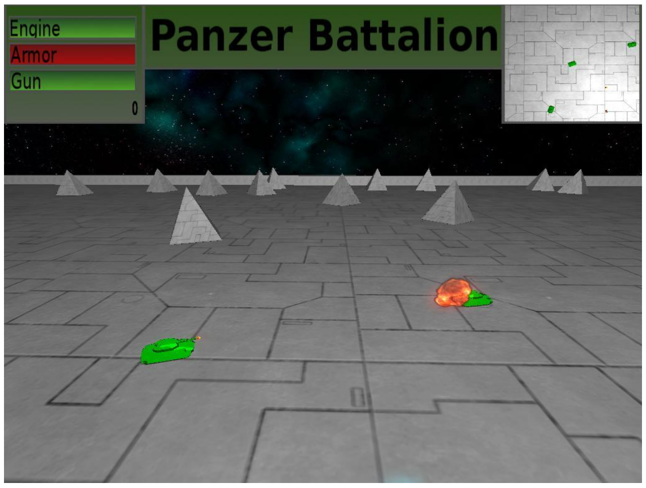

Figure 4: Before a rollback: a tank A (left) shoots a tank B (right).

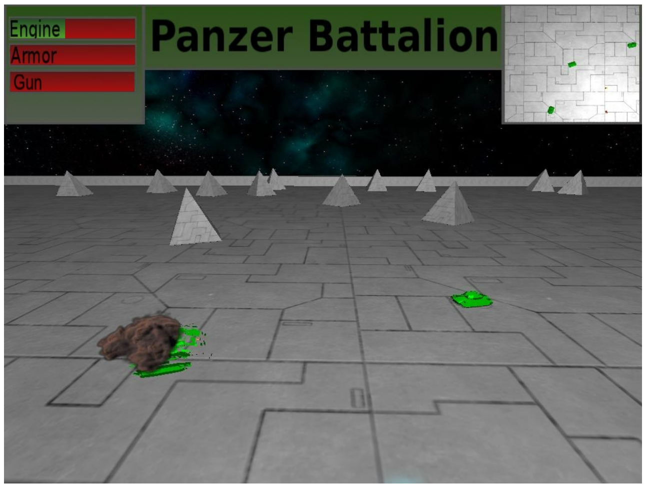

Figure 5: After the rollback: Tank B shoots tank A.

A well-known problem of Time Warp is given by inconsistencies on one peer causing inconsistencies on other peers. Suppose that peer A sends a message m, and then later receives a late message m' which contains an event that happened before that of m, and because of which m should *not have been sent*. Peer A is now aware of this fact and carries out a rollback, but since it already has sent m to all other peers (and m should not have been sent), the other peers already have received and executed m, leading to further inconsistencies (see Figure 6 a)). The game instance that received this late message now knows of this fact, but of course other peers do not. The message m should never have been transmitted. This problem is solved through implementation of *null-messages*. These are messages saying that a certain message should not have existed and was false. In our previous example, after receiving m', A transmits $\bar{m}$

which voids the already sent message m. Any other instance upon receiving of $\bar{m}$ (see Figure 3 b)) does a rollback, and skips the execution on the message m that was voided, thus synchronizing its state with the other instances (see Figure 3 c)).

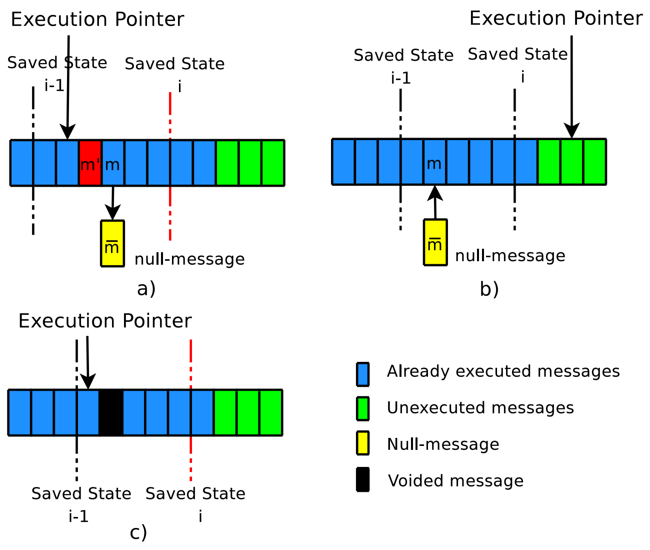

Figure 6: A null-message arrives and triggers a rollback.

However rollbacks are rather expensive and should be avoided as much as possible. Messages lagging only several milliseconds need not cause a time warp. In addition, a full time warp may not always be desirable, even if it prevents a situation that should not have happened. As an example imagine a situation, where a player A kills player B. However a rollback occurring a few seconds later shows that B actually was supposed to survive the shot. A looses the frag and B is warped back, most likely in the middle of the battle, only to die again, which ends up being unpleasant for all players.

Even with rollbacks and null-messages it is hard to be certain that the states in all instances are the same. Additional measures that are used is that all messages should be self-sufficient. This means that every message contains all the data required to make the change in the game state without depending on information not contained in the message. As an example, a message bearing date of object translation should not be implemented as: "translate object A by a vector V", as the end result depends on the current state of the object A. Instead the movement should be done as such: "move object A to position P". Another example would be shooting. Instead of "the object A shot" a message looking like "an object A positioned at P with rotation angle R shot" is more appropriate. This ensures that if there was an inconsistency in, say, a position of an object, after receiving the next move message the inconsistency would be corrected.

3.4 Network Latency Emulation

In order to assess the effect of network latency in a decentralized gaming scenario, "Panzer Battalion" implements an artificial latency unit. Messages are delayed in a queue between the outer and inner queue, called *middle queue* (see "MQ" in Figure 2). A message that should be moved from outer to inner queue is temporarily stored in the middle queue for some predefined time dt. This artificial lag can be adjusted per game instance during gameplay. It must be noted that the resulting behavior is actually different from Local Lag, since during its stay in the middle queue, the message is not seen by the inner queue, and hence not taken into account when executing other messages.

It is interesting to note that the render loop consumes most of the computing performance of the game, and itself introduces a Local Lag. As a result, even if the emulated network lag of the middle queue is set to $dt = 0$, several messages are often passed from the middle queue to the inner queue at once or in short successions, making them executed in the same pass in the inner queue, and the inconsistencies caused by them are fixed in a single rollback.

4. EXPERIMENTAL RESULTS

The experiments were held at our laboratory. We used several Linux PCs interconnected by FastEthernet running at 30fps average. In these first experiments we aimed at finding general dependencies of Time Warp rollbacks on network latency. Second we wanted to examine the behavior of human players in a networked game with high latency in a decentralized, P2P system using Time Warp. Both issues are treated in the next two sections.

4.1 Rollbacks

In a first set of experiments in each experiment two persons played against each other for five minutes. The network lag in the middle queue was set to $dt = 0, 200, 400$ and 600 ms. Each experiment was run two times, resulting in a total of eight experiments. The games were run at 60 frames per second on average. We recorded the number of rollbacks that occurred within these 5 minutes, and the maximum time a message had been late for each rollback.

Table 2 shows the average number of rollbacks that occurred at each peer, depending on the emulated network lag. It can be seen that this number first rises sharply, but decreases after 200 ms. An explanation for this is provided below.

Lag (ms)	Average number of rollbacks
0	285
200	630
400	462
600	301

Table 2: Average number of rollbacks for different lags.

In our experiments for each rollback we computed how late each message causing the rollback was (a rollback can be caused by more than one message), and for each rollback recorded the maximum value of these times. Figures 7, 8, and 9 show the histograms of these lateness values.

The first game shown in Figure 7 was recorded without any artificial lag and caused on average 285 rollbacks per player. Even if the artificial lag created by the middle queue was set to $dt = 0$, inconsistencies occurred, for once due to the render loop, and second, due to small offsets of the peer clocks. As a result, over 50% of those rollbacks were caused by messages late less then 50 ms, and nearly 90% less then 100 ms. It is interesting to note that in these experiments the effects of rollbacks were not visible to the players.

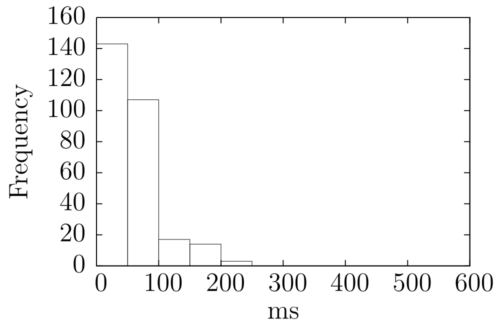

Figure 7: Number of rollbacks depending on how late a message was (No Lag).

Figure 8 shows the rollback lateness distribution of the same game with a 200 ms lag. In five minutes there were 630 rollbacks per player on average. The players noted the game at this point was somewhat irritating, and the effects of the rollbacks were often visible. However, the game still was acknowledged as being playable.

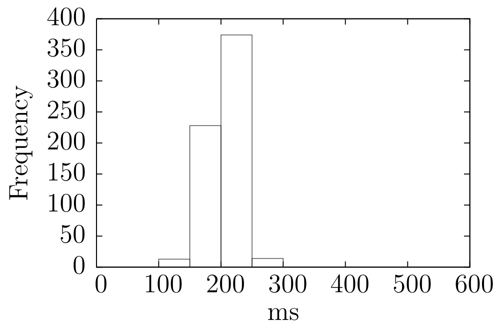

Figure 8: Number of rollbacks depending on how late a message was (200 ms lag).

Finally, Figure 9 shows the rollback lateness distribution in the case of a 400 ms lag. There were 462 recorded rollbacks per player on average, which is about 27% less than in the 200 ms case. This is due to having many late messages executed in single a rollback, and also, as will be pointed out in the next section, due to a change of tactics by the players, shifting from a dynamic to a more static gameplay. The effects of rollbacks here were very annoying. Players stated that the game in effect was no more playable.

In the 600 ms case players were no more able to control their gameplay. Rollbacks were actually clearly visible as such. The gaming experience was disastrous, even though the number of rollbacks decreased to the order of magnitude of the zero lag case (see next section).

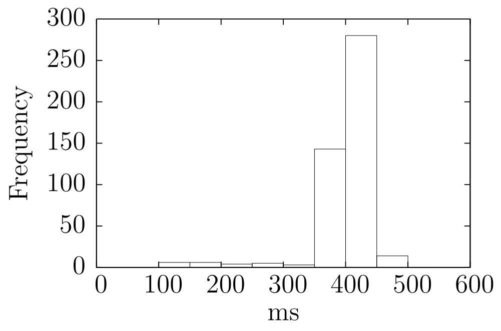

Figure 9: Number of rollbacks depending on how late a message was (400 ms lag).

4.2 Player Behavior

Another set of games were played in which the playing *behavior* was monitored. In each experiment three players were involved. The player input was recorded, and categorized into two categories: *dynamic* and *static*. Dynamic includes moving the tank, evading being shot, racing for power-ups etc. Static input includes aiming and shooting enemies. The games started with 0 ms lag, and during the game the lag was constantly increased to the values 200, 400 and 600 ms. Table 3 shows the percentage of dynamic vs. static input.

Lag (ms)	Dynamic (%)	Static (%)
0	59.3	40.67
200	48.9	51.0
400	26.4	73.5
600	21.0	78.9

Table 3: Dynamic vs. static.

The experiment shows that with the lag increase, tactics started being more static. In the end when the lag was around 600 ms, for most cases the game could be broken down to the following: a player would respawn, rush to the killing ground, where the battles were fought over and over again. Then he/she would stop and spend the rest of that life aiming at and shooting other players, with minimal evasion tactics. When asked about why this tactic was chosen, players replied that it was getting hard to earn the frag, and thus the general tactic was to prefer to be able to shoot someone, and increase the risk of being shot yourself. Aiming and assessing where the opponent actually is was more difficult when moving, and thus movements were stopped.

In addition the players replied that it was getting hard to guess the intentions of other players, and often realizing too late that they were in danger. While this is partially due to to the lag itself, players said that introducing a clearly visible rollback often created confusion, for example it was harder to keep track of one's relative position to other players in short distance battles. This was more visible with the local lag being greater. Players first noticed this with the lag being about 400 ms, and in games with artificial lag of 600 ms shot distance, they tried to avoid fights at short distance.

5. CONCLUSIONS

In this paper we introduce "Panzer Battalion", a testbed for assessing the effects of rollbacks on gameplay in a tank battle game, and how these rollbacks depend on network conditions. We put considerable efforts into the development of "Panzer Battalion", in order to generate a realistic positive gaming experience. Due to the development from scratch, we were able to make important design decisions for our testbed without being forced to alter third party source code.

In a first set of experiments we evaluated the effect of network lag and rollbacks. We showed that for increasing network lag, after reaching a maximum, the number of rollbacks drops again, due to the fact that more and more messages cause only one rollback.

We also showed how gamers change their behavior, away from a dynamic to a more static behavior in case the network lag grows too large. In general for large lags, Time Warp still may provide consistency, but this comes with a price. The game gets increasingly unplayable, and when using Time Warp rollbacks too far into the past must be avoided.

6. REFERENCES

[1] S. Aggarwal, H. Banavar, A. Khandelwal, S. Mukherjee, and S. Rangarajan. Accuracy in dead-reckoning based distributed multi-player games. In *NetGames 2004*, 2004.

[2] T. Beigbeder, R. Coughlan, C. Lusher, J. Plunkett, E. Agu, and M. Claypool. The effects of loss and latency on user performance in unreal tournament. In *NetGames 2004*, 2004. http://www.sigcomm.org/sigcomm2004/netgames.html.

[3] J. Brun, F. Safaei, and P. Boustead. Server topology considerations in online games. In *NetGames 2006*, 2006.

[4] M. Claypool. The effect of latency on user performance in real-time strategy games. *Computer Networks*, 49(1):52–70, 2005. Special issue: Networking issues in entertainment computing.

[5] E. Cronin, B. Filstrup, A. R. Kurc, and S. Jamin. An efficient synchronization mechanism for mirrored game architectures. In *NetGames 2002*, 2002.

[6] M. Dick, O. Wellnitz, and L. Wolf. Analysis of factors affecting playersŠ performance and perception in multiplayer games. In *NetGames 2005*, 2005. http://www.research.ibm.com/netgames2005/.

[7] T. Fritsch, H. Ritter, and J. Schiller. The effect of latency and network limitations on mmorpgs (a field study of everquest2). In *NetGames 2005*, 2005.

[8] S. Gargolinski, C. S. Pierre, and M. Claypool. Game server selection for multiple players. In *NetGames 2005*, 2005. http://www.research.ibm.com/netgames2005/.

[9] S.-Y. Hu, J.-F. Chen, and T.-H. Chen. Von: A scalable peer-to-peer network for virtual environments. *IEEE Network*, 20(4):22– 31, 2006.

[10] T. Iimura, H. Hazeyama, and Y. Kadobayashi. Zoned federation of game servers: a peer-to-peer approach to scalable multiplayer online games. In *NetGames 2004*, 2004.

[11] D. R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(3):404–425, 1985.

[12] P. Kabus, W. W. Terpstra, M. Cilia, and A. P. Buchmann. Addressing cheating in distributed mmogs. In *NetGames 2005*, 2005. http://www.research.ibm.com/netgames2005/.

[13] B. Knutsson, H. Lu, W. Xu, and B. Hopkins. Peer-to-peer support for massively multiplayer games. In *IEEE Infocom 2004*, 2004.

[14] K.-W. Lee, B.-J. Ko, and S. Calo. Adaptive server selection for large scale interactive online games. *Computer Networks*, 49(1):84–102, 2005. Special issue: Networking issues in entertainment computing.

[15] D. Liang and P. Boustead. Using local lag and timewarp to improve performance for real life multi-player online games. In *NetGames 2006*, 2006.

[16] M. Mauve. How to keep a dead man from shooting. In *Proc. of the 7th International Workshop on Interactive Distributed Multimedia Systems and Telecommunication Services (IDMS) 2000*, 2000.

[17] M. Mauve, J. Vogel, V. Hilt, and W. Effelsberg. Local-lag and timewarp: Providing consistency for replicated continuous applications. *IEEE transactions on Multimedia*, 6(1):47–57, 2004.

[18] D. L. Mills. Network time protocol (version 3). RFC 1305, March 1992.

[19] J. Müller, A. Gössling, and S. Gorlatch. On correctness of scalable multi-server state replication in online games. In *NetGames 2006*, 2006.

[20] K. L. Morse, L. Bic, and M. Dillencourt. Interest management in large-scale virtual environments. *Presence: Teleoperators and Virtual Environments*, 9(1):52–68, 2000.

[21] W. Palant, C. Griwodz, and P. Halvorsen. Consistency requirements in multiplayer online games. In *NetGames 2006*, 2006.

[22] L. Pantel and L. C. Wolf. On the suitability of dead reckoning schemes for games. In *NetGames 2002*, 2002.

[23] J. D. Pellegrino and C. Dovrolis. Bandwidth requirement and consistency resolution latency in multiplayer games. In *NetGames 2003*, 2003.

[24] A. E. Rhalibi, M. Merabti, and Y. Shen. Aoim in peer-to-peer multiplayer online games. In *Proceedings of the 2006 ACM SIGCHI international conference on Advances in computer entertainment technology*, 2006.

[25] G. Schiele, R. Suselbeck, A. Wacker, J. Hahner, C. Becker, and T. Weis. Requirements of peer-to-peer-based massively multiplayer online gaming. In *Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGRID 2007)*, pages 773–782, 2007.

[26] S. D. Webb, S. Soh, and W. Lau. Enhanced mirrored servers for network games. In *NetGames 2007*, 2007.

[27] T. Yasui, Y. Ishibashi, and T. Ikedo. Influences of network latency and packet loss on consistency in networked racing games. In *NetGames 2005*, 2005. http://www.research.ibm.com/netgames2005/.

[28] Y. Zhang, L. Chen, and G. Chen. Globally synchronized dead-reckoning with local lag for continuous distributed multiplayer games. In *NetGames 2006*, 2006.