Aalto University
School of Science
Degree Programme in Computer Science and Engineering

Joni Toiviainen

Consistency-responsiveness Tradeoff Evaluation for Consistency Maintenance in Multiplayer Online Games

Master's Thesis
Espoo, March 03, 2016

Supervisor: Professor Jukka K. Nurminen, Aalto University
Advisor: Teemu Kämäräinen M.Sc. (Tech.)

Aalto University
School of Science
Degree Programme in Computer Science and Engineering

ABSTRACT OF
MASTER'S THESIS

Author: Joni Toiviainen

Title:
Consistency-responsiveness Tradeoff Evaluation for Consistency Maintenance in Multiplayer Online Games

Date: March 03, 2016 **Pages:** vii + 62

Major: Software Technology **Code:** T-106

Supervisor: Professor Jukka K. Nurminen

Advisor: Teemu Kämäräinen M.Sc. (Tech.)

Multiplayer online gaming has become a massive part of the game industry. Games that use network connections are known to be vulnerable to communication problems like latency, jitter and packet loss. These problems may cause players to suffer from delayed responsiveness or weird game entity behaviors.

Games typically fight against network problems by using consistency maintenance methods which need to perform a balancing act between the game's consistency and responsiveness. A huge variety of techniques exists, but there has not been a clear guideline about how a suitable technique should be selected.

This Thesis proposes a categorization for the consistency maintenance techniques based on their ability to handle time or data. Categorization is also evaluated with a user study by implementing a single technique from each category and testing them with a simple game.

Results from the evaluation are further used to form an analysis for each proposed category. Analysis is used to make small conclusions about the advantages and disadvantages for each category together with an suggestion of how to balance with the consistency-responsiveness ratio to maintain the playability of the game.

Pessimistic techniques are considered to provide good consistency in the cost of responsiveness and vice versa for optimistic techniques. PPT is considered as a good solution for a game that does not require real-time simulation, while PDT works well with a game that requires high consistency and may tolerate long response times. OPT is considered as a suitable solution for a game which contains easily predictable shared state entities and ODT is recommended for a game that contains clear decision points.

Keywords: consistency-responsiveness trade-off, latency, consistency maintenance, latency compensation, multiplayer online games

Language: English

Aalto-yliopisto
Perustieteiden korkeakoulu
Tietotekniikan koulutusohjelma

DIPLOMITYÖN TIIVISTELMÄ

Tekijä:	Joni Toiviainen		
Työn nimi:	Yhteneväisyyden ja reagointikyvyn tasapainon tutkiminen verkkomoninpelien yhteneväisyyden ylläpidossa		
Päiväys:	3. maaliskuuta 2016	**Sivumäärä:**	vii + 62
Pääaine:	Ohjelmistotekniikka	**Koodi:**	T-106
Valvoja:	Professori Jukka K. Nurminen		
Ohjaaja:	Diplomi-insinööri Teemu Kämäräinen		

Verkkomoninpelit ovat nousseet massiiviseksi osaksi peliteollisuutta. Verkkoyhteyksiä käyttävät pelit ovat tunnettuja siitä, että ne kärsivät helposti yhteyteen liittyvistä ongelmista kuten viiveestä, yhteyden huojunnasta ja pakettihävikistä. Nämä ongelmat voivat aiheuttaa pelin reagointikyvyn heikkenemistä tai pelissä esiintyvien objetien erikoista käyttäytymistä.

Pelit kamppailevat verkon ongelmia vastaan yleensä käyttämällä erilaisia yhteneväisyydenhallintamenetelmiä, jotka tasapainottelevat pelin yhteneväisyyden ja reagointikyvyn välillä. Laaja valikoima erilaisia tekniikoita on jo saatavilla, mutta niiden valintaan liittyvää selkeää ohjeistusta ei ole vielä kehitetty.

Tämä työ esittää tavan luokitella yhteneväisyydenhallintatekniikat perustuen niiden kykyyn hallita aikaa tai tietoa. Tuloksena syntyneet kategoriat myös testataan käyttäjätestauksessa pienen pelin avulla, johon toteutetaan yksi tekniikka jokaisesta kategoriasta.

Testauksen tuloksista muodostetaan analyysi jokaiselle ehdotetulle kategorialle. Lopuksi analyysiä käytetään esittämään pieniä johtopäätöksiä jokaisen kategorian vahvuuksista ja heikkouksista. Lisäksi esitetään ehdotus siitä, miten yhteneväisyyden ja reagointikyvyn tasapainottelukertoimia on mahdollista hyödyntää pelin pelattavuuden ylläpidossa.

Pessimististen tekniikoiden päätellään tuottavan hyvää yhteneväisyyttä alennetun reagointikyvyn kustannuksella, kun taas optimiset tekniikat mahdollistavat nopean reagointikyvyn alennetun yhteneväisyyden kustannuksella. PPT toimii hyvin sellaisten pelien kanssa, jotka eivät vaadi reaaliaikasimulaatiota. PDT puolestaan toimii hyvin niiden pelien kanssa, jotka vaativat korkean yhteneväisyyden ja pystyvät hyväksymään pitkän reagointiajan. OPT toimii hyvin, mikäli pelin jaetun tilan objektit ovat helposti pääteltävissä. ODT:ta puolestaan suositellaan käytettäväksi, kun pelin päätöksentekopisteet ovat helposti määritettävissä.

Asiasanat:	yhteneväisyys-responsiivisuuden tasapainotus, latenssi, yhteneväisyyden ylläpito, latenssin kompensaatio, verkkomoninpelit
Kieli:	Englanti

Acknowledgements

I would first like to thank Prof. Jukka K. Nurminen and thesis advisor Teemu Kämäräinen for all their support and guidance. Our conversations gave a numerous amount of new ideas and sights about the thesis topic. Thank you for the valuable information that helped me throughout the thesis.

I would also want to thank all user study participants for their effort. Without your help, the evaluation part of this thesis would not have been possible. Thank you for spending your time and being a part of this thesis.

I also want to thank my friends and family for being supportive throughout the time I have worked with my studies. You gave me plenty of new ideas and support throughout my long days spent with this thesis. I am the luckiest man in the world to have you all around.

Above all, I would like to express my profound gratitude to my beloved Erika for the continouos encouragement and support during my years of study. These accomplishments would not have been possible without your love and support. Words cannot describe how much I love you.

Espoo, March 03, 2016

Joni Toiviainen

Abbreviations and Acronyms

CI	Confidence interval
CM	Consistency maintenance
CPU	Central processing unit
CSV	Comma-separated values
C-S	Client-server
DIA	Distributed interactive application
FPS	First-person shooter
IR	Input response time
MOG	Multiplayer online game
ODT	Optimistic delay technique
OPT	Optimistic presence technique
P2P	Peer-to-peer
PDT	Pessimistic delay technique
PPT	Pessimistic presence technique
RAM	Random access memory
RR	Remote response time
RTS	Real-time strategy
RTT	Round-trip time
UX	User experience

Contents

Chapter 1

Introduction

Multiplayer online gaming has become a massive part of the game industry. Some multiplayer games like Dota 2 [3] and Counter Strike: Global Offensive [2] currently have millions of dedicated players around the world. It is needless to say that online gaming has established its place in the gaming mainstream and also deserves a great focus on the fields of research.

Multiplayer online games use network connections to communicate between participated computers around the globe. These networks are known to be vulnerable to network challenges like latency, jitter and packet loss, which are also commonly known as a *lag*. Lag may issue odd behavior to game entities by making players to not hit targets, making characters to move in weird fashion or by halting the screen for some amount of time.

Even some popular games have suffered from the negative effects of lag right after their release date. One of these games is Battlefield 4 [1], where some players were affected by the lag effect called *rubber banding* that makes players to warp back in time by their positions. Developers took the issue as a top priority problem as it had a huge negative impact on the gaming experience. Rubber banding is also only a single example from huge variety of negative effects that can affect the game when it is affected by a lag. All these lag issues should be avoided as they are concluded to have a huge impact on the gaming experience by making players frustrated or annoyed [21].

Most games try to minimize the negative effects of the lag with consistency maintenance techniques. These techniques try to ensure that the game runs smoothly and consistently when the latency impacts the network connection. Efficient usage of these techniques is not trivial, as techniques can be complex and may be used individually or by mixing multiple techniques together. Different techniques also have advantages and disadvantages that must be considered when selecting a good set of techniques to be used.

1.1 Problem statement

The main objective for this Thesis is to study and evaluate how consistency maintenance techniques should be selected for multiplayer online games that have some specific interaction and consistency requirements. Objective can be written as a research question in the following way.

RQ: *How to select a suitable consistency maintenance technique for a networked multiplayer game with a particular responsiveness and consistency requirements?*

To achieve the answer for the question, we need to study advantages and disadvantages of different consistency maintenance techniques. As there are huge variety of different techniques, we need to categorize techniques so we can restrict our evaluation process. Therefore, the main research question can be divided into following three subquestions.

RQ1: *How to perform the major categorization for the consistency maintenance techniques commonly used in the multiplayer online games?*

RQ2: *What are the major advantages and disadvantages for each consistency maintenance technique category?*

RQ2: *What factors should be considered when deciding which consistency maintenance technique to use with the multiplayer online game with some specific responsiveness and consistency requirements?*

1.2 Scope of the Thesis

Thesis concentrates to study the major consistency maintenance technique algorithms that fight against lag in multiplayer online games. We assume that all games use a client-server architecture with a single server instance and therefore we will exclude multi-server environments and peer-to-peer (P2P) consistency maintenance problems out of the scope. However, it does not mean that these games would not be able to benefit from the results.

Thesis will focus on software issues and inconsistency prevention systems that are affected by the network delays. We will exclude all external physical and logical challenges that are related to network lines or data transfer systems which are not part of the actual game engine or logic code.

1.3 Research methods

Thesis gathers background and related work information from journal articles, conference papers and reports from research institutions. Information is used to receive a solid understanding about the current state of the topic. In addition, some minor details are also gathered from websites to provide up-to-date statistics.

Gathered knowledge is used to plan and implement a small simple test game for the empiric study. Implementation is done with the C++ language using the SDL and Boost Asio libraries together with the Emscripten cross-compiler. Game was tested with 26 people to gather player experience statistics about the suitability of different consistency maintenance technique categories. Statistic information was used and compared with background information to make conclusions and answers to presented research questions.

1.4 Structure of the Thesis

Chapter 1 contains the introduction about the Thesis. Chapter 2 dives into the theory and the current state of the research about the topic. Chapter 3 contains the planning and implementation phase for the testing environment where we build the demo game. Chapter 4 contains the testing phase where real human players are used to test different consistency maintenance methods. Chapter 5 contains a discussion and analysis about the evaluation results and finally Chapter 6 contains the conclusions based on the results from the previous chapters.

Chapter 2

Background

The definition of a multiplayer online game (MOG) refers to a multiplayer game where players use a client software to process a network communication with a server or directly with other players. Communication is used to distribute game events to maintain a consistent shared state between all connected participants, which are also known as *nodes*. Issued game events are typically distributed immediately between the participating nodes, which allows participated players to interact with the same game world in real-time.

Each node is running a game instance that has a *game state*, which defines a current state and behavior of all entities within a game world. Game state is updated regarding to the game events and the inputs provided by the players. The way how the game state is updated is specified by the networking architecture and the way how the player interacts with the game.

Network communication contains data transfer challenges like packet loss and delays, which may make the game to behave in an inconsistent or unresponsive way. These challenges are typically fought against with consistency maintenance (CM) techniques that try to balance between the consistency and the responsiveness of the application. Achieving an absolute consistency with absolute responsiveness is considered to be impossible because of the a *consistency-responsiveness tradeoff* of the techniques. [31]

This chapter starts by gathering information about distributed interactive applications (DIAs) that are used as the basis structure for all MOGs. Then we continue by exploring what is the definition for a consistency and responsiveness and how do they relate on each other. And finally we study what kind of consistency maintenance technique categories exists and introduce some examples from each category.

2.1 Distributed interactive applications

DIAs are computer programs that are connected by a communications network and can interact with each other in a shared virtual environment [12]. Shared virtual environment contains a *global state* that presents a position and behavior of all entities within the environment. Global state is replicated to all nodes as a *local state*, which presents the state of the environment on a single node. User may interact with the shared virtual environment by issuing events on a local computer, which is also referred as a *terminal*. Issued events are delivered to other nodes directly or through a centralized node. Demonstration about state replication is shown in Figure 2.1.

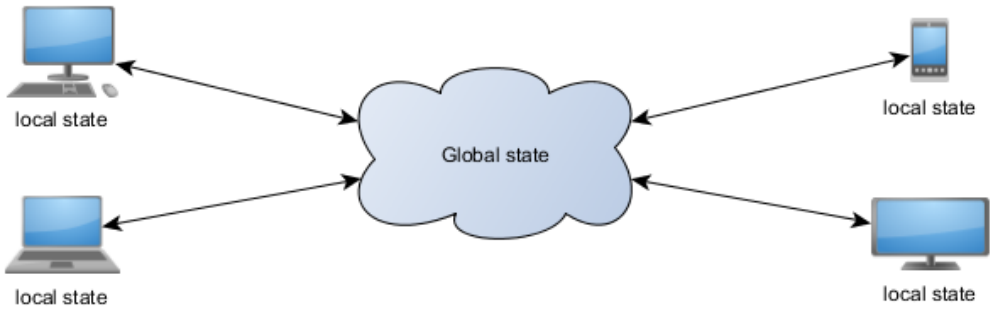


Figure 2.1: Terminals maintain a local version of the DIA's global state.

User issued events are used to make changes to global state in addition to application's simulation events. However, authoritative modification to the global state is performed by the *decision points*. These decision points are either dedicated servers or users computers, which may either deny or allow state changes requested by other nodes. The way how decision points communicate with each other is based on the used networking architecture. [8]

DIAs can be categorized based on two different key features: a used network topology and the different types of interactions that a DIA provides for the users [31]. Network topology can be considered as a main feature to define how different nodes communicate and form the networking structure for the application. Different interactions define how the users may communicate with the virtual environment and how events are applied to states.

2.1.1 Network architectures

Network architecture defines the main decision points and the way how different nodes may communicate with each other. It is an important aspect of DIA as it defines the propagation connections for the state synchronization and event distribution. These communication architectures can be split into three categories: centralized, distributed and hybrid solutions [4, 29, 31].

Centralized solution contains a node instance that acts as an *server* and performs a two-way communication with other nodes known as *clients*. This kind of network architecture is typically known as a client-server (C-S) architecture. C-S requires all communication to go through the server node, which is in contrast to distributed architectures that allow nodes to directly communicate with each other. Distributed architectures can be also referred as peer-to-peer (P2P) architectures, where each node acts as a *peer*. Hybrid solutions are used combine communication strategies from the C-S and P2P. The concept of the mentioned topologies are shown in the Figure 2.2.

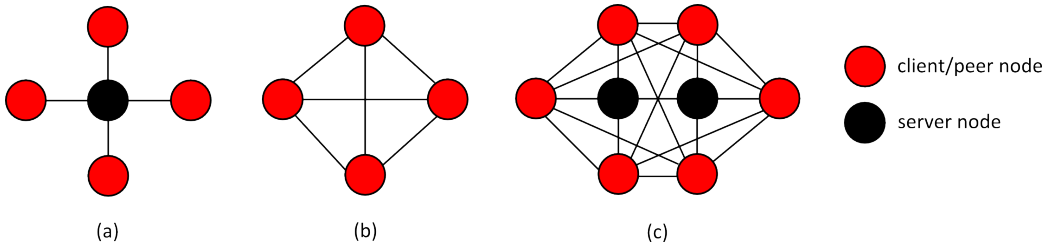


Figure 2.2: DIA network topology architectures: (a) C-S (b) P2P (c) hybrid

Each topology is considered to have advantages and disadvantages. Advantages of the C-S architecture are that it can be implemented to contain administrative features like cheating prevention and network data flow control [29]. However, the server may become a performance bottleneck impacting the speed of the whole DIA [4]. Advantages of the P2P are that it is straightforward to implement and there is no server as a performance bottleneck [4, 29]. Serverless architecture may provide better responsiveness to P2P when compared to C-S. On the other hand, P2P disadvantages are that it does not scale well without additional hierarchical structure [4, 29]. Hybrid solutions may be used to combine the benefits from P2P and C-S, but with the cost of adding more complexity to the architecture.

2.1.2 User interaction

User interaction types can be categorized based on how the entity state can be changed and how the participants take turns to interact with the entity. State of an entity can be changed either in *discrete* or *continuous* way. Discrete applications change the entity state by a result of events from the users or from the application logic, while continuous applications change entity state in a response of the passage of time in addition to other events. Participant interaction turns can be taken in *turn-based* or *concurrent* way. Turn-based applications allow only one user per time to interact with the entity at one time instance, while concurrent applications allow simultaneous actions on a same entity at the same time. The concept of the categorization is shown in the Figure 2.3. [31]

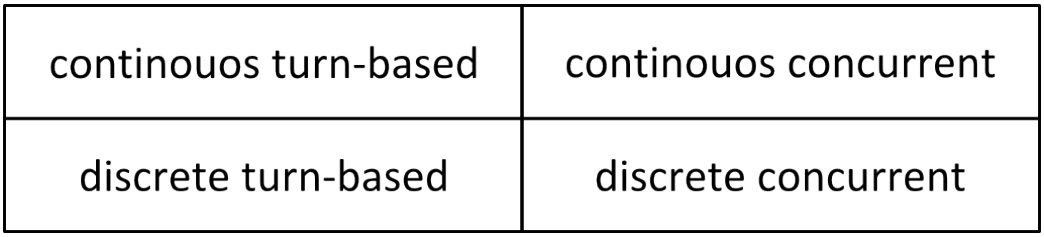


Figure 2.3: Dia categorization taxonomy based on the user interaction.

Example of continuous and turn-based applications are tennis games, where simulation is continuous but players take turns to interact with the ball. One example of a discrete and turn-based application is a traditional chess, where players take turns and entities move to target spots. Example of a discrete and concurrent application is a collaborative editing system, where a multiple users make changes to editable target. Finally an example about a continuous and concurrent application is a typical simulation based MOG like Call of Duty, which allows multiple players to interact with a continuously simulated world. The focus of this Thesis concentrates on the DIAs that belong in the continuous categories.

2.2 Consistency-responsiveness trade-off

Continuous and concurrent DIAs have four major features that are necessary to keep the application to mimic real-world interactions. First feature is the *causality*, which requires application to be able to handle events in a correct order and switch from one state to another. Second is the *concurrency* that requires multiple users to be able to interact with the same entities at the same time. Third one is the *simultaneity*, which requires simultaneous events to be shown as simultaneous events to all participants. The last feature is the *instantaneity* that requires a short response time for actions. These features are also listed within the Table 2.1. [7, 31]

causality	update state according to received events.
concurrency	allow users to interact with entities at the same time.
simultaneity	show simultaneous events simultaneously.
instantaneity	provide short response time for input events.

Table 2.1: Four major features for continouos and concurrent DIAs.

Achieving an absolute simultaneity with absolute instantaneity is considered to be impossible in DIAs due to network delays [13, 31]. Instantaneity requires the application to maintain fast *responsiveness*, which is typically achieved be applying events to the local state immediately. However, due to network delays, the simultaneity is not satisfied as events are applied and processed on remote nodes at different time points. As a result of non-simultaneity, local and remote states start to diverge towards inconsistency.

By the means of not being able to provide absolute simultaneity with absolute instantaneity, some amount of inconsistency must be tolerated by the MOG for the sake of the responsiveness [28]. A research by Brun et al. [8] stated that the game's consistency and responsiveness are highly related to each other. Their suggestion of *the playability space* defines that by decreasing the level of consistency it is possible to increase the the level of the responsiveness and vice versa. A similar conclusion was made by the study from Savery [24], which stated that application's CM can be though as a balancing act between the consistency and the responsiveness.

2.2.1 Consistency

Consistency refers to the game's ability to keep remote and local game states to as similar as possible. If these states are considered similar at all points of time, application can be considered to maintain an *absolute consistency*. Maintaining an absolute consistency in a practical DIA is considered to be impossible due the clock asynchrony between the nodes and the network delays that impact the game's event processing. However, CM techniques can be used to maintain consistency near the absolute consistency. [9, 13]

According to a study from Delaney et al. [13], consistency can be considered to refer to the causality, concurrency and simultaneity of DIAs features that were listed in the Table 2.1. By causality, consistency tries to ensure that events are applied in a correct order to global game state, which is then updated to new state to maintain a cause-effect order. By concurrency, consistency tries to resolve simultaneous events on a same entity in a reasonable way. Finally, the consistency tries to ensure simultaneity by trying to keep local game states between nodes to as similar as possible.

Consistency may also be defined as a set of *consistency requirements*. A study from Palant et al. [20] suggests to create the consistency requirements by focusing on the user-centered consistency, where clients do not necessarily have to be in similar state with other clients. Study suggest that requirements could be split into physical model restrictions, game critical state distribution and ensuring that player actions are reasonable also in other players views.

Study from Savery et al. [27] has a bit different approach by focusing on a entity-centric consistency requirements by suggesting that each entity-event relationship could have a different consistency requirements. They suggest that each entity type, interaction type, tolerable inconsistency and game-critical event should be considered separately. Study also categorizes the main consistency requirements into three categories: state divergence, propagation delay and timeline divergence. State divergence is the amount of difference between the view of the users at a certain point in time. Propagation delay is the amount of time that passes from the local state change to be propagated to other nodes. Timeline divergence is the extent how the player experience of game world events differ.

2.2.2 Responsiveness

Responsiveness refers to the time for the application to register and response on a input event [13]. Input events may be issued with input peripherals like a keyboard, mouse or gamepad. Application responds to events by providing a feedback with a output device like a computer screen or speakers. The time between issuing an event to receive the associated feedback may also be called as a *response time* [27] or *input lag* [16].

The amount of acceptable response time is determined by the way how the player views and interacts with the game [11]. Claypool and Claypool [10] found that omnipresent aerial games, like real-time strategy (RTS) games may tolerate delays up to 1000ms. In contrast, Raaen and Grønli [22] found that fast-paced games may be considered to tolerate only 45ms to 60ms. Response time tolerance varies between all games and it should be considered as a game specific value that should be evaluated for each game independently.

Networked applications may divide response time into *remote response time* (RR) and *input response time* (IR). IR is the time from issuing an input to receiving the associated feedback on the same terminal. RR is the time that passes before a remote user sees the locally issued action. [6] Both RR and IR are affected by the *local delays*, which are the delays introduced by the local hardware and software on each node [23]. Because RR contains the network transmission line, it is also affected by the network challenges like delays and packet loss. Challenges may also affect IR if the system requires a remote node to provide a response before local state can be updated.

Ivkovic et al. [15] found that local delays on a single node may vary from 23ms to 243ms. Highest local delays were introduced by the gaming consoles, where all delays were ranging from 130 ms to 243 ms. In contrast, fast action games in a PC environment did have the lowest local delays ranging from 23 ms to 65 ms. Similar results was achieved by a study from Raeen and Petlund [23], where authors found that the average local lag in a popular fast action Unreal Tournament 3 FPS is between 33 ms and 95 ms.

Responsiveness can be considered as a metric of the DIAs instantaneity feature that was listed in the Table 2.1. The instantaneity feature requires DIAs to provide short response times for input events, which can be also expressed as a requirement of a fast responsiveness. In contrast, long and noticeable delays are considered to provide slow responsiveness, which is typically considered to be annoying [21].

2.3 Consistency maintenance techniques

Consistency maintenance (CM) can be considered as a key feature in networked interactive applications. CM techniques try to ensure that all nodes keep their game states to as similar as possible while also trying to ensure that application stays responsive. Without any kind of CM, states would start to diverge and players would face strange phenomenas like warping characters or other jerky behaviors that would degrade the UX of the game.

CM techniques can be categorized into optimistic and pessimistic techniques based on their relation to consistency-responsiveness tradeoff [13, 31].

Optimistic CM techniques
: Optimistic techniques allow node to make immediate local state updates to maintain a high responsiveness. Node may also use technique to perform speculative computations to predict remote state updates before they arrive. However, if predictions go wrong, they must be corrected with some repair techniques like Time Warp. [13, 31]

Pessimistic CM techniques
: Pessimistic techniques require node's local state to be synchronized with the global state. Node may not perform any predictions and the local state is only updated when the same update is synchronized with all other nodes. [13, 31]

Optimistic techniques are especially good for applications that have easily predictable state updates or the cost of correcting divergence is relatively low. Pessimistic techniques are preferred when a high consistency is required and a degrade in responsiveness is tolerable. In general, optimistic techniques prefer responsiveness over consistency and vice versa for pessimistic techniques.

It is also completely allowed to combine multiple consistency maintenance techniques together to form the final solution. Final solution may be built from optimistic and pessimistic techniques, where a hybrid solutions may take advantages from each category. For example, optimistic techniques could be used to keep world simulation running smoothly while pessimistic techniques could be used to ensure that game-critical events are executed in a consistent way. According to some previous studies [11, 27], different game entities and events should be treated to contain different consistency requirements to other entities within the game world.

CM techniques may also be even further categorized based on different characteristics. There seems to not be a general way to do the categorization but several studies have made some suggestions. In this Thesis, we will form

a taxonomy which is based on the optimistic and pessimistic technique categorization. This makes a categorization suggestion from Savery et al. [28] to not directly fit into our requirements as it does not separate optimistic and pessimistic techniques. Their study suggests to split a client-server CM design space into three factors, which are the way how the inconsistency is prevented, how it is repaired and where the game critical decisions are made. Pessimistic CM techniques require states to be synchronized, which makes the suggestion to use the factor of repairing inconsistent states applicable only with the optimistic techniques. For the same reason, we ignore game-critical decisions as a CM categorization factor. Delaney et al. [13] would split CM techniques based on the major manipulation targets. However, their suggestion would also consider network packet transmissions and application architectures, which are not in the scope of this Thesis. A research from Stuckel and Gutwin [30] would split network latency management techniques into delay hiding, delay revealing and delay adding techniques. By collecting the information from all these three mentioned categorizations, we can construct a CM technique taxonomy that is shown in the Figure 2.4.

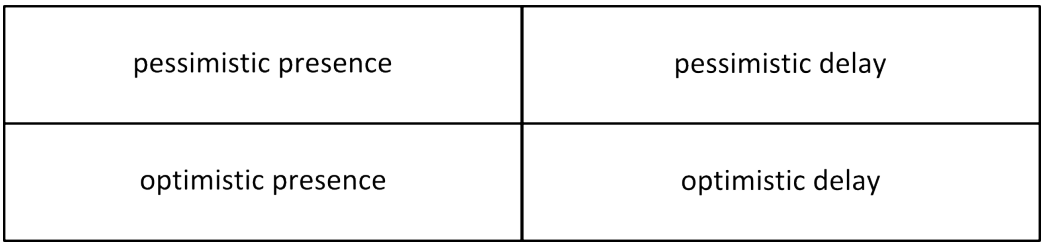


Figure 2.4: Taxonomy for CM based on the technique behavior.

Taxonomy categories are based on the optimistic and pessimistic behavioral axes. Presence techniques control game consistency-responsiveness by managing game entities and events with synchronization and prediction methods. Delay techniques introduce delays to various points of the system to compensate network delays. These artificially introduced delays are selected such they fit into the application's responsiveness requirements. Taxonomy is not meant to be exclusive and different techniques may be combined to form a final solution.

2.3.1 Pessimistic presence techniques

Pessimistic presence techniques (PPTs) require game state updates to be applied to each interested participant before the next simulation step can be processed. PPTs ensure high consistency by requiring strict synchronization of the game states. However, techniques may dramatically reduce application's responsiveness as all nodes must wait until event has reached all nodes.

One of the most basic PPT is the *pessimistic serialization* [14], which requires events to be sent to centralized node that broadcasts events to other nodes in a consistent order. Simulation can proceed into next step after all nodes have acknowledged that they have received and processed the event. Pessimistic serialization is strongly impacted by the network delays as the node with the highest lag does determine the impact on the responsiveness.

Another common PPT is the *pessimistic locking* [14], where nodes must acquire a lock before they may perform changes to shared context. The lock management is based on the lock requests and releases. When a node wants to handle the lock target, it must request a lock from the lock holder. Lock may be granted if it is free, but otherwise the requesting node must wait until the lock is released. Requests may be blocking or non-blocking, but without the lock the node cannot edit the lock target. When a node has finished manipulating the lock target, it will release the lock so it can acquired by other nodes. The *editable granularity* determines the object set that is protected by the lock. Coarse granularity requires less lock requests from the nodes, but also implies to less concurrency. Fine-grained locks allow better concurrency but require more lock requests. It is up to application developer to determine which kind of granularity should be applied based on the application's requirement of concurrency.

PPT techniques try ensure good consistency but their impact on responsiveness seems to violate DIA's instantaneity feature. Techniques are considered to be suitable choice when game-critical events are processed [27]. However, they should not be considered as a main CM technique with interactive MOGs. A study by Savery et al. [27] tested how serialization works as MOG's CM technique among other techniques. Serialization was considered as the worst technique from all tested techniques. Results show that serialization performance becomes much worse when the latency gets higher, where especially response times are straightly related to introduced network delays.

2.3.2 Optimistic presence techniques

Optimistic presence techniques (OPTs) allow some divergence between the nodes to achieve better responsiveness. Responsiveness is maintained by allowing nodes to apply local events into the local game state before the event is applied on the remote nodes. OPT allows the use of prediction techniques to continue simulation even when remote updates are not yet received.

Two of the most basic OPTs are the *optimistic serialization* and *optimistic locking* [14]. Optimistic serialization works like the pessimistic version by requiring that events are sent to centralized node that broadcasts events to other nodes. Unlike pessimistic version, optimistic version does not require node to wait until synchronization before events are transmitted, which improves the response time for events. Optimistic locking is based on the same idea than the pessimistic version, but it may assume that lock can be achieved without waiting until a lock is really granted. By assuming that lock is granted, the node may perform update operations to lock target before the response is received. When the response is finally received, it will determine how the node should behave. Node can continue normally if the lock is approved, while lock denial requires lock target to be rolled back.

One popular method for OPT is to use a prediction technique called *dead-reckoning*, which uses the previously known states to extrapolate the current state. Predictive methods can be used to continue the simulation when the game is impacted by the network conditions by predicting the remote state update before it is received. This is typically done by using the position, heading and velocities of the game entities to extrapolate their next positions. [24]

A typical method for extrapolating the position of a game entity may be calculated by using a basic physics formula like shown in the Equation 2.1. The extrapolated position $P(t_1)$ is calculated by using the previously known position $P(t_0)$, velocity $V(t_0)$, acceleration $A(t_0)$ and the time difference Δt from the previously received remote update to the current time. [4, 29]

$$P(t_1) = P(t_0) + V(t_0)\Delta t + (1/2)A(t_0)\Delta t^2 \tag{2.1}$$

As shown in the equation, the technique assumes that the velocity and the acceleration remain constant until a next state update is received. This makes the technique quite vulnerable to non-predictive behavior, like sudden movement direction changes or warp-movements like teleportation from one position to another. Non-predictive behavior typically increases the amount of inconsistency between the local and remote view, and it must be repaired when the next state update is received.

An example about the inconsistency with the prediction can be seen in the Figure 2.5. The gray path indicates the true motion path and black lines present the predicted path of the car. Points P_0 and P'_0 present a first point where a remote state update is received and the first inconsistency correction must be executed. In this case, a warp correction is used to restore the car into consistent state by directly warping the car into correct position. Points P_1 and P'_1 present a second point where the next remote update is again received and the position is again restored to maintain state consistency. It is easy to deduct that the frequency of the remote state updates and the freedom of movement behavior is highly related to the amount of inconsistency introduced by the technique.

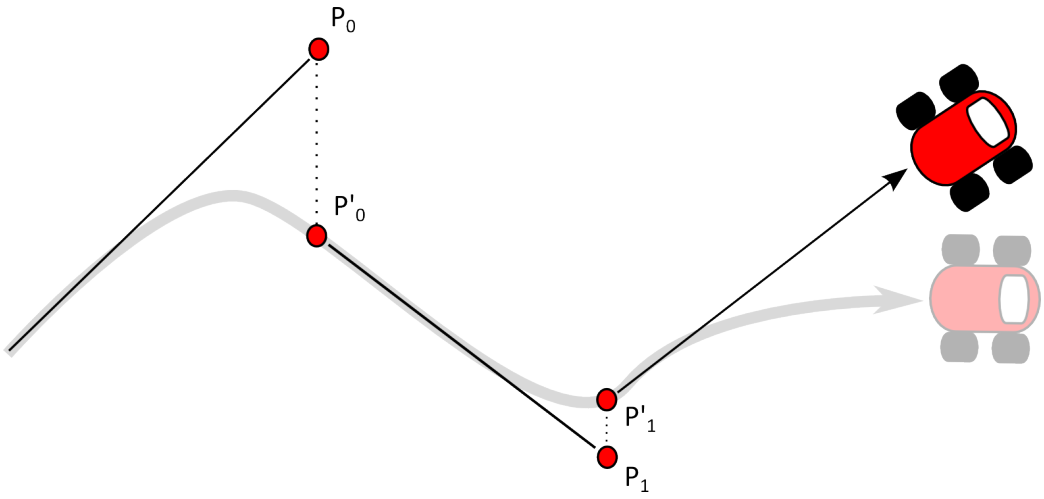


Figure 2.5: Dead reckoning prediction errors in entity motion path.

Prediction techniques are also used to decrease the amount of data required to be sent over the network. This can be done when all nodes (including the server) use a same prediction technique. Nodes may use a difference threshold, which is a tolerance limit for a state difference. If the server detects a entity's movement that exceeds the defined difference threshold, it will broadcast a game state update about the movement. This can greatly reduce the amount of transferred network data, especially when game entities have a easily predictable motion paths. [29]

OPT techniques are generally considered good for optimizing responsiveness by enabling simulation to continue before remote response is received. However, their impact on consistency with unpredictable entity movements may demand application to use costly convergency methods.

2.3.3 Pessimistic delay techniques

Pessimistic delay techniques (PDTs) compensate network latency by introducing delays on the game state updates. Delays are used to wait until all nodes have been synchronized, which removes the need of state roll-backs.

One approach to PDT is to delay local event processing with a technique called *local lag*. It is based on the idea to introduce delay to local event execution to improve the change to execute events at the same time with a remote node. Delaying events on the local node compensates the time required for the event to be transferred to remote nodes over the network. [19]

Let us consider a local lag example where we have two nodes like shown in the Figure 2.6. Local node issues an event E at time the t_1. Event is locally delayed by amount of D_1, which corresponds to the network latency D_2 that appears in the connection from local to remote node. By using an execution delay on the local node, the event is executed at the time t_3 in both nodes.

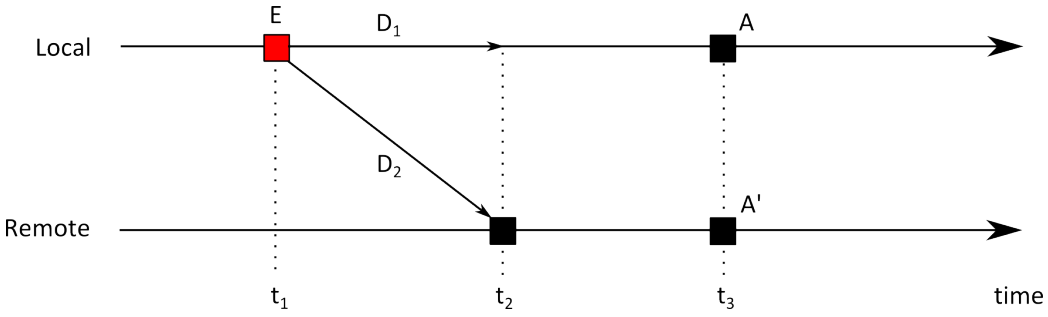


Figure 2.6: Local lag event execution example.

There has been plenty of studies regarding on how to select a suitable local delay value for a local lag. Mauve et al.[19] suggested that the delay value should be selected between the maximum average network delay of the nodes and the application's maximum acceptable response time. However, network delays may change during the application execution and therefore an adaptive solutions have been proposed. A study from Chen [17] introduced an approach where the delay value is calculated from the maximum acceptable response time and periodically measured network latency average. Khan et al. [18] also used network load as a factor, but also introduced a way to use entity based calculations for the delay value for particular object classes. These adaptive techniques introduce a bit more complexity to application structure but keeps the delay value in a balance with the network latency.

2.3.4 Optimistic delay techniques

Optimistic delay techniques (ODT) allow nodes to maintain a responsive simulation by the cost of possible divergence between the game states. Techniques may give immediate feedback to user by applying local events immediately to the local game state. However, when a remote state update is received and inconsistency is detected, the game state must be rolled back in time into a correct state.

One approach to ODT is to use *predictive time management* [13]. It is based on the idea that deterministic events can be pre-empted before they actually happen. For example, if two bounding objects are closing to each other, the game logic could pre-empt that they are going to collide after next 50ms. Pre-empted collision event could be sent to other nodes in advance to compensate network transfer delays. Eventhough technique can work well with deterministic events, it cannot pre-empt non-deterministic events like sudden player movements that are quite usual in MOGs.

Another approach to ODT is to use *remote lag* technique, which buffers game state updates before they are applied. Technique makes the client to run behind the server's time by introducing a small delay before remote state updates are applied. Delay allows client to receive remote state updates so the client always knows the past and the future of the global game state. Local simulation runs by interpolating between the states in the state buffer.

According to Bernier [5], technique was implemented in the well-known Half-life (1998) FPS. By default, the game server sends state updates to clients on every 50 ms, but allows the value to be changed by the players. Clients will use these updates as *position history entries*, which contain a timestamp, the origin, angles and any other interpolative data for the object. Entries are used to determine final positions of the game entities by smoothly interpolating between entries with the target time.

Technique introduces inconsistency as clients do not immediately detect events from other clients. For example, let us consider a shooting action that comes from the client A. Action is directed to precisely hit the character owned by the client B. However, client A is currently lagged by some amount of time and therefore action is delayed before it reaches server. When the action reaches the server the consequences must be determined whether the shooting actually hit the target. In the client A perspective, shooting should hit the character owned by the client B. However, in the client B perspective is seemed that she/he has dodged the possible hit. This kind of inconsistency can be solved by allowing either client or server to determine whether the character was actually hit.

According to Savery et al. [28], remote lag can eliminate almost all posi-

tion corrections on game entities at the cost of adding the complexity to make game critical decisions on diverged player views. It can be also considered to make it easier to provide smooth animations to players.

2.3.5 Convergency techniques

State divergency is commonly found in applications that use optimistic CM techniques. When a node detects divergency it typically handles it with one of the three common techniques: tolerate, warp or smooth correct [28].

Tolerate

Tolerating means that the inconsistency will not be repaired. This is due that inconsistency on some game entities may not have any affect on the gameplay or they can tolerate amounts of inconsistency before they do impact the game. Decision whether to tolerate can be done for example, by defining a tolerance threshold, which must be exceeded before any correction is processed. [28]

Warp

Warping means that the inconsistency is repaired by immediately moving the target entity into a correct position. Moving is typically done immediately when the remote update is received. Technique ensures that on each remote update, the local state of the entity matches the remote state which took place at the time when the remote message was sent. [28]

Smooth correction

Smooth correction is a technique, which modifies the position of the entity to converge into correct position over a time. Correction may not even be noticeable by the user because the correction is processed smoothly and also typically without any jerky motion. [28]

A research from Savery el al. [27], describes that each entity can be considered to have individual consistency requirements. Authors suggest that non-interactable entities like clouds in the sky, shattering glasses and some animations can be usually considered to tolerate any amount of inconsistency. This is due that they do not typically have any kind of impact on the actual gameplay and can therefore be considered to be inconsistent-safe.

A study from Savery and Graham [25], experimented that players are highly sensitive to sudden entity position changes. Authors especially noticed that players get frustrated when they are controlling an entity and performing actions against another entity that suddenly warps into another position.

This might be due that player decisions were based on the game state that was followed by an unpredictable state with jerky motion, which leads into decreased game UX.

Chapter 3

Implementation

This chapter explains how we implement CM techniques into a game framework which is used to construct a game for the evaluation phase. We start by explaining what kind of game we are going to use and how the base structure of the implementation is built. Then we proceed to explain how the framework handles the time and the object model especially with the shared state objects. Finally we proceed and close the chapter by describing how we implement an CM technique from each CM technique category.

3.1 The game

Our target is to build and evaluate a game that is highly sensitive to the effects of the consistency-responsiveness trade-off. In other words, we need to build a game which requires fast player reactions to game state changes. Previous studies have shown that FPS and racing games can be considered to require fast response to player input [10]. However, these genres do not fit into our needs because we want to keep our game implementation as simple as possible and also want players to be interactive between each other.

Our requirements can be satisfied by constructing a clone of an old classic game called *Pong*. Pong is a simulated table-tennis game where two players control paddles which are used to hit a ball back and forth. Players are able to earn score when an opponent misses a ball and the first player which reaches the maximum points wins the game. Pong contains a small and simple scene where players make decisions based on the movements of a ball and the opponent. This makes Pong a suitable choice to study the effects of the CM techniques for evaluating the consistency-responsiveness trade-off. The visual look of our version of the game can be seen in the Figure 3.1.

Implementation contains five shared state objects, which are the paddles,

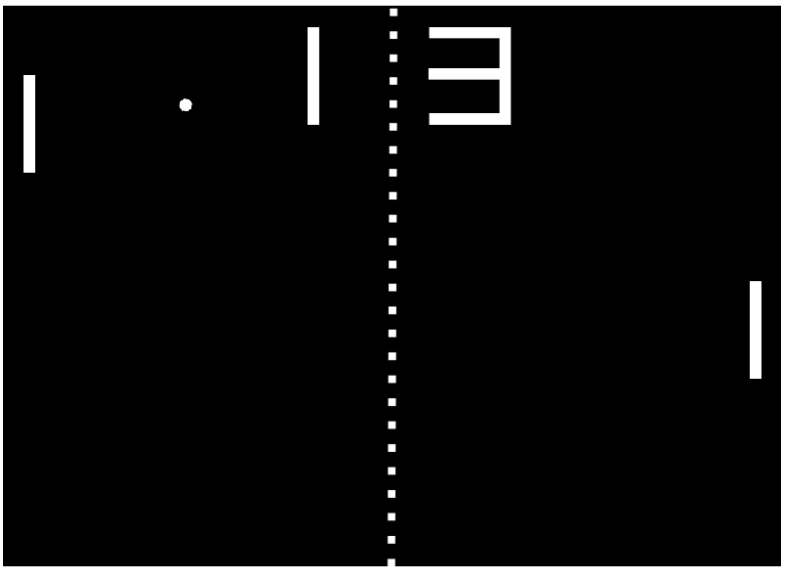


Figure 3.1: In-game screenshot from the game.

the ball and the point indicators. Point indicator synchronization is simplified by using non-stateful objects that are updated via using point event distribution. The paddles and the ball are considered as shared state entities that also require interpolation and extrapolation method implementations for positions and orientations. Top and bottom walls are considered as non-moving and non-stateful objects as well as the goal sections at the right and the left side of the scene.

The game allows players to control their avatar paddles up and down with the keyboard controls. Ball movement will be re-oriented when it hits a paddle or a wall. Right player will receive a point if the ball hits a goal section at the left and vice versa for the left player. A game round lasts until either player receives 10 points or the time limit of one minute is reached.

Our target is to compare four different CM strategies so we create a game that has four rounds. Each round contains a different CM strategy and the order of strategies is randomized for each player pair. We also gather UX between the rounds by using simple query scenes for the user feedback. Query contents are handled and described in the next chapter.

3.2 Framework overview

We extend the target application framework with the networking components described in this chapter. The original application framework is built as a generic game framework that can be used to construct different types of games. Framework is written in C++ and has an additional library dependencies on SDL, SDL-image and OpenGL.

We want our game to playable by the web browsers, so we decide to cross compile the client side code into a JavaScript application. LLVM-to-JavaScript compiler called Emscripten[1] will be used to generate an optimized high performance JavaScript code and the server application will be compiled as a native application.

Application dependency structure is split into modules as shown in the Figure 3.2. Game implementation uses framework components to construct and run a game instance which contains all required game entities and simulation methods. Client and server implementations use the game library to construct a new client or server executable of the game.

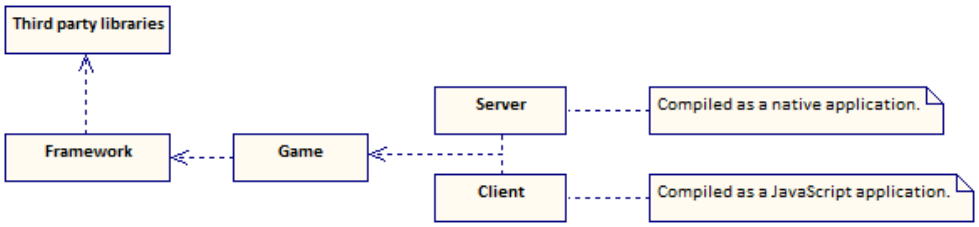


Figure 3.2: Framework dependency diagram.

Networking support for the server application is implemented by using the WebSocket++[2] library, while the networking support for the client is built with the WebSocket API provided by the Emscripten. The WebSocket protocol is used as the communication channel between the clients and the server. WebSocket is a standardized[3] and quite broadly supported by the currently used browsers. It also provides a way for users to participate into our experimentation remotely without users having to install any new software or plug-ins on their computers.

[1] https://github.com/kripken/emscripten
[2] http://www.zaphoyd.com/websocketpp
[3] https://tools.ietf.org/html/rfc6455

3.3 Time synchronization and RTT

The original framework contained a simple clock implementation where the time was measured directly from the system time. A direct system time measurement is typically easy to use when the application runs a local simulation that does not require state synchronization over the network. However, it is not suitable for the distributed applications that must run a timed related simulation simultaneously as each node may be running on a different system time. Difference could lead into problems with the the object state synchronization where the packages typically contain a time related data.

A clock synchronization support requires that the old clock implementation is extended to manage time offsetting. The new clock implementation allows the offset to be given as a negative or positive value that will be added into the results on each virtual time query. Framework uses this corrected time for all game simulation events but uses the system time for the local engine specific events like periodic tasks and local statistics.

Time offset calculation time points are shown in the Figure 3.3. Node_1 starts the procedure by creating a time offset network message. The message is time stamped by t_0 that presents a time when the message is sent to Node_2. Node_2 receives the message and adds a time stamp t_1 to indicate the receive time. Node_2 may then perform some operations like message parsing and handling between times t_1 and t_2. Then the message is sent back to the node_1 stamping it with the time stamp t_2 that this time presents the point when the message is sent. Node_1 will add the time stamp t_3 at the point when the message is received back. All collected time stamps are queried from the clock as an virtual times that may contain already added offset values.

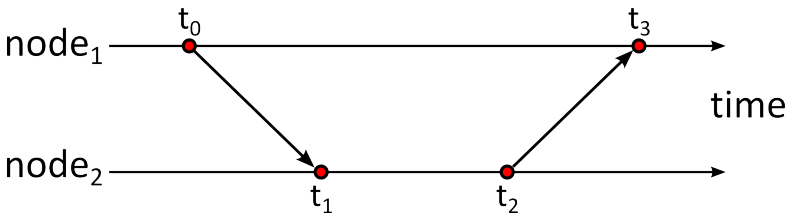


Figure 3.3: Time offset calculation procedure.

All four time points are used to calculate the offset value for the clock with the formula shown in the Equation 3.1. Equation assumes that the same amount of network latency affects both transfer directions and therefore allows it to calculate the time difference by using the specified time points.

The calculated offset is only applied when the node acts as a client. This kind of behavior makes the server to run the simulation in a correct virtual time which further simplifies the state synchronization procedures.

$$\text{time-offset} = \frac{(t_1 - t_0) + (t_2 - t_3)}{2} \tag{3.1}$$

Procedure also provides a way to calculate the round-trip time (RTT) with the formula shown in the Equation 3.2. RTT is calculated by solving the amount of time passed from the time when the message was sent to the time when the message was received back subtracted by the time that was used by the remote node to process operations.

$$\text{RTT} = (t_3 - t_0) - (t_2 - t_1) \tag{3.2}$$

A clock synchronization process is launched as a periodic task with a five second interval upon the application startup. At the client side, procedure measures and applies the time offset to keep the virtual time consistent with the server and to also calculate and store the connection RTT. Server does not need the offset value so it uses the procedure to only calculate RTTs for all connected clients.

3.4 Object model

Framework splits object model into the shared and non-shared objects. Non-shared objects are only used locally and do not require synchronization with the other network nodes, while shared objects contain a list of states that present values of the object in certain points in time and must be synchronized with the other network nodes. This kind of approach was also used in the Timelines implementation presented by Savery and Graham [26].

Framework calls shared objects as *stateful objects* that are used with different state types. Each different state type must implement a state interface that requires an implementation of an interpolation, extrapolation, difference calculation and a serialization operation. Stateful object itself is a template class that is able to manage the list of states by providing access to set, assign, get and clear operations.

The *set* function may be used to define a state indirectly into a specified point in time as can be seen in the Algorithm 1. Algorithm takes a state S as an input parameter and generates a state update event that is queued into the event queue of the application. This makes S to be applied by the stateful object during the next application update tick.

Algorithm 1 Set a state into a stateful object

```
1: procedure SET(S)
2:     event ← create-state-update-event(S)
3:     insert-into-event-queue(event)
4: end procedure
```

A variation of the Algorithm 1 is shown in the Algorithm 2, which takes a time offset input parameter t and the state S which are then combined into a new state that is sent into the normal version of the *set* algorithm. The time offset value t is given as an offset of the current time, so the function adds the current update milliseconds and a possible state update delay into the offset to construct the final timestamp for the event. Idea behind adding a state update delay is that we are able to locally determine all new state updates to be offset by a certain amount of time.

Algorithm 2 Set a state into a stateful object with time offset

```
1: procedure SET(t, S)
2:     t ← t + application-update-time
3:     t ← t + application-state-update-delay
4:     S.time ← t
5:     set(S)
6: end procedure
```

The *assign* function can be used to define a state directly into a specified point in time. Pseudocode of the implementation can be seen in the Algorithm 3. Procedure takes a state S as an input parameter and iterates over the list of states to find a place where to insert the given state. State count may be reduced by removing the most oldest state definition after the specified state has been inserted into the list. Decision whether the latest state is removed is decided by comparing the current count of states with the state count limit specified by the used CM technique. However, it should be noticed that Algorithm 3 requires state S to contain a absolute timestamp. To make the interface more programmer-friendly, we also provide an Algorithm 4 for specifying a state with a time offset like in the Algorithm 2.

The *get* function is used to get an state for a time offset from the stateful object. Pseudo-code for the implementation is shown in the Algorithm 5. Algorithm first appends the previous application update time into the offset to form an absolute time. Time is then used to find a correct state by iterating over the state list. Interpolation is used when the target time is located between two existing states and is allowed by the CM technique.

Algorithm 3 Assign a state into a stateful object

```
1: procedure ASSIGN(S)
2:     state ← states.first
3:     found ← false
4:     while found = false do
5:         if state = states.end then
6:             append-to-back-of-the-list(states, S)
7:             found ← true
8:         else if state.time = S.time then
9:             state.value = S.value
10:            found ← true
11:        else if state.time > S.time then
12:            insert-before(states, state, S)
13:            found ← true
14:        end if
15:        state ← state.next
16:    end while
17:    if states.count > cm-strategy-cache-size then
18:        remove-from-front-of-the-list(states)
19:    end if
20: end procedure
```

Algorithm 4 Assign a state into a stateful object with time offset

```
1: procedure ASSIGN(t, S)
2:     t ← t + application-update-time
3:     t ← t + application-state-update-delay
4:     S.time ← t
5:     assign(S)
6: end procedure
```

Extrapolation is used when the target time is after all known states and is allowed by the CM technique. Algorithm will return a dummy value if the state list is empty. Implementations for extrapolation and interpolation must be constructed by state objects. Our implementation contains states for position and orientation, where position uses 3d-vector that uses linear interpolation/extrapolation and orientation uses spherical linear interpolation/extrapolation for quaternions.

3.5 Consistency maintenance techniques

Framework requires all CM technique implementations to define state cache size and a way to resolve whether it is allowed the use the state extrapolation or interpolation. Techniques are also further divided into a server and client side implementations as they require different kind of operational structures.

Framework contains implementations for four different CM techniques where each technique is selected from a different CM technique category. Nature of the category defines how the technique adjusts the consistency-responsiveness trade-off with the time and the available data but is also used to determine which nodes run a game logic instance.

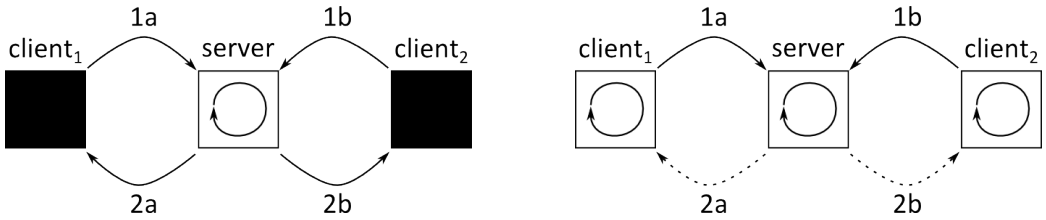
Framework requires clients to send shared state object updates to server for validation and processing of the global game state update as shown in the Figure 3.4. Pessimistic clients (a) do not process game logic, like car controlling, at the client side as all control events are directly send (1a and 1b) to server that validates and applies them into the global game state. Server distributes (2a and 2b) shared object state changes to all clients after it has finished the game state update. In contrast to centralized logic, optimistic clients (b) process logic in each node. Control events routing is is similar to pessimistic technique, but distribution from the server to clients is only done when a correction or a user generated object state change update is detected.

3.5.1 Pessimistic presence technique

Framework implements PPT as a centralized serialization technique. Technique requires that all client events are sent to the server which applies them into the global game state. State changes are distributed to clients when the server has received an confirmation message from the clients. A confirmation message is a request that tells the server that the sender client has received previously sent updates and is now ready for the next application update step. Server does not tick the game logic before all clients have confirmed

Algorithm 5 Get a state from the stateful object

```
1: procedure GET(t)
2:     t ← t + application-last-update-time
3:     if states.empty = false then
4:         if states.first = states.last then
5:             return states.first
6:         else
7:             if states.first.time > t then
8:                 return states.first
9:             else if states.last.time < t then
10:                if extrapolation-allowed = true then
11:                    return extrapolate(t, states.last.previous, states.last)
12:                else
13:                    return states.last
14:                end if
15:            else
16:                for each state in states do
17:                    if state.time = t then
18:                        return state
19:                    else if state.time > t then
20:                        if interpolation-allowed = true then
21:                            return interpolate(t, state.previous, state)
22:                        else
23:                            return state.previous
24:                        end if
25:                    end if
26:                end for
27:            end if
28:        end if
29:    end if
30:    return dummy(t)
31: end procedure
```



(a) Shared state update routes and logic location in a pessimistic CM technique.

(b) Shared state update routes and logic locations in a optimistic CM technique.

Figure 3.4: Shared state update routes and logic locations in CM techniques.

that they are waiting for the server, which makes the game simulation speed to highly coupled with the latency of the clients.

Client side implementation uses a simple event handling logic for the serialization as shown in the Algorithm 6. Client will response to server sent update event immediately by sending an update event as a confirmation response. Client also immediately applies all object state updates when they arrive.

Algorithm 6 PPT-Client: handle an event received from the server.

```
1: procedure PPT-CLIENT-HANDLE-REMOTE-EVENT(event)
2:     if event.type = "update" then
3:         send-update-event-to-server()
4:     else if event.type = "object-state-update" then
5:         apply-object-state-update(event.data)
6:     end if
7: end procedure
```

Server side implementation for the remote event handling is shown in the Algorithm 7. Technique requires an update event from both clients before server ticks the game logic. Logical step will typically update game object states which makes the system to distribute state updates to clients. Server also collects and applies all input events that are sent from the client to server. Input events are applied to the avatar controller that is assigned for the client that did sent the event.

3.5.2 Pessimistic delay technique

Framework implements the PDT technique as a server side version of the local lag technique. Both pessimistic CM techniques in the framework are

Algorithm 7 PPT-Server: handle an event received from a client.

```
1: procedure PPT-SERVER-HANDLE-REMOTE-EVENT(event)
2:     if event.type = "update" then
3:         if update-requester-set-size() = clients-count then
4:             tick-game-logic()
5:             clear-update-requester-set()
6:             send-update-event-to-clients()
7:         else
8:             add-into-update-requester-set(event.connection)
9:         end if
10:    else if event.type = "input-message" then
11:        apply-input-to-controller(event.data)
12:    end if
13: end procedure
```

required to maintain game logic at the server, which is also used as a point to apply local delay to all events in PDT. As an pessimistic technique, the implementation only requires simple remote event handling algorithms for the object state updates and input events as shown in Algorithms 8 and 9.

Algorithm 8 PDT-Client: handle an event received from the server.

```
1: procedure PDT-CLIENT-HANDLE-REMOTE-EVENT(event)
2:     if event.type = "object-state-update" then
3:         apply-object-state-update(event.data)
4:     end if
5: end procedure
```

Algorithm 9 PDT-Server: handle an event received from a client.

```
1: procedure PDT-SERVER-HANDLE-REMOTE-EVENT(event)
2:     if event.type = "input-message" then
3:         apply-input-to-controller-with-delay(event.data)
4:     end if
5: end procedure
```

Applying a state update delay allows events to be distributed to all clients and to provide a feedback at the same time when the delay is selected as a maximum latency of current clients. Technique adapts changes in network conditions by checking the maximum latency with a five second interval and making updates on the local lag with correction rounded up to next 50s.

Selected latency value is added to all next state updates and it should compensate the transfer time of object state event messages from the server to clients.

3.5.3 Optimistic presence technique

OPT allows stateful objects to use extrapolation to predict current and future state of an object. New stateful object states are sent to the server but are also immediately applied to the local game state which makes the game to produce immediate feedback for the user.

Extrapolation uses two previously known states to predict the current state with an extrapolation scalar that is calculated with an extrapolation function implemented in the state structure. OPT implementation allows the extrapolation processing shown in the Algorithm 5. Extrapolation can be only processed when the state history contains at least two states.

The algorithm for handling OPT events contains an important aspect that is not required with pessimistic CM techniques. A client must be able to apply remote updates after local state update are applied. This kind of behavior ensures that the corrections sent by the server are applied over the local updates that may contain divergent data. Framework implements ordering of the events by using an event priority management, where low priority events are processed after high priority events. An remote event handling algorithm for OPT clients is shown in the Algorithm 10.

Algorithm 10 OPT-Client: handle an event received from the server.

```
1: procedure OPT-CLIENT-HANDLE-REMOTE-EVENT(event)
2:     if event.type = "object-state-update" then
3:         event.priority = low
4:         wait-for-processing(event)
5:     end if
6: end procedure
```

3.5.4 Optimistic delay technique

ODT requires the application to delay remote events by running two different virtual times. One time is used to simulate the locally controlled avatar while the another is used for all remotely controlled objects. The virtual time for the remote objects is lagged behind the the time used for the local avatar making the remote objects to be positioned by their past transformations.

Our technique implementation slightly follows the algorithm introduced by the Bernier [5]. Each node must be able to detect which shared state objects are considered as owned by the node. These owned objects are simulated by using the current virtual time while the non-owned objects subtract a remote lag offset to simulate past states. A procedure to read a value from a shared state object is shown in the Algorithm 11.

Algorithm 11 Read a state from a shared state object when using ODT.

```
1: procedure GET-VALUE(statefulObject)
2:     if statefulObject.isOwned then
3:         return statefulObject.get(0)
4:     else
5:         return statefulObject.get(remoteLag)
6:     end if
7: end procedure
```

The remote lag amount is calculated at the server when the technique is activated. An formula for this calculation is shown in the Equation 3.3 where d_x values present the one-way network delay and a_x values indicate the artificial network delay assigned for the client x. Calculation result is further rounded up for the next 50s and then sent to both clients which apply the value to be used with all non-owned shared state objects.

$$\text{remote-lag} = -(d_1 + d_2 + a_1 + a_2) \tag{3.3}$$

It should be noted that the amount of remote lag is always negative, which makes the state query for the non-owned stateful objects to always point into the past. This kind of behavior allows the target node to first receive information about the real state of the non-owned object and to interpolate a precise state by using two known states.

Technique requires the game to process game critical decisions in multiple places. Remote paddles are simulated by their past positions, which makes their simulation imprecise for collision detection. Decision whether a paddle collides with a ball is determined and distributed from the client which owns the paddle. This kind of functionality could lead into situations where the

local client would see a ball to go through the remote paddle. We solve this by allowing the local client to preempt the collision and correct it when the state update is received from the owner of the collision participant. Goal sections are processed similarly where the paddle owner decides whether the ball collided with the goal section behind the paddle.

Chapter 4

Evaluation

This chapter describes the used methods and results for the consistency-responsiveness trade-off evaluation for the created game. Chapter starts with an explanation about how the testing environment setup is built. Then the chapter proceeds to describe how the measurements for the responsiveness, consistency and playability are implemented and what kind of results they provided. Chapter is then closed by gathering the evaluation results as a consistency-responsiveness trade-off ratio analysis.

4.1 Testing environment setup

The server environment is built on the top of a virtual server machine hosted on a high speed campus network. The machine is running Ubuntu 14.04 on a single CPU environment with 1024 MB of RAM. The game client application is set available by running an Apache HTTP server (ver. 2.2.22) instance, while the game server is compiled and executed as a native application.

The server application has permission to write data into the file system. This data contains all log entries from the internal log system and statistic data collections from the game session specific measurements. Log entries are used to debug the application in the case of any errors, while the statistic data collections are used to study how different CM techniques operated during the game sessions.

Data collecting can be divided into automatic and user specific statistics. Automatic data is collected by the background processes during the game session. This data contains consistency and responsiveness measurements and is not noticeable by the user. In contrast, the user specified data contains the answers that are given by the user for each presented query. After each session is closed, a new line containing the session specific data will

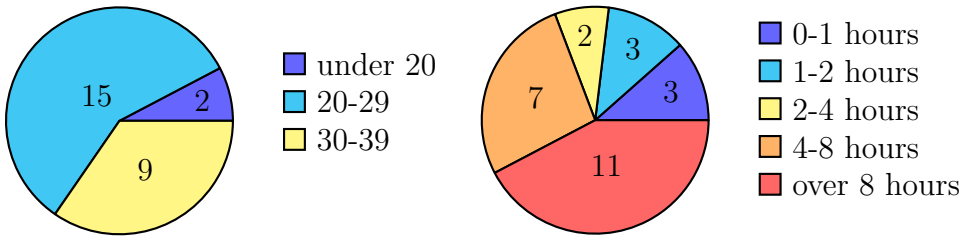
be appended into a CSV file. This data will be appended even in the case where the player does leave the game before the test is over. This kind of functionality allows us to collect results also from the partial game sessions.

Our server application contains support for management of connection specific artificial delays. We use this feature to provide additional delay to each game session to simulate real-world latency problems. A collection of four different delays are assigned to connections in a round robin pattern. Our selection of delays are 0ms, 25ms, 50ms and 125ms, where the 0ms contains only the delay from the real connection latency. When a delay is assigned to a connection, it will be assigned into incoming and outgoing channel. For example, this kind of functionality increases the connection RTT by 250ms when the delay of 125ms is assigned to a connection.

4.2 User study participants

User study was participated by 26 persons who were recruited by using personal invites, Facebook, Twitter and forums at gamedev.net and TIGSource. As an addition to queries performed after each round, we also queried the age group and the playing hours per week at the start of the test.

Each query contained five possible choices. First query asked the participant to select an age group, where the user was able to select from the following categories; under 20, 20-29, 30-39, 40-49 and 50 or over. Second query asked user to select a playing hours per week group that contained following options; 0-1, 1-2, 2-4, 4-8 and over 8 hours per week. The participant distribution between the categories can be seen in the diagrams shown in the Figure 4.1. Most of the participants were 20-29 years old and were playing digital games at least over 8 hours per week.



(a) Age group distribution. (b) Playing hours per week distribution.

Figure 4.1: Results from the user study initial queries.

4.3 Responsiveness measurement setup

Game responsiveness is measured by using the response time for the local input events. The total delay is the time from the user to give an input to get and associated feedback as shown in the Figure 4.2. The game section contains the processing delay (t_2 - t_3) of the target application, while the system sections contain hardware device delays (t_0 - t_1 and t_4 - t_5) together with operating system and software delays (t_1 - t_2 and t_3 - t_4). Our client side application is processed remotely as a web browser application, which makes our measurement to be unable to measure system specific times as they would require additional equipments installed on each local system. This limitation makes our responsiveness calculation to focus only to the response time measurement from t_2 to t_3 with an formula *response-time*$= t_3 - t_2$.

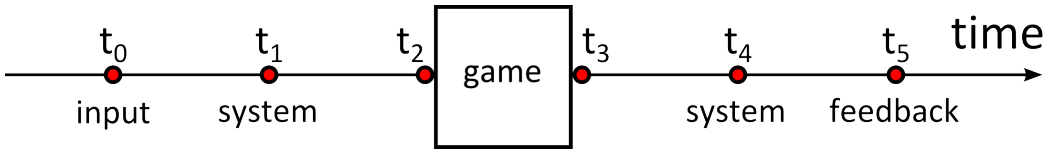


Figure 4.2: Total delay for an user input.

The procedure to capture t_2 and t_3 requires us to use time stamps with the issued events. All locally polled input events are tagged with a time stamp that describes when the event was detected by the game. Time stamps are further used to tag state updates which are issued as a reaction for the input events. Response time is calculated when the rendering system renders a positional state which contains a tagged time stamp.

Each client first stores all detected response times into a local container which is cleared when the values are sent to the server by an five second interval periodic task. Value message contains the average response time from all collected response times from the last five seconds. When the server receives an response time message, it stores the value into a statistics file.

We also provide user queries about how users were able to detect possible responsiveness problems with the controlled paddle. Queries were shown after each round and the user was able to provide an answer by using a scale from one to five. Number one means that the user felt that the controlled paddle did respond very poorly and number five means that user felt that the paddle was responding very well.

4.4 Responsiveness measurements

The results from the responsiveness measurement were collected with the technique described in the previous section. Each technique was measured separately such that we were able to collect technique specific responsiveness statistics from each game session. The measured responsiveness results can be seen in the Figure 4.3.

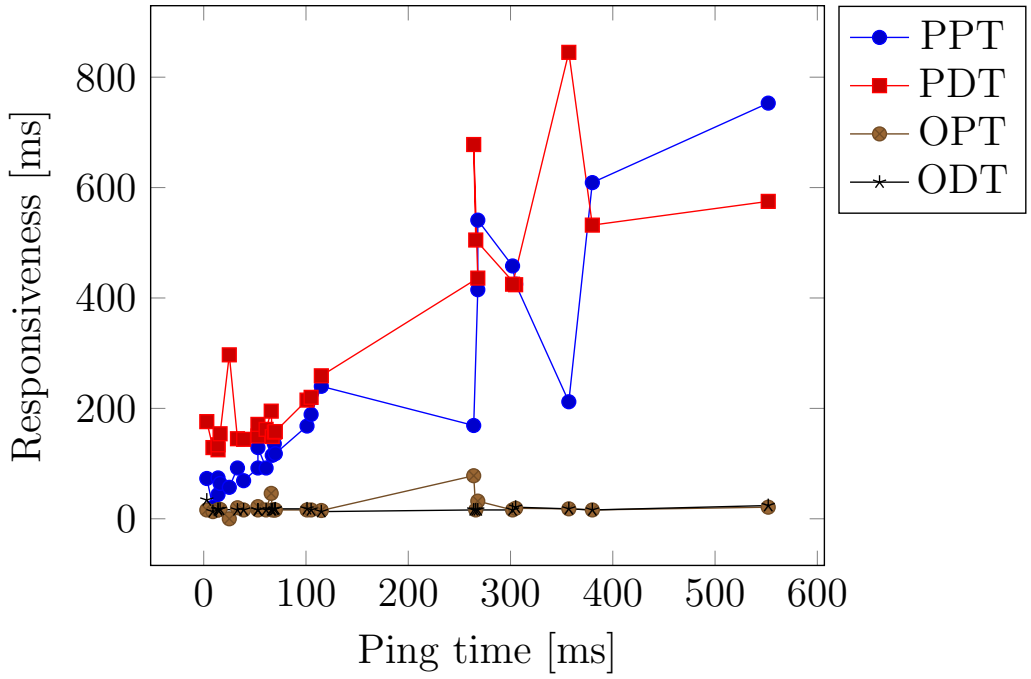


Figure 4.3: Technique specific responsiveness results.

The measurement results show that the responsiveness in pessimistic techniques is highly related to the ping time of the connection. In contrast, optimistic techniques seems to be not affected by the connection ping as their responsiveness stays close to 17ms, which relates to one frame lag in a game that runs with a 60fps frame rate. The best responsiveness was provided by the ODT, while the PDT had the worst responsiveness for the paddle.

The Table 4.1 shows a summary of the responsiveness measurement results. PDT contains the worst responsiveness average and median with 291ms and 168ms, while ODT has the best responsiveness with 17ms average and 16ms median. Table 4.1 also contains columns for ping related responsiveness average and median. These ping related values are calculated by dividing the measured responsiveness value with the connection latency.

Technique	Average	Median	Ping-rel average	Ping-rel median
PPT	218	136	2.93	1.77
PDT	291	186	6.14	2.47
OPT	21	16	0.59	0.24
ODT	17	16	0.79	0.24

Table 4.1: Responsiveness measurement result averages and medians. Lower is better.

According to Section 2.2.2, fast paced games may only tolerate delays up to 45ms - 60ms. This can be provided by both optimistic techniques as they keep the response time below 45ms regardless to the connection latency. In contrast, our PDT should not be considered as suitable for fast paced games as it does keep the response time constantly above 100ms. Our PPT is able to produce enough responsiveness when the connection latency is 50ms or lower as it seems to be quite highly coupled with the connection delay.

In Section 2.2.2, we also studied that some researches have found that slow paced RTS games may even tolerate delays up to 1000ms. This is not a problem for optimistic techniques as they are capable to provide fast responsiveness at any latency level. However, pessimistic techniques may even fail to satisfy this limit if the connection latency increases enough. For example, with a connection latency of 1000ms, PPT would delay responsiveness above the one second limit as it requires events to be serialized. PDT would also behave in a similar way as it delays all events at least with an amount of connection latency.

The results from the responsiveness query also show that the participants were more satisfied with the responsiveness provided by the optimistic techniques. Results were collected after each round where the participants gave values from one to five, where one indicates worst value and five would be the best possible value. The average and the median of the results can be seen from the Table 4.2.

As seen from the Table 4.2, optimistic techniques were considered superior with their responsiveness capability. Our ODT implementation was considered as the best of the four techniques, while PDT was considered as the worst technique to maintain the responsiveness. OPT was considered as the second best with an almost similar points than ODT, while PPT was graded with an almost similar grade than PDT. Query results also seem to reflect the measured values shown in the Figure 4.3.

Table 4.2 also contains columns for ping related average and median for the query answers. These values describe the relationship of the given query

Technique	Average	Median	Ping-rel average	Ping-rel median
PPT	2.4	3	0.09	0.03
PDT	2.2	2	0.07	0.03
OPT	4.2	4	0.14	0.05
ODT	4.3	4	0.15	0.06

Table 4.2: Responsiveness query result averages and medians as a normal and ping related values. Higher is better.

answer regarding to the amount of the connection latency. Results show that optimistic techniques were capable to produce almost two times better responsiveness for each latency millisecond than pessimistic techniques. By combining the measurement results from the Table 4.1 and Table 4.2, it is possible to deduce that optimistic techniques produce much more responsive gaming experience than pessimistic techniques. This supports the findings from the previous researches studied in the Section 2.3, where we made the major categorization of the techniques based on their optimistic and pessimistic behavior and studied their relationship with the application's responsiveness.

4.5 Consistency measurement setup

Game consistency was measured by calculating the state divergence between the nodes. Only stateful objects are suitable for the measurement as the static objects are stateless and therefore they can be considered as immune to inconsistency. Our game implementation contains three suitable objects for this purpose; the ball and both avatars.

We adapt a state divergence metric used in the paper by Savery et al. [27]. Difference amount is expressed as a percentage of the screen size, which can be calculated by the formula shown in the Equation 4.1. Formula calculates the difference of the positions in x and y axises and divides them with the screen width and height. The result value is divided by two and converted into percentages by multiplying the value by 100%.

$$\text{ball-state-divergence} = \left(\frac{|x_1 - x_2|}{\text{screen-width}} + \frac{|y_1 - y_2|}{\text{screen-height}}\right) : 2 \times 100\% \quad (4.1)$$

Equation 4.1 is suitable for calculating a difference for the ball movement but not viable for the avatar difference as the avatar only moves in y-axis.

Therefore we introduce a new formula shown in Equation 4.2 to calculate the difference of the remote avatar. Formula uses differences in the y-axis and divides them with the screen height multiplied with the percentages to get the difference in the required format.

$$\text{avatar-state-divergence} = \frac{|y_1 - y_2|}{\text{screen-height}} \times 100\% \tag{4.2}$$

A message route example for calculating the state divergence for an entity is shown the the Figure 4.4. $Client_1$ starts the procedure at the point t_0 by creating a message that contains the currently used position state for the non-owned entity. Message is then passed to the server that immediately routes the message to $Client_2$ at the point t_1. When the $Client_2$ receives the message at the point t_2, it compares the remote value to the local value and measures the state divergence value as shown in the Equation 4.1. Calculation result is then send to the server, which receives the value at point t_3 and stores the value into a statistics file.

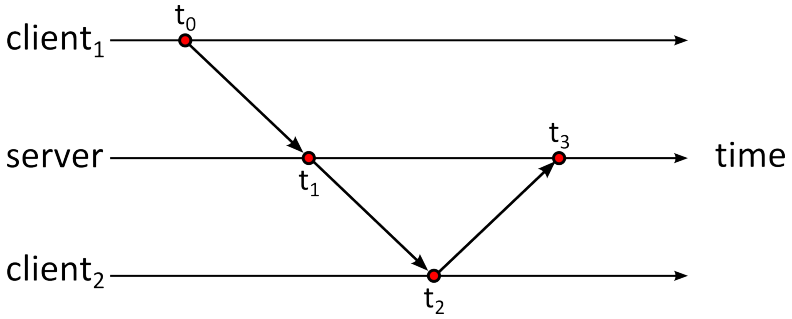


Figure 4.4: Consistency measurement message route.

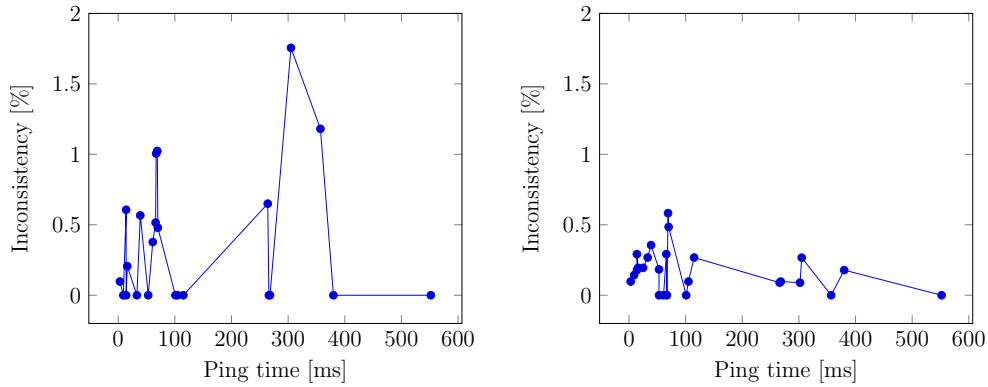
The measurement procedure is implemented as a periodic tasks that is launched when a game round begins. Periodic task is processed with an five second interval on both clients.

As an addition to automatically measure the consistency, we also provided user queries about how the user actually recognizes inconsistency corrections. These queries were shown after each game round and the user is able to answer by using a scale from one to five, where one means that user was constantly recognizing odd movements and five means that user was not able to detect any corrections.

4.6 Consistency measurements

The results for the consistency measurement were collected by using the technique described within the previous section. As with the responsiveness measurement, consistency was measured by collecting values separately such that we were able to collect technique specific consistency statistics from each game session. In addition, the procedure split the measurement into two sections; the ball and the remote avatar.

Inconsistency results for the ball with the pessimistic techniques can be seen from the Figure 4.5. Results show that PPT is able to keep the inconsistency below 2%, where PDT may even keep the inconsistency below 0.6%. PPT inconsistency variation seems to increase when the connection ping time increases, while PDT keeps the variation quite steady.



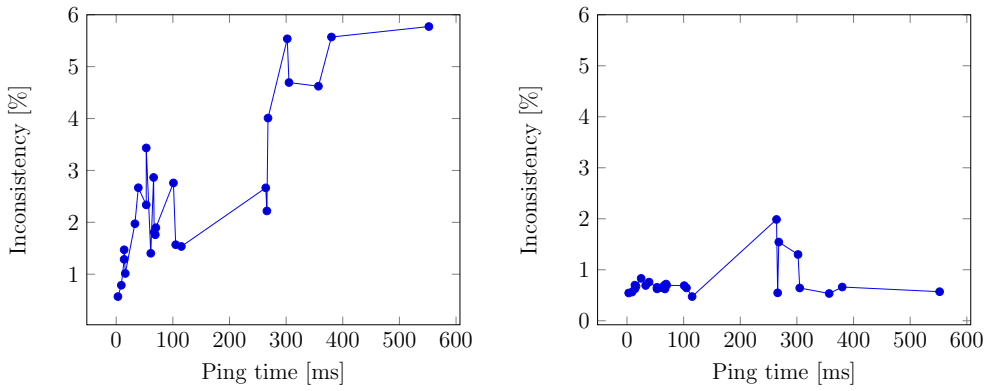
(a) PPT (b) PDT

Figure 4.5: Ball inconsistency results for PPT and PDT.

The Figure 4.6 shows the results for the remote avatar consistency measurement with pessimistic techniques. Results show that the inconsistency for the remote avatar grows when the connection latency increases with PPT. Inconsistency may even reach almost up to 6%, while PDT is able keep it below 2%. These results also support the measurements from the Figure 4.5, where PDT was producing better consistency than PPT.

Three high inconsistency nodes at 275ms - 300ms shown in the PDT diagram in Figure 4.6 may be a result of the connection jitter, browser environment or bad clock synchronization. However, all other nodes within the same chart shows that PDT is able to keep the inconsistency below 1%.

The results for the PPT show that the inconsistency increases rapidly when the connection ping time increases. This may be due to same problems



(a) PPT (b) PDT

Figure 4.6: Remote avatar inconsistency results for PPT and PDT.

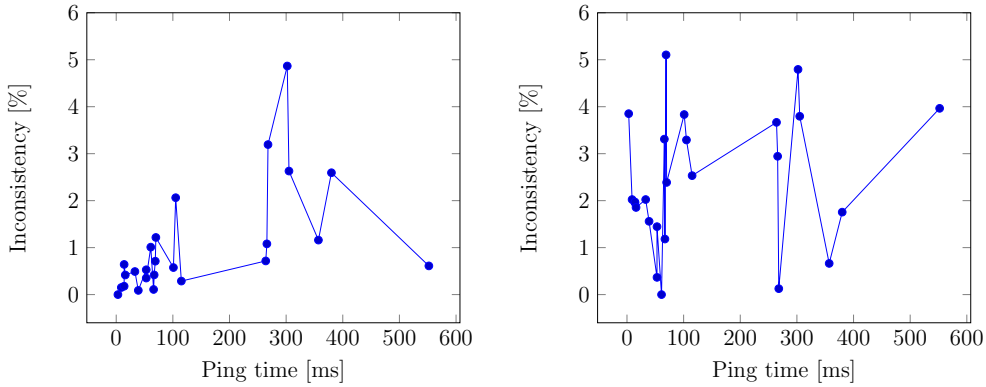
than with PDT but can be also a result of the PDT implementation as the implemented serialization technique allows client to perform the rendering of the updated state immediately when the client receives the acknowledgement from the server.

The ball inconsistency measurement results from the optimistic techniques are shown in the Figure 4.7. OPT seems to be able to keep the inconsistency around one percentage when the connection ping time is between 0ms to 100ms. When the ping time increase above 100ms, the inconsistency variation increases rapidly. In contrast, ODT inconsistency contains a big variation that seems not to be so closely related with the connection latency. However, ODT technique does allow clients to simulate game in the most divergent environment, which may produce this kind of behavior.

Results for the remote avatar consistency with optimistic techniques can be seen from the Figure 4.8. Here the results show that the OPT is nearly able to keep the inconsistency around 1%, which makes it to perform almost as well as PDT. OPT chart also contains two peaks at around 25ms and 75ms which may be caused by connection jitter, clock asynchrony or other possible problems.

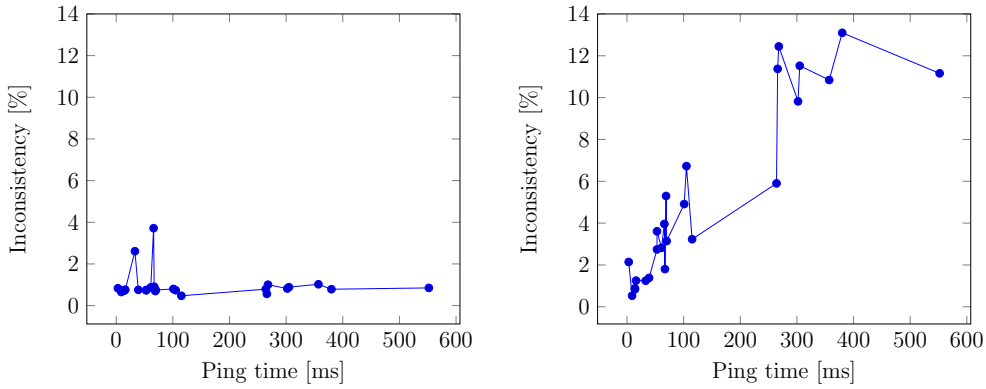
The Figure 4.8 shows that the ODT increases the inconsistency quite rapidly when the connection ping time increases. Inconsistency may even reach up to 10% when the connection latency increases above 300ms. This kind of behavior can be a result of the remote lag, which may allow the view between the clients to divergence almost linearly with the connection latency.

All results from the ball consistency measurement procedure are summarized in Table 4.3. Pessimistic techniques seem to provide better consistency than optimistic techniques. Results show that PDT provides the best consis-



(a) OPT (b) ODT

Figure 4.7: Ball inconsistency results for OPT and ODT.



(a) OPT (b) ODT

Figure 4.8: Remote avatar inconsistency results for OPT and ODT.

tency with an average inconsistency of 0.34% and median 0.18%, while ODT has the largest state divergence with average of 2.42% and median 2.02%. However, differences get smaller when we calculate the average and median for the ping related inconsistency. This calculation tells us the relationship between the measured inconsistency and the connection latency. While this may give us a more precise results, it does not change the classification order of the techniques.

Summarized results from the remote avatar measurement procedure are shown in Table 4.4. PDT can be still considered as the best technique as states only divergent on average 0.76% and median 0.66%, while ODT is still considered to make the largest state divergence with an average of 5.31% and

Technique	Average	Median	Ping-rel average	Ping-rel median
PPT	0.61%	0.21%	0.01%	0.00%
PDT	0.34%	0.18%	0.01%	0.00%
OPT	1.04%	0.61%	0.01%	0.01%
ODT	2.42%	2.02%	0.10%	0.03%

Table 4.3: Ball inconsistency summary from the measurement procedure. Lower is better.

median 3.61%. Interestingly, PPT provided a much worse state consistency for the remote avatar than OPT. However, this might again be a result from the serialization technique implementation, as the technique applies state changes immediately when they arrive to the client.

Technique	Average	Median	Ping-rel average	Ping-rel median
PPT	2.65%	2.22%	0.04%	0.03%
PDT	0.76%	0.66%	0.02%	0.01%
OPT	0.97%	0.79%	0.03%	0.01%
ODT	5.31%	3.61%	0.07%	0.05%

Table 4.4: Remote avatar inconsistency summary from the measurements. Lower is better.

The results from the consistency query can be seen in Table 4.5. Participants were asked to describe how much odd movements they noticed during the game. Results show that the ODT was producing least odd movements regarding to players, which does not correspond to the results we received from the consistency measurement procedure. However, it should be noted that odd movement in a optimistic technique is typically a side effect of a state correction, which is not the same thing than the view inconsistency. This makes the query not so strictly relevant to the actual inconsistency of the game as the users were actually asked about the amount of state correction they noticed during the game.

This assumption also describes why OPT was graded as the worst technique by the participants, while it was capable to provide good consistency measurement results. Technique used a state prediction that is capable to extrapolate entity movement, which makes the local state vulnerable to many corrections. These corrections were applied directly into the state objects, which made entities to warp into corrected positions. This relation between the bad grade and warping corrections is similar to findings from the previous

Technique	Average	Median	Ping-rel average	Ping-rel median
PPT	3.0	3	0.10	0.04
PDT	3.7	4	0.10	0.05
OPT	2.7	2	0.13	0.03
ODT	3.9	4	0.14	0.05

Table 4.5: Consistency query result averages and medians as a normal and ping related values. Higher is better.

researches studied in Section 2.3.5, where studies show that warp corrections are considered as a quite frustrating way to correct state inconsistency.

In contrast, ODT only uses interpolation between already known states to make optimistic assumptions for the state update and therefore may also require less and smaller corrections to states. These corrections are done with the smooth correction convergence method, which is considered to provide good and unnoticeable corrections in regards to findings from the previous researches studied in Section 2.3.5. The query points received by the ODT also provide similar results than what we found from the previous studies in Section 2.3.4, where some researches found that remote lag technique may eliminate almost all visible corrections to game states.

A reason why PPT received such low points may be due to simulation jerkiness that grows linearly when the connection latency increases. The simulation delta time is tightly coupled with the connection ping as the server requires confirmation from both clients after each simulation step. This behavior makes game entities to warp with a magnitude that is related to the maximum latency between the clients and the server. Query results show that participants ranked PPT almost as bad as OPT. This may be due that PPT state updates behave almost similarly than noticeable and frustrating warp corrections that were used with OPT.

As a summary, PDT can be considered as the best technique based on the overall measurement and queried results. ODT can be considered as the worst technique to prevent state divergence, even though it provided the best grade from the user queries. Our classification order for the techniques based on their consistency capabilities is PDT, PPT, OPT and ODT, where PDT can be considered as the best technique to prevent inconsistency between the nodes.

4.7 Playability results

The results from the consistency and responsiveness measurements do not itself provide a clear view how the players experienced the game. Therefore, we also provided a query about the playability, where participants were asked give an grade to their gaming experience after each round. Users were able to give an grade by selecting an value from one to five, where one would be the worst and five would be the best possible grade.

The results from the query show that both optimistic techniques provided higher playability scores than either of the pessimistic techniques. ODT seems to provide the best playability with the average grade 3.7 and median 4.0. PPT was considered as the worst technique with an average grade of 2.2 and median 2.0. Both pessimistic techniques were graded below the average playability level, while optimistic techniques were considered a bit above it. The collected results from the query can be seen from the Table 4.6.

Technique	Average	Median	Ping-rel average	Ping-rel median
PPT	2.2	2.0	0.08	0.03
PDT	2.3	2.0	0.08	0.04
OPT	3.2	3.0	0.12	0.03
ODT	3.7	4.0	0.14	0.04

Table 4.6: Results from the playability query. Higher is better.

Table 4.6 also contains columns for ping relative grade averages and medians. Ping related grade is calculated by dividing the given grade with the connection latency. The results from the ping relative value columns show similar classification ordering than non-ping relative columns.

We are also able to calculate relative playability against measured responsiveness and inconsistency. The Table 4.7 shows results from these relative calculations. Relative responsiveness describes a user given grade per responsiveness millisecond, while relative ball and remote avatar contain a grade per inconsistency percentage. Each value is calculated by dividing the playability grade with the corresponding measurement result.

The results from the Table 4.7 show that ODT is able to get highest playability grade per responsiveness milliseconds. In contrast, PDT has the worst responsiveness classification per playability but provides the best playability per ball inconsistency percentage. OPT seems to get quite good classification in all three categories, while PPT produces worst results in almost all categories.

Technique	responsiveness	ball	remote avatar
PPT	0.02	2.74	1.26
PDT	0.01	11.42	3.51
OPT	0.18	7.28	3.99
ODT	0.22	3.49	1.52

Table 4.7: Average relative results for the playability. Higher is better.

4.8 Consistency-responsiveness trade-off

To study the trade-off effect between the consistency and the responsiveness, we need to study both values by evaluating their relationship. To do this, we use the measurement results from the previous sections and combine them to form a trade-off value results. The first value combination is shown in the Table 4.8, which contains a summary of average values.

	Resp.	Ball incon.	Remote avatar incon.	B/R	RA/R
PPT	218	0.61%	2.65%	0.003	0.012
PDT	291	0.34%	0.76%	0.001	0.003
OPT	21	1.04%	0.97%	0.050	0.046
ODT	17	2.42%	5.31%	0.142	0.312

Table 4.8: A result comparison for the average measurement values.

The Table 4.8 contains three columns for the average results received from the previous sections. Column *Resp.* for the responsiveness, *Ball incon.* for the ball inconsistency and *Remote avatar incon.* for the remote avatar inconsistency. Additional column *B/R* stands for a ratio between the ball inconsistency and the responsiveness, while column *RA/R* contains ratios between the remote avatar inconsistency and the responsiveness.

The results in the Table 4.8 shows that both optimistic techniques have a much higher ratio to produce inconsistency per a single responsiveness millisecond. In contrast, pessimistic techniques have a very small ratio to produce inconsistency per a millisecond spent on the responsiveness. The PDT offers the best ratios, where it averagely produces inconsistency with an ratio 0.001 and 0.003. The worst ratios are clearly produced by the ODT, which uses ratios 0.142 and 0.312 by an average.

The second value combination is shown in the Table 4.9, which contains a summary of median values. Median values produce similar results than

average values, while the difference between the optimistic and pessimistic techniques is a bit smaller. PDT can be still considered to produce best ratios with ratios 0.001 and 0.004, while ODT clearly produces the worst ratios with 0.126 and 0.226.

	Resp.	Ball incon.	Remote avatar incon.	B/R	RA/R
PPT	136	0.21%	2.22%	0.002	0.016
PDT	186	0.18%	0.66%	0.001	0.004
OPT	16	0.61%	0.79%	0.038	0.049
ODT	16	2.02%	3.61%	0.126	0.226

Table 4.9: A result comparison for the median measurement values.

We are also able to evaluate ping relative values, which gives us ratios per connection latency milliseconds. The Table 4.10 contains the ping relative values from the previous sections, but does also provide similar ratio columns as provided within the Table 4.8 and Table 4.9.

	Resp.	Ball incon.	Remote avatar incon.	B/R	RA/R
PPT	2.93	0.01	0.04	0.003	0.014
PDT	6.14	0.01	0.02	0.002	0.003
OPT	0.59	0.01	0.03	0.017	0.051
ODT	0.79	0.10	0.07	0.127	0.089

Table 4.10: Results for the ping relative average measurement values.

The results in Table 4.10 show that the ball inconsistency ratio difference between the OPT and the pessimistic techniques gets smaller. Also the remote avatar inconsistency ratio in ODT is much smaller than the ratio produced by the normal average and median calculations. There is no difference for the classification order of the techniques, while PDT and ODT still contain the best and the worst ratios, respectfully. Also the median value calculation shown in the Table 4.11 does classify techniques in a same manner. Results are similar than shown in the median calculation Table 4.9.

Results show that optimistic techniques trade consistency into increased responsiveness and vice versa for pessimistic techniques. Pessimistic techniques provide much less inconsistency per used responsiveness milliseconds than their optimistic versions. Optimistic techniques are capable to provide much better responsiveness especially at high latency levels in the cost of introduced state divergence.

	Resp.	Ball incon.	Remote avatar incon.	B/R	RA/R
PPT	1.77	0.00	0.03	0.000	0.017
PDT	2.47	0.00	0.01	0.000	0.004
OPT	0.24	0.01	0.01	0.042	0.042
ODT	0.24	0.03	0.05	0.125	0.208

Table 4.11: Results for the ping relative median measurement values.

Based on the results calculated for previous tables, we are able to deduce the classification order for the techniques. The order of the techniques based on their trade-off ratios is PDT, PPT, OPT and ODT, where PDT can be considered as the best technique to keep the game in a consistent state with the cost of increased response time.

Chapter 5

Discussion

Chapter 4 successfully evaluated a simple real-time game, where four different CM technique categories were used to compensate a connection latency. Evaluation results were collected and analyzed by forming different kinds of relational results between the measured and queried values. Each evaluation section classified techniques based on their ability to maintain the evaluated value. A summary of the classifications can be seen in Table 5.1.

	Responsiveness	Consistency	Playability	Trade-off ratio
PPT	3	2	4	2
PDT	4	1	3	1
OPT	2	3	2	3
ODT	1	4	1	4

Table 5.1: Measurement classification order for each CM technique from the evaluation sections. Lower classification is better.

User study participants were asked to rank each technique based on the responsiveness, consistency and the playability after each round of the game. The queries asked users to give an grade for each category, where the number one was considered as the worst and the number five as the best possible grade. A summary of the query results and their confidence intervals (CI) can be can be seen from the Table 5.2. All CI values are calculated by using the 95% confidence level.

Pessimistic techniques were reasonably ranked with a low classification and responsiveness grade as their response time is highly related to the connection latency as we studied in the Section 4.4. PDT produces the worst responsiveness, since it delays game events with the maximum latency rounded up to next 50s. This rounding operation adds additional and perhaps un-

	Resp.	Resp. CI	Cons.	Cons. CI	Play.	Play. CI
PPT	2.4	2.0 ... 2.8	3.0	2.4 ... 3.6	2.2	1.7 ... 2.7
PDT	2.2	1.8 ... 2.6	3.7	3.2 ... 4.2	2.3	1.8 ... 2.8
OPT	4.2	3.9 ... 4.5	2.7	2.2 ... 3.2	3.2	2.8 ... 3.6
ODT	4.3	4.0 ... 4.6	3.9	3.5 ... 4.3	3.7	3.2 ... 4.2

Table 5.2: Average points and their confidence intervals (CI) from the user queries. Higher value is better. CI values use the 95% confidence level.

necessary margin to the response time, which made PDT as not capable to produce response times that would be shorter than 100ms. Based on the previous researches studied in the Section 2.2.2, a 100ms responsiveness is considered to exceed the responsiveness requirements for fast paced games that may tolerate delays up to 45ms - 60ms. Evaluation shows that this requirement could be satisfied by our PPT implementation if the connection latency is 50ms or lower. PPT is more strictly coupled with the maximum connection latency and does not use additional delay that would add unnecessary margin to application's response time. Both pessimistic techniques did also rapidly increase the response time when the connection latency increased. These measurement findings are also supported by the user query results shown in the Table 5.2, where responsiveness (Resp.) grades given for the pessimistic techniques are much worse than the grades given for the optimistic techniques. The difference between the techniques can be considered as quite significant as the responsiveness CIs (Resp. CI) measured for the PPT and PDT do not overlap with the CIs calculated for OPT and ODT.

Optimistic techniques provided instantaneous feedback as they could apply state changes locally before they were applied at remote nodes. Both techniques were capable to maintain application's responsiveness at around 20ms, which is nearly instantaneous responsiveness feedback in a game that uses one frame lag and runs at 60fps. Evaluation shows that the difference between optimistic and pessimistic techniques becomes more wider when the latency level increases. Optimistic techniques are able to adapt connection delays and still maintain immediate responsiveness, while pessimistic techniques have direct impact on their response delays. These results also support the findings from some previous researches studied in the Section 2.3, where studies found that optimistic techniques should be capable to provide fast responsiveness by using immediate state updates. Furthermore, fast responsiveness seems to be highly related to the game's playability regarding to our measurements in the Section 4.7. Results show that optimistic techniques were capable to produce 9 - 22 times better playability for each responsive-

ness millisecond with 0.18 (OPT) and 0.22 (ODT) against 0.02 (PPT) and 0.01 (PDT). This clear difference is also supported by the user query results shown in the Table 5.2, where the playability CIs (Play. CI) for OPT and ODT contain a clear difference against pessimistic techniques. Based on these results, it is quite clear that optimistic techniques are capable to provide better playability than pessimistic techniques.

Based on the measurements in the Section 4.4, presence techniques are capable to provide better responsiveness than delay techniques. The average ping relative responsiveness for the PPT was 2.93 and 6.14 for the PDT. This means that PDT delays the response time about two times longer than PPT when the connection latency is increased by one millisecond. However, it should be noted that our PDT implementation has a 100ms minimum response delay, while our PPT does not have this kind of restriction. The difference between optimistic techniques provided smaller difference, while the average ping relative responsiveness for OPT was 0.59 and 0.79 for ODT. This difference may be due to inaccuracy of the measurement or the browser environment problems, while both techniques were using similar kind of local state update strategy that should provide quite identical results.

Results from the Section 4.6 show that pessimistic delay technique is able to produce less inconsistencies than pessimistic presence technique. PDT was capable to keep average inconsistency level of the ball at 0.34%, while PPT almost doubled this with 0.61%. PDT also produced less remote avatar inconsistency with an average inconsistency level of 0.76% in contrast with 2.65% provided by the PPT. PDT's ability to define events to be applied in the future seems to provide more consistent state changes between nodes. In contrast, PPT applied all incoming events at the moment they were received from the server, which led into temporary divergence between the nodes. Interestingly, optimistic techniques produced inversed results, while the presence technique produced less state divergence than the delay technique. OPT was capable to keep average ball inconsistency level at 1.04%, while ODT doubled this with 2.42%. OPT was also capable to keep remote avatar inconsistency level at 0.97% in contrast with 5.31% provided by the ODT. The high inconsistency level of ODT was predictable as the technique allows states to diverge by maintaining the playability with reasonable decision points. However, it seems that OPT is capable to provide much less inconsistency than ODT and even almost similar consistency level than PPT. Findings from the Section 2.3.2 are in contrast with our measurement results, while OPT methods should be highly vulnerable to state divergence when they are used to predict player movement. A possible reason why OPT received such good consistency results may be due to inaccuracy of our measurement procedure. Procedure took state comparison snapshots at every

5 seconds, which can produce a behavior that does not detect some of the extrapolated states before they are converged. Perhaps a better way to measure inconsistency could be implemented by comparing states when a remote state correction is applied.

Interestingly, ODT received the highest average points from the consistency query, even though it was considered as the worst technique to prevent state divergence. One possible reason for this can be that ODT allowed the state interpolation, which enables the technique to use smooth correction convergence technique described in the Section 2.3.5. It may be that users were not notifying these corrections as long as the game was running smoothly. Similar conclusions were made in the research by Savery et al. [28], where three different small games were studied with some CM technique combinations. Authors found that players were preferring smooth corrections and local view consistent game critical decisions over the global consistency. Also the research from Palant et al. [20] introduced similar conclusions, while authors suggested to keep the local view and decision points consistent even though it could lead into global state inconsistency. Our ODT implementation used decision points that were based on the game entity ownership, which may have reduced some amount of corrections at the local view and therefore produced a smooth simulation. The results also seem to support findings from the previous researches studied in the Section 2.3.4, where some studies concluded that ODT may be capable to eliminate almost all visible state corrections if game decision points are handled properly.

It is also noticeable that OPT had much worse points than ODT from the consistency query. One possible reason for this can be the state convergence technique, which replaced state container states without any smooth corrections. This kind of state management can make big state corrections that may move game entities in a warping manner. Findings from the previous researches studied in the Section 2.3.5 show that warping behavior may lead into decreased game UX as players may get frustrated when game entities warp around the game scene. If these corrections could be hided or corrected smoothly with a different convergence technique, OPT would most likely get much better results.

The best consistency-responsiveness trade-off ratio is provided by the PDT with average ratios 0.001 and 0.003. The ratio is about 15-142 times better than the ratios provided by the OPT and ODT, while it is also about 3 times better than the ratio provided by the PPT. This means that PDT was highly capable to prevent state divergence by increasing the response time. However, this does not mean that PDT should be considered as the best technique to keep the nodes in a consistent state. It still requires the highest decrease in the application's responsiveness, which seems to have big

impact on the application's playability as was studied in the Section 4.7.

There are also some other problems within our evaluation, as we did not include any kind of measurements for the amount and amplitude of the required corrections. Perhaps this should have been done for better consistency analysis as our techniques were using two kinds of convergence techniques. For possible further research, one solution could be to use a research by Savery et al. [28] as an reference to determine what kinds of components could be used for more precise measurements.

Our evaluation also did not provide any kind of results for the implementation complexity. During the implementation, the PPT was the easiest to implement. It did not require any kinds of corrections as the incoming state was immediately applied to the target shared state object. PDT was also quite easy to implement as we were able to define events to be applied in the future. The biggest difference between the complexity of the technique implementations was that our optimistic techniques must run the game logic in all nodes, where pessimistic techniques only run logic at the server. This behavior made optimistic techniques much harder to implement as we were required to define the correct decision points for the events. All decision points for OPT were assigned to the server, while ODT used node specific decision points that made final decisions based on whether the node owned the shared state entity. If the decision point strategy would be changed, it would most likely lead into different kinds of measurement results.

Our user study was participated only by 26 persons, which was less than what we expected. However, we still received quite good CIs for each user query as shown in the Table 5.2 and therefore we may expect that these results are quite significant. Similar amount of persons was also recruited for a research by Savery et al. [28]. Their study evaluated three different kind of small games, where each game was tested with different CM related behavior. Authors found that good CM strategy does not require players to have identical local states. It is more important that players have reasonable game critical decisions and state outcomes. Our measurements and conclusions also support their findings as our ODT was measured as the best technique, while it also produced the highest amount of state inconsistency. ODT implementation also used smooth corrections, which provided reasonable state outcomes as corrections did not warp entities around the game scene.

Chapter 2 presented a study were CM techniques were divided into four categories, where split was based on their optimistic or pessimistic behavior against time or data. This categorization did provide a quite clear distinction between the techniques that were implemented within the implementation. Categorization helped us to build an generic stateful object container that

was capable to handle all state change events for each category. Our implementation of the stateful container may be a bit too complex for a game which uses only one or two from the provided categories. Perhaps it would be wiser to build an stateful container that is directly optimized for implemented techniques to decrease the management complexity for the state updates.

Our study only evaluated one game type, where players were not performing tightly coupled interaction with each other. It is likely, that players mainly focused only on the owned paddle and the ball. Games that use similar middle point interaction entities could benefit our results, while games with a tightly coupled player interaction may contain different types of requirements. Our study did emphasize how the responsiveness, decision points and a suitable state convergence method can improve the game's playability. This may not be a case with the games that require tightly coupled interaction between the players. Tightly coupled interaction can require high consistency level to produce good playability and may contain problematic decision points, which are in contrast to our game implementation.

Chapter 6

Conclusions

Consistency maintenance is a huge topic in the field of multiplayer gaming, where a malfunctioning synchronization or an invalid timing of game events can make the game totally unplayable. Network connections are almost always affected by some sort of latency and jitter that may cause problems with the transmission of the network messages. Network problems can be fought against with different consistency maintenance techniques, which require a balancing act between the responsiveness and the consistency.

This Thesis proposed that consistency maintenance techniques could be categorized based on their ability to handle time or data either in an optimistic or a pessimistic way. Optimistic techniques are able to provide fast feedback with the cost of increased state divergence, while pessimistic techniques sacrifice responsiveness to gain increased shared state consistency. Techniques using the time manipulation may either pessimistically add an additional delay to local events or by optimistically adding a delay for the remote events. Data manipulation based techniques may either pessimistically require event serialization or optimistically allow node to predict future states for the shared state objects.

The solution how to select an consistency maintenance technique for a game should be always based on the game behavioral requirements. A technique from the PPT category could be used as an solution for a game that does not require real-time simulation, while it is typically simple to implement and may produce a good consistency between the nodes. Technique also allows the game logic to be located at a centralized server, which can validate and handle data streams and may also be used to prevent cheating. As a downside, PPT has a strong impact to the responsiveness and may produce warping movement even at the low latency levels.

A technique from the PDT category could be used for a game that is capable to tolerate delay in the responsiveness. PDT provides a very good

and precise consistency and may also be used with interpolation to perform smooth movement. It can be also used to put all game logic into a centralized server and may also be simple to implement. However, a PDT has the strongest impact to the responsiveness, which makes it not suitable for fast paced games that may require 45ms or lower response times.

Fast paced games could benefit from the optimistic techniques as they are capable to provide instant feedback. A technique from the ODT can be a good solution for a game which contains a clear set of decision points and can tolerate large amounts of divergence between the nodes. However, the implementation may become quite complex if the decision point strategy cannot be easily determined.

A technique from the OPT should be considered for a game that contains shared state objects that are easily predictable. The technique may reduce the required amount of transmission data and may produce smooth extrapolation results when implemented correctly. As a downside, it may introduce big corrections to states when the game has shared state objects that are not easily predictable.

None of the techniques should be considered as a supreme solution, as each of them have downsides. Sometimes games could benefit by combining different techniques to form a hybrid solution. For example a technique from the OPT could be combined with a technique from the PDT to add a small delay to local events to decrease the high inconsistency ratio of the OPT. These combinations may be used to balance the consistency-responsiveness ratio, which indicates how the consistency maintenance will trade-off between the inconsistency percentages and responsiveness milliseconds. This should be separately considered for each game to find the most optimal and playable solution that is capable to provide the most enjoyable gaming experience for the players.

6.1 Future Work

This Thesis evaluated how consistency maintenance techniques could be categorized and how they balance with the consistency-responsiveness trade-off. The evaluated game was somewhat slow paced and did not require users to be directly interacting between each other. A user study should be arranged for a fast paced game where users would directly interact with each other.

Evaluation only studied each consistency maintenance category separately, while many games tend to combine consistency maintenance techniques to form the final solution. A further research should be done about how the presented categories could be combined and what kind of results they

would provide with the same measurements as we used within this Thesis.

Thesis did not also consider decision point problems, convergence techniques or data reduction capabilities in the main technique categorization. Further study should be arranged to see whether these features should be taken into account within the categorization.

By overall, the results from this Thesis could be used with other DIA environments as well. Games are not the only type of networked applications that may benefit from the techniques and results that were received from this Thesis.

Bibliography

[1] Addressing battlefield 4 rubber banding issues. webpage, 2016. http://battlelog.battlefield.com/bf4/news/view/addressing-bf4-rubber-banding-issues/ Accessed: 2016-01-31.

[2] Counter strike: Global offensive. webpage, 2016. http://blog.counter-strike.net/ Accessed: 2016-01-31.

[3] Dota 2 official blog. webpage, 2016. http://blog.dota2.com/ Accessed: 2016-01-31.

[4] Grenville Armitage, Mark Claypool, and Philip Branch. *Networking and Online Games: Understanding and Engineering Multiplayer Internet Games*. John Wiley & Sons, Ltd, 1st edition, 2006. ISBN: 978-0-470-01857-6.

[5] Yahn W Bernier. Latency compensating methods in client/server in-game protocol design and optimization. Game Developers Conference 2001, Valve, 2001.

[6] Sumeer Bhola, Guruduth Banavar, and Mustaque Ahamad. Responsiveness and consistency tradeoffs in interactive groupware. In *Proceedings of the 1998 ACM Conference on Computer Supported Cooperative Work*, CSCW '98, pages 79–88, New York, NY, USA, 1998. ACM.

[7] Nicolas Bouillot and Eric Gressier-Soudan. Consistency models for distributed interactive multimedia applications. *SIGOPS Oper. Syst. Rev.*, 38(4):20–32, 2004.

[8] Jeremy Brun, Farzad Safaei, and Paul Boustead. Fairness and playability in online multiplayer games. In *Consumer Communications and Networking Conference, 2006. CCNC 2006. 3rd IEEE*, volume 2, pages 1199–1203, Piscataway, NJ, USA, 2006. IEEE.

[9] Peng Chen and Magda El Zarki. Perceptual view inconsistency: An objective evaluation framework for online game quality of experience (qoe). In *Proceedings of the 10th Annual Workshop on Network and Systems Support for Games*, NetGames '11, pages 2:1–2:6, Piscataway, NJ, USA, 2011. IEEE.

[10] Mark Claypool and Kajal Claypool. Latency and player actions in online games. *Commun. ACM*, 49(11):40–45, 2006.

[11] Mark Claypool and Kajal Claypool. Latency can kill: Precision and deadline in online games. In *Proceedings of the First Annual ACM SIGMM Conference on Multimedia Systems*, MMSys '10, pages 215–222, New York, NY, USA, 2010. ACM.

[12] Declan Delaney. *Latency Reduction in Distributed Interactive Applications using Hybrid Strategy-Based Models*. PhD thesis, National University of Ireland, May 2005.

[13] Declan Delaney, Tomás Ward, and Seamus McLoone. On consistency and network latency in distributed interactive applications: A survey part i. *Presence*, 15(2):218–234, 2006.

[14] Saul Greenberg and David Marwood. Real time groupware as a distributed system: Concurrency control and its effect on the interface. In *Proceedings of the 1994 ACM Conference on Computer Supported Cooperative Work*, CSCW '94, pages 207–217, New York, NY, USA, 1994. ACM.

[15] Zenja Ivkovic, Ian Stavness, Carl Gutwin, and Steven Sutcliffe. Quantifying and mitigating the negative effects of local latencies on aiming in 3d shooter games. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, CHI '15, pages 135–144, New York, NY, USA, 2015. ACM.

[16] Chris Lewis, Jim Whitehead, and Noah Wardrip-Fruin. What went wrong: A taxonomy of video game bugs. In *Proceedings of the Fifth International Conference on the Foundations of Digital Games*, FDG '10, pages 108–115, New York, NY, USA, 2010. ACM.

[17] Chen Ling. An adaptive consistency maintenance approach for replicated continuous applications. In *"Proceedings of the 2005 11th International Conference on Parallel and Distributed Systems*, volume 1, pages 795–801, Piscataway, NJ, USA, 2005. IEEE.

[18] Abdul Malik Khan, Sophie Chabridon, and Antoine Beugnard. A dynamic approach to consistency management for mobile multiplayer games. In *Proceedings of the 8th International Conference on New Technologies in Distributed Systems*, NOTERE '08, pages 42:1–42:6, New York, NY, USA, 2008. ACM.

[19] Martin Mauve, Jürgen Vogel, Volker Hilt, and Wolfgang Effelsberg. Local-lag and timewarp: providing consistency for replicated continuous applications. *IEEE Transactions on Multimedia*, 6(1):47–57, 2004.

[20] Wladimir Palant, Carsten Griwodz, and Pål Halvorsen. Consistency requirements in multiplayer online games. In *Proceedings of 5th ACM SIGCOMM Workshop on Network and System Support for Games*, NetGames '06, New York, NY, USA, 2006. ACM.

[21] Tseng Po-Han, Wang Nai-Ching, Lin Ruei-Min, and Chen Kuan-Ta. On the battle between lag and online gamers. In *2011 IEEE International Workshop Technical Committee on Communications Quality and Reliability (CQR)*, pages 1–6, Piscataway, NJ, USA, 2011. IEEE.

[22] Kjetil Raaen and Tor-Morten Grønli. Latency thresholds for usability in games: A survey. NIK-2014, Norsk Informatikkonferanse, 2014.

[23] Kjetil Raaen and Andreas Petlund. How much delay is there really in current games? In *Proceedings of the 6th ACM Multimedia Systems Conference*, MMSys '15, pages 89–92, New York, NY, USA, 2015. ACM.

[24] Cheryl Savery. *Consistency maintenance in networked games*. PhD thesis, Queen's University, September 2014.

[25] Cheryl Savery and Nicholas Graham. Reducing the negative effects of inconsistencies in networked games. In *Proceedings of the First ACM SIGCHI Annual Symposium on Computer-human Interaction in Play*, CHI PLAY '14, pages 237–246, New York, NY, USA, 2014. ACM.

[26] Cheryl Savery and T. C. Nicholas Graham. Timelines: simplifying the programming of lag compensation for the next generation of networked games. *Multimedia Systems*, 19(3):271–287, 2012.

[27] Cheryl Savery, T. C. Nicholas Graham, and Carl Gutwin. The human factors of consistency maintenance in multiplayer computer games. In *Proceedings of the 16th ACM International Conference on Supporting Group Work*, GROUP '10, pages 187–196, New York, NY, USA, 2010. ACM.

[28] Cheryl Savery, Nicholas Graham, Carl Gutwin, and Michelle Brown. The effects of consistency maintenance methods on player experience and performance in networked games. In *Proceedings of the 17th ACM Conference on Computer Supported Cooperative Work & Social Computing*, CSCW '14, pages 1344–1355, New York, NY, USA, 2014. ACM.

[29] Jouni Smed and Harri Hakonen. *Algorithms and Networking for Computer Games*. John Wiler & Sons, Ltd, 1st edition, 2006. ISBN: 978-0-047-01812-5.

[30] Dane Stuckel and Carl Gutwin. The effects of local lag on tightly-coupled interaction in distributed groupware. In *Proceedings of the 2008 ACM Conference on Computer Supported Cooperative Work*, CSCW '08, pages 447–456, New York, NY, USA, 2008. ACM.

[31] Xin Zhang. *An Information-Theoretic Framework for Consistency Maintenance in Distributed Interactive Applications*. PhD thesis, National University of Ireland, April 2011.