TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

IVAN DUBROVIN

SHARED WORLD ILLUSION IN NETWORKED VIDEOGAMES

Bachelor thesis

Checker: professor Petri Ihantola

ABSTRACT

COMMITTEE: Thesis of Ivan Dubrovin
Tampere University of Technology
Bachelor of Science Thesis, 17 pages, 0 Appendix pages
December 2016
Degree Programme in Computing and Electrical Engineering, BSc (Tech)
Major: Software Systems
Examiner: Professor Petri Ihantola

Keywords: game, replication, lag, multiplayer, network

This thesis aims to discover pivotal points of maintaining shared world illusion in networked videogames. The paper was inspired by huge amount of games that include multiplayer components, even if it does not fit thematically. With overabundance of multiplayer games on the market, paper aims to shed a light on why exactly only few of them have acceptable, faultless networked virtual environments, while most others struggle with it.

In this thesis, the difference between single- and multiplayer game's structure is glanced on. Then, architectures like p2p and client-server, are observed from videogame point of view, with each having its pros and cons. This is followed by the establishment of challenges of replication, introduced by networking component. Finally, some contemporary solutions are presented, along with example of integration of them in a commercial product.

Paper is concluded by summarizing the aspects of making coherent shared worlds, and noticing that not only there's little to none standards of implementation for them, but also the nature of competitive market drives developers to keep the details of well-made replication mechanisms secret, thus gaining an advantage over other competitors.

TABLE OF CONTENTS

LIST OF FIGURES

ABBREVIATIONS AND SYMBOLS

AoI	Area of Interest
EULA	End-User Licence Agreement
FPS	Firs Person Shooter
ICMP	Internet Control Message Protocol
MMORPG	Massively Multiplayer Online Role Playing Game
Netcode	Implementation of aspects related to networked play
NVE	Networked Virtual Environment
P2P	Peer-to-peer
PvE	Player versus Environment
TCP/IP	Communication protocol set to distinguish machine in network
TSS	Trailing State Sync
UDP	User Datagram Protocol

1. INTRODUCTION

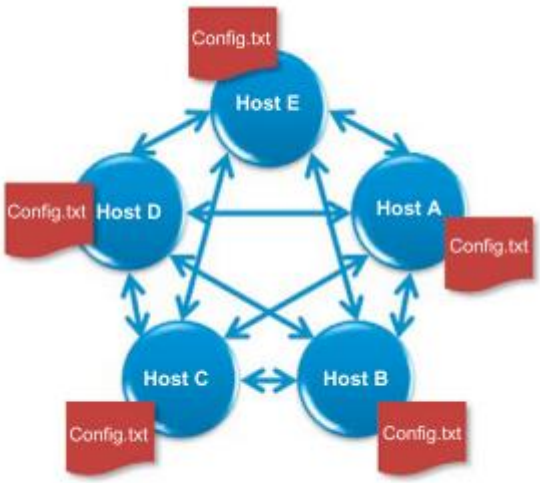

Figure 1. *Peer-to-peer configuration file distributed. (Steed, Oliveira et al. 2009, 2010, p.130)*

Nowadays the game without multiplayer is regarded as something incomplete, and players tend to avoid such games. Moreover, the multiplayer component adds to longevity of the game's life cycle, which is why most of the developers tend to want to include at least some kind of online multiplayer interaction, be it simple high score table, cooperative extras, or full-fledged competitive multiplayer.

In this thesis methods of maintaining consistent and coherent shared world illusion in networked videogames are observed. The aim of this paper is to establish essential challenges with real time networking, world and event replications, as well as provide an insight to current and future solutions to said complications. The act of keeping all of the participating game worlds, or networked virtual environments, NVEs up to date with one another is called replication. It aims to ensure that all of participants have a consistent model of the game state and is the absolute minimum problem which all networked games have to solve, with other, more concrete problems derived from it. Videogames are exquisite over other distributed systems, since the replication is the end goal, rather than a mean unto an end. Solution mainly follows either one of principles: *active* and *passive* replications (Schiper, Charron-Bost et al. 2010, p20).

Networked videogames also have to be able to deal with problems networking component introduces. As seen on *Figure 1*, connecting multiple participant machines, hosts, to each other is not a trivial task. Imagine Host A wants to play with Host B. Firstly, they need to

find each other within the network, for example over internet. After finding both need to update their configuration files, with the relevant information about each other for the session. After that they both start exchanging messages about their actions making both players see each other's actions replicated at their screen. The connection established is represented on *Figure 1* by arrow between hosts A and B.

But what if Host C needs to join both his friends, Host A and B, and play with them? Then Host C needs to inform all players, or participants, about his intentions. After it is done, new configuration file, which includes new, as well as old players' information, needs to be distributed across all hosts. Each player now has a way to connect to other two, and the game continues. Extrapolate the situation for player D, E, and so on, and soon each player is in need of keeping individual connections to all other players as well as accepting incoming connections from them, represented on *Figure* 1. Not only does this gets confusing to figure out and is also heavy on network, this also means that failure on one players side will start chain reaction for all remaining players. It is clear that a better way to network players together is needed.

Networking itself can be imagined as layered model, with application layer being on top, followed by transport, network, link and physical layers. This is also called *TCP/IP stack* (Steed, Oliveira et al. 2009, 2010, p.72). Each of this layers are responsible for a specific action. For the purpose of the paper, only application, transport and some of the aspects of network layers are in focus.

Network layer is associated with internet address or *IP,* the 32-bit number, comprising four bytes most commonly represented as 127.0.0.1. This protocol is unreliable, meaning that a packet sent to network is not guaranteed to arrive. *IP*, along with header *TCP* additions, resolves and identifies machine within the network. (Steed, Oliveira et al. 2009, 2010, p.73) Using example from *Figure 1*, the layer allows hosts to find and identify each other in the web.

After discovering all participants, the need to identify what process sent state data is meant for arises. Here's where the transport layers come in. The layer consists of *TCP* and *UDP* protocols. Transmission Control Protocol, or *TCP* is a reliable connection-oriented service, which implements its reliability without help from network infrastructure on a transport level. User Datagram Protocol, or *UDP* is an unreliable connectionless service, which requires no connection setup, meaning that clients just send messages to servers, and servers do not need to explicitly keep track of active sessions etc. The protocol has no congestion control, ordering or reliability either. This is desired in many NVEs, since the networked data changes so fast, that resending previous outdated data is not only of no use, but also detrimental to consistency of simulation. On practice, *UDP* segment includes source port and destination port, both 16 bit in range, coupled with length of data package, its checksum and the payload itself. The port mechanism allows for multiple independent networked applications to function on same machine without interfering with

one another. (Steed, Oliveira et al. 2009, 2010, p.84). Game developers have to properly chose libraries that utilize these protocols, because specific combinations may not work with specific networking setups as intended, and ideally developers aim to abstract of how the state data is being delivered, putting more focus on what that data is and the order that data is transferred in. Referring back to *Figure 1*, all our hosts should have the information needed to start exchanging messages, that contain information about user state, for example their movement. This, however, does not solve the issue of saleability completely.

The growth of videogame industry coupled with almost mandatory presence of at least some sort of networked component even in purely local, single player experiences, on top of developers' seeming inability to implement said components perfectly is the main reason for this paper existence.

The work is carried out based on literature, as well as presentation, held on a convention.

2. GAME AS SHARED SYSTEM

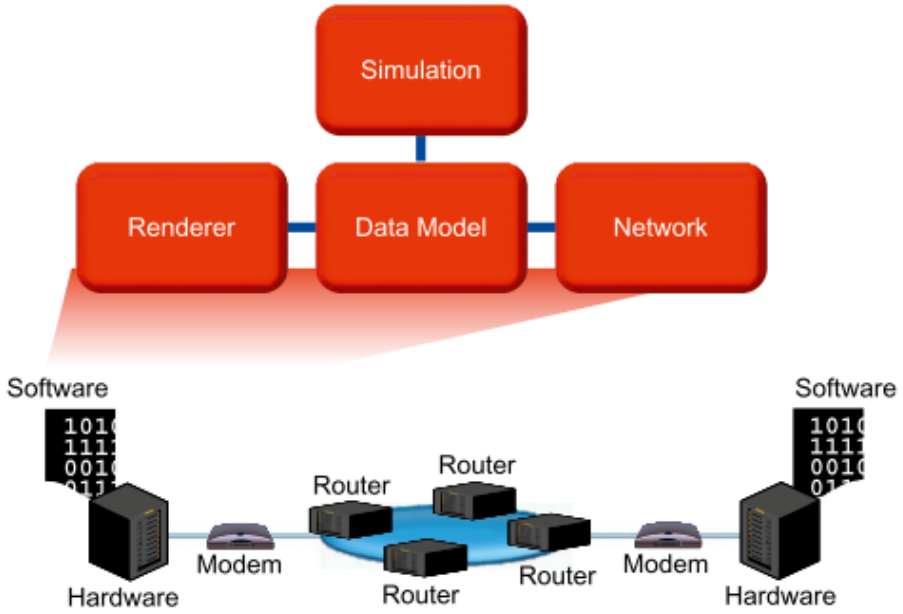

Figure 2. *Data coupled software (Steed, Oliveira et al. 2009, 2010, p.53)*

Shared or distributed computing system is basically a collection of processors interconnected by a communication network in which each processor has its own local memory and other peripherals, and the communication between any two processors of the system takes place by message passing over the communication network (Sinha 1997). Games are prime examples of said systems, since the architecture has multiple to some degree dependant participants, players, exchanging state messages among the network. Messages contain information about objects that changed their states, with states being anything from orientation or position changes, to more complicated events initializations like beginning to shoot. The data needs clear separation between each machine's local and networked parts. Client or host thus usually consists of renderer, data model, networked and simulation components. Renderer is in charge of displaying visual representation of data, provided by data model, which in terms, is responsible for accounting for local data states, partially coming from simulation, as well as remote data's states, received through networking component. For example, when player moves, the simulation makes sure the position is updated in data model. Renderer then displays the movement, while networking component messages other players about the change in player's position. In networked applications each client is thus responsible for simulating correct states and changes locally as well as for accepting interactions from the network, and sending its own changes to it, as is shown on *Figure 2*.

There are many architectures to facilitate such messaging, however in this paper the focus is on peer-to-peer and, more commonly used, client-server communication models. For sake of simplicity, models will be explained using the bird flock example, from Networked Graphics book (Steed, Oliveira et al. 2009, 2010). Basically, there are multiple hosts, each having local flock of birds, as well as network simulated flocks of other hosts.

2.1 Peer-to-peer

Basic peer-to-peer model consists of two or more separate machines, called hosts. With this communication model, each host is responsible for sending updates to all the other participating hosts. In order to be able to send updates, it is necessary for a host to have records on the necessary information concerning each remote host. These records have at least address/port data, but any additional information may be stored, such as network statistics (Steed, Oliveira et al. 2009, 2010). Each machine thus receives this configuration file, forming a structure as shown on *Figure 1*. Now all hosts know about the initial states, the messaging between hosts can begin. In those messages, states of local flocks are transmitted directly to all other hosts, while also receiving and rendering the states of remote flocks. The architecture is thus rather demanding on bandwidth and is susceptible to failure, due to the number of critical points.

For keeping consistency across multiple simulations connected in this way, active replication is implemented. Active replication allows for state messages from all players are sent to all players in the network, with state being simulated deterministically and independently on each client. This is also called *lock-step synchronization* and *state-machine synchronization*. This replication model on practice mostly implemented by just broadcasting player inputs. One of the main drawbacks of such model is that it is fragile, meaning that all players must be initialized with an identical copy of stat and maintain complete representation of it all the time. State updates must be implemented identically on all clients, with even the smallest differences, for example due to floating point number rounding differences, amplified into desynchronization bugs, which render simulations unplayable.

While the *p2p* architecture has several editions with host having more or less authority over others, it is being used in games where the real-time requirement is more loose, or where the number of participating clients is two. It is more commonly used in fighting games, where the action is one on one, or in *PvE* games, where players interact with partially networked simulation, and not directly with or versus each other.

2.2 Client-Server

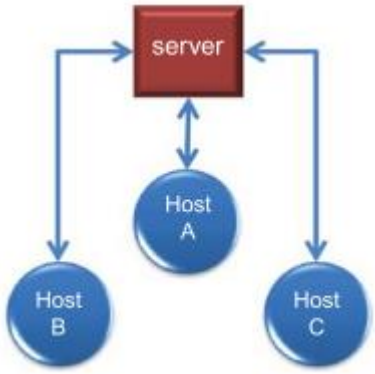

Figure 3. *Server-client (Steed, Oliveira et al. 2009, 2010, p.152)*

Alternatively, more commonly used contemporary solution is client-server architecture. Here all communication between players is done via central server. The architecture of the client resembles closely that of a host in the peer-to-peer net- work, with the Renderer, Datamodel, Network and Simulation components (Steed, Oliveira et al. 2009, 2010). Unlike peer-to-peer, clients do not have to keep track of other participants, other than the server. This is shown on *Figure 3.* This is achieved by utilizing multicast, the mechanism that requires users to join a group using *ICMP* control packets in order to receive and deploy messages among this group. This allows not only for hosts to join and leave with seemingly no detrimental effect on other participants' configurations, but also for server not needing to concern itself about package distributing, since the underlying multicast network would do it instead.

For replication consistency, it generally makes sense to implement a form of passive replication, due to this architecture's central point being the server. Passive replication transfers inputs from clients to a single machine and updates are broadcast to all players. The main advantage of this method is it is robust to desynchronization and allows strong anti-cheating measures to be implemented. The cost is enormous burden being placed upon the server.

This model is currently used for games where more than two players interact with each other, since its saleability is virtually limited only by server's bandwidth. One of the other benefit of such system, is that the it allows clients to have less than perfect, fluctuating quality wise connections without rendering other world participants invalid. This architecture is used in most modern shooter games, as well as in massively multiplayer online games, coupled with saleability adjustments to the server side.

3. CHALLENGES

The complications in implementing sustainable networked videogame can occur in plethora of different places. Due to the complexity of distributed systems, classifying individual challenges and choke points might seem ambiguous at some points. However, some distinction must be made to somehow classify and put them in unarranged order. The main aim of well *NVE* is to keep all individual simulations as consistent in regards to each other as possible, dealing with lag due to following complications.

3.1 Network bottleneck

The most obvious choke point in shared systems is networking component. Ideally each individual client simulation works independently from server's and other clients, but in reality things are not always straight forward.

Latency is a measure of how long it takes for a message to reach its destination (*A Dictionary of Computer Science.* 2016). Thus, if server has highest authority, any start of action on client side will be followed by a delay, during which the message would need to reach server and the reply would be received. This time is called round trip time.

Consider example of two cars starting moving in a driving game (Yasui, Ishibashi et al. 2005), where both players have their simulation clocks synchronized at the beginning. Both players receive signal message to start race simultaneously, and both of them are about to start moving at the same time. Because accelerate is a direct control, each expects his/her own car to accelerate immediately. However, the event that includes the accelerate action will be delayed by the network, and thus each player sees themselves leap ahead, while their opponent is on the starting line. Because of network latency, without any compensation for the delay, each player sees their opponent slightly behind where they actually are on the track. (Steed, Oliveira et al. 2009, 2010, p.356)

Bandwidth is another limiting factor of network. Depending on the game, some actions may require more state data to be transferred through the network, resulting in bandwidth usage peaks. It is a common practice to limit maximum amount of transferred data for such occasions, but not receiving state changes in time violates consistency of simulation.

3.2 Real time requirements

Most games are working in real time, as opposed to turn based ones. According to a study (Pantel, Wolf 2002), latency above 500 milliseconds is not acceptable, and under 50 milliseconds imperceptible. Thus consistency between local and remote simulations is of paramount. However, maintenance of such close consistency may not benefit users, since in the time the message reaches client, other, perhaps interfering, events could occur, making the original state change invalid.

The main attraction for the subject of this thesis is that there are several times: time like present, and time like past. Cone of uncertainty is, in simple terms, time of casual future, as in the predicted future events, that might occur, based on previous observations. For the sake of simplicity, it is safe to conclude that, using the specific prediction mechanisms allows to predict with some certainty which events casually precede others, and therefore help with predictions of such future events. For example, if running players face a wall, they turn right in most cases. Thus, it can be predicted, that any running player that runs into any wall will turn right, until the update with real action is received. If such assumption is made, player will seamlessly see predicted action of other players, instead of their discrete states one after another.

Also, the validity of each of client simulation can suffer at times. Without authorities like server's simulation, which is considered the main and only true simulation with validity of state changes and events being tested upon it, the question of which simulation is true arises. One client's simulation can lag behind on updates, thus seeing the "more than acceptable" outdated states of other client. Thus, the client sees and acts on states as if they are present, making his actions valid from his perspective. However, when the validity of action is checked and invalidated, user experiences undesired event when his actions seems not to have any effect, even though his simulation suggested otherwise. The effect is exaggerated even more when clients interact directly with each other.

3.3 Development

Developing heavily time-dependent NVEs is complicated, mostly because developers have to deal with time in their codes. Little to none object oriented languages provide developers with cohesive data structures and methods to cope with time in local simulation, not to mention remote ones (Savery, Graham 2013, 2012). Because of this, developers often have to use solutions and libraries done by other developers, with all their flaws and quirks, while adding on their own logic on top of existing one, thus hurting reliability of end product.

Engines have complications dealing with networked environments too. The main problem from game engine point of view is how to reliably receive, analyse and display data, coming from network in suitable time and manner. Some engines use device input/output sharing, which relays hosts' actions, like button presses to others, thus evolving each NVE in the same manner. This method however has been rendered faulty, since it is heavy on the bandwidth, is reliant on low latency and introduces constrains like inability to join NVE late without rewinding all previous actions from the start, which is deemed unacceptable in our time. (Steed, Oliveira et al. 2009, 2010) It is obvious that more sophisticated mechanism needs to be implemented instead.

Lastly, security issues need to be addressed. As with any shared system, the possibility of manipulating messages that NVEs use to communicate their states to gain unfair advantage over other participants, or simply break other clients' simulation, exists. The problem is exaggerated by the huge popularity of some networked games, with thousands of people willing to take advantage of any mishap left in the game code, even though most of games EULAs deem such actions illegal and punishable by the law.

4. SOLUTIONS

There are numerous solutions to challenges listed above, next some of key principles will be explained. They are used as basis's in most frameworks for NVE development either under the hood, as a build in high abstraction level of building block, or as library extensions.

4.1 Area of interest management

Some games have to replicate a plethora of different things, on top of just other players and their actions. For example, contemporary shooters have all sorts of actors, like vehicles, weapons, mines, debris and uncontrollable bodies of killed players. These all can interact and be interacted with by various ways, bodies block bullets but move if are hit etc., so networking them is needed. However, the situation, when the amount of such objects is so high, that the network's bandwidth would not suffice is very likely. Thus the need to prioritise which objects need to be networked firstly, and which can be left to be updated after a delay, when the load on network is lower arises. Such prioritization is called *area of interest management* and it varies in type, being static, rule based or geometric (Lysenko, 2014).

Static *AoI* implies partitioning of game world in smaller individual parts. These parts are working independently and usually are either completely separated instances of simulation or smaller regions. To each such partition, maximum number of players can be set, so the worst case scenario can be accounted for. The method loses its appeal if many players tend to locate themselves in specific regions. For example, it is common for *MMORPG*s to have high level areas, where the majority of players reside. Many games adopted *instancing*, which is a method to create new separated similar area and populate it with overflow of players.

Geometric algorithms of managing area of interest assigns priorities to object based on their geometric attributes. Simplest example of such is distance to a player: if networked object is close enough so the player can see it, it will have higher priority than more distant or smaller objects. Distance, however is not always the best option to prioritize networking. Noticeability is also a big factor that should be accounted for, since in

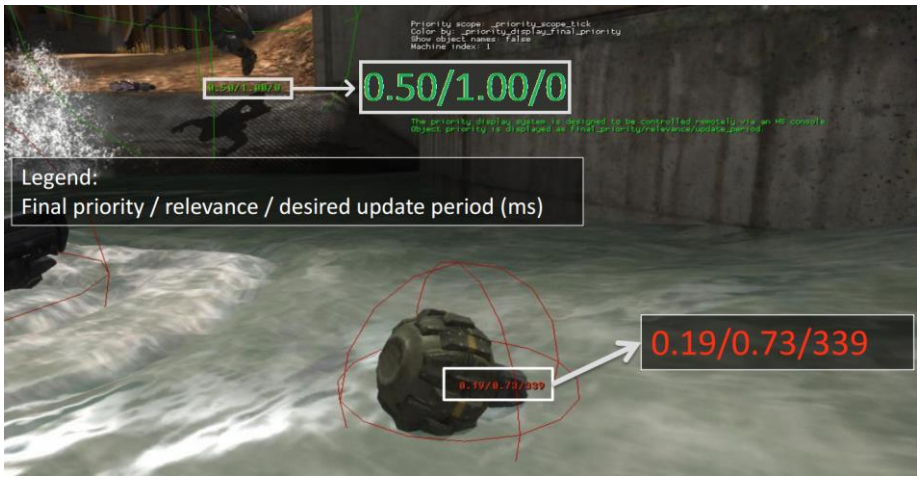

Figure 4. *Prioritization system's debugging view (Aldridge, 2011)*

general, it's acceptable to leave some lesser noticeable things up to predictions, while concentrating on things more noticeable.

Rule based management decides what to replicate based on the game's rules. If a game has invisible units, for example, replication is not needed, since other players cannot see them. Irrelevant information, such as player's inventory contents, or his selected skills etc. needs no replication either, and well defined partitioning of relevant data that needs replication and that does not helps reduce state update size, thus saving bandwidth.

There's no common approach that suits every game, so often game developers mix and match affirm mentioned techniques to create their own. One of such developed systems is related to *Halo Reach*, developed by *Bungie*, multiplayer shooting game involving 16+ players, vehicles and hundreds of replicated objects. In the presentation (Aldridge, 2011) the custom mechanism managing prioritization of networked objects was shown and explained. In this mechanism, the flow control layer, which is a separate prioritization layer of simulation in the game's netcode, decides when to send packet and what size it should be. This packet then is filled with state data until full by the replication itself, prioritizing high-priority data.

Prioritization itself, much like rule based area of interest management, is based upon each individual client's view and simulation state and is calculated for every object for every client. Much like in geometric AoI technique, core metrics are distance and direction, but unlike basic implementation, size and speed also affect priority. So, if an object, for example SUV is launched by explosion into the air, its priority will go up, because fast moving objects need to be replicated more accurately. This is done to avoid *shot by dead man* problem discussed before, making sure that updates are frequent enough that the player is guaranteed to see what hits him, in this case flying automobile, before the dam-

age is dealt. On top of it, shooting and damage boost priorities of certain things. For example, updates two players shooting each other will be most highly prioritized for the players, leaving less meaningful actions to delay. As seen from debug view of the system *Figure 4* the grenade, as do all the objects, has three values associated with it: final priority, relevance to the player and desired update period. Based on relevance and update periods, the final priority is calculated and based on this priority the objects are queued for replication. In this case, grenade has relatively low priority, 0.19 out of 1.00, because it is dropped off killed players body and is not armed. Dead player's body can be vaguely seen on the top left of *Figure 4*, with its priority of 0.5, top relevance and 0 ms desired update period. This makes sense, because the game is about shooting other players, and thus player models are on top of the priority chain, for most of the time.

Game's replicating mechanism guarantees that all states will be updated eventually, with average full update time of around 3 seconds. Without prioritization, the real-time shooting would not have been possible, due to sheer amount of state data that needs to be sent and received. However, thanks to the clever *AoI* management mechanism, the all-out multiplayer mayhem is indeed a reality.

4.2 Lag compensating/reducing algorithms

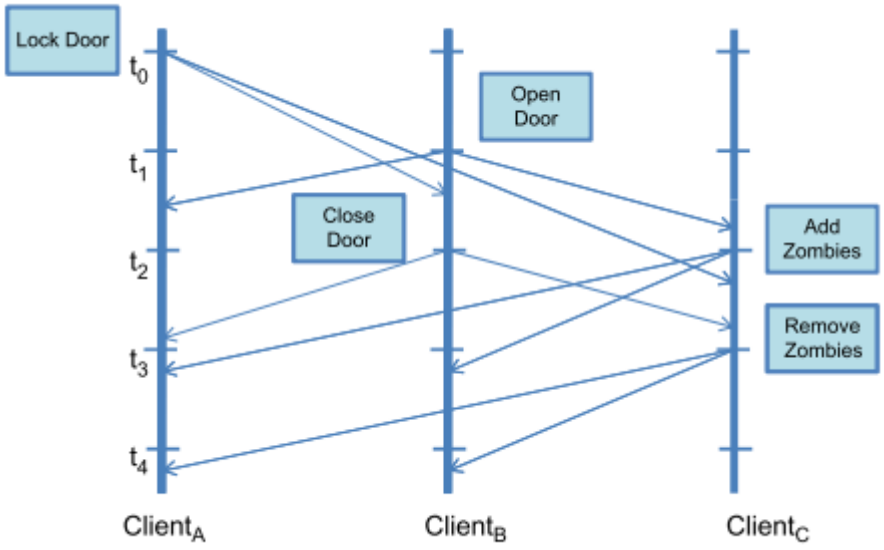

Figure 5. The opening door example involving three clients and rollback (Steed, Oliveira et al. 2009, 2010, p.366)

Assuming the client is not *dumb*, meaning that it has calculation of its own to perform onto received data from the server, it usually has a plethora of algorithms that provide smooth transitions between received state updates.

Optimistic algorithms do not reason about the orders of events, but execute them as they arrived in queue, in which every event has a *time warp*. Each event is time-stamped and each receiver processes events as the receiver receives them. If and when the receiver receives an event that occurred before the time of the last event processed, the receiver *rolls back* in time to a point in the simulation before the event that caused the rollback (*the straggler*). The events are then replayed forwards in time in simulation time order including the straggler. The main complication with time warp is that it can cause events to be undone. This is done with *antimessages*: messages that cause other hosts in the simulation to correct their simulations by rolling back. Consider the following example, shown in *Figure* 4: there are three collaborating clients, Client A, Client B, Client C. Client A opens a door, but Client B attempts to lock it before it receives the open door message. Client C activates a group of zombies behind the door whenever it sees the door open. The outcome now depends on the simulation time ordering of Client A 's and Client B 's events. The lock door message from Client A happens first (t_0), but before receiving this, Client B sends an open door message. When it does receive the message is has to rollback its state and sends an antimessage telling the other doors that the open door message is invalidated by sending close door message. Client C on receiving the open door message sends Add Zombies message. It then gets the lock door message and close door message, so on the next time-stamp it sends the Remove Zombies to rollback its state. Two key problems exist with time warp: maintaining enough state data to rollback and dealing with cascades of antimessages. Common solution derives from three techniques: making periodic checkpoints, keeping the event stream with enough information to undo needed amount of events and implementing *TSS*, technique where two versions of environment state are kept, one up to date and one "behind". Upon updating trailing state, the full sequence of events is known, which makes the state update correctly. If the state is different from up to date one, a rollback is triggered. Time warp can be implemented either on the server side, or client's, and is used in most Source engine based games, like Half-Life 2 and Counter Strike Source, alleviating the apparently dead players shooting problem as well as "fireproof" player problem (the problem where one player shoots the other, seeming to hit him locally, but in the time it takes for the message to reach server, other player has changed his location, thus invalidating the hit). (Steed, Oliveira et al. 2009, 2010, p.358-367)

Client can also *predict ahead* of authoritative state for some aspects of world state. This is usually used for extremely time-sensitive NVEs, like FPS games. In this method, the client can simulate state locally, with server acknowledging the state later. For example, when rotating camera in FPS, local simulation applies rotation immediately, without waiting for server to authorize it. The input is sent to the server, for it to distribute state changes to other participants, but it never overrides control inputs, since the player is free to rotate however and wherever he sees fit. Other use of predicting ahead is player movement. Client has full control over its movement, and can predict where it will go in the near future. The server can override the movement change if it imposes some specific

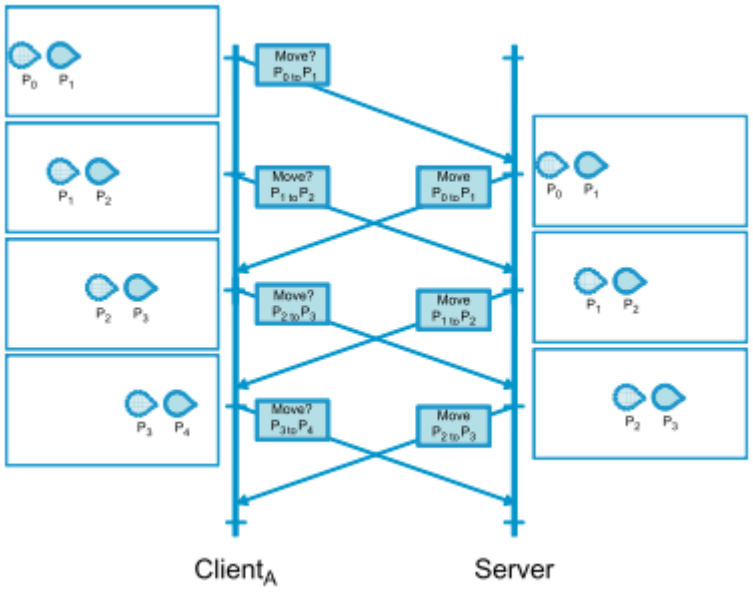

Figure 6. Successful movement predicted ahead (Steed, Oliveira et al. 2009, 2010, p.369)

constraint that the player's local environment has no information about. Multiple commands will usually be in the pipeline, so it is possible to predict multiple ones ahead.

Figure 6 illustrates successful series of confirmations of predict ahead, with two messages. The client moves and gets its moving confirmed by a server, with movement happening one or two rendered frames (about 0.03 seconds) ago.

If the server was to correct one move command message, the simulation must adopt to the newest state, forming branches of possible predicted outcomes as shown in *Figure 7*. In this situation, one player moves horizontally and one vertically, and depending on the server's response, one of branches is closest to truth. Top branch illustrates first player being blocked by other, second tries to preserve predicted position by moving around the obstacle and third makes client try to move again. In practice, the situation is more complex, requiring more branches.(Steed, Oliveira et al. 2009, 2010)

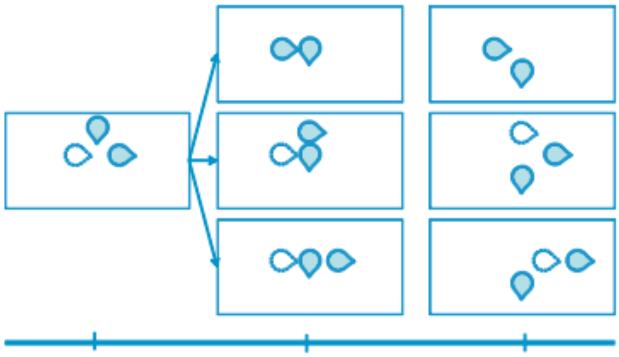

Figure 7. *Available options preserving predicted place (Steed, Oliveira et al. 2009, 2010, p.370)*

4.3 Programming solution

Novel "*timeline*" model (Savery, Graham 2013, 2012) allows access to temporal states of shared data, making it available to modify in future, as well as in its past. Timeline variables also can be shared, which allows for closer approximation of time perception between players.

Timelines model aims to help programmers with time management, providing object of reference to time. Fuelling a tank is used as an example for simplistic explanation of how timelines work. It mainly consists of events having among other things their time of occurrence attached to them, which helps to interpolate, filling in the gaps between two known values, as well as extrapolate, predict the future unknown value, more precisely. On top of that, clients' timelines can communicate changes, adjusting or overruling each other state data, or simply providing it for lag correction algorithms to make both simulations more accurate in comparison. (Savery, Graham 2013, 2012)

One problem that the mechanism solves is called stale state, in which the local representation of remote state updates needs additional mechanisms to ensure that it has enough reliable data to predict following state change, beginning to render locally the things that are likely to happen in a moment. Moreover, the solution aims to mitigate the differences of client's simulations by having an access to all state data of each individual client for any wanted time. Timelines also help with input delaying, commonly used for extremely timing sensitive games. Input delaying makes both players inputs register with the same deviation, making the simulations seemingly accept inputs at the same global time, varying this delay based on each client latency. Using Timelines also allows developers to seamlessly switch between network and local desired lag, just by utilizing state change's time.

5. SUMARRY

Maintaining shared world illusion in real-time networked videogames is a complicated task, that consists of many aspects, resigning at different levels of obstruction. Challenges emerge from the lowest transportation layers, making choosing the optimal set of transporting protocols for optimal performance. These all have their pros and cons, and choosing poorly may result in chain effect, leading to the need to rewrite huge parts of the game's code.

Choosing the right architecture affects replication greatly too. Whether it is p2p, client-server, or a mixture of both, it is crucial to understand each one's strengths and weaknesses, from both software architecture and commercial points of view. Running dedicated servers costs money and is considered a golden standard for commercial products, however for some cases it is not needed, leaving development companies more resources for implementing other parts of the game better or faster.

The need to use networks affects user experiences as well as development in its own unique way. The amount of state data translated needs to be as small as possible, but enough for accurate replication on other players' simulations. Tricking a player to believe that events of the game happen in nearly real-time, even though sending, receiving and processing state data takes time, is a merit not every development team can achieve. Developers also need skills as well as imagination to work with multiple simultaneous times, for each player's simulation, allowing some, and denying other conflicting events.

All these aspects combined, prove to be one big challenge of game developers of this century. The task is so vast, that whole studios are specializing in this aspect alone, and often are contracted to make already made game networked one. As it stands, there's little standards and no singular solution for all possible types of games, so each studio has to deal with complication the best it can, applying discussed techniques to make multiplayer as seamless and latency-free as single player is. Implementing good netcode can not only be one of the biggest selling point of the game, but also push game industry forward to develop in this field, making shared world illusion a reality.

6. REFERENCES

A Dictionary of Computer Science. 2016. 7 edn. Oxford University Press.

PANTEL, L. and WOLF, L., 2002. On the impact of delay on real-time multiplayer games, 2002, ACM, pp. 23-29.

SAVERY, C. and GRAHAM, T.C.N., 2013, 2012. Timelines: simplifying the programming of lag compensation for the next generation of networked games. *Multimedia Systems,* **19**(3), pp. 271-287.

SCHIPER, A., CHARRON-BOST, B. and PEDONE, F., 2010. *Replication: Theory and Practice.* Berlin, Heidelberg: Springer Berlin Heidelberg.

SINHA, P.K., 1997. *Distributed operating systems: concepts and design.* New York: IEEE Press.

STEED, A., OLIVEIRA, M. and BOOKS24X7, I., 2009, 2010. *Networked graphics: building networked games and virtual environments.* Oxford; San Francisco, Calif: Morgan Kaufmann.

YASUI, T., ISHIBASHI, Y. and IKEDO, T., 2005. Influences of network latency and packet loss on consistency in networked racing games, 2005, ACM, pp. 1-8.

LYSENKO, M., 2014, Replication in network games, 0fps.net, 2014.

ALBRIDGE, D., "I shot you first" Gameplay Networking in Halo Reach presentation, GDC 2011.