Automated Network Compression for Online Games

Manuel A.W. Herrmann

MASTERARBEIT

eingereicht am
Fachhochschul-Masterstudiengang

Interactive Media

in Hagenberg

im November 2016

Declaration

I hereby declare and confirm that this thesis is entirely the result of my own original work. Where other sources of information have been used, they have been indicated as such and properly acknowledged. I further declare that this or similar work has not been submitted for credit elsewhere.

Hagenberg, November 23, 2016

Manuel A.W. Herrmann

Contents

Declaration iii

Kurzfassung vi

Abstract vii

1 Introduction 1

2 State of the Art 3
2.1 Common Connection Issues and their Cause 3
2.1.1 The “Last Mile” Problem 3
2.1.2 Nodal Delay and Nodal Packet Drop 3
2.1.3 Bit Errors 4
2.1.4 ISP Traffic Management 4
2.1.5 Out-of-Order Delivering and Jitter 5
2.2 Detection of Congested Connections 5
2.2.1 Congestion Control in TCP 5
2.3 Common Encoding Techniques and Algorithms 6
2.3.1 Serialization 6
2.3.2 Bit Packing 6
2.3.3 Delta Encoding 7
2.3.4 Entropy Encoding 7
2.3.5 Bound and Quantization 8
2.3.6 Selective/Variable Updating 8

3 Implementation of the C.A.N.P. Framework 10
3.1 General Functionality 10
3.2 The Monitoring Module 11
3.3 The Encoding Module 12
3.3.1 Encoding Tables 12
3.3.2 Delta Encoder 13
3.3.3 Serializer 14
3.3.4 Quantizing 16
3.3.5 Selection Module 17

3.4 The Control Module 18

4 Evaluation **20**
4.1 Isolated Look at the Compression Methods 20
4.1.1 Compression Rate 20
4.1.2 Performance Tests 26
4.2 Testing the Framework 30
4.2.1 Test Arrangement 30
4.2.2 Synthetic Tests 32
4.2.3 Organic Test 38

5 Conclusion **42**

A Inhalt der CD-ROM **44**
A.1 PDF-Dateien 44
A.2 Projekt 44
A.3 Websites 44
A.4 Sonstiges 44

References **45**
Literature 45
Online sources 45

Kurzfassung

Diese Arbeit beschäftigt sich mit dem Projekt „Connection Adaptive Networked Physics“, das ursprünglich entwickelt wurde um ein ebenes Spielfeld für alle Teilnehmer eines Onlinespiels zu schaffen, unabhängig davon, ob sie über eine ausreichende Internetverbindung verfügen oder nicht. Das funktioniert, indem das C.A.N.P.-Framework für jeden Spieler, dessen Verbindung zu *irgendeinem Zeitpunkt* während einer Session nicht den Anforderungen an die benötigte Bandbreite entspricht, ein maßgefertigtes Netzwerkprotokoll erstellt, das, indem es überflüssige oder weniger wichtige Informationen stärker komprimiert oder gar weglässt, die benötigte Bandbreite an die Verbindung des Spielers anpasst. Im Laufe der Projektentwicklung ergab sich jedoch noch zusätzlich, dass eine Automatisierung bestimmter Teile des üblichen Netzwerkmoduls eines Onlinespiels nützlich wären, sowohl um die Effektivität einiger verwendeter Kompressionsmethoden wie *Bit Packing* und *Delta Encoding* zu erhöhen, als auch um die Arbeit, die bei der Implementierung des Netzwerkteils sonst anfällt, deutlich zu reduzieren. Damit wurde „Automatisierung“ ein weiterer zentraler Punkt des Projekts. Neben der Projektvorstellung konzentriert sich die Arbeit vor allem auf Leistungs- und Stabilitätstests des Frameworks, und gibt zu Beginn außerdem einen Überblick über die häufigsten Verbindungsprobleme sowie beliebte Kompressionsmethoden für Netzwerke.

Abstract

This thesis covers the project "Connection Adaptive Networked Physics", which was originally developed to create an even playing field for all participants of an online game, whether they possess a sufficient internet connection or not. This is made possible by creating a custom network protocol, which, by applying stronger compression to or even discarding less important or redundant information, matches the needed bandwidth to a player's connection if it drops below minimum requirements at *any time* during a session. During the project development it appeared that automation of certain parts of the usual network module of an online game would be useful, not only to boost the effectiveness of some of the utilized compression techniques like *bit packing* and *delta encoding*, but also to significantly reduce the amount of work needed to implement the network section. With this, "automation" became another focus of the project. Apart from the project review, the thesis mainly concentrates on performance and stability tests of the framework, and gives an overview of the most common connection issues as well as popular compression techniques for networks in the beginning.

Chapter 1

Introduction

"Everyone should be able to participate in online games with the same chances, no matter their connection". This was the thought that inspired the project "Connection Adaptive Networked Physics", a networking framework that is able to adjust the bandwidth taken by stream of data between a game server and a client to the bandwidth actually being available, preventing congestion of the connection and all the nasty side effects (like latency spikes and packet loss) coming with it. Having lived in a dorm where the connection speed varied extremely throughout the day, from several mb/s early in the morning down to sometimes a mere 50 kb/s in the evening, the author experienced it first hand what it means to try to play a bandwidth demanding online game (like almost any MMORPG) on an insufficient connection. This made the motivation behind developing the project a very personal one.

Having taken part in the development of several online game prototypes in the past years, this problem often showed up during their presentations as well. With public Wi-Fi connections, and even a phone used as a hot spot one time, the bandwidth available for the presentations often turned out to be less than needed, resulting in suboptimal performance of the presented projects. This was especially annoying as implementing the network components and setting up communication between server and client already takes a lot of time, so for such short lived prototypes, optimizing the networking was simply too much. This ultimately inspired the second part of the project: Almost complete automation of the networking process.

Ordinary network communication takes time to implement. Connections have to be managed, messages between server and client have to be handled, but most importantly, lots of methods for fabricating and decoding messages have to be implemented. The tedious process of writing encoding and decoding methods, in which fields and parameters are encoded one by one, is something any game programmer that tried himself at low level network communication might be familiar with. This is where the C.A.N.P.

framework aims to take a lot of time off the developers hands: With the help of encoding tables, all fields of any objects given to the framework are automatically encoded and converted to messages, with additional networking specific benefits compared to conventional object serialization.

With this, the project's goals are clear: More convenience in setting up the networking component for online games for the developers by automating a big part of it, and more accessibility for bandwidth demanding online games for the players by having the game adapt to their connection.

Chapter 2

State of the Art

2.1 Common Connection Issues and their Cause

Before going into greater detail with the project, it is also important to expand on its circumstances. That means introducing and explaining the most popular encoding techniques used in networking for online games, of which some have been implemented in the project, as well as taking a look at the idea of *congestion control* and how it already is implemented in the different versions of TCP. But prior to that, common roots of connection problems are investigated. Knowing those will give a better insight on how to deal with the connection issues later in the framework.

2.1.1 The "Last Mile" Problem

Often being the root cause of bandwidth limitations, the "last mile" describes the link between the end user and the first node(s) in a network chain. The heavy branching of lines connecting all end users to the first node means an exponential rise in cost for upgrading the "last mile" to newer technology, and with the last miles often being in the hand of a single ISP there is little competitive drive for the ISP to upgrade, which holds true especially in rural areas (see [8]).

2.1.2 Nodal Delay and Nodal Packet Drop

The total delay a packet experiences on its way through an end-to-end network is the sum of all nodal delays. A single nodal delay consists of the processing delay, the queueing delay, the transmission delay and the propagation delay (see [6, 7, 4, 11]).

Processing delay is the time it takes the node to read the header, check for errors and determine its destination (the link it will be pushed on). The duration of this varies depending on the complexity of the processing, with simple packet forwarding being almost negligible and more complex opera-

tions like network address translation or deep packet inspection pushing it to up to 1ms, therefore significantly adding the the total delay if multiple nodes in the network are affected.

Queueing delay is the time a packet has to wait in the node's buffer before being pushed on the link that was determined earlier. When the load factor of this link exceeds 100% (meaning the queue receives data at a faster rate than the link can take) the queueing delay will first rapidly increase before the queue buffer finally overflows, causing the packet to be dropped. This is a primary cause of packet loss. Some packets may even be dropped shortly before the buffer overflows if the node uses RED (random early drop). This signals impending congestion to the sender, which creates a small time window where the sender can throttle its output before total congestion occurs.

The *transmission delay* is the time it takes to push all bytes of the packet onto the link. Naturally, bigger packets take longer, but even at the mtu (maximum transmission unit) this delay is basically negligible relative to the total latency.

Lastly, the *propagation delay* describes the time needed for the packet to travel through the physical link to the next node. The delay starts when the first part of the packet (the header) is sent and stops when it arrives. It is primarily dependent on the length of the link.

2.1.3 Bit Errors

As written earlier, packets are checked for bit errors during the processing stage of every node (done via cyclic redundancy checks using check sums), and detected ones are dropped. This is another major cause of packet loss. Bit errors are mainly caused by the natural noise of physical links and signal interference, but also by distortion, self synchronization issues and attenuation. Long copper wires and wireless transmission suffer the most from this. The relation of successfully transmitted bits to corrupted ones is called *bit error rate* (BER) and is a popular indicator for connection quality. In practise, the bit error rate is typically at 1^{-12} or less for the wired part of a connection (and the Ethernet specification allows for BERs of up to 1^{-10}), and while wireless connections can have much worse rates, they are usually corrected automatically using Forward Error Correction (see [6]).

2.1.4 ISP Traffic Management

A common strategy employed by ISPs is to shape the network passing through their nodes, mostly to avoid congestion, especially during peak hours. This means packets may be distributed to different channels depending on their type, and maximum throughput may be throttled for all or individual types of traffic and even individual senders depending on their

behavior (e.g., an ISP may choose to throttle the throughput from a sender transmitting large amounts of peer-to-peer traffic when the links are almost at full load to ensure fair service) (see [12]).

2.1.5 Out-of-Order Delivering and Jitter

Both of these issues are side effects of the problem that packets have to enter a queue on every node. Jitter is caused by these queues being of unpredictable length every time a packet arrives, while a node that was previously congested and is now open again may cause a newly sent packet to be routed through a faster route using this node, overtaking older packets in the process (see [10]).

2.2 Detection of Congested Connections

2.2.1 Congestion Control in TCP

The sending behavior of TCP can be divided into two parts: The *congestion recovery* phase and the *congestion avoidance* phase. When establishing a connection, TCP takes the advertised bandwidth of the recipient and sets it as the upper limit for transmitted data per second. It will also initialize a variable called the *congestion window* for the sender, which defines the actual amount of data per second sent and starts at one (packet), but will increase every time data has been acknowledged. The rate at which the window increases is defined by the current phase and the TCP version:

During the recovery phase, the congestion window of TCP version Tahoe is increased linearly for every ack, and with the additional data increasing the window as well, it grows at an exponential rate (doubling every round trip time), until it reaches the advertised bandwidth, or a packet drop is detected. In the case of a packet drop, Tahoe sets a control threshold at half of the reached window size and then resets the window back to its start size. This time, the recovery phase is interrupted at the control threshold and transitions into the avoidance phase where every new ack only increases the window logarithmically, resulting in linear growth of the window, e.g. +1 packet per second (see [5], p. 30 et seq.).

The advantage of the avoidance phase is that in case of congestion, the amount of lost packets is minimal, whereas a congestion during the recovery phase may cause data loss in the order of half the congestion window or higher, since the window still increases rapidly due to incoming acks while the lost packets need to time out before they are detected, or duplicate acks from the sender need to arrive. Additionally, the avoidance phase means a significantly higher time until congestion is encountered again, resulting in less throughput throttles.

To further avoid the risk of big data loss during the recovery phase, TCP

Vegas alternates between exponential and linear growth every RTT (see [2], p. 12). Other versions of TCP each use slightly altered versions of congestion control (e.g., TCP Reno skips the recovery phase in case of duplicate acks, and restarts right at the control threshold), but the essence stays the same: If congestion in form of packet loss or delay is detected, the output is throttled severely, which is a big reason why UDP is usually preferred for networking in games. Here, transmission needs to stay real-time, simply buffering and throttling the game state broadcast is no option. The solution is, instead of sending the information slower, to decrease the bandwidth needed by it through heavy compression.

2.3 Common Encoding Techniques and Algorithms

2.3.1 Serialization

Serialization is a popular method to turn any primitive data type or object into a stream of bytes with the goal of being able to safely store or transmit the data for later reconstruction. In order for the reconstruction method to properly decode the stream it is imperative for it to know exactly how it was encoded, either by having mirrored serialize and deserialize methods, by exchanging this information beforehand or by abiding a standard. Such a standard is often provided by the programming language either through native support like in Java or Python or through libraries. The advantage of this is the simplicity of the process, objects often require not much more than to simply being marked as *serializable* and already the encoding and decoding works. Creating a custom serialization standard can however also save a lot of space, e.g., by using *bit packing*. A primitive data type like a signed 32 bit integer can store a number higher than two billion, but in games this magnitude is seldom needed, yet it still always takes 32 bits of space. If it is known that a certain property never exceeds 24 bits and the decoding side knows when to expect this, the serialization can free up the 8 bit of unused space (see [3], p. 4 et seq.). It is hard to compress the serialized binary code any further, but the result of other compression techniques can still often be serialized. Thus, serialization is usually the last step when encoding a network message.

2.3.2 Bit Packing

The method used to compress data by cutting unused bits and packing what is left in a new container is called bit packing (see figure 2.1). It uses an encoding protocol that specifies both the data types and the bits used for all values packed together, so the resulting buffer does not contain any encoding information. Bit packing often uses bit shifting and other bitwise

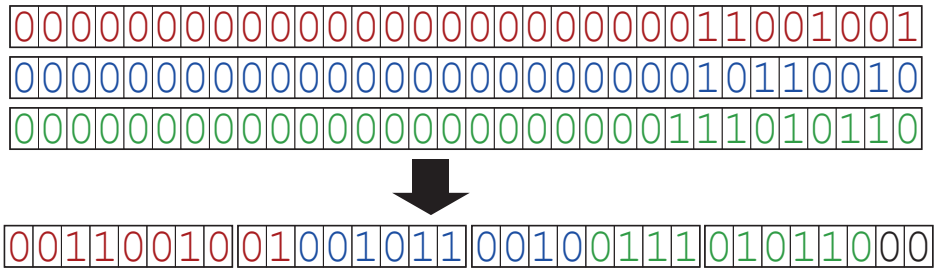


Figure 2.1: Example Bit Packing: Three 32-bit integers (in binary representation) packed into four bytes, using 10-bit containers.

operations to extract exactly the relevant bits and to place them at the right position in the target container. Byte arrays are commonly used as containers, especially when they are needed for networking, but 32 or even 64 bit integers can be used just as well for better performance.

2.3.3 Delta Encoding

Delta encoding serves a similar goal like serialization, reducing the size of a transmitted number, but achieves it in another way. As the name delta suggests, instead of always sending the absolute value of, e.g., a transform of the current state, only the difference relative to an earlier state is sent (see [1], p. 3 et seq.). Because latency is a big factor in online games, most state updates are done via the unreliable protocol UDP. This means that in order for delta encoding to work, the client has to acknowledge a successfully synced game state periodically. After receiving this ack, the server can then encode all further updates relative to it and thus severely reduce the magnitude of the transmitted values. Combining this with bounding the values for serialization results in a big reduction in message size. A downside to this technique is that the server always needs to keep a buffer of recent game state snapshots he can encode his messages relative to.

2.3.4 Entropy Encoding

Entropy encoding is a loss-less compression technique that is based on the frequency of single digits or characters in a message. A very popular method is Huffman Encoding, where a so called Huffman Tree (see figure 2.2) is created based on entropy and is traversed every time a character is encoded or decoded. This can save a lot of space when there is a big set of characters, but only a few of them make out the majority of a typical message, because these ones will be represented using less bits. As both serialization and entropy encoding produce a bit stream, these two compression techniques

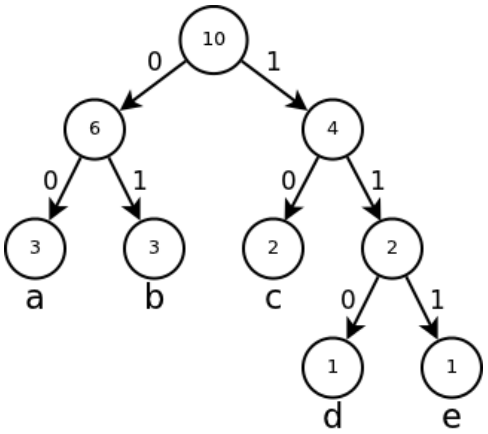


Figure 2.2: A simple Huffman Tree. When reading the tree, traversing it to a character's leaf determines the bit code representing that character.

are practically not used together.

2.3.5 Bound and Quantization

All compression methods shown so far achieve loss-less compression, bound and quantization though do not. It is an alternative approach to cutting down the magnitude of values without relying on acknowledgments, by redefining the minimum steps a value can change and then bounding the value to a minimum and maximum, essentially creating a custom-bit integer data type, similar to a custom serialization standard. The position of a game entity is normally defined with floating point accuracy, which is usually way more precise than it needs to be for a simulation on the client. Under the assumption that due to latency and package loss a perfect and loss-less simulation of the server state cannot be achieved anyway, it might be enough to send the translation of game entities in 20 cm steps (so that, e.g., transmitting a length of 12 would equate to 2.4 m) and letting interpolation do the rest (see [9]). This is especially effective when combined with delta encoding as it may eliminate the need to update an entity altogether when its state on the server didn't change enough to snap into the next step on the scale.

2.3.6 Selective/Variable Updating

The biggest reduction of a network message's size can be achieved by simply not sending it. Carefully dividing the game entities into gameplay relevant and non gameplay relevant for the player and then only syncing the relevant ones can increase network performance considerably. The best results are achieved when this distinction changes dynamically over the game. Enemies

that are far away from the player in a game with limited interaction distance are not gameplay relevant as long as they stay at distance. The key is to not make the player notice the unsynced entities, which may also be the case when game entities flood the screen, like different players in social hubs. Here, employing variable update rates reduces network throughput without impacting player experience too much.

Chapter 3

Implementation of the C.A.N.P. Framework

The "Connection Adaptive Networked Physics" framework is composed of three major modules: A monitoring module that keeps track of packets and measures the connection quality of all individual clients, an encoding module that encodes objects into ready-to-send messages using several different encoding techniques and algorithms, and a control module that uses the information provided by the monitoring module and tries to find the perfect encoding tailored to the connection. Apart from these three modules, the framework also houses general network communication functionality, like establishing and maintaining a connection and sending or receiving packets on a reliable or a non-reliable channel.

3.1 General Functionality

For establishing new connections, pushing packets onto the socket and receiving and decoding unknown messages, the *Canet* class provides all necessary functionality. Server and client both use the same classes, though one of them, the server, needs to have a fixed port and IP-address so a connection can be initiated by the client, which is done through transmitting a fixed protocol. The user needs to register his custom server as a listener in his Canet-instance, so he can receive events for incoming, accepted and dropped connections, and, most importantly, incoming packets and their content. Once client and server have successfully connected, the user receives a *CanetClient* object that from here on out regulates all activity on this connection.

CanetClients fetch all packets that were sent from the address of their connection directly from the socket. Afterwards, the Canet class will read the leftovers and scan them for new connection attempts. To differentiate between user-sent messages and internally exchanged information, all pack-

ets receive a preposed 1-byte flag. All user-sent messages are first collected in two separate queues, one for reliable and one for unreliable messages. During the update phase of the framework, after fetching all packets from the socket, all queued messages are packed together into packets with a maximum size limit (maximum transmission unit) of 1472 bytes, which minimizes packet count while preventing packet fragmentation. The resulting packets are then pushed onto the main queues, where they are pushed onto the socket together with all the internal packets just moments later. Naturally, user-sent messages are unpacked again and returned one by one at the destination, so the user receives them the exact same way they were sent.

3.2 The Monitoring Module

The monitoring module consists of a single class called *PacketTracker* that every CanetClient runs its own instance of. Every packet sent and received must pass through here so the tracker can assign a number to outgoing and read the number of incoming packets. Numbers are assigned ascending and range from 0 to 32,767 (cycling back to 0 when exceeding the max value), thus using up 15 bits of memory. An additional bit is used to separate between reliable and non-reliable messages, resulting in a total of 16 bits of extra space per packet.

The packet numbers are primarily used to identify missing, out-of-order and duplicate packets. Upon arrival of a packet, the tracker compares its number with the number of the last received packet. If it is not the next one in the series, all numbers in-between are marked as *missing* and the received packet as *out of order* (in the case of a higher number). In the case of a lower number, the list of currently missing packets is checked, removing the *missing*-mark if the packet was contained and discarding and marking the packet as *duplicate* if not, or if its number equals the last received packet's one. Packet numbers that are marked as missing will wait a short amount of time (in case the packet is simply late) before their missing-status is considered final. The stats calculated from identifying all abnormal packets are stored in the corresponding CanetClient, where the control module and the user, should he be interested, have access to them.

Similar to the list of currently missing incoming packets, all outgoing packets sent through the reliable channel will temporarily be stored until an acknowledgment packet is received or the packet timed out, in which case the process repeats and it is sent again, until the ack is finally received. Resent packets receive a new packet number, thus eliminating the chance of the number being no longer contained in the list of missing packets at the destination client and the packet being falsely detected as duplicate and discarded.

For measuring the bandwidth used by current packets, packet sizes are

added up over time the same way the counts for abnormal packets are. The size of a packet is determined by the contained message, encoding information from the encoding module, the previously mentioned 4 bytes of tracking information and the 52 bytes needed for the datagram headers (24 for the ethernet frame, 20 for the IPv4 header and 8 for the UDP header).

The round trip time is measured by regularly pinging the destination client, which also provides information about the connection status. If pings have not been answered for a certain amount of time, the connection will be declared dead and dropped. Pings may be delayed a bit should the outgoing packet queue be too full so an overloaded socket cannot distort the result.

3.3 The Encoding Module

The encoding module is the heart of the framework and houses all the different encoding and compression techniques used to turn objects into ready-to-send messages of the smallest possible size. It generally differentiates between loss-less compression, which is applied to all objects as a baseline, and lossy compression, which is only applied to various degrees when the control module deems it necessary due to detected connection issues.

The user can access the encoding module via the *EncodingManager* class, of which an instance is automatically created within the CanetClient when establishing a new connection. The EncodingManager does not encode single parameters like primitive data types, but whole objects. This makes using it incredibly easy, as during the broadcast of the server game state, one simply needs to pass the entities to the EncodingManager one by one and will receive a completely encoded message in form of a byte-array which can then be sent immediately via the CanetClient. This extra step makes it also possible to directly transmit byte-arrays that have not been generated by the EncodingManager but by the user himself, bypassing the "connection-adaptiveness" and using the framework just for its connection handling and message transmission.

3.3.1 Encoding Tables

When encoding a passed object, the EncodingManager needs to know which of its fields the user wants to encode and transmit. For this it uses a class called *EncodingTable* (see figure 3.1). Encoding tables can either be created automatically, which happens when the encoding manager first comes in contact with the class of an object, either by en- or decoding, or by the user. While the automatic version is often sufficient, it will always include *all* fields of the objects, so in order to be more selective, a user generated encoding table with manually chosen fields is recommended. Apart from the field count, a manual encoding table also allows to tweak encoding parameters for each individual field, which can both improve the quality and compression

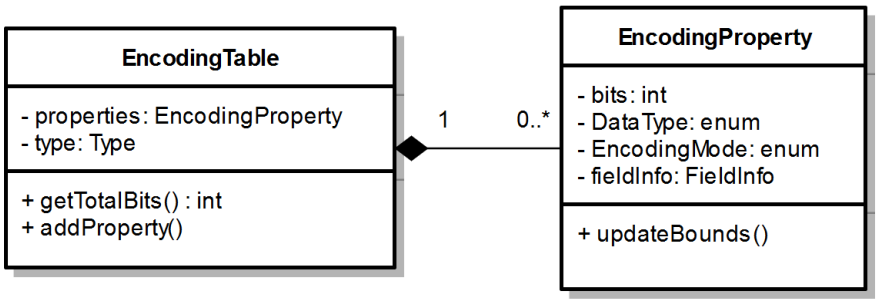


Figure 3.1: A simplified class diagram of the *EncodingTable* class and its properties.

of the encoding. It is important that manually created encoding tables are created on both the server- and the client-application so that objects can be successfully decoded again.

Often some fields of an object only require initial or irregular transmission, like the type of a certain entity that only needs to be shared on the spawn message. For this purpose it is possible to create multiple encoding tables per class, and the user can choose which one he wants to use when encoding objects.

Encoding tables also contain dynamically tracked information about the contents of fields which are used by the serializer and the quantizer. During the server run time, this information often leads to a change in the optimal encoding settings for individual fields. When this happens, the modified encoding table is synced immediately with the client to replace the old one.

3.3.2 Delta Encoder

Objects that were passed to the encoding manager are first processed by the delta encoder. On server side, the delta encoder is responsible for building game state snapshots out of all the objects passing through, and after receiving the first client confirmation, encoding new objects relative to certain game states. On client side, it tries to reconstruct the by the server defined game states from incoming messages and decodes them according to the instructions.

Game state snapshots are created by collecting and storing the values of passed objects inside a buffer. A snapshot is considered complete, when the update method of the delta encoder is first called after having previously collected and encoded objects. Each snapshot is assigned an ascending number, cycled in the range of 0 to 255, which means the maximum number of snapshots that can be stored at once is 256. Every object passing through

the delta encoder, whether it is actually encoded or only used to build the game state snapshot, receives the number of the snapshot it is currently building, so that the client will be able to allocate it. The client now faces the task to ascertain whether a snapshot is complete, meaning all objects of it have arrived. Since the client has no way of knowing which and how many entities were actually sent by the server, determining this needs to be done by looking at the packets in which the objects arrived. For this purpose, the lowest and highest numbered packet is tracked for each snapshot. During the verification phase, the packet tracker is asked if all adjacent and intermediary packets have arrived, guaranteeing a complete snapshot on success. Even with one or two missing packets the snapshot may possibly still be complete, since those packets may have contained only internal information, but since the client has no knowledge of the missing packets' contents, it is safer to assume that the snapshot is incomplete.

Snapshots are verified multiple times per second, but not on every update. During the verification phase, all not yet examined snapshots are checked. The most recent complete one is immediately reported to the server, who now knows that the client will be able to decode all entities encoded relative to this snapshot. The encoding of an entity is simply done by searching the client-confirmed snapshots for an earlier version of the entity and subtracting the older values from the current one, returning only the difference (the delta). Next to the current snapshot's number, delta encoded objects also receive the number of the snapshot they were encoded to, so the client knows how to decoded it again.

Once every entity was featured in at least one completely received game state snapshot and only the difference of their field values are transmitted, floating point accuracy issues and (later on) quantized values can lead to the client game state drifting significantly apart from the server game state. To avoid this problem, the server regularly sends a whole snapshot completely uncompressed by delta encoding, which the client detects and then prioritizes to verify and report to the server. Doing so lets the client state snap back into the server state and prevents the amassing of errors before they become notable.

3.3.3 Serializer

Turning the final property values of objects into a byte array is the job of the *Serializer* class. The class has two layers, and while both work similarly (storing the data types in the byte array using their binary representation), the first layer works uncompressed and writes into the buffer in byte steps using the same amount of bytes the data type takes up in memory, the second layer advances bit by bit and can encode data types compressed using any number of bits specified. This second layer is implemented using a nested class *BitSerializer* and is the standard procedure for all properties marked

Program 3.1: The serialization method for unsigned integers of up to 64 bits. Bits are read one by one starting from the least important bit until the specified amount is reached.

```
1 public void serializeUInt(ulong value, int bits)
2 {
3     if (bits < 0)
4         return;
5     for (int i = 0; i < bits; i++)
6     {
7         buffer[(pos + i) / 8] = (byte)(buffer[(pos + i) / 8] << 1 |
8           (((byte)(value >> (bits - 1 - i)) & 1)));
9     }
10    pos += bits;
11 }
```

as *compressed* in their object's encoding table.

The bit serializer employs the earlier discussed bit packing technique (see section 2.3.2) to read and write values bit by bit using bit shifting and bit masking with the total amounts of bits defined by the encoding table (see program 3.1). Since both server and client possess the same encoding table containing the information about order of encoding, kind of data type and amount of bits used, the serializer adds no additional flags to the byte array, jut the pure, compressed values of the encoded object's fields. The (bit) serializer is able to encode and decode the following data types:

- **Booleans:** Easiest to encode and decode, by simply reading or writing a single bit into or from the array (using the bit serializer). Compared to regular serialization using 8 bits per boolean, this means a guaranteed net saving of 87.5% used space.
- **Signed and unsigned integers:** As their binary representation can be read as-is, these are relatively easy to encode and decode using bit shifting and bit masking as shown in the highlighted method above (program 3.1). As the process is the same for integers of any length from *byte* to *long*, the same en- and decode methods apply to them. In case of signed integers, an additional bit containing the sign bit is used.
- **Floating point values:** Their binary representation is harder to fetch and also divided into mantissa, exponent and sign bit. As the biggest part, the mantissa, uses all its available bits most of the time, simply cutting bits like with integers is no viable strategy. Thus, for loss-less encoding, *float* and *double* are serialized normally using four or eight bytes written into the byte array. They can however be significantly compressed if quantization is in use (see section 3.3.4), in which case their value is already stored as an integer and they are serialized as

such.

- **Strings:** These are first converted into byte chunks using an utf-8 encoder which are written into the byte array byte by byte using integer serialization after the amount of encoded bytes was written into it for decoding purposes. String is the only data type that breaks from "no additional serialized information", necessarily so, as it is also the only one not bound by a fixed container size.
- **Enumerations:** *Enums* work by using an underlying data type, which is normally byte, but can also be specified to be any other integer data type. This underlying type is detected when the encoding table is first created and used to encode and decode an enum as an integer.

Especially when quantization is used, the most commonly serialized data type is an integer, which, when using the bit serializer, requires information about the amount of bits used in the encoding from the encoding table. The table in turn can receive this information through two ways: By the user explicitly specifying it when adding the respective field to the table, or by making use of *automatic bound tracking*.

Every parameter of the encoding table automatically tracks the biggest and smallest value ever encoded and can thus quantify how many bits are needed to encode it. These bounds are of course based on past information and could be exceeded in a future update, which is why every parameter is padded using a safety buffer of a few bits, the amount depending on how long values have been tracked. The longer they have, the less the chance of the actual bound not having been discovered, and so the buffer slowly shrinks from padding to the full data type container size down to only one bit. This way, eventually the perfect serialization settings are found even without this information given by the user.

As delta-encoded values are typically much smaller and need less bits to encode, they are tracked separately. The bound also depends a lot on packet loss, since higher loss results in higher delay until a complete game state snapshot was successfully reported by the client to the server, so it is tracked separately for every client as well. In order to not needing to start the tracking from zero, the biggest known bounds from active connections are shared when a new connection is established.

3.3.4 Quantizing

Quantizing is available for floating point values and integers and happens in the *EncodingManager* directly before serialization. As mentioned earlier, because of the nature of the binary representation of floats, serializing them using bit packing does not free up unused space in the way it works for integers. However, quantizing them into integer containers works really well. This works by predefining a fixed minimum increment for all floats that

should be quantized in the encoding table and, before serializing, multiplying the floating point value by one divided by the minimum increment and then casting it to an integer, flooring all precision smaller than the minimum step in the process. After deserializing on client side, the float is multiplied by the minimum step again to return to its old value, but with lost precision. The total amount of bits needed to encode floating point values this way is the sum of the bound obtained via maxima and minima tracking as well as the bits needed to express the minimum increment. The most efficient way to define this minimum increment is as

$$Increment = \frac{1}{2^n},$$

where n is the amount of bits needed to express the increment. Because of this, the minimum increment is initialized and stored as the amount of bits that should be used on it, not as an actual value.

Integers are quantized fairly straightforward, by cutting one or several bits, starting at the least important bit and going up from there. Cutting off just one bit means that the minimum increment for integers changes from one to two, cutting off two bits changes the increment to four, three bits to eight, and so on. This operation has different effects on the precision of compressed integers though – the bigger the value on average, the less it is notable, and in reverse, cutting off four bits of an integer that does not exceed eight will probably make the compressed value unusable. Because of this, the bounds determined by the minima and maxima tracking of the encoding table are used to define the severity of the quantization applied to each entity field. Just as the bounds, this will often change at run time and is therefore included in the encoding table syncing.

Aside from a field's global quantization strength stored in the encoding table, it also varies on a case by case basis when looking at the object that is currently being encoded. This local quantization can depend on the object's priority and distance to player (if applicable) and its strength decided by the selection module. It is segmented into four strengths: No quantization, normal quantization, double quantization and quad strength quantization. Since this varies from object to object and the rules for its application need to change and adapt fast when the connection quality is dropping, the quantization strength used is stored as a two-bit flag in front of the object's encoded values.

3.3.5 Selection Module

The class *SelectionModule* gets its name from its task to select the proper encoding for every outgoing object. As mentioned earlier, this includes the four different degrees of quantization, but the module may also elect to not encode and send an object at all during this update (see section 3.4). In this

case the encoding manager will return *null* instead of the processed byte array, but the user does not have to check for this as empty arrays or null will not be transmitted regardless. This also means that entities the user attempted to send to a client may possibly not arrive there so knowledge of this behavior is important.

When creating an encoding table, the user has the choice to assign a priority from zero to three to it, and therefore the object the table is used to encode. He also can feed the target client's position in the game world to the CanetClient, which forwards this information to the selection module. The module uses this position to attempt to calculate an object's distance to the player, which also requires the user to mark the respective properties in the encoding table so the module knows where to look. It needs either the distance or the priority to work, if none are given, it is skipped. Otherwise, it will consult the *encoding plan* created by the *control module* to determine the object's fate, which normally means increasing compression the lower its priority and the bigger its distance to the player. The only objects exempt from this are the ones with priority zero. This highest priority means that the object will never be quantized other than the first layer of float quantization (if the user chooses to) and it will always be transmitted. The selection module and therefore the possibility to adapt compression to a connection quality drop is therewith bypassed, so it should be reserved for only the most important entities.

3.4 The Control Module

The control module, implemented in the class *CompressionControl*, is the third major part of the complete framework. It is responsible for detecting a bad connection quality and reacting appropriately by assembling a compression plan that the selection module can use to look up how to compress specific objects.

As seen earlier (section 2.1), bandwidth is the number one limiting factor for bad connections. High packet round trip time and packet loss are mostly side effects of packet congestion, caused by trying to send more data than the bandwidth allows. Because of this, saving bandwidth is the goal of the C.A.N.P. framework, and packet loss the perfect indicator for dropping quality. The basis for this approach is taken from the various *congestion control* algorithms employed by the different versions of TCP, where packet loss is also the trigger for throughput throttles (see section 2.2.1).

The module makes its move every time new information about packet loss is received from the client. The update rate is variable but currently set at ten updates per second, so that the packet loss information reaches the server fast, even if some reports are lost during connection issues. On its arrival, if the packet loss percentage is above a tolerated level, the module

attempts to create a compression plan containing compression actions for each category of entity. For this, it uses the information about all in the last update to the client broadcasted entities gathered by the selection module. These units can be grouped into 16 categories resulting from each of the four categories for *distance to player* and *priority*, and as player movement and time since the last update and the next is very little, it can be assumed that the amount of entities in each category will remain mostly the same, and so each category can be compressed accordingly to reach the total compression quota needed to offset the packet loss percentage.

A compression plan is created by balancing the cost of a possible compression action against its benefit. The cost represents the effect the compression action will have on the player's gaming experience, e.g., compressing high priority units near the player will have a huge impact on the playability of the game, while the compression of low priority, far away units may barely be notable. Same goes for the compression action itself, as not sending a unit at all is obviously worse for the experience than lightly quantizing it.

Benefits are calculated in the form of bandwidth savings in relation to the total bandwidth. Reducing the update rate also reduces the bandwidth per second taken by all affected unity by the same amount, and quick tests have shown that every level of quantization reduces the bandwidth taken by an entity broadcast by roughly 5%. With this information, the plan is now created by gradually adding compression actions to it until the packet loss quota is reached, starting from the lowest cost action. After the plan has been completed, all entity categories that have been elected for reduced update rate are split amongst update ticks. Update ticks are counted in cycles of four ticks, of which half-updates are skipped on two non-consecutive ones, and quarter updates on three consecutive ones. By dividing all categories using these update rates amongst those four ticks, the module attempts to keep the data sent on every tick of roughly equal size.

After the compression plan has been finalized, the control module will wait until new packet loss information, that results entirely from entities affected by the plan, arrives, which is at least double the round trip time. If packet loss is still reported, the module will slowly continue adding and changing compression actions until the packet loss is finally below tolerated levels, if the packet loss has been resolved, it will begin to slowly remove compression actions until either packet loss is reported again or the plan was completely decomposed without encountering new packet loss, meaning the connection issues were resolved. In the case where the connection issues could not be resolved by reducing needed bandwidth, meaning the packet loss is of abnormal origin (like a cable failure) or the congestion occurs somewhere out of reach (like on a node outside the local ISP), compression is reduced as well, as there's no point to worsening the player's gaming experience further when it is not helping to increase playability.

Chapter 4

Evaluation

4.1 Isolated Look at the Compression Methods

While the compression methods seem to work as they should, their individual effectiveness differs a lot, so they are tested independently from each other to reveal the differences. Apart from the average achieved compression rate, performance is also an important part. The framework uses *reflection* to access fields of encoded objects automatically instead of requiring the user to create encode and decode methods for all messages he wants to send. This automation makes the networking much more comfortable and easier, but it should come at the price of decreased performance – reflection is, after all, several times slower than accessing the fields directly. With the massive amounts of objects an online game server has to process every second, the algorithm's run time is not insignificant, and thus, the impact of using reflection in it is tested as well.

4.1.1 Compression Rate

Probably the most important factor for evaluating the compression modules is the compression rate they can achieve. This holds true especially for the adaptive serializer and delta compression, as they attempt loss-less compression and are therefore bound to limitations, whereas with the help of the lossy compression applied through quantization and reduced update rates the size of an entity can be reduced a lot further. For testing all the modules' specific compression, entities of different sizes with a variety of different fields are encoded and the result compared to their raw size. These tests are also repeated at different levels of packet loss to check for potential influence and the modules' stability when exposed to a bad connection – a core requirement, given that the framework aims to be effective at exactly these types of connections.

Table 4.1: Details of the objects used for compression tests.

Entity	Total Fields	Floats	Ints	Bools	Enums	Initial Bits
Small	8	6	1	0	1	232
Medium	14	6	5	1	2	376
Big	21	9	8	2	2	576

Serializer

The initial effectiveness of the serializer mostly depends on how the encoding tables were set up. A properly, manually created table can already reduce the size of an encoded entity quite a bit, since it gives the user great control over the precision of the encoding of floating point decimals. Additionally, it allows integers to be marked for lossy compression via quantization, a feature that is disabled by default since they are often used for IDs and indices, meaning quantization would make the information transmitted by them completely worthless. More on this though later in the lossy compression evaluation.

The compression rate achieved by the serializer also differs a lot depending on how many fields encoded objects possess and how big their range of values is. Because of this, all tests will be done three times, using a different type of entity each time, ranging from a small one just carrying basic transform information as well as an ID and a type, to a big one with many additional information such as equipped weapon, ammunition, health, shield, energy, current target, active effects and their duration and some more, with values ranging from 10 to 10,000 depending on the field. The exact entities used can be viewed in table 4.1.

As explained in the project description chapter, the serializer slowly learns the bounds of all fields of encoded objects and therefore the bits needed to encode them. If the bits used have changed, the corresponding encoding table is synced to the client and can be used as soon as the client's confirmation is received. For this test, the bits needed to encode one of the three entities are checked every time a table has been acknowledged. Once no more changes occur, the serializer has reached its maximum compression rate, which can be calculated by comparing the current bits per entity to the initial amount.

The serializer reaches its full potential after the sixth iteration of the encoding tables, which is mostly because of the set rate at which it adapts to measured min and max values. The achieved compression rate ranges from 60.3% to 72.7%, almost a quarter of the original size, which is a very good result (see table 4.2). A higher amount of fields seems to result in a better compression rate, though it is also important how big the ranges of values stored in those fields are, and both the medium and big entity added several

Table 4.2: Test Results Compression Rate (Serializer): Total bits needed to encode a specific entity type at every iteration of its corresponding encoding table.

Entity	0.	1.	2.	3.	4.	5.	6.	7.	8.	CR
Small	232	173	137	116	104	96	92	92	92	60.3%
Medium	376	260	194	156	132	116	108	108	108	71.3%
Big	576	393	291	232	193	168	157	157	157	72.7%

fields with rather small value ranges, and especially integers which do not need additional bits for their decimals, thus achieving a better result. For the final bandwidth consumed by the sent packets, bigger entities also have the added benefit of having a bigger ratio of compressible information (its field values) to non-compressible one (the prefixed networking and encoding information), so using them will see a bigger relative reduction in total bandwidth used.

When exposed to packet loss, the serializer remains stable. Even if the synchronization of an encoding table iteration fails initially, being sent on a reliable channel means that it will be repeated until successful. The encoding manager can buffer up to 16 pending iterations, so the serializer can continue to improve the encoding tables even during the delayed acknowledgment. Even at a disastrous 50% packet loss, the only difference to a flawless connection is varying delay of new iterations being confirmed for use. With this, the adaptive serializer can be seen as a great success in regards to compression rate and reliability.

Delta Encoding

The delta encoder's compression rate depends on a lot more things than the serializer's. It aims to reduce the size of the transmitted values by only encoding the difference to a recent game state that both the server and the client have complete information about. The final compression rate thus depends on just how much it can shrink the values, which in turn depends on how old the game state they are encoded relative to is.

In the perfect scenario, the client successfully reconstructs every single game state, can confirm this to the server immediately and the connection has no round trip time, meaning the server receives the confirmation before it encodes the next state. With this, all states would always be encoded relative to the last one and the delta encoder would achieve its maximum compression rate. However, apart from localhost testing, this scenario will not happen very often. Instead, due to round trip time, the last by the client confirmed state that the server knows of will often lag one or several states behind, the client will not always be able to confirm the last state due

Table 4.3: Test Results Compression Rate (Delta Encoding): Total bits needed to encode a specific entity type at every iteration of its corresponding encoding table, starting from the highest possible CR without delta encoding.

Entity	0.	1.	2.	3.	4.	5.	6.	CR
Small	92	86	80	77	74	71	71	22.8%
Medium	108	102	98	95	92	89	89	17.6%
Big	157	145	133	127	121	115	115	26.5%

to packet loss, confirming every state generates traffic that, albeit small, might be the tipping point when dealing with bad connections, and finally, confirmations can get lost before reaching the server, again due to packet loss. Confirmations are not sent on a reliable channel, as detecting the lost packet and resending it would take way too long so by the time the ack finally reaches the server the confirmed state would already be too old to use.

In summary, delta encoding reacts badly to packet loss and high round trip times. Since the C.A.N.P. framework aims to supply a networking framework that remains stable and adapts to even the worst connection conditions, it is very much necessary to expose the delta encoder to these weaknesses to determine his ultimate usefulness. For the first test however, its theoretical potential is explored by fabricating the "perfect scenario", similar to how the adaptive serializer was tested. The test begins *after* the serializer has completed its learning process and reached the minimum non-delta encoded bit size per entity, then delta encoding will be activated so the serializer can adapt to the new, smaller delta values. The compression rate is then calculated by comparing the final bit size to the old non-delta bit size (see table 4.3).

With a range of 17.6% to 26.5% CR, delta encoding is not nearly as effective as the adaptive bit packing employed by the serializer, and that is just with optimal conditions. It is most effective on the big entity, which adds several high-range-value fields that only change slowly between states, so ideal conditions for delta encoding, while the more low-range-value fields of the medium sized entity can already be encoded using very few bits before delta encoding, meaning the algorithm cannot reduce it much further.

The more interesting test is however how these compression rates hold up under not-so-ideal situations. As soon as the delta values increase due to the problems explained above, the serializer is forced to use more bits per field to still guarantee a successful transmission of all entities, otherwise lag spikes will result in the most important bits to be cut off, making the values relatively useless. There is already some protection in place for this in the form of a few bits buffer per field, but as soon as the serializer needs to make use of this breathing room the bits used to encode the affected field is raised

Table 4.4: Test Results Compression Rate (Delta Encoding) under bad conditions: Total bits needed to encode a specific entity type at different levels of latency and packet loss.

Packet Loss	0ms	50ms	100ms	200ms	300ms	500ms
0%	71	73	75	75	77	79
5%	75	75	77	77	77	79
10%	75	77	77	77	79	79
20%	77	79	79	79	79	79
30%	79	79	79	79	79	79

Table 4.5: Test Results Compression Rate (Delta Encoding) under bad conditions: Additional compression rate relative to pre-delta encoding.

Packet Loss	0ms	50ms	100ms	200ms	300ms	500ms
0%	22.8%	20.7%	18.5%	18.5%	16.3%	14.1%
5%	18.5%	18.5%	16.3%	16.3%	16.3%	14.1%
10%	18.5%	16.3%	16.3%	16.3%	14.1%	14.1%
20%	16.3%	14.1%	14.1%	14.1%	14.1%	14.1%
30%	14.1%	14.1%	14.1%	14.1%	14.1%	14.1%

for all entities of this type as precaution. In this test, both packet loss and latency are now gradually increased using the small entity (see table 4.4).

These results translate to a compression rate relative to pre-delta encoding as seen in table 4.5.

As can be seen, already the slightest disturbance reduces the effect of delta encoding, and some of the values chosen for testing are not even uncommon – a 5% packet loss on a 50–100 ms latency is not a rare sight for Wi-Fi users, and that already reduces the additional compression rate gained by 28.5%. It is also striking that even under horrible conditions, the additional compression does not get reduced below 38.1%. This is due to the fact that only three of the entity's eight fields have constantly changing values (x-position, z-position and y-rotation), while the other five do not change, so the increased delta values do not affect their delta compression. With three of eight fields affected, so 37.5%, a 38.1% loss of additional compression rate makes perfect sense – these fields have apparently reverted completely back to pre-delta encoding sizes. This is a pretty bad result, since it means that if all of the transmitted fields had constantly changing values, delta encoding had actually zero effect when exposed to bad connection conditions. Since enduring those is a requirement for the framework, it can be concluded that delta encoding was a bad choice to include for an encoding method and should probably be excluded, since an *increasing* bandwidth is

Table 4.6: Test Results Compression Rate (Quantizing): Total bits needed to encode the test entities for all three levels of quantization.

Entity	none	level 1	level 2	level 3
Small	92	85	76	58
Medium	108	101	92	70
Big	157	147	132	101

Table 4.7: Test Results Compression Rate (Quantizing): Compression rates for all three levels of quantization.

Entity	none	level 1	level 2	level 3
Small	0%	7.6%	17.4%	37%
Medium	0%	6.5%	14.8%	35.2%
Big	0%	6.4%	15.9%	35.7%

very counterproductive when the connection quality suddenly drops.

Quantization

Quantization, together with variable update rates, marks the last resort of the compression package and is only used when the current connection quality is deemed too low for reliable transmission. While the "compression rate" of the variable update rates can be easily calculated since it is constant (half updates = 50% CR, quarter updates = 75% CR etc.), the effect of quantization as implemented in the framework is not that straightforward, since it depends on the amount, type and value ranges of an object's fields. On fields that contain big values on average, a larger amount of least important bits is cut than on fields with little average values. To get an idea of how much savings this nets in practice, the three levels of compression available to the control module are applied to the three entity types after the adaptive serializer has locked the field sizes (see table 4.6).

As the results show, objects with fewer but higher average value fields (like the small entity) achieve the highest compression rate (see table 4.7). With over a third of the bits cut from every field on average using level three quantization, the precision loss is bound to be clearly visible. The goal though is to take this loss in exchange for maintaining real-time game state transmission by avoiding congestion thanks to the reduced throughput. Whether this quality loss is worth it remains to be seen when testing the adaptiveness to bad connection conditions.

Table 4.8: Test Results Bit Shifting (Encoding): Run time in milliseconds per 100,000,000 ints processed.

Test Nr	BitConverter	Byte Wise	Bit Wise
1	1035	3144	38921
2	1054	3001	39057
3	1022	3122	40462
4	1087	3094	39434
5	1160	3143	41224

4.1.2 Performance Tests

For performance tests, the algorithms in question will process an object a certain amount of time, with the object containing 14 fields – six floats for position and rotation, six ints for unit stats, id and priority, and two enums for unity type and equipped weapon. The test results presented are done on an Intel machine (Core i7-4700MQ 2.4GHz, 8.00 GB RAM) with no other programs running.

Serializer

The serializer has two potential performance problems: It uses the earlier mentioned reflection to get and set an object's fields, and on top of that, it makes use of excessive bit shifting in its bit packer component. Compared to byte wise serialization, where all eight bits of a byte are shifted at once, the bit packer shifts them one after another, and additionally, it also needs to shift the target byte, inevitably decreasing performance. This performance difference is measured by encoding a 32 bit integer 100,000,000 times using both the bit- and the byte-wise variant, as well as the provided *BitConverter* class that can be used to convert any primitive data type to a byte-array and back. As the processing of a single integer is still quite fast, an excessive amount of repetitions like this is necessary to get normalized results. The test is repeated five times.

As can be seen in table 4.8, even at 100,000,000 repetitions, the run time still varies a lot. Still, the performance difference between the different approaches is easily visible. With an average of 1,071.6 ms, using the bit converter proved to be the fastest method by far. Switching to the byte-wise approach, the average run time of 3,100.8 ms means a run time increase of +189%, however, this method gains the benefit of being able to cut one or more bytes during the encoding to save bandwidth. With being able to cut single bits as well, the bit-wise approach can save even more bandwidth, at the cost of a +1,184% run time increase compared to byte-wise, and +3,616% compared to using the bit converter. As anticipated, with an av-

Table 4.9: Test Results Bit Shifting (Decoding): Run time in milliseconds per 100,000,000 ints processed.

Test Nr	BitConverter	Byte Wise	Bit Wise
1	692	3295	32859
2	774	3382	31994
3	750	3567	35973
4	747	3383	31348
5	762	3438	36883

Table 4.10: Test Results Reflection (Encoding): Run time in milliseconds per 10,000,000 objects processed.

Test Nr	Manual Encoding	Reflection
1	9322	40296
2	9523	40129
3	9810	40965
4	10096	44135
5	10028	42591

erage duration of 39,819.6 ms, the performance of the bit packer has proven to be significantly worse than the other two methods. It should be noted however, that there is still room for optimization: Changing the method from copying the data bit by bit to copying bit-chunks the size of free bits available in the target byte should improve the run time significantly.

For the decoding methods, the results look pretty similar (see table 4.9). Compared to encoding, the bit converter as well as the bit-wise approach are both faster, while the byte-wise approach slows down slightly - the reason for that is most likely a sub-optimally optimized algorithm.

The next potential weak point of the serializer, reflection, is tested by creating hard coded encoding and decoding methods and comparing them to the automated approach that gets a list of all fields and uses that list to access them via reflection.

As expected, processing objects via reflection results in a significantly higher run time (see table 4.10 and 4.11). With 41,623.2 ms for encoding and 46,661 ms for decoding (average per 10,000,000 objects processed) it is 327% resp. 362% slower than the 9,755.8 ms resp. 10,104.4 ms of the manual approach. Unsurprisingly, decoding takes slightly longer for both approaches than encoding, since decoding requires values to be written into the objects' fields while encoding only reads them, and is more apparent in the approach using reflection. Here, decoding takes 12% longer than encoding. That being said, the gained benefit from not having to write hard-coded encoding

Table 4.11: Test Results Reflection (Decoding): Run time in milliseconds per 10,000,000 objects processed

Test Nr	Manual Decoding	Reflection
1	9977	46199
2	9979	46674
3	9888	47333
4	10324	47900
5	10354	45199

Table 4.12: Test Results Delta Encoding: Run time in milliseconds per 10,000,000 objects processed.

Test Nr	Building Gamestate	Delta Encoding	Total Processing
1	21058	41259	62317
2	21147	43304	64451
3	21583	42408	63991
4	20896	43884	64780
5	21179	41606	62785

and decoding methods for every type of entity or message brings a lot of convenience, and also eliminates the need of having to maintain them in a mirrored fashion.

Delta Encoding

Delta encoding uses reflection even more than the serializer. Whenever an object is encoded or decoded, it is used to read its fields and store their values to build the game state. When delta encoding is applied to an object, its current field values are read and the delta values then written back, again via reflection. It is already known how much more expensive using reflection compared to manual encoding is, but it is still interesting to know the impact delta encoding has on the total run time of a complete object encoding.

As can be seen in table 4.12, with an average duration of 21,172.6 ms per 10,000,000 objects processed, just building the game state with the processed objects increases the total run time of the encoding or decoding by 45–51% of the serializer's run time, so even if no delta encoding is actually used, just having the option of enabling it at any time is costly, performance wise. Actually activating it increases its total run time to 136–15% of the serializer's, based on the 63,664,6 ms average.

As opposed to the serializer, both encoding and decoding result in the same run time, since the exact same operations are performed: The object's values are added to the game state, and if delta encoding is enabled, its

Table 4.13: Test Results Selection Module: Run time in milliseconds per 10,000,000 objects processed.

Test Nr	Selection Process
1	6915
2	7055
3	7087
4	7029
5	7207

values are read and processed, then written back one by one. The only thing different is the order in which they are applied – on encoding, the object is added to the game state before delta encoding, while on decoding, the object needs to be restored to its absolute state (via delta decoding) first. Because of this, the measured run times for encoding represent the ones for decoding as well.

Selection Module

Even though the selection module has a very short task – looking up how to compress an object in the encoding plan – one of the criteria used in the evaluation is its priority, the other the distance to the player, which requires checking an object's position, and both are retrieved via reflection. Even though the amount of calls is substantially lower than for serialization or delta encoding (three to four, depending on whether the game world is 2D or 3D), it still has an impact on performance that is now quickly measured (see table 4.13).

Compared to the other encoding modules, the average run time of the Selection Module turns out relatively small, with just 7,058.6 ms per 10,000,000 objects processed. Compared to Serialization and especially Delta Encoding, it is a rather small increase in the total run time of the encoding and decoding process, but significant enough to be considered. It should however be noted, that the run time ratios can differ wildly depending on how many fields the encoded objects possess. As the amount of reflection calls in the selection module remains a static three to four, the less fields an object has, the greater the Selection Module's run time's part of the total run time becomes.

Quantization

Quantization does not have any significant impact on performance as it is just a simple bit shift per field to cut off (or restore) one or several bits before serialization, which, for the 14 field object, took just a little over 300 ms per

10,000,000 objects processed on average. Naturally, this makes it a highly efficient way of reducing used bandwidth, albeit with the known limitation of being lossy.

Total Run Time

In total, the time it takes to process the specified 14 field object averages at 112,646.4 ms for encoding and 117,684.2 ms for decoding per 10,000,000 runs if all encoding methods are in use. Most of this time is split between delta encoding and serialization, with delta encoding taking up 54.1%–56.5%, serialization 39.6%–37%, the selection module 6%–6.3% and the quantization barely noticeable 0.3% of the total run time. Because of the disappointing test results of both run time and compression rate for delta encoding, its use for the main tests of the adaptive framework is deemed too little and it will thus be excluded.

In a more realistic scenario, where maybe 50 of these entities have to be broadcasted to 10 different players 20 times per second, the total of 10,000 encoding would result in a total of 112.6 ms required processing time just for the encoding, if the machine these tests were run on was used as a server. This is actually a quite significant amount, considering the server also needs to run the game logic. It can be concluded that performance wise, the framework needs more optimization, should it be intended to be used on mid-end computers.

4.2 Testing the Framework

4.2.1 Test Arrangement

To evaluate the overall success of the project, a few tests are necessary. The general procedure of the tests will be to run a deterministic scenario multiple times, first using the framework without and then with adaptiveness enabled, recording the results and then comparing the data. Most of the tests will be of synthetic nature, with simulating the connection issues on a virtual network in a controlled manner.

The Scenario

For the deterministic scenario, a simple server/client setup is created using Unity3D (see figure 4.1).

On the server, several game objects roam a plane in a very simple manner, they run in big circles at fixed speeds. These entities are split into four types, each being assigned one of the four priority levels, using different colors and simple models to distinguish them easily. A total of 98 units is used, eight of the highest priority (level zero) representing other players and 30

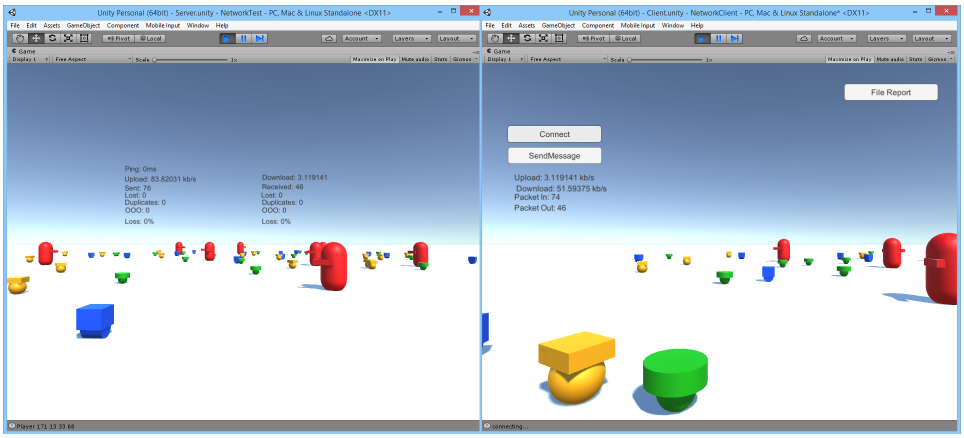


Figure 4.1: Test setup using Unity 3D. Two separate instances run server and client.

each for the other three priorities. Initially, entities are placed around the center of the plane randomly but will keep this starting position throughout all tests. The transform of these entities (position and rotation) is transmitted to all clients at a rate of 30 updates per second on an unreliable channel using the automated object encoding of the encoding manager. When a new connection is coming in, the server will first send the game state on a reliable channel as a spawn command before starting with the updates.

The client is set up to be initially empty and must initiate the connection to the server. Once the connection is established, it spawns and places the entities as instructed on the plane and will then start to update their transform using the incoming game state updates.

Both server and client receive a HUD with information on the current network stats (latency, incoming/outgoing bandwidth, packets sent/ received/ lost etc.).

Test Data Capturing

Game states are written to log files on both server and client for later data extraction. The server writes the complete game state containing position and rotation of all active entities every time it is transmitted to the client, while the client writes all received entity information immediately to its log file on arrival. To keep the log files synchronized, every state receives a time stamp. The time for log files is tracked separately from the game time and starts with the spawn command, for the server when it is sent and for the client when it is received, so normal latency is compensated for.

For easier visual comparison and also to find points of interest in the log files easier, a replay system is created that can reproduce the test results in

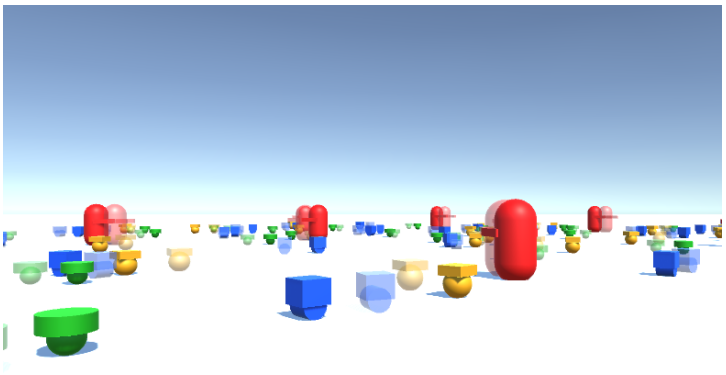


Figure 4.2: With the replay system, server and client state can be overlayed for better visual comparison.

visual form. The replay system is able to play log files in real time, pause anywhere, jump to specific time stamps in the test or fast forward/rewind. Additionally, the system is able to play the two log files at the same time, overlaying the replays. The entities of one of them are shown with a transparent material. With this it should be very easy to spot discrepancies between server and client states and gather data (see figure 4.2).

Evaluation Criteria

While visual comparison has its place, it is important to have statistical evidence as to which approach is superior. The most important criterion will be how much the client's game state differs from the server one (at synchronized times), which can be compared by pairing corresponding entities from server and client state and taking the absolute differences between their transforms. This comparison can be further restricted to entities that are of higher interest to the player, like the ones closer to him, the ones with higher priority, and especially the ones representing other players.

4.2.2 Synthetic Tests

The synthetic tests are executed in a virtual network using a loopback-adapter. Since no bandwidth limiter working with packets sent over localhost could be found, the *UdpClient* class is extended with a custom one. The limiter tracks incoming or outgoing bandwidth at 30 samples per second and blocks any traffic that would exceed a certain limit by first putting the packets into a short queue to simulate the queues at any network node, and discarding incoming packets completely once the queue is full. Whether this simulation is accurate to real networks will be shown later in the organic tests.

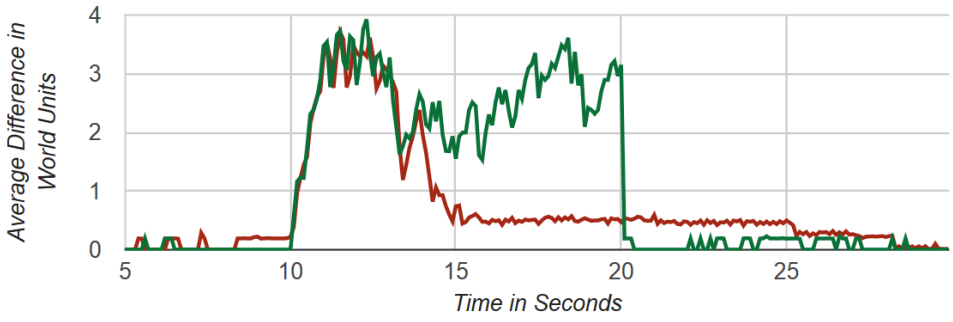


Figure 4.3: First Test: Average position differences between server and client, comparison between the adaptive (red) and the non-adaptive (green) approach.

Preparation

To set up appropriate bandwidth limits, the simulation is run once to quantify the base bandwidth needed to sustain it. It initially starts at 219 kb/s, and, after the adaptive serializer has fixed the field sizes, shrinks down to 74 kb/s on average.

To make the test runs perfectly deterministic, the simulation of the connection issue is automatized. The bandwidth limiter will be applied at exactly 10s of test begin, and removed again after it has been in place for 10s. Entities will first start moving when then connection is established.

First Run

For the first test, the bandwidth is limited to 40 kb/s, so a little over half of what is needed for a lossless transmission. For the comparison, the average position difference between server and client state for every entity is compared over a time span of 25 seconds, starting 5 seconds before and ending 10 seconds after the bandwidth limitation (see figure 4.3).

As can be seen in the line chart, the adaptive approach works: Within three seconds, lossy compression is engaged, causing the congestion to resolve. After five seconds, packet loss has completely disappeared, the slightly elevated difference of 0.5 world units on average remains due to the quantization and reduced update rates. For the rest of the five seconds connection issues, the game would be completely playable, whereas the non-adaptive approach continues in a highly inaccurate state with on average 2–3 world units difference, so 400–600% worse than the adaptive approach. Its only advantage is that as soon as the congestion resolves at the 20 second mark, all differences vanish immediately and the simulation resumes normally, whereas the adaptive approach does not lower its compression until five seconds later, and needs another three seconds to completely return to loss-less compres-

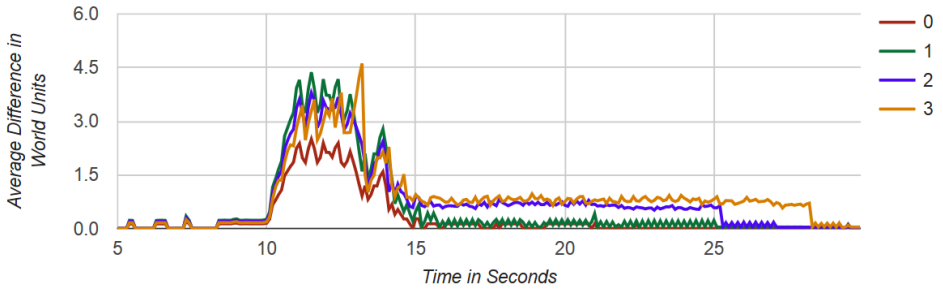


Figure 4.4: Average position differences between server and client with entities split up according to their priority.

sion.

The accuracy measured above is the average for all entities, but since the adaptive system compresses less important and further away entities first, a closer look at how the lossy compression is distributed over different entity categories is taken. Ideally, highest priority entities (representing other players) should be still nearly identical to the server game state even during enabled lossy compression.

Looking at the test result data, it seems this goal has been reached (see figure 4.4). After the initial five seconds needed to resolve the congestion, both priority zero and priority one units show much less difference between server and client state than the other two priorities. As they are not affected by lossy compression, the priority zero units also recover completely as soon as the connection issues resolve at 20 seconds, with priority one taking five seconds longer, priority two three additional seconds and the remaining entities the full 30 second test.

An interesting outlier can be seen during the three seconds of unhandled congestion: Priority zero units, the players, suffer distinctly less from the connection issues than the other entities. This can be explained with these entities being transmitted first, while the rest is sent afterwards in no preferred order, so the first arriving packets containing priority zero entities have a higher chance to make it into the packet queue, which might have processed some packets since the last update and thus has new room for a few extra ones.

Splitting the entities by their distance to the player gives a less distinct result, with very close units even being less accurate on average than more far away entities at one point during the recovery phase (see fig 4.5). It seems that an entity's priority is more influential than its distance to the player. But while less distinct, the influence of distance is still clearly visible. Closer entities are more likely to deliver more accurate results, and also return to their loss-less transmission faster.

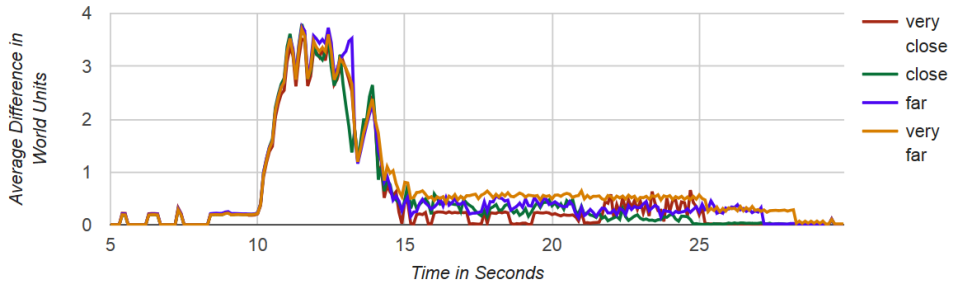


Figure 4.5: Average position differences between server and client with entities split up according to their distance to the player.

Value Tweaking and Improvements

After evaluating the first test, the adaptive system looks promising. There are however two areas that could see improvement: The *slow recovery phase*, which leaves the player with inaccurate, lossy transmission for a long while even after the connection issues have resolved, and the *slow attack phase*, which lets the player feel all the repercussions of a congested connection for several seconds before the adaptive system engages. An attempt to fix or at least contain these issues will now follow.

The recovery phase can be sped up by either increasing the amount that the lossy compression is reduced by every recovery attempt, and by reducing the compression more often. The current system keeps track on how many recovery steps were successful in a row (meaning no packet loss occurred despite lowered compression) and gradually increases the compression reduction steps the longer the series gets.

While increasing these values may lead to a faster recovery phase after connection issues have resolved, doing so should be handled with care. If connection issues remain for a longer period of timer, the game experience may suffer, as every time the recovery fails and the temporary bandwidth threshold is surpassed, congestion will occur for a moment before the adaptive system realizes its error and increases compression again. Whether this will be problematic will be shown by speeding up the recovery phase in the next test.

Shortening the attack phase is a little less straight forward. In its current iteration, the control module attempts to adjust the outgoing bandwidth to the client's available one by setting the compression rate equal to the detected packet loss, then waiting a bit on new packet loss data, increasing the compression even more if packet loss is still reported. This waiting period is the reason why finding the right compression amount takes several seconds. When the congestion begins, the very first packet loss reported will be much

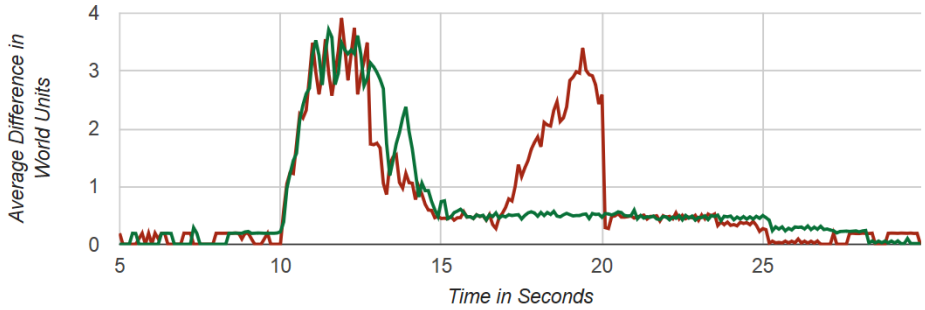


Figure 4.6: Second Test: Average position differences between server and client, comparison between the previous (green) and the new (red) adaptive approach.

lower than the final value, reason being that all network stats are always reported as the average in the last second of traffic. It takes roughly one second plus the current round trip time for the final packet loss value to reach the server, and even if the congestion has resolved, it takes the same time until no packet loss is reported anymore. This makes one second waiting periods essential, since that's the earliest time at which the packet loss is guaranteed to be the sole result of an underestimating compression rate. Reducing the waiting time between attacks is thus not enough, it would result in overshooting the compression rate almost guaranteed. Instead, the following modifications are tried:

- Packet loss is reported for only half the last second instead of the whole.
- The control module will wait a full half second for the final packet loss data before its first attack.
- The attack delay is reduced to half a second plus current rtt, matching the new packet loss measurement.

After this and the changes to the recovery phase, the test is repeated.

Second Run

Looking at the result, the extent of just how problematic hitting the temporary bandwidth threshold with a failed recovery really is, becomes apparent: Another congestion occurs and the position differences revert back almost to the worst case scenario (see figure 4.6). With this, the recovery speed used for this test has proven to be way too fast, either it needs to be reduced again to old values or another solution must be implemented.

On the other hand, the sped up attack phase seems to work. Good results start coming in slightly earlier, and side effects seem to stay away as well.

Given though how much the delivery of new stats and the attack phase were sped up, the results look slightly disappointing. At any rate, the new settings for the attack phase can be kept.

Additional Improvements

The last problem that remains to get solved therefore is the recovery phase. Slowing it down seems acceptable, since even though this means the lossy compression will remain active for longer after the connection issues have already resolved, as the tests have shown, even with lossy compression the client state does not sheer off nearly as much as during congestion. However, even with slowed down recovery phase, hitting the bandwidth threshold would hurt, as while the congestion is building up, compression is still being reduced. With the delay in getting the packet loss information to the server, the recovery phase stays active way too long, resulting in a congestion much worse than necessary.

To solve this issue, the recovery phase is redesigned to work more carefully. Now, whenever the compression is reduced, the reduction is only applied for a short moment and then reverted back. The control module then waits a bit for the following network stats to check whether the short term reduction caused any packet loss, before finalizing the reduction and trying an even lower level. This should stop congestions from going out of control and works around the latency issue. To guarantee the recovery to fail at least once so the consequences can be measured, the bandwidth limitation phase is increased to 20 seconds this time.

Third Run

The test results are rather disappointing. Even though the previous peak of inaccuracy caused by congestions is not reached this time, the system fails to recognize when a compression reduction first causes an exceedance of the bandwidth limit, visible at the disregard of the first, small rise in client inaccuracy at roughly 20 seconds (see figure 4.7). It is assumed that the fault for this lies within the packet queue built into the bandwidth limiter: Because packets exceeding the limit are first queued, as the compression reduction is reverted quickly after its introduction, the exceeding packets will not cause the queue to overflow, meaning no packets are actually lost. The inaccuracy is then caused by the increasing latency. This also may be the way to solve this problem successfully: For the fourth and final test, latency is checked whenever a compression reduction is reverted. Should a considerable rise be detected, it can be assumed that nodal queue delay was at fault and a congestion is imminent, which means that the recovery has failed.

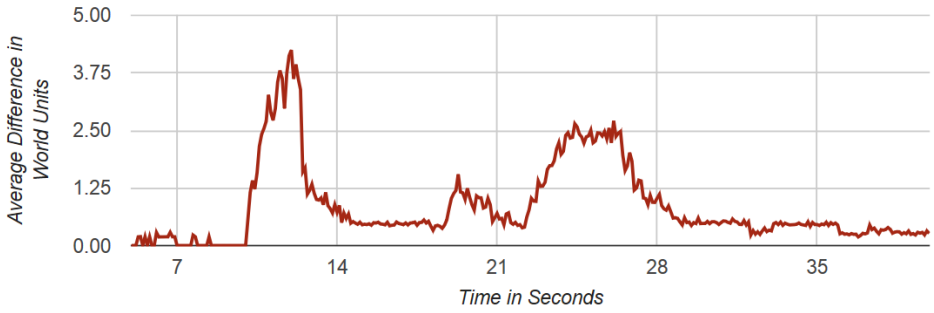


Figure 4.7: Third Test: Average position differences between server and client, attempt to contain a congestion during the recovery phase.

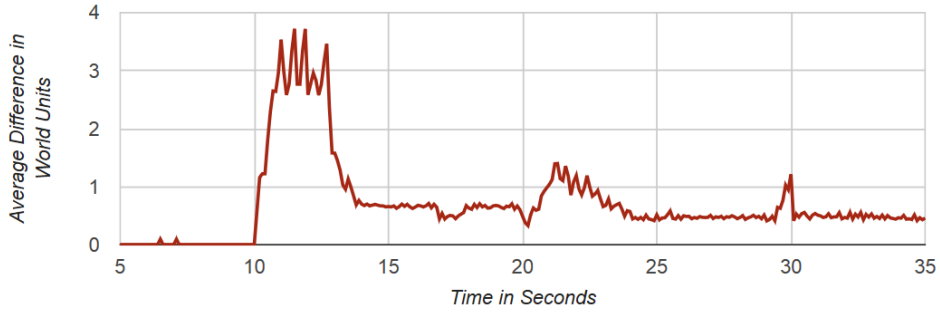


Figure 4.8: Fourth Test: Average position differences between server and client, attempt to contain a congestion during the recovery phase, this time considering latency.

Fourth Run

With considering the latency for rating the success of a recovery attempt, the results look a lot better (see figure 4.8). Failed attempts seem to be recognized immediately, and the disturbance in the client state, while still considerable, is at nowhere the level it was before. With these improvements, the framework will finally be tested on a physical network.

4.2.3 Organic Test

The test is prepared by starting server and client on two separate PCs with internet connection and around 100 km distance between them. The server is connected to the internet via LAN cable, while the client uses a Wi-Fi connection with acceptable signal strength. The server has access to 100 mb/s of upload and download so the connection issues are caused purely

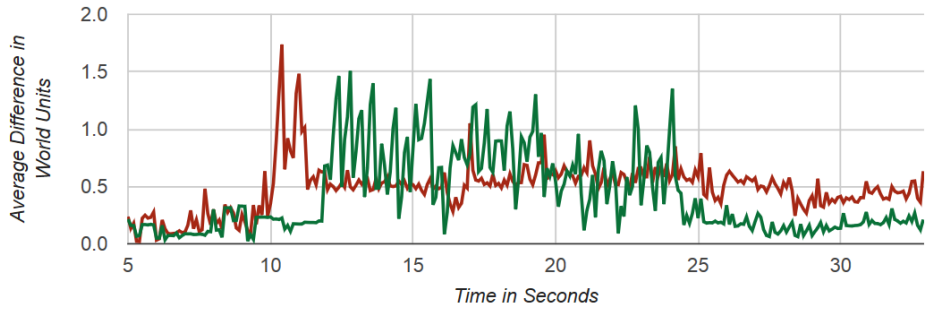


Figure 4.9: Organic Test: Average position differences between server and client, comparison between the adaptive (red) and the non-adaptive (green) approach. Begin and end of connection issues is not completely synchronized in this test.

clientside. After a few difficulties, the connection can be established and the simulation is executed. Since it now runs over a real internet connection, the connection quality can be worsened clientside by clogging it with other applications that leave over less bandwidth than needed for a loss-less transmission. The connection is being clogged for roughly 10 seconds, after that the full bandwidth is immediately restored. This is repeated two times, once with disabled adaptiveness, and once with enabled one, using the modifications and improvements gained from the synthetic tests.

The results are less impressive than the ones of the synthetic tests. The congestion caused by clogging the client's connection has a less severe impact on the precision loss of the game state transmission than before, while the average distance of the adaptive approach has not really improved, it still sits a little over 0.5 world units. The benefit of using the adaptive approach over the non-adaptive one is thus greatly reduced. The one thing it still delivers though is consistency, the average distance stays more or less the same during the test while for the non-adaptive approach it varies greatly. The reason for this can be seen when looking at the connection stats during the test: Compared to the synthetic tests, latency fluctuates a lot more during a congestion, causing entities to stop moving for a split second when a batch of packets has been lost and then snapping back to their actual position when another batch makes it through.

Another downgrade becomes apparent when looking at how long the recovery phase needs to completely restore loss-less compression after the connection issue has resolved. Additional issues that were absent in the synthetic test, like jitter, latency spikes and background packet loss, make it a lot harder for the control module to decide whether the new connection stats point towards a congestion or not. Not a single increase in inaccuracy during

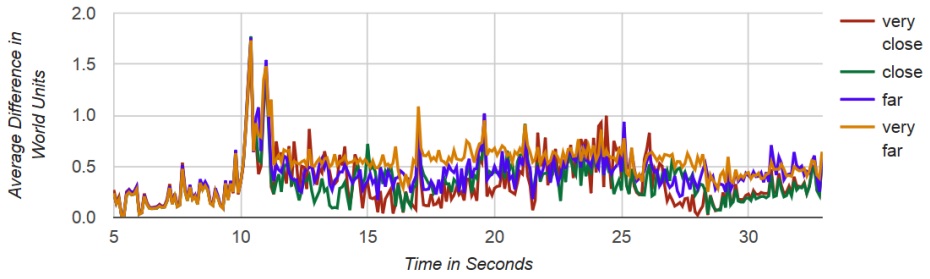


Figure 4.10: Organic test: Average position differences between server and client with entities split up according to their distance to the player.

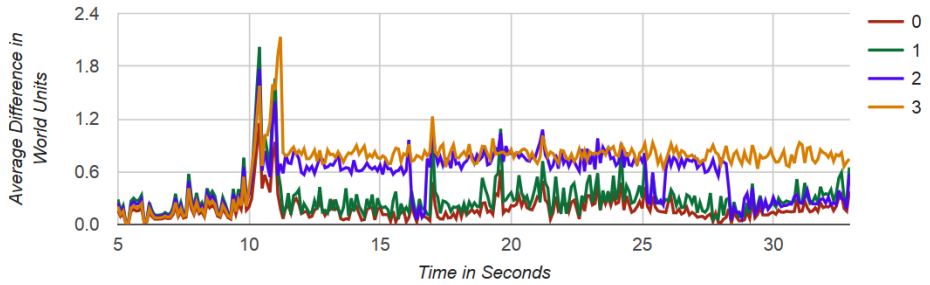


Figure 4.11: Organic test: Average position differences between server and client with entities split up according to their priority.

the lossy phase can be seen, which points towards all recovery attempts being classed as failures despite not actually causing congestion, meaning the compression rate used was higher than needed, and the adaptive approach remained below its potential.

A quick look is also taken at the effect of entity distance to player (see figure 4.10) and priority (see figure 4.11). As can be seen, the entity priority still has a major impact on the accuracy of the transmission. Both priority zero and priority one entities achieve much better results than two and three, just like with the synthetic test before. But while the synthetic test achieved almost full precision for higher priority units (see figure 4.4), for the organic test, the system fails to do so due to the additional connection issues introduced. The decline in precision is also clearly visible when splitting the entities up by distance to the player. While, similarly to the synthetic test (see figure 4.5), the difference in precision is much less distinct than it is for priority, entities closer to the player still fare better on average. Again though, the benefit of being very close to the player is a lot less than it was for the synthetic test, the almost loss-less transmission that was achieved

there has worsened significantly.

With the more detailed view on entity precision the problems of the framework have become even more apparent. Especially the control module's failure to reduce compression when the connection could clearly take it but was falsely interpreted becomes very apparent when looking at the precision chart split up by priority (see figure 4.11): The precision of priority two entities periodically improves severely, indicating an essentially successful (the client showing only improvements means the bandwidth is still sufficient) compression reduction attempt, but snaps back to old values without applying the changes, suggesting the failure of the attempt.

Chapter 5

Conclusion

Rating the success of the project is a difficult task, especially when so many components factor in. When looking at it as a learning experience, then yes, it was definitely successful. The gained information on how efficient the popular network compression techniques are and how they are holding up when exposed to suboptimal connection conditions, together with the general knowledge about networking in online games acquired through implementing and testing the project will be immensely helpful when working on online games in the future. When looking at the different components of the framework, the results are mixed, with some working reliably and delivering good results, while others have shown questionable performance. And finally, looking at the functionality of the framework as a whole, while, after some improvements to the project during the test, it definitely showed potential, the final organic test revealed that a lot more work has to be done before it could be used effectively in a real online game.

The part that ultimately had the most success was the automation of entity encoding. Using reflection proved to be expensive, yet still fast enough to warrant the gained convenience. The dynamic encoding tables that allow for the huge compression benefit of bit packing without the developer having to figure out and keeping track of the value ranges for all the fields of his entities was working exceptionally and remained completely stable during connection issues. The bit packing itself turned out to be a little taxing on performance due to excessive bit shifting, but the additional compression rate enabled through it made it worth it.

Using lossy compression in the form of quantization and reduced update rates to match a target bandwidth under the assumption that a controlled loss of precision is preferred to uncontrolled congestion was mostly successful as well, especially when it was used selectively. Compressing less important objects first and stronger than more important ones successfully allowed them to retain almost their full precision during the times of limited bandwidth.

The least successful part, especially during the organic test, turned out to be the detection of connection issues. Interpreting the connection stats correctly worked, after some improvements, more or less reliably during the synthetic tests, but when additional, random connection issues like jitter, latency spikes and background packet loss were introduced with a real internet connection, the framework's control module had obvious difficulties making the correct decisions. For the whole adaptive functionality to be ready to use in a real online game, a more sophisticated congestion detection algorithm has to be developed first.

Also very disappointing was the the performance of delta encoding. While delivering a decent amount of additional, loss-less compression under good connection conditions, most of this was lost during bad ones. Coupled with the highest run time by far out of all the encoding methods, it ultimately was classed as unsuited for use in the framework.

If nothing else, the project has shown that automating big parts of the networking process in online games can work. Even if it ultimately failed to accomplish all the planned parts of the automation perfectly, the potential of the framework could be seen during the synthetic tests, and with more work being put into connection surveillance and congestion detection, it might find use in a real online game some day.

Appendix A

Inhalt der CD-ROM

Format: CD-ROM, Single Layer, ISO9660-Format

A.1 PDF-Dateien

Pfad: /

Herrmann_Manuel_2016.pdf Masterarbeit

A.2 Projekt

Pfad: /project

*.zip . . . . . Komprimiertes Projekt, zwei Versionen für Server und Client

A.3 Websites

Pfad: /websites

*.html . . . . . Für Onlinequellen genutzte Websites

A.4 Sonstiges

Pfad: /images

*.pdf . . . . . Original Vektorgrafiken

*.jpg, *.png . . . . . Original Rasterbilder

References

Literature

[1] Ashwin R Bharambe, Venkata N Padmanabhan, and Srinivasan Seshan. "Supporting Spectators in Online Multiplayer Games". In: San Diego, CA: SIGCOMM HotNets-III, 2004 (cit. on p. 7).

[2] Lawrence S. Brakmo, Sean W. O'Malley, and Larry L. Peterson. "TCP Vegas: New Techniques for Congestion Detection and Avoidance". *SIGCOMM Computer Communication Review* 24.4 (Oct. 1994), pp. 24–35. URL: http://doi.acm.org/10.1145/190809.190317 (cit. on p. 6).

[3] Carl Gutwin et al. "Improving Network Efficiency in Real-time Groupware with General Message Compression". In: *Proceedings of the 2006 20th Anniversary Conference on Computer Supported Cooperative Work*. CSCW '06. Banff, Alberta, Canada: ACM, 2006, pp. 119–128. URL: http://doi.acm.org/10.1145/1180875.1180894 (cit. on p. 6).

[4] Ramaswamy Ramaswamy, Ning Weng, and Tilman Wolf. "Characterizing network processing delay". In: *Proceedings of the IEEE Global Communications Conference (GLOBECOM*. 2004, pp. 1629–1634 (cit. on p. 3).

[5] Scott Shenker, Lixia Zhang, and David D. Clark. "Some Observations on the Dynamics of a Congestion Control Algorithm". *SIGCOMM Computer Communication Review* 20.5 (Oct. 1990), pp. 30–39. URL: http://doi.acm.org/10.1145/381906.381931 (cit. on p. 5).

Online sources

[6] Inc. Data Expedition. *Loss, Latency, and Speed*. URL: http://www.dataexpedition.com/support/notes/tn0021.html (cit. on pp. 3, 4).

[7] Fleshgrinder. *(Nodal) Processing Delay*. URL: http://networkengineering.stackexchange.com/a/12558 (cit. on p. 3).

[8] *Last mile*. 2016. URL: https://en.wikipedia.org/wiki/Last_mile (visited on 11/28/2016) (cit. on p. 3).

[9] *Networked Physics - Snapshot Compression*. 2015. URL: http://gafferongames.com/networked-physics/snapshot-compression/ (visited on 11/28/2016) (cit. on p. 8).

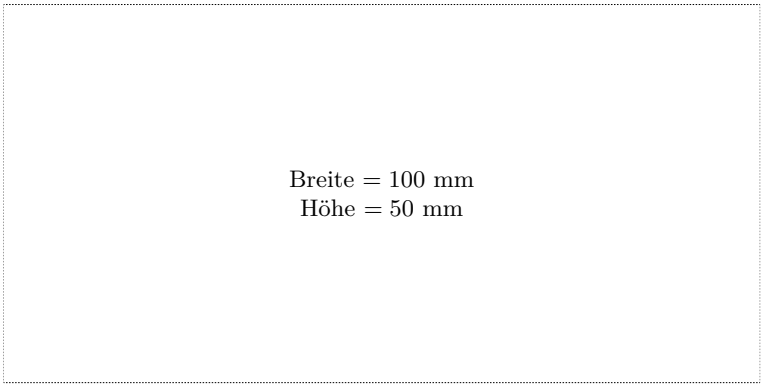
[10] *Quality of service*. URL: https://en.wikipedia.org/wiki/Quality_of_service (cit. on p. 5).

[11] Gary Shute. *Network Delays and Losses*. 2016. URL: http://www.d.umn.edu/~gshute/net/delays-losses.xhtml (cit. on p. 3).

[12] *What is internet traffic management?* URL: http://consumers.ofcom.org.uk/internet/internet-traffic-management/ (cit. on p. 5).

Messbox zur Druckkontrolle

— Druckgröße kontrollieren! —



— Diese Seite nach dem Druck entfernen! —