Information is the resolution of uncertainty.

– Claude Shannon.

University of Alberta

Dynamically learning efficient server/client network protocols for networked simulations

by

Sterling Orsten

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

Examining Committee

Abstract

With the rise of services like Steam and Xbox Live, multiplayer support has become essential to the success of many commercial video games. Explicit, server-client synchronisation models are bandwidth intensive and error prone to implement, while implicit, peer-to-peer synchronisation models are brittle, inflexible, and vulnerable to cheating. We present a generalised server-client network synchronisation model targeted at complex games, such as real time strategy games, that previously have only been feasible via peer-to-peer techniques. We use prediction, learning, and entropy coding techniques to learn a bandwidth-efficient incremental game state representation while guaranteeing both correctness of synchronised data and robustness in the face of unreliable network behavior. The resulting algorithms are efficient enough to synchronise the state of real time strategy games such as Blizzard's Starcraft (which can involve hundreds of in-game characters) using less than three kilobytes per second of bandwidth.

Acknowledgements

I would like to thank my supervisor, Michael Buro, for helping me to have a truly rich set of experiences throughout my academic career. From my first experimental undergraduate course to my most recent term as a teaching assistant, not only did you open the doors for me to work on interesting and practical projects, where I met numerous talented colleagues in the process, but you gave me the opportunity to directly contribute to the futures of dozens of undergrads, and to help develop a curriculum that will continue to influence countless more. More than just a supervisor, you've been a great friend to me.

I would additionally like to thank my graduate student advisor, Edith Drummond, for going above and beyond the call of duty on numerous occasions to ensure that my tenure as a graduate student was successful.

I would also like to thank my undergraduate students from both of my teaching assistantships. Your passion and enthusiasm for learning new concepts and mastering skills surprised and delighted me, and helped me discover a new side of myself through the joys of teaching. I wish you all bright futures, and hope to cross paths with many of you again in our future endeavours.

Also deserving of thanks are the many friends who sat through rambling explanations of my work and offered helpful insights into how I should develop it. You've been a tremendous help to me. I would like to thank Chris Friesen specifically for offering constructive criticism, and introducing me to the field of arithmetic coding, which became one of the pillars upon which my work is based. Without you, my thesis would have taken a very different form, and may even have failed to materialise at all.

I would further like to express my gratitude to my parents, whose encouragement and support gave me the resolve to see my graduate studies through to completion. Isaac Newton once said "If I have seen further, it is only by standing on the shoulders of giants". You two have always been my giants, and I appreciate the many opportunities you have given me. I would not have made it this far without you.

Finally, I would like to thank my little sister Katie. The fervor and dedication you've shown to your own discipline has proven to me that you will do great things with your life, and it sets my heart at ease to know that you are pursuing your dreams. You make me very proud, and inspire me to strive onwards to set a good example.

Table of Contents

List of Tables

List of Figures

List of Symbols

$P(E)$	The probability of event E occurring
X	A random variable
w_i	An outcome of a random variable
$E(X)$	The expected value of random variable X
Chapter 3	
$I(X)$	The information content of random variable X
$H(X)$	The Shannon entropy of random variable X
C	An arithmetic coder
M	The set of all possible messages
m	A specific message within M
$[a, b)$	A half-open interval, the set $\{x \in R : a \leq x < b\}$
$c(m)$	The cost to encode message m with arithmetic coder C
Chapter 5	
t_i	A point in time, possibly corresponding to a frame, tick, or snapshot
$f(t)$	The value of a particular field, as a function of time
$g(t)$	A polynomial function which predicts the value of a particular field, an approximation of $f(t)$
a_i	A coefficient of the polynomial function $g(t)$
T	A matrix based on the time values of sample points
A	A column vector consisting of the coefficients $\{a_i\}$
F	A column vector consisting of sample points from $f(t)$
U	A row vector based on the time value where we are making a prediction
P	A row vector used to make a prediction at a specific time value
N	A row vector consisting of the integer numerators of P
d	A scalar integer denominator of P
U'	The derivative of U with respect to t
P'	The derivative of P with respect to t
s_i	Shifted points in time, defined as $t_i + k$
$h(t)$	A polynomial function on s, whic predicts the value of $f(t)$
b_i	A coefficient of the polynomial function $h(t)$
S	A matrix similar to T, but based on s_i instead of t_i
B	A column vector similar to A, consisting of the coefficients $\{b_i\}$
V	A row vector similar to U, but based on s instead of t
Chapter 6	
δ	The difference between the actual and predicted values of a field
$f_X(x)$	The probability mass function of random variable X
$F_X(x)$	The cumulative distribution function of random variable X
x	An integer value
i	The index of the bucket containing x
j	x's offset within the bucket i
a_i	Boundaries of bucket intervals
I_i	The interval of bucket i, defined as $[a_i, a_{i+1})$
μ	The mean of a distribution
σ	The standard deviation of a distribution
λ	The parameter for an exponential distribution
γ	The decay factor for a recency weighted average

Chapter 1

Introduction

In the early 1990s, it became feasible to play multiplayer video games over the internet for the first time. Multiplayer capabilities quickly became one of the pillars of PC gaming, ranging from popular first person shooters such as Quake III, Unreal Tournament, and Counterstrike, to real time strategy games such as Warcraft, Command and Conquer, Age of Empires, and Starcraft. Throughout the first decade of the 2000s, multiplayer games only continued to grow in importance, as consoles such as the Xbox and Playstation 2, and their successors, the Xbox 360 and Playstation 3, launched their own online services, while integrated platforms such as Steam became a central component of online gaming on PCs. In modern games, good multiplayer offerings are essential to boosting the longevity of a game, and can make the difference between receiving average and excellent scores from reviewers. Some of the most financially successful franchises in all of modern gaming, such as Activision's Call of Duty, are essentially built on the strength of their multiplayer support.

Despite this, networked multiplayer remains an intrinsically challenging feature for game developers to support. Creating the illusion that players are participating in the same shared experience, when they are in fact playing on independent machines whose communications experience nontrivial latencies, requires careful design and robust testing. Fundamentally, game developers are trying to keep a large quantity of data synchronised between two or more machines, in spite of both limits to the amount of information that can be transmitted between those machines in a certain time frame (throughput) and an intrinsic delay between information being sent by one machine and received by another (latency).

Synchronisation strategies can be broadly classified into two categories. Explicit strategies revolve around transmitting representations of data from an authoritative peer (such as a server) to a non-authoritative peer (such as a client). Implicit strategies revolve around transmitting representations of events, from the originator to their peers, while peers perform identical simulations in parallel with one another. Explicit strategies afford the developer more control over what is visible to who, and are less prone to diverging, but require comparatively more bandwidth and more programming effort. Implicit strategies require less bandwidth, but rely on calculations working identically across multiple machines with varying architectures, clock speeds, and amounts of available memory, and afford less control over what information is present on a given machine at a given time, rendering these strategies vulnerable to information-revealing hacks.

In order to build a robust network protocol, developers need to balance issues such as available bandwidth, CPU requirements, response times, and resistance to hacking and cheating, all on top of the significant challenges inherent in designing and building an enjoyable game.

In this thesis, we present a networking library designed to simplify many of these issues, by adopting a flexible, data-driven, explicit synchronisation model, and using prediction, learning, and entropy coding techniques to drastically reduce bandwidth requirements. Ideally, the user need simply specify what server-side information he or she wants made available to each client, and the software library will determine what needs to be transmitted, and when, in order to keep the clients synchronised with the server.

The core principle behind the library is that archived snapshots of the state of the game over the past several frames can be used to make deterministic short-term predictions on both server and client about what will occur next, and network traffic will only be generated when the true state on the server differs from the predicted state. Furthermore, these differences are encoded in a way that achieves very strong compression by learning from the statistical properties of the game state itself, at runtime, requiring no input from the game developers. The result is a networking library that makes it very easy to synchronise large amounts of game state using minimal network traffic.

This thesis is organised into the following chapters.

Chapter 2 summarises the most popular publically known synchronisation techniques used in commercial multiplayer videogames over the past two decades.

Chapter 3 summarises some of the fundamental algorithms in the field of data compression, and their theoretical properties.

Chapter 4 presents an overview of the structure of our networking library, and describes the basic flow of information from server to client, through the library.

Chapter 5 presents the field prediction techniques we developed to make our short term predictions about the game state, and shows how they can be implemented very cheaply by factoring out common calculations and precomputing them.

Chapter 6 presents the compression methodology we developed to encode the differences between our actual and predicted game state, and shows how we can cheaply approximate the statistical distributions required to get strong compression out of entropy coding.

Chapter 7 describes the experiments we used to evaluate the performance of our library. It includes a simulation of how Blizzard Entertainment's Starcraft, a complex real time strategy game that makes use of extensive implicit synchronisation techniques, would perform using explicit synchronisation via our library instead.

Chapter 8 highlights the advantages of the techniques we have presented, and finishes by suggesting areas for future research for those interested in explicit synchronisation techniques.

Appendix A contains more specific information about the architecture of our library, and information useful to understanding how it would be used in a real program.

Appendix B contains several tables describing bucketing schemes used by the compression techniques shown in Chapter 6. They may be of interest to anyone seeking to implement similar techniques.

Chapter 2

Related Work in Game Networking

2.1 Internet Protocols

The two most common protocols used over the internet are the Transmission Control Protocol (TCP/IP) and User Datagram Protocol (UDP/IP).

TCP offers a stream oriented protocol. Two peers first form a connection with a three-way handshake (one peer sends a connection request, the other peer seconds an acknowledgement of the connection request, and the first peer acknowledges that acknowledgement). Once the connection is formed, either peer can write bytes into a two-way stream. TCP guarantees that these bytes can be read out by the opposite peer in the exact order that they were written in (although it makes no guarantees that they will arrive all at once). If this guarantee cannot be provided, the connection will be broken. TCP transparently handles resends, packet fragmenting and reconstruction, and check-summing. This allows for large messages, such as files or web pages, to be transmitted easily. The Hyper-Text Transfer Protocol (HTTP), File Transfer Protocol (FTP), Simple Mail Transfer Protocol (SMTP), and Post Office Protocol (POP) all operate over TCP.

UDP offers a connectionless, datagram-oriented protocol. Messages are sent out as individual datagrams, with a source and destination address. UDP provides no guarantees that messages will arrive, or that, if they do arrive, they will arrive in the order they were sent. Furthermore, UDP provides no mechanism for determining whether or not any particular message reached its destination, although ICMP response packets are generated when the destination cannot be reached (for instance, a machine is offline, or the destination port is not open). UDP's lack of guarantees are offset by much lower overhead. While messages sent over TCP must be buffered by the operating system until they are acknowledged, messages sent over UDP can be discarded the moment they are transmitted.

Almost all modern application-level network traffic is carried over TCP or UDP. Numerous other protocols exist for routers and devices to communicate with each other to carry out routing and provide services such as the Domain Name System (DNS) which is used to map human-readable URLs to machine-interpretable Internet Protocol (IP) addresses, however, from the point of view of an application programmer, we are primarily concerned with TCP, which can provide reliable communication at the cost of higher overhead, and UDP, which can provide fast, low overhead communication at the expense of reliability.

2.2 Co-Simulation

Perhaps the simplest networking model possible, which was deployed in the original Doom[17] (Id Software, 1993), is peer-to-peer co-simulation. Under this model, every machine playing a networked game is treated as an equal, authoritative peer, and each machine simulates the entire game. The game is advanced in a series of logical steps, commonly referred to as ”frames” or ”ticks”. Before each ”tick” occurs, each peer transmits the actions its local player desires to take to each other peer, and receives the desired actions for those peers’ players in turn. The most common scenario is one player per peer, but having multiple players on a single machine is not impossible. Once every peer has received the input from every other peer, they can all advance one frame, carrying out the exact same calculations on every single machine. In order to avoid situations where machines are blocking on input messages travelling over the network, peers can transmit their input desired for frame n on frame $n - k$, for some constant k. This introduces a built in ”latency” to all player actions, as they cannot take effect until k frames have passed. However, if this value is well chosen, peers will almost never have to wait for messages that are still traveling through the network.

If written correctly, the same calculations will be performed on the same initial data, producing the same results, and each machine will have an identical representation of the state of the game. This requires careful attention on the part of the programmer. Pseudorandom numbers must be generated from the same seeds in the same order, and any differences between machines need to be isolated from the frame stepping calculations. For instance, different machines might return different addresses for dynamically allocated blocks of memory, or give different integer file handles for opened files. Therefore, numeric comparisons between these types of objects cannot be used. This means that dynamically allocated objects cannot be stored in, for instance, a balanced binary tree keyed on their memory addresses.

A correct implementation of co-simulation has the following key advantages:

1. The state of the game known to each machine is identical. There is never any need to account for differences between what two different peers can see.

2. Game logic can be developed independently from networking, except in the area of user input. There is no need to tweak the networking code to account for new types of objects or changing object behaviors.

3. The game is completely robust to cheating in the form of hacks that change the state of the

game. Any player whose data has been changed will rapidly go out of synch with the other peers, ruining his own game experience without affecting theirs.

However, all co-simulation implementations share the following key disadvantages:

1. As all peers need to have an identical model of the game state, there is no elegant way to allow new peers to join a game in progress. Either a secondary synchronisation model must be used to accurately reproduce the entire internal state of the game, or a record of all in-game events must be transmitted and the new peer must simulate the entire game from start to present as quickly as possible. Similarly, peers which drop from the game cannot re-join without using one of the above techniques.

2. The maximum speed of the simulation is capped by the speed at which the slowest peer can simulate. Thus, having even a single player playing on an older machine can slow down the game for everyone involved.

3. Peers are required to send messages to (and receive messages from) every other peer in the game. This has the practical effect of limiting the number of players at which the game can remain playable. Many residential internet service providers will actually impose a cap on the number of peers a single customer can communicate with at once.

4. As all game logic and state data is present on each peer machine, the game is completely vulnerable to cheating in the form of hacks that expose the state of the game. This has little effect in cooperative games, but can ruin competitive games, by allowing a hacking player to observe his opponent's position, decisions, and strategies.

Despite its early discovery, this technique is still the core technique employed by developers of real-time strategy games. Bettner and Terrano[3] deployed this technique for Age of Empires (Ensemble Games, 1997), enabling gameplay with up to eight players and hundreds of units without overtaxing the 28K dialup modems that were still prominent at the time. Reverse engineering[13] has shown that this technique was also used for Starcraft (Blizzard, 1998), which could save out a record of all input commands received during the course of a game to produce a "replay" file that could be used to play back the results of games.

The susceptibility of clientside simulation to hacking was one of the factors which influenced the development of the Open Real Time Strategy (ORTS) project [4] (Buro, 2002). However, as recently as Dawn of War II (Relic, 2009) and Starcraft II (Blizzard, 2010), leading professional real time strategy game engines still rely on peer-to-peer co-simulation techniques, as evidenced by freely available map-revealing hacks for both games.

2.3 Server-Client Networking

The most common alternative networking methodology, used by Quake (Id Software, 1996), has one machine explicitly designated as the server, while all other peers are designated as clients, and connect exclusively to the server via a star topology. Under this model, the server simulates the sole authoritative representation of game state, and communicates with clients to synchronise their view of the game state with the ground truth on the server.

As opposed to co-simulation, server-client games are much more flexible. Clients can join and leave a game in progress, and the performance of any particular client affects only their gameplay experience. Individuals with poor connections or slow computers will not slow down the game for everyone else. Furthermore, there is no requirement that clients have a complete model of the game state, or a complete representation of the game rules simulated on the server. This opens up a number of interesting opportunities:

1. Limited visibility: The server can choose to send to each client only the information that the player(s) on that client would be able to see. This eliminates information-revealing hacks and cuts down significantly on cheating.

2. Dedicated servers: The server can be run as a separate program, perhaps on a specially built machine with a fast internet connection. By eliminating the need for the server to spend CPU cycles preparing graphics or sound, server logic can be computed faster and more clients can be supported in a single game.

3. Proprietary servers: The genre of massively multiplayer online RPGs is built around proprietary servers which are never publically released. By selling a subscription to a service, instead of distributing a digital product, it is much easier to thwart piracy.

4. Server-side modifications: Special game variations or minor upgrades need only be distributed to servers. Thus, significant gameplay advancements can be introduced without requiring the majority of the player base to download a patch or a new client, which is excellent for public beta testing, community modifications of commercial games, or massively multiplayer online games.

The fundamental difficulty of server-client networking is that any and all information which is to be visible to the client must be transmitted explicitly. For most games, this includes:

1. The positions, orientations, and animations of all players and/or mobile entities (mobs) visible to a particular player.

2. The private (but player visible) state of the player.

3. The private (but player visible) state of items and equipment owned by the player.

4. The private (but player visible) state of allies or units loyal to the player.
5. Noteworthy events that require special effects, sounds, or messages to be displayed to the player.

Network programmers must carefully consider the importance of each piece of information and decide how much data will be synchronised, how frequently it will be replicated across the network, and whether or not to use reliable or unreliable networking protocols.

Conventional wisdom states to transmit short lived information, such as the positions of moving objects and entities, via UDP, for maximum speed, and because there is little consequence to losing any single packet, as the lost data will be overwritten with the next update. Important information, on the other hand, such as object creation or destruction, or rare state changes, should be sent via a reliable protocol, whether by TCP or by explicitly managing acknowledgements and resends of UDP datagrams.

2.3.1 Dead Reckoning

Typically, the rate at which a game server sends out updates to the clients (somewhere between ten and thirty times per second) is slower than the rate at which the client will render the simulation. This can lead to “jerky” behavior where objects remain at one position for several frames and then instantly “jump” to their new position.

Dead reckoning, as defined by Aronson[1], is a technique to mitigate this behavior. In addition to transmitting positions and values of varying fields, higher order derivatives such as velocities are also transmitted. This allows the client to make a prediction as to the location of an object for a few frames after receiving an update. This technique was used in Quake II (id Software, 1997) to hide the difference between the network framerate and the rendering framerate (which varied from client to client).

2.3.2 Predictive Contracts

Furthermore, an intelligent server can make use of a concept known as “predictive contracts”. Under this concept, the server remembers what information it has sent to the client, and is able to simulate the predictions that the client will be making, and compare them to the ground truth values on the server. Under this model, updates only need to be sent for a particular object if the difference between the actual and predicted values exceeds some chosen error tolerance.

2.3.3 Cubic Splines

Dead reckoning and predictive contracts do not solve the issue of jumpy visual perception of objects on the game client, they merely reduce the size and frequency of the jumps that occur. Caldwell[5] demonstrated a method of smoothing out discontinuities using cubic splines. At any point in time, the client can model the value p and first derivative v of all fields for which smoothing must be done. When an update comes in, a new value p' and first derivative v' will be delivered. Given a suitable timestep ϵ, it is possible to construct cubic spline function $f(t)$ such that $f(t) = p, f'(t) = v, f(t+\epsilon) = p' + \epsilon v', f'(t+\epsilon) = v'$. We could think of it as smoothly transitioning between two different trajectories, the trajectory we are currently on, and the trajectory the server tells us we should be on.

Of note is that the ϵ value actually increases the perception of lag, because it introduces a short interval of time between receiving a value and displaying it to the user.

2.3.4 Latency Compensation

Bernier[2] developed a novel method for dealing with the fundamental lag between a client's perception of an object and the object's actual position on the server. By having the server store snapshots of the positions of important objects for the last several frames, as well as measuring the latency between the server and client, the server could estimate the position that an object had been on a particular client when the client took a particular action. Any interactions that depended on objects being in a particular configuration relative to one another could thus take effect as they would have if objects were in the specific configuration they were on a client machine, when the player chose to invoke such an action.

This was deployed in Half Life (Valve Software, 1998), primarily under the context of aiming in competitive multiplayer. When a player aimed and fired his weapon, the server was able to rewind the positions of all players to where they were on the attacker's machine when he fired the shot. Hit detection could thus be performed with the positions visible to the client, and the effects factored in to the server's game state.

While this greatly improved the feeling of responsiveness in Half Life's multiplayer modes, there were some interesting side effects, in that it was possible to duck behind cover, and still take damage for several frames, because your opponents still perceived you as being visible.

Note that, while slightly more challenging to implement, this technique could also be applied to implicitly synchronised multiplayer games. In this case, any archiving or rewinding needs to be performed identically by all machines running the game. Machines also need to come up with an agreed-upon scheme for determining "when" an action is to have taken place, in a manner that is resistant to cheating by individual clients who either wish to speed up their own actions or delay the actions of their opponents.

2.3.5 UnrealScript

Unreal (Epic Games, 1998) adopted a more integrated approach. It used a proprietary scripting language known as UnrealScript [16] to handle the logic for all game entities. UnrealScript contains syntactic support for indicating which code should run on the server, which code should run on the clients, which machines should be considered authoritative over which pieces of data, and how data should be replicated. Furthermore, function calls that cross server/client boundaries can be automatically transformed into remote procedure calls.

UnrealScript's networking capabilities work by replicating the values of variables which change across the network at regular intervals. Objects can be given different relative priority levels, which allow them to be updated more or less frequently. Additionally, function calls across the network can be transmitted either reliably or unreliably, based on how they are declared inside of a "replication statement" in the UnrealScript source code. This allows quite a large degree of fine tuning in terms of what gets sent when, and the relative amounts of bandwidth dedicated to each object.

This technique is particularly elegant because it allows game logic to be written in a single place, regardless of whether it needs to run on the server or the client. All the network protocol glue needed to get the different components to work together is handled by the UnrealScript virtual machine.

2.3.6 ORTS

A very similar technique was used in the ORTS project [4] mentioned earlier. In ORTS, all game objects are defined in an object oriented language via a set of scripts known as "blueprints". Methods and fields on these scripts are marked up with access level specifiers, determining whether they are visible to clients. Access specifiers on fields control whether or not they are replicated to any client which can see the object, to only the client who owns the object, or to no one at all (in the case of hidden variables known only to the server). Actions performed by an ORTS client (which could be a user interface for a human player, or an AI bot) are performed as remote procedure calls to public methods on these objects. This allows almost all game logic to be described in blueprint files, while simultaneously providing the interface clients will use to interact with the game.

The actual replication of state variables is in ORTS is accomplished by a differencing scheme, which ensures that only those variables which have actually changed generate network traffic. This differencing scheme is followed by a layer of ZLib compression, to further reduce the amount of network traffic that is sent.

2.3.7 The "Quake 3" Networking Model

Quake 3 (Id Software, 1999) was the first game known to abandon the concept of reliable / unreliable packets in favor of a robust delta encoding scheme. The concept is simple. The server takes snapshots of the game state at regular intervals, and stores them in an archive. Each time a snapshot is taken, enough information is transmitted to each client to allow them to reconstruct the complete visible state of the game. Those clients then send back acknowledgements that a particular snapshot was received. These acknowledgements can be bundled with packets carrying ordinary player input, which often have plenty of room to spare.

Once the server has received acknowledgement that a client has received frame i, it can now rely on the fact that the client knew the state of the game at frame i for all future sends. For instance, if we are sending frame $i + 3$ to this client, we only need to send information about what has changed in the last three frames. This will often be fairly minimal, simply the new positions of any moving objects, and maybe a small amount of information corresponding to a new object that appeared, or a property that changed on an existing object. In other words, we exploit frame-to-frame coherency to minimise the amount of data that needs to be sent.

For this to work, clients must also archive the state of the game as they receive it. However, these archives do not need to be very large. The moment the client receives a packet from the server that is encoded as a delta from frame i, the client can throw out all frames prior to i, because the server now knows the client has access to frame i and it will always make use of that information. Similarly, once every client has confirmed that they have at least received frame i, the server can discard snapshots of previous frames, since it can also form a delta encoding from a frame at least as recent as frame i.

Under this model, there is no need to specifically retransmit dropped packets, in essence, all "new" data is continuously resent until it is either acknowledged or deprecated by further changes. This has the advantage of keeping the game state inherently synchronised, whenever a complete snapshot is received, the complete state of the game, as visible to the particular player, can be reconstructed based on the transmitted deltas and an archived copy of the game state.

A key advantage to this networking model is that there is no need to explicitly decide what information needs to be sent "reliably" or "unreliably", and neither the server nor the client need to worry about waiting for acknowledgement packets or interpreting messages in the correct order. The main disadvantage is that the server is constantly consuming bandwidth sending delta encodings, and the size of the delta encoding (dependent on the number and degree of changes) increases with the number of dropped packets. As packet sizes grow, latencies increase, and large packets are more readily dropped by routers during periods of network congestion. More significantly, any packets which exceed the maximum transmission size and are fragmented stand a compounded chance of being dropped. To mitigate this, Quake 3 incorporated a fast per-packet Huffman encoding to reduce the sizes of transmitted packets.

Note that this model is still compatible with dead reckoning and other techniques for reducing visual choppiness on the client. It is simply that updates are now being delivered in a rapid and concise manner, which in and of itself helps to smooth out some of the problems of discontinuous movement.

In essence, the Quake 3 model reduces network traffic to a handful of redundant copies (based on latency and packet loss) of every change in network-visible game state that occurs during the course of the game. This is particularly effective for first person shooters, where the number of players and mobile objects is often capped at a small, manageable number, but can become problematic for larger environments such as real time strategy games (where each player may command not one, but hundreds of units) or massively multiplayer online RPGs (where hundreds of players may be present in an area at once). In these games, enough movement and action can occur at once to seriously bloat packet sizes past the MTU, causing packet fragmenting and corresponding increases to latency and drop rates.

Chapter 3

Related Work in Compression

3.1 Shannon Entropy

Compression has its roots in information theory, which deals with the concept of information content. Put simply, information content is a measure of the fundamental amount of information required to describe particular concept. The most common measure of information content is Shannon entropy[15], which can be interpreted as the number of bits required to describe an arbitrary message drawn from a well defined space of possible messages.

If the distribution of the messages we are concerned with is uniform, that is, no message is any more or less likely than any other, then the minimum description of an arbitrary message requires a number of bits logarithmic in the number of messages. Specifically, if we have b bits to work with, then we can uniquely identify $n = 2^b$ messages, therefore, if we must represent n messages, we will require $b = \log_2 n$ bits.

However, Shannon showed that if the distribution of messages is not uniform, that the expected number of bits required to represent a message drawn from this distribution can be less than $\log_2 n$ bits.

Consider a random variable X, which has a number of discrete outcomes $\{\omega_1, \ldots, \omega_n\}$. If we wish to describe a particular outcome of X, we can use $\log_2 n$ bits to describe an index $i \in \{1, \ldots, n\}$, corresponding to a particular ω_i.

What if, instead, we chose to describe a position in the probability space of X, corresponding to an outcome ω_i. That is, if we imagine the interval $[0, 1]$ divided up into subintervals, each corresponding to a particular ω_i, and each having length equal to $P(\omega_i)$. We know these subintervals would sum up to an interval of length 1 because the probability mass function of X must sum up to one. The amount of information that we would need to uniquely identify any particular subinterval would be $-\log_2 P(\omega_i)$, a concept known as the information content of ω_i, or $I(\omega_i)$. The notion is that outcomes which are common have low information content, while outcomes which are very rare have high information content.

Shannon entropy is therefore simply the expected information content of the $random$ outcome of a particular random variable. It is defined thus:

$$H(X) = E(I(X)) \tag{3.1}$$

$$H(X) = \sum_i P(\omega_i)I(\omega_i) \tag{3.2}$$

$$H(X) = \sum_i P(\omega_i)(-\log_2 P(\omega_i)) \tag{3.3}$$

$$H(X) = -\sum_i P(\omega_i)\log_2 P(\omega_i) \tag{3.4}$$

$$\tag{3.5}$$

As a simple example, consider the flip of a weighted coin C, with probability $p = P(C = H) = 1 - P(C = T)$ of landing on heads. The Shannon entropy of this coin is

$$H(C) = -(P(H)\log_2 P(H) + P(T)\log_2 P(T)) \tag{3.6}$$

$$H(C) = -(p\log_2 p + (1-p)\log_2(1-p)) \tag{3.7}$$

For a fair coin $p = \frac{1}{2}$, we get $H(C) = 1$, but for a biased coin, say, with $p = \frac{9}{10}$, we get $H(C) \approx 0.46$. There is less entropy in the outcome of the biased coin, because we already have a pretty good idea of what the outcome will be.

Most compression schemes operate by transforming the input data into a form whereby the actual amount of memory used is much closer to the Shannon entropy of the data. That is, the lower the entropy of the original data relative to its size, the more it can be compressed. Note that, as the actual information content of the data does not change (the original contents can still be recovered), the output of a compression scheme will have high entropy relative to its size. This is why you cannot compress the same file multiple times and expect it to continue getting smaller.

3.2 Huffman Coding

One of the simplest and most widely used compression techniques is Huffman coding[8]. This technique assumes that the message in question is composed of a stream of symbols, drawn from a known alphabet. Huffman codes replace each symbol with a binary code of variable length, chosen to minimise the expected length of the resulting binary message, if the stream of symbols follows some known distribution.

The type of coding created is known as a "prefix code", meaning that no symbol is represented by a code which is a prefix to the code representing any other symbol. This is advantageous because it allows symbol codes to be appended to one another directly in a stream of bits, without any overhead to indicate when the coding for one symbol ends and the next begins. Shannon-Fano coding[15] also produces a prefix code based on the known probabilities of the symbols in an alphabet, but Huffman was able to show that his algorithm produced optimal prefix codes for a given distribution. Huffman coding is one of the main algorithms at work in the freely available open-source compression library ZLib[6].

It is worth noting that the compression ratios of Huffman coding are constrained by the need to emit discrete codes for each symbol, comprised of a whole number of bits. Thus, even though it is an optimal prefix code for the task of data compression, it has difficult representing symbols whose Shannon entropy is not a whole number (any symbol whose probability is not an integer power of two).

For instance, an alphabet comprised of three symbols, a, b, c, where $P(a) = P(b) = P(c) = \frac{1}{3}$, can at best be represented by a prefix code such as $a = 0, b = 10, c = 11$, or some other permutation, with the expected cost to represent any particular symbol at $\frac{1+2+2}{3} \approx 1.667$. By contrast, the Shannon entropy of such an alphabet is $-\log_2 \frac{1}{3} \approx 1.585$.

It is possible to improve the performance of Huffman coding by grouping sequences of symbols together, and running Huffman coding on the sequences of symbols instead of individual symbols. In the above example, by grouping symbols into groups of 5, we now have $3^5 = 243$ different equally likely sequences. As no sequence will require more than 8 bits to represent, and each sequence contains 5 symbols, a Huffman code on this new alphabet will spend a little less than 1.6 bits per symbol, much closer to the Shannon entropy.

However, it must be noted that the number of sequences is b^n for an alphabet with b symbols and sequences of length n. Constructing the full join probability distribution for such a scheme may be prohibitive, as may storing the corresponding Huffman tree.

3.3 Arithmetic Coding

3.3.1 Theory of Arithmetic Coding

Arithmetic coding[14] follows a different approach. Instead of emitting separate binary codes for each symbol, the entire message is encoded into a single, arbitrarily high precision number. Let M be the set of all possible messages. Our coder is a function $C : M \mapsto \{[a, b) : a, b \in R, 0 \leq a < b \leq 1\}$, which maps each possible message to a nonoverlapping subinterval of $[0, 1)$. If $C(m) = [a, b)$, then any number $x \in [a, b)$ will be considered equivalent to the original message m. We can therefore choose the number $x^* \in [a, b)$ such that x^* has the fewest possible digits, and transmit or store just those digits. On the receiving end, once x^* is known, we can recover the message m for which $x \in C(m)$.

In general, as x^* must always fall within $[a, b)$, a subinterval of $[0, 1)$, we know that $0 \leq x^* < 1$. We can therefore treat an n bit message unambiguously as the binary numerator of a fraction with denominator 2^n, giving us arbitrary precision within the range $[0, 1)$.

Consider the minimum number of bits in order to represent some number within the interval

$[a, b)$. If we were to spend exactly n bits, we could represent 2^n different numbers, spaced $\frac{1}{2^n}$ apart. Therefore, for any interval $[a, b)$ where $a + \frac{1}{2^n} \leq b$, there is at least one number which can be represented by n bits within the interval $[a, b)$. Let us solve for n given arbitrary a and b.

$$a + \frac{1}{2^n} \leq b \tag{3.8}$$

$$\frac{1}{2^n} \leq b - a \tag{3.9}$$

$$\frac{1}{b-a} \leq 2^n \tag{3.10}$$

$$\log_2(\frac{1}{b-a}) \leq n \log_2(2) \tag{3.11}$$

$$n \geq \log_2(\frac{1}{b-a}) \tag{3.12}$$

Thus, according to Equation 3.12, as long as we spend at least $\log_2(\frac{1}{b-a})$ bits, we can represent some number $x \in [a, b)$. Note that $b - a$ is simply the length of the interval $[a, b)$, which we can refer to with l. Thus, the minimum whole number of bits require to represent a given message is given by Equation 3.13.

$$n = \lceil \log_2(\frac{1}{l}) \rceil \tag{3.13}$$

Consider the case in which we have $M = \{m_1, m_2, \ldots, m_k\}$, and $C(m_i) = [a_i, a_i + l_i)$. If we let $P(m)$ be the probability that we will encounter message m, then an upper bound on the cost $c(m)$ to encode m using C can be computed as in Equation 3.21.

$$E[c(m)] = \sum_{i=0}^{k} P(m_i) c(m_i) \tag{3.14}$$

$$= \sum_{i=1}^{k} P(m_i) \lceil \log_2(\frac{1}{l_i}) \rceil \tag{3.15}$$

$$< \sum_{i=1}^{k} P(m_i)(\log_2(\frac{1}{l_i}) + 1) \tag{3.16}$$

$$= \sum_{i=1}^{k} P(m_i)(-\log_2(l_i) + 1) \tag{3.17}$$

$$= \sum_{i=1}^{k} P(m_i)(1 - \log_2(l_i)) \tag{3.18}$$

$$= \sum_{i=1}^{k} (P(m_i) - P(m_i) \log_2(l_i)) \tag{3.19}$$

$$= \sum_{i=1}^{k} P(m_i) - \sum_{i=0}^{k} P(m_i) \log_2(l_i) \tag{3.20}$$

$$E[c(m)] < 1 - \sum_{i=1}^{k} P(m_i) \log_2(l_i) \tag{3.21}$$

A lower bound can similarly be computed, using Equation 3.12 instead of Equation 3.21, resulting in Equation 3.22.

$$E[c(m)] = -\sum_{i=0}^{k} P(m_i)\log_2(l_i) + \epsilon, \epsilon \in [0, 1) \tag{3.22}$$

We are constrained in that we must assign our values for $\{l_1, l_2, \ldots, l_k\}$ such that $l_i > 0$ and $\sum_{i=1}^{k} l_i \leq 1$.

If we assign our partition our interval such that the length of each subinterval $l_i = P(m_i)$, then $E[c(m)] = (-\sum_{i=0}^{k} P(m_i)\log_2(P(m_i))) + \epsilon$, which is simply the Shannon entropy of M, plus a fraction of a bit (in practise, this fraction is due to the fact that a whole number of bits must be emitted, although the expected cost can still be fractional). Thus, an idealised arithmetic coder, with access to the exact probability distribution of the space of messages we wish to send, encodes messages whose lengths are expected to be within one bit of the Shannon entropy of that space of messages.

Contrast this with Huffman coders, which are required to emit up to an extra bit over Shannon entropy on a per-symbol basis. For instance, if we are encoding a stream of characters which are independent and identically distributed such that any character is ‘a’ with probability 0.99 and ‘b’ with probability 0.01, Huffman coding must emit a single bit for each character, whereas Arithmetic coding will emit on average only 0.08 bits per character, provided that the overall message still emits a whole number of bits. For messages which contain multiple symbols, Arithmetic coders rapidly outperform their Huffman predecessors.

3.3.2 Implementation of Arithmetic Coding

In practise, we will neither have access to the complete set of all possible messages, or the probability distribution of that set. However, if we break our message down into a sequence of symbols, we can form a good approximation.

Consider a message comprised sequentially of three symbols, drawn from three respective alphabets, $s_1 \in S_1, s_2 \in S_2, s_3 \in S_3$. We wish to assign the message $m = [s_1, s_2, s_3]$ an interval whose length is equal to $P(s_1, s_2, s_3)$. From the definition of conditional probability, we can rewrite this probability as $P(s_1)P(s_2|s_1)P(s_3|s_2, s_1)$. Thus, if we know the conditional probabilities of symbols appearing based on the previous symbols in the message, we can still compute the size of the interval to assign to m.

In practise, we can take a working interval, initialised to $[0, 1)$, and progressively shrink it for each symbol we encode. For instance, if $P(s_1) = \frac{2}{5}$, we might shrink the interval to $[0.2, 0.6)$ (any interval of size 0.4 will do). If $P(s_2|s_1) = \frac{1}{2}$, we might further shrink the interval to $[0.2, 0.4)$. Finally, if $P(s_3|s_2, s_1) = \frac{1}{4}$, we might shrink the interval to $[0.25, 0.3)$. The final interval, of length 0.05, corresponds to the overall probability $P(s_1, s_2, s_3) = \frac{1}{20}$.

This means that we can encode arbitrarily many symbols into a message, by simply using each one to shrink our working interval until we arrive at the final interval. Symbols can be drawn from different alphabets, and can even depend on the specific values of prior alphabets. As long as we shrink the working interval by a factor equal to the conditional probability of the current symbol appearing at that point in the stream, the cost of encoding our overall message will remain nearly equivalent to the Shannon entropy of the message we are encoding.

When we are decoding a message, we can follow the same process. We can determine the first symbol by looking at which subinterval our code number falls into, and then shrink our working interval to that subinterval. We can repeat this process to decode further symbols, tracing the exact process by which the working interval was transformed during encoding. This process is illustrated in Figure 3.1.

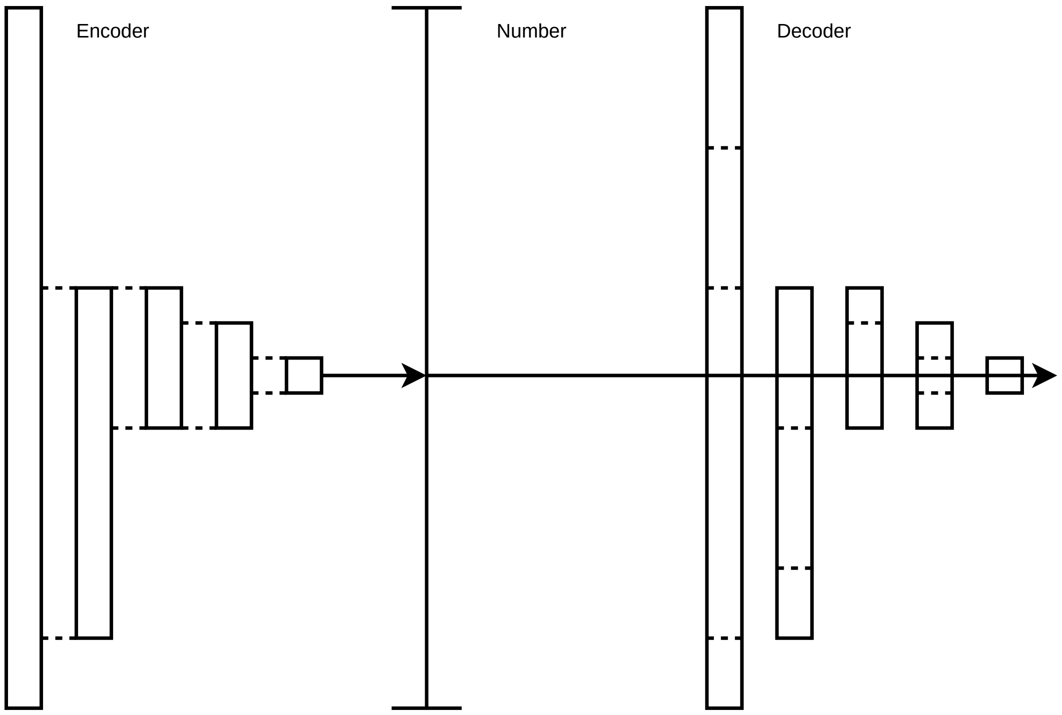


Figure 3.1: Conceptual Overview of Arithmetic Coding

In practise, we will not have infinite precision numbers to work with when encoding or decoding our arithmetic code. However, it turns out we will not need them. As we encode symbols, shrinking our working interval, we will notice that at times the lower bound and upper bound of our interval will agree on one or more digits. Once this condition is detected, we can emit those digits immediately, as any number falling between these boundaries will also share those same digits. Once we have emitted the known digits, we can shift them out, and "promote" the remaining digits. For instance, if we encode a symbol, and our working range shrinks to $[0.01100001, 0.01101010)$, we can emit the digits "0110", and renormalise our working range to $[0.0001, 0.1010)$. We could think of the contents of our “low” and “high” bounding variables as sliding windows into the real variables, giving us effectively infinite precision, although we can only work with 32 or 64 bits at a time. This process is illustrated in Figure 3.2.

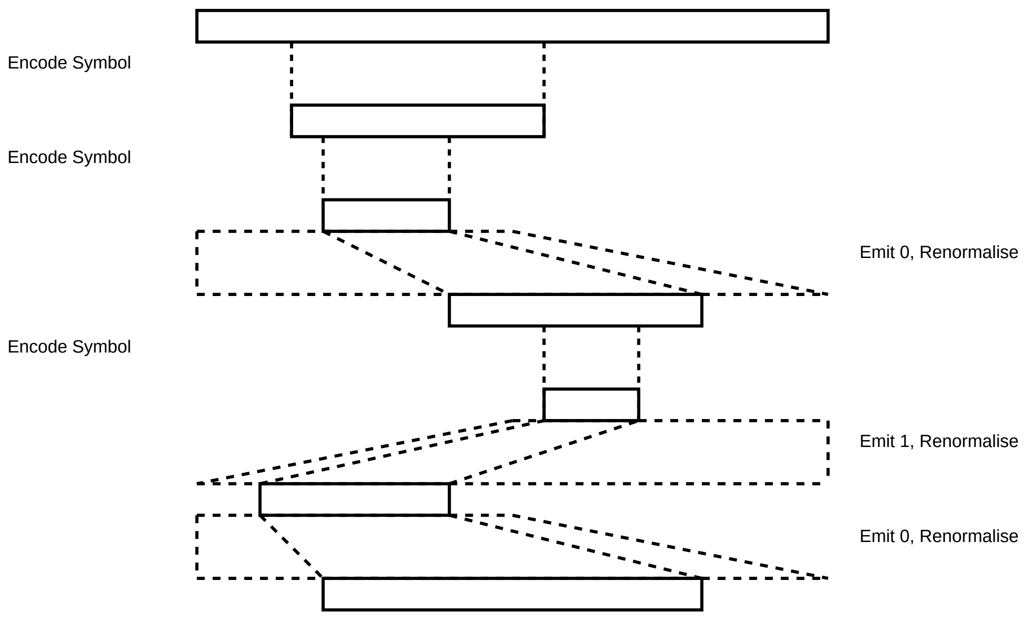


Figure 3.2: Arithmetic Encoding Example

Our decoder will work in the exact same fashion, except that in addition to modelling a window into the “low” and “high” values, it must also model a window into the coded number itself. This number will be used to decide which subintervals we select (and correspondingly, which symbols will be emitted). Whenever we shift our “low” and “high” variables, we must shift away the same number of bits from our “code” variable, and read in new bits from our encoded message. Note that, when we read the last bit in from our coded message, we may still have several symbols to decode. As our code represents a number, we simply assume that all remaining digits are zero, and continue decoding until the final symbol has been decoded. This process is illustrated in Figure 3.3.

It is worth reiterating that there is no need for each encoded symbol to have been drawn from the same alphabet, or to be encoded using the same probability distributions, as long as it is clear from context when decoding which alphabet and distribution will be used for the next symbol in the stream. Similarly, it is not necessary to indicate how many symbols have been encoded as long as it will be clear from context when the last symbol has been decoded from the stream. This allows very sophisticated file formats and communication protocols to be described entirely with an arithmetically coded message, and the only actual “metadata” required is the length of the coded message itself, which is usually available from context.

Arithmetic coders can be written using 32-bit or 64-bit floating point operations, but there is no real need to do so. The simplest arithmetic coders can be written using straight integer values for the low and high bounds, and the code itself. In order to encode a subinterval, you call a function

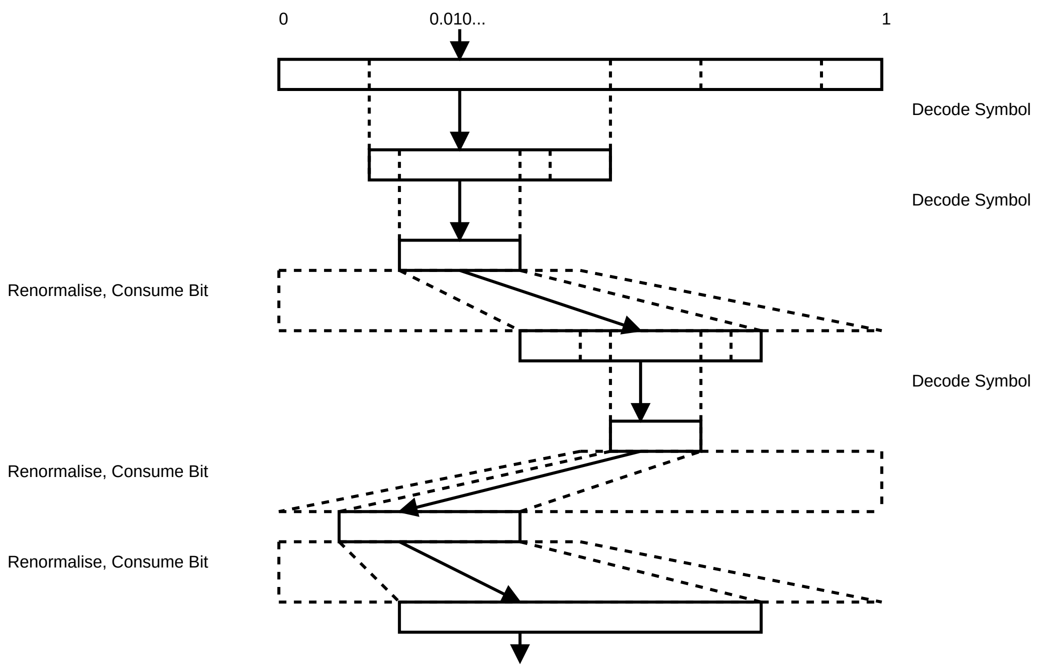


Figure 3.3: Arithmetic Decoding Example

Encode(a, b, d), which accepts three unsigned integers representing the subinterval $[\frac{a}{d}, \frac{b}{d})$. The coder simply determines the range between the low and high bounds, splits it into d equally sized portions, and shrinks the working interval to consist only of portions a through $b - 1$.

This makes arithmetic coders a natural match for statistical distributions which are estimated by counting the frequencies of observed symbols. For instance, Table 3.1 shows how we can use the observed frequencies of symbols from an alphabet to encode any particular symbol via an arithmetic coder.

Table 3.1: Example Frequency Table for Arithmetic Encoding

Symbol	Frequency	Encoder Call
s_1	5	Encode$(0, 5, 17)$
s_2	8	Encode$(5, 13, 17)$
s_3	4	Encode$(13, 17, 17)$

Decoding is a slightly trickier task. You must first call a function Decode(d), which, as before, splits the working interval into d equally sized portions. It then determines which of these portions contain the code number, and returns the index of that portion. You must manually compare this index to the $[a, b)$ intervals for each symbol that could have been encoded. Once the correct symbol has been identified, you then call Confirm(a, b), which shrinks the working interval according to whichever symbol was encoded.

In fact, at the lowest level, arithmetic coders are not aware of symbols at all. The function Encode(a, b, d) is simply a request to the encoder to encode something such that when the corresponding Decode(d) is called, a number in the interval of $[a, b)$ is returned. This has some interesting properties. For instance, you can call Encode($i, i+1, d$) for some $n < d$, and the corresponding call to Decode(d) is guaranteed to return i. As this technique allocates an equally sized subinterval for each possible value of i, it is a great way to store integers which are drawn from a range whose size is not a power of two, if those integers are roughly uniformly distributed within that range.

3.3.3 Range Coding

Arithmetic coders are sometimes known as range coders [11, Martin, 1979]. Theoretically, arithmetic coders refer to coders which produce a rational number in the interval $[0, 1)$, while range coders produce a whole number within an interval on the whole numbers. The algorithms themselves are equivalent. Historically, the term “range coder” tended to refer to an arithmetic coder that was implemented exactly as specified in Martin’s paper. These coders would emit code a byte at a time, which meant that the working interval grew somewhat smaller than in an arithmetic coder which would emit code a bit at a time. This process was claimed to be twice as fast as a bit-by-bit arithmetic encoder, while incurring less than a 0.01% cost in the size of the compressed stream.

However, range coders were also interesting for a different reason. From 1978 onward, several United States Patents were issued to IBM Corporation on arithmetic coders and various uses. However, Martin’s paper predated all but one of these patent applications. After Patent No. 4122440[7] expired in 1997, Martin’s paper could be shown to be “prior art”, and range coders implemented exactly as presented in Martin’s paper were safe from potential patent infringement litigation. This was of great interest to the open source community. Most of the patents related to fundamental arithmetic coding have since expired.

It is worth noting that the actual performance of an arithmetic coder implementation will often not precisely match the Shannon entropy of the message being encoded. Some implementations, for instance, divide the working interval into equally sized subintervals. In these cases, whenever the working interval is not wholly divisible by the given denominator, a certain amount of the working interval is discarded, not assigned to any potential symbol. This slightly inefficient way of dividing intervals is beneficial because it eliminates the need to catch and handle integer overflows during the algorithm.

The slightly weaker compression observed for range coders is partially due to this phenomena. Range coders allow their working interval to grow smaller before emitting data and renormalising, so the integer remainders when dividing an interval into pieces occupy a comparatively larger amount of the interval, resulting in greater wasted space.

More importantly, the actual performance of the coder is highly dependent on the quality of the probability estimates used to select the fractions to pass into the encoder. Most applications of arithmetic coding, including the one presented in this thesis, put special care into how they acquire estimates for the probability that a particular symbol will appear at a particular point in the stream.

3.3.4 Prediction by Partial Matching

"Prediction by Partial Matching"[12] (PPM) is the name given to a family of algorithms based on combining learned statistical information with arithmetic coding to achieve data compression. In this case, the conceptual target is textual data, although PPM-style techniques can be successful in a wide range of contexts.

PPM works on the notion of a context, typically the last n observed characters. For each possible sequence of n characters, a frequency table is stored and populated, listing the number of times any particular character has followed the sequence in question. When compressing a stream of characters, the simplest behavior is, for each character, to look up the context of the previous n characters, and then to encode the current character using the frequency table built up thus far, after which point the frequency for the present character can be incremented.

However, on occasion, a particular string will be observed, and there will be no frequency data for characters following the string in question. In fact, if we are starting with empty frequency tables, this situation will occur frequently as we begin encoding text. The solution, in PPM, is to additional store contexts for smaller sequences of characters. If no frequency information exists for characters following a string of n characters, then PPM checks the context for the $n - 1$ most frequent characters. If that fails, PPM checks the context for the $n - 2$ most frequent characters, and so on down to a single "zero-level" context, which simply stores the statistical frequency of characters encountered in the source text so far. In fact, these "lower order" models can be learned while encoding characters using higher order models. This is where the "Partial" in Prediction by Partial Matching comes from. We try to match part of the preceding text to some context where statistical frequencies are known. If such a context is unavailable for a large part, we use a smaller part.

Lastly, there will occasionally be situations where a particular observed string does have a nonempty context, that is, we have frequency information for symbols following our string, but we have an observed frequency of zero for the current symbol in our stream. In this case, we would normally assign the symbol an estimated probability of zero, but of course, we cannot encode a zero-length interval, because we will not be able to find a bit string that represents a number which exists inside this interval, to say nothing of our ability to perform further encoding.

There are a number of known solutions for handling the "zero frequency problem". One solution is to assign each symbol an initial pseudo-frequency of one. This is somewhat similar to the Bayesian concepts of prior and posterior distributions, in this case, we assume a uniform distribution across symbols as our prior distribution, and iteratively refine towards our correct distribution.

Another solution is to have an "escape" symbol, which always has some nonzero frequency, and which can trigger some lower-order, simpler encoding. In PPM, the escape symbol might be designed to trigger an encoding in the next smaller context, so an escape symbol at the 3-character context would trigger an encoding at the 2-character context (and if our particular character still had zero frequency in that context, we could emit another escape character). Another alternative might simply be to follow the escape symbol with an encoding of the character drawn from a uniform distribution. Indeed, we may need to resort to this anyway, if a symbol has zero frequency in the 0-character context.

There is one last detail to be considered. What statistical frequency should be assigned to the escape symbol? One solution is to use a fixed pseudocount of one, which always reserves some small interval of space for representing the escape symbol, though this space shrinks as more and more characters are encountered. Another solution is to increment the pseudocount of the escape symbol every time a zero-frequency symbol is encountered. That is, the probability of encountering an unseen character is estimated by the ratio of unique characters to total characters in the source text thus far. This is the basis for the PPM-D algorithm.

Consider what sort of compression performance we would be able to get if we could narrow down the possibilities for the next character in an ASCII-encoded stream of English text to only two, each with a 50% probability of occurring. In this scenario, we would be able to encode each character with only one bit, achieving $8 : 1$ compression. The current record holder for the Hutter Prize, "durilca'kingsize", is a PPM-based algorithm capable of $7.8257 : 1$ compression over a gigabyte-sized text dump of the English Wikipedia from 2009[10].

Chapter 4

Library Overview

Our work has resulted in the creation of a software library designed to drastically reduce the amount of effort required to develop a networked video game. It has, as its chief design goals, the following principles:

1. Reliability: The library should "just work", with relatively few things that can go wrong, and safeguards to warn the user of anything that can.
2. Ease of use: The library must not require extensive knowledge of networking, and should have a minimalistic public interface.
3. Non-intrusiveness: The library must not require extensive modification of game code, such as requiring game classes to inherit from interfaces or implement specific methods.
4. Efficiency: The library should strive to use CPU and memory resources as efficiently as possible.

Additionally, the library is written in standard C++ and should be both platform and endian-ness independent.

4.1 View-Based Architecture

The library's primary use is in transmitting data between pairs of views. These views, one on the server, the other on the client, are the main point of contact between game code and library code. The server view's job is to observe some subset of game state, for instance, the properties of a single object, and produce a "snapshot". The library will then reproduce the snapshot on the client, and provide it to the client view, which could, for instance, use the received data to drive model placement and animation in the renderer, or configure user interface elements, etc. Clients have their own "visible set" of views, and each time the server takes a snapshot of the game state, only enough information is transmitted to each client to reproduce the snapshots corresponding to their

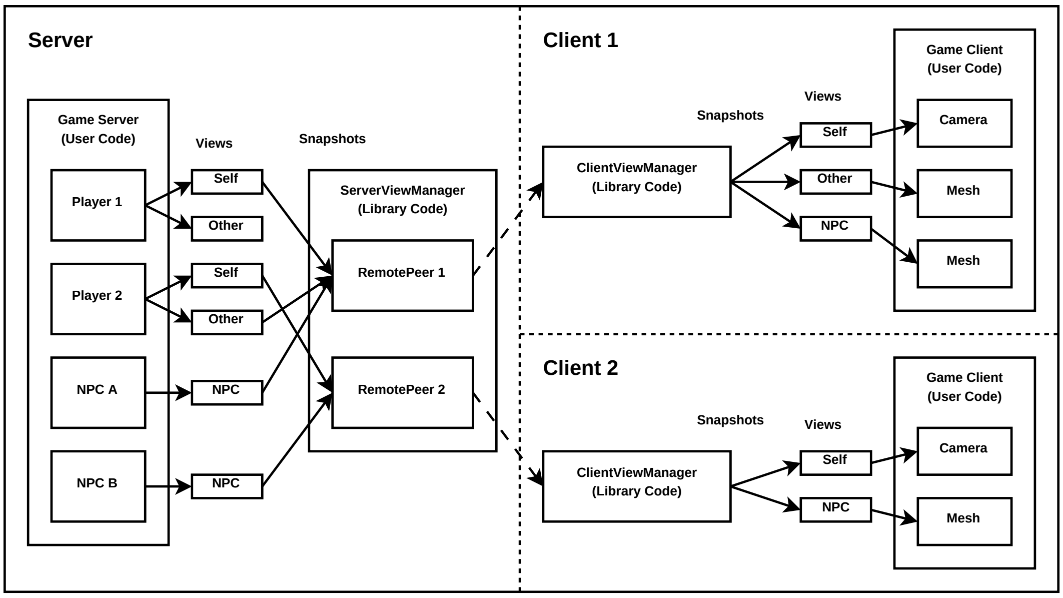


Figure 4.1: Library Usage Overview

visible set. No information whatsoever is leaked between clients, even to the extent of using separate pools of network IDs whenever there is a need to refer to specific views.

The use of snapshots, instead of the more straightforward approach of writing and reading individual bits and bytes, has two chief advantages. The first is that previously captured or received snapshots can be archived, and used to make predictions about the contents of future snapshots. The second is that the different fields within each snapshot, and the different types of snapshots themselves, can be explicitly recognised, allowing the library to learn separate policies for transmitting each field in the most efficient way possible. A secondary benefit is an added degree of robustness. As the order in which values are packed into the snapshot structure on the server can be completely different from the order in which values are used on the client, the process of adding new fields or entire objects to be synchronised is much less error prone. There is essentially no risk of introducing subtle errors that would otherwise be very difficult to debug.

4.2 Predictive Delta Encoding

Once snapshots are acquired, there are three steps that are performed for every field of every view in a client's visible set.

1. The server and client simultaneously and identically make a prediction about the value of the field, based on previous snapshots known to both machines.

2. The server creates an "error term" between the predicted value and the value actually observed in the snapshot, and transmits that error term to the client via a form of entropy coding. The client uses its predicted value and the error term to reconstruct the actual value.

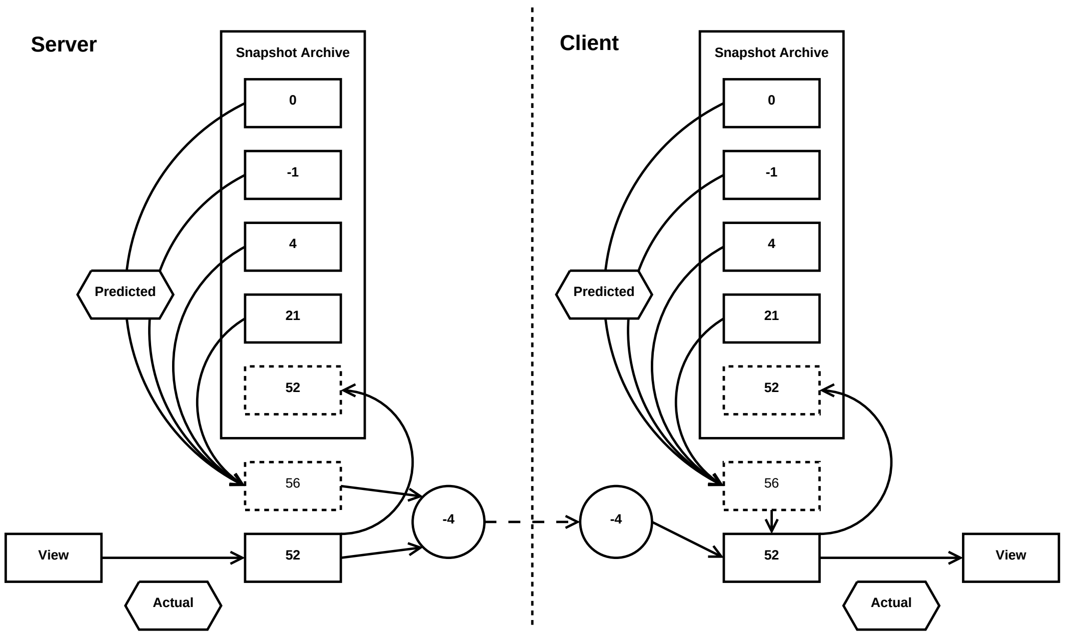


Figure 4.2: Overview of Delta Differencing Scheme

3. The server and client simultaneously and identically learn about the statistical distribution of error terms for the particular field in question. This distribution is used to improve the efficiency of transmitting future error terms.

While each of these steps will be elaborated on in more detail later, the chief concept to note is that the server and client collaborate in developing a “shared context” of information about the game, in the form of snapshot histories for views and dynamically learned statistical distributions over error terms. This allows the library to not only learn about the game being played, but about each client’s specific experience of the game being played, and adapt to changes in that experience. It is absolutely critical that we be robust to these sorts of variations and changes, because varied, asymmetric gameplay is a **goal** of game designers, and cannot be avoided.

4.3 Justification for Learning Network Protocols

One might rightfully ask why we should bother with learning a network protocol at all. Most games whose synchronisation models are public knowledge do not attempt to learn about the content being synchronised while the game is running. Instead, they have static, usually hand-coded synchronisation solutions for each part of the game that is transmitted.

The best answer to this question is to consider the trade-offs made in developing a network protocol from scratch. A simple, robust protocol can be constructed very quickly if one is not concerned about space efficiency, simply stuff the state of every game object into one large message and send it every frame. This will work, but for anything other than the very simplest of games, message sizes will rapidly grow large. As message sizes become larger and larger, they are more prone to fragmenting into multiple packets, and the resulting torrent of packets can increase latencies and drop rates, choke off the user's network connection. This could ultimately cause the network connection to fail as expensive resends need to be carried out more and more frequently.

It is surprisingly easy to reach this point, as even in 2010, the maximum message size that can be sent over UDP over the internet tends to be 1440 bytes. Simply sending the three dimensional coordinates of a series of game objects as single-precision floating point values, and eschewing any other form of information, you will reach that limit after 120 objects. If velocities are sent, only 60 objects can be synchronised. Orientations, represented as Euler angles, drop us to 40 objects, or, as quaternions, to 36 objects. We are now well below the capability to handle 64-player online first person shooter games, and we haven't even touched player avatars, weapons and projectiles, items on the map, event notifications, objectives, vehicles, etc.

The alternative is to have a software engineer design some protocol to reduce your bandwidth usage. He might start with a delta encoding scheme, introduce predictive contracts, and then start on identifying effective ranges of variables, systems of early-out flags and variable bit length encodings, and all manner of tricks dependent on his specific knowledge of the game in question. This will generally need to be done early in development, and updated frequently as gameplay mechanics are added, removed, and tweaked. This process is labor-intensive and error prone, as subtle assumptions that are true one day cease to be true the next.

On the other hand, a well designed network subsystem that can use learning to achieve efficient encoding of game state without requiring manual tweaking or any game-specific knowledge gives you the best of both worlds. Designers and software engineers can spend their time working on important gameplay related decisions without worrying about the low level networking code, while retaining most of the space efficiency of a hand-coded network protocol that takes advantage of knowledge about the content of the game.

As an example, consider developing a real time strategy game, in which units are placed on a grid. Initially, the game may only support maps up to 256x256 in size, and units might only move one tile in any particular frame. An enterprising software engineer might decide to take advantage of these properties, transmitting initial coordinates as a pair of bytes, and transmitting movement in a mere three or four bits. Later in development, however, maps might be made larger, or units might be allowed to move to fractional locations between tiles, or an ability might be added that allows units to teleport. If our software engineer is lucky, the game will immediately crash, and he can begin the laborious process of stepping through the server and client in a debugger looking for the discrepancy. If he is unlucky, failure might only occur during rare corner cases that are difficult to reproduce, let alone debug.

On the other hand, a library that has no preconceptions about the game, but can learn to take advantage of trends, tendencies, and observed statistical distributions, would be able to efficiently express the sorts of data and events that it observes without committing to any particular representation. This ability to transmit data efficiently while remaining very low maintenance is key to achieving the design goals listed at the start of this chapter.

Chapter 5

Accurate Field Prediction

In order to achieve the greatest compression possible, it is important to have accurate predictions of the values that fields will take on. The better the predictions are, the more tightly the error deltas will be distributed around zero. This means a smaller range of symbols to transmit, with each symbol appearing more frequently, which is a major win for all forms of entropy coding.

In addition to our desire for accuracy, we have the absolute requirement that any information we use to make a prediction is available on both the server and the client, so that both machines can make a deterministically identical prediction. Unless we have this property, we cannot correctly reproduce a field value from the error term alone.

The simplest prediction technique is to assume values have remained unchanged from some point in time where the values were known to both the client and the server. Under this assumption, the error term would simply be the difference between the previous value and the current value, which gives a standard delta-encoding from that frame. Generally, as fields change over time, it makes sense to use the most recent frame that the server knows the client has access to, to minimise the number and magnitude of changes that will have occurred.

However, we can do much better than this, since the client and the server will actually have access to the state from **all** frames that were transmitted, should we choose to archive them. If we treat the values of a particular field from a number of prior snapshots as the data points of some hidden function that governs the behavior of this field, we can model it and use our model for prediction.

5.1 Constructing Polynomial Predictors

Consider a scenario in which the client has received and acknowledged several frames, which we will refer to as $\{t_0, t_1, \ldots, t_{n-1}\}$. Note that these t_i values need not be sequential or ordered in any particular way, but $t_i \neq t_j \forall i \neq j$.

As both the server and client, we are interested in predicting the value of a specific field at frame number t_n. Typically, t_n will be a value greater than any of the other t_i values were are using, but

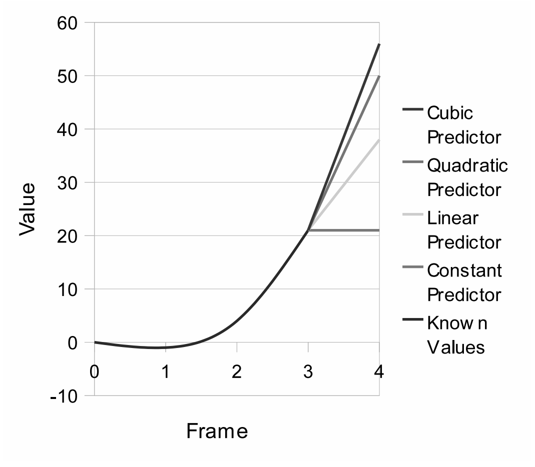


Figure 5.1: Comparison of Polynomial Predictors applied to function $f(x) = x^3 - 2x$

there is no requirement for this to be the case.

For our purposes, we will assume that the value of this particular field is governed by some arbitrarily complex, unknown function $f(t)$, for which $f(t_0), f(t_1), \ldots, f(t_{n-1})$ are all known. $f(t_n)$ is known to the server, but not the client. We will approximate $f(t)$ by a function $g(t)$, and treat $g(t_n)$ as our prediction for the value of $f(t_n)$. The server can then send $\delta = f(t_n) - g(t_n)$ to the client, which can reconstruct $f(t_n)$ as $f(t_n) = g(t_n) + \delta$.

Consider approximating $f(t)$ with the polynomial function $g(t) = \sum_{i=0}^{m-1} a_i t^i$, as is shown in Figure 5.1. While polynomial functions have limited expressiveness, they can achieve a high degree of accuracy in approximating continuous, differentiable functions around a specific point, as is done with Taylor series approximations. Unfortunately, we cannot rely on having access to the values of $f(t)$'s derivatives at each point in time, however, we can still select values for $a_0, a_1, \ldots, a_{m-1}$ which form a reasonable approximation. One way to do so is to select a number of previous frames, and construct a polynomial function that precisely passes through each point $(t_j, f(t_j))$ corresponding to the frames we have selected. In other words, we will construct a polynomial such that $g(t_j) = \sum_{i=0}^{m-1} a_i t_j^i = f(t_j)$ holds for each such t_j. Thus, we have the following system of equations.

$$\begin{array}{ccc}
\sum_{i=0}^{m-1} a_i t_0^i & = & f(t_0) \\
\sum_{i=0}^{m-1} a_i t_1^i & = & f(t_1) \\
\vdots & & \vdots \\
\sum_{i=0}^{m-1} a_i t_{n-1}^i & = & f(t_{n-1})
\end{array}$$

These equations can be expressed in matrix notation via Equation 5.1, or more succinctly via Equation 5.2.

$$\begin{bmatrix} 1 & t_0 & t_0^2 & \dots & t_0^{m-1} \\ 1 & t_1 & t_1^2 & \dots & t_1^{m-1} \\ 1 & t_2 & t_2^2 & \dots & t_2^{m-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & t_{n-1} & t_{n-1}^2 & \dots & t_{n-1}^{m-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{m-1} \end{bmatrix} = \begin{bmatrix} f(t_0) \\ f(t_1) \\ f(t_2) \\ \vdots \\ f(t_{n-1}) \end{bmatrix} \tag{5.1}$$

$$TA = F \tag{5.2}$$

If we have $m = n$, we can solve for a unique, exact value of A via $A = T^{-1}F$. It is worth noting that T is a square Vandermonde matrix, where each row has a different base. The Vandermonde determinant, $\det(T) = \prod_{1\leq i<j<n}(t_j - t_i)$, is merely the product of a number of small positive integers, and is therefore guaranteed to be nonzero, meaning that our T matrix will always be invertible.

The following techniques will assume this is how we are solving for A, however, it is worth noting that in cases where $m > n$ (we have more sample points than the order of the function we are approximating), we can use any one of a number of linear regression techniques to solve for A, provided the formula for A consists of some matrix independent of F multiplied by F. For instance, an ordinary least squares regression would approximate A with $(T'T)^{-1}T'F$, in which case $(T'T)^{-1}T$ becomes our independent matrix.

We can thus express $g(t_n)$ as in Equation 5.6.

$$g(t_n) = \sum_{i=0}^{m-1} a_i t_n^i \tag{5.3}$$

$$= \begin{bmatrix} 1 & t_n & t_n^2 & \dots & t_n^{m-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{m-1} \end{bmatrix} \tag{5.4}$$

$$= UA \tag{5.5}$$

$$= UT^{-1}F \tag{5.6}$$

Note that U contains only the powers of the approximation time, t_n, T contains only the powers of the prior sample times $t_0, t_1, \dots, t_{m-1}$, and F contains only the values of the field at the prior sample times $f(t_0), f(t_1), \dots, f(t_{m-1})$. This means that for a particular approximation time and set of sample times, for instance, "Approximate frame 10 based on frames 8, 7, 5, and 2", both U and T will be constant for the duration of the frame, and a row vector P can be precalculated, resulting in Equation 5.8.

$$P = UT^{-1} \tag{5.7}$$

$$g(t_n) = PF \tag{5.8}$$

Therefore, predicting the value of a given field is as simple as sampling the values of that field at several points in the past, and taking a dot product between a precalculated "predictor" vector and the fetched "sample" vector.

Note that if T^{-1} is calculated as $\frac{1}{|T|}(C_{ji})$, where C_{ji} is the transposed matrix of cofactors of T, then the only division operation used in forming P is the division by the determinant of T. If instead the determinant is stored, we can use Equation 5.11, with a "numerator" vector and a "denominator" constant.

$$P = U\frac{1}{|T|}(C_{ji}) \tag{5.9}$$

$$g(t_n) = U\frac{1}{|T|}(C_{ji})F \tag{5.10}$$

$$g(t_n) = \frac{UC_{ji}}{|T|}F \tag{5.11}$$

This is important, because it allows us to make our prediction using integer values, which is desirable because integer differences are reversible, that is, if $c = a - b$, then $a = b + c$. This is not always true with floating point values, which would create small differences between the ground truth values stored on the server, and the reconstructed values created in the client by adding the transmitted error term to the predicted value. Since the client would use these slightly different values to make future predictions, the server and client would diverge over time. This could be solved by predictive contracts, as shown by Aronson[1], but it is simpler to just use integers (which can also represent fixed-point real values), and be explicit about how much precision is needed.

There are some concerns with using integer values, however. For one, an integer division carries with it an inherent discarding of the remainder, akin to a forced floor function on our predictions. If this is a concern, slightly different predictors could be formed which use additional constant terms on the numerator to force behavior equivalent to different types of "rounding".

Furthermore, performing the multiplications and summations first, followed by the division operation, could lead to overflow concerns. In practise, the actual values in the numerator vector and scalar denominator will be small. Later in this chapter, we will show that the predictors can be calculated in a slightly different but equivalent way that takes into account only the time differences between the frame being predicted and the frames being used as sample points.

However, even small coefficients can cause an overflow if we are using the full range of values for a particular integer field. In these cases, it will be necessary to use a larger integer type or special hardware instructions to catch the overflow and deal with it appropriately. For instance, if 32-bit integer fields are used, a 64-bit integer could be used to store the result of the dot product, and the subsequent division operation, at which point it can be safely cast back to a 32-bit integer.

5.2 Learning the Best Predictor

We can use the work in the preceding section to generate predictors corresponding to polynomials of any order.

Having such a list of predictors, we can apply each one to every field and gather information about the statistical distribution of error deltas (differences between the actual and predicted values) for each predictor. In Chapter 6, we will show that the statistical distribution of values is key to being able to transmit them via entropy coding. In addition, we will show that, for a known distribution, the expected cost of transmission via entropy coding can be calculated cheaply. Therefore, it is not unreasonable to simply learn the distribution of error deltas produced by each predictor, and send the error delta produced by the predictor whose distribution has the cheapest expected representation.

Simultaneously, the client will be learning these same distributions (as a necessary part of our entropy coding implementation). The client can therefore deterministically predict which predictor the server will have selected at any point in time, apply that same predictor, and recover the actual value by summing the error delta onto the predicted value.

To be explicit, the server must perform the following steps:

1. Select the predictor with the cheapest expected representation cost
2. Make a prediction using that predictor
3. Form the error delta δ as the actual value minus the predicted value
4. Encode δ according to its known distribution
5. Make all other predictions, and form their corresponding δ terms
6. Record each δ term in its corresponding distribution

The client must perform these steps:

1. Select the predictor with the cheapest expected representation cost
2. Make a prediction using that predictor
3. Decode δ according to its known distribution
4. Recover the actual value as the predicted value plus δ
5. Make all other predictions, and form their corresponding δ terms
6. Record each δ term in its corresponding distribution

By following these steps, the server and client maintain the invariant property that they each model the exact same distribution of error terms corresponding to each predictor for each field. This allows them to continue to select the same predictor each time a value must be transmitted, which in turn allows them to continue to make the same predictions, calculate the same error terms for each predictor, and thus learn the same distributions of error terms.

Note that, under this setup, the server and client can continue learning about the performance of predictors that are not currently being used, because both can deterministically produce the error terms that would have been transmitted had other predictors been used.

In practise, we do not need to factor every value into our distributions, nor do we need to select the best predictor every time we wish to select a value. In fact, we could decide that once we have established that certain predictors are statistically worse than other predictors, we can stop learning about them, saving us valuable CPU cycles. If fact, if we so chose, we could stop learning about all distributions after we had gathered a certain amount of data, speeding up our process drastically.

What is important is that whatever policies we use in regards to this, they function identically on the server and client, even accounting for the slightly different order that things are calculated. If the distributions are allowed to go out of synch even a little, entropy coding will completely break and the client will be unable to decode anything the server sends.

It is a valid question to ask why multiple predictors are needed. Higher order polynomial functions seem capable of subsuming the representative strength of lower order polynomial functions (since nothing prevents the higher order coefficients from being equal to zero). However, there are cases in which a lower order polynomial makes a better predictor. This is because the sample points that are used to form the approximating function are not equally valuable. Samples from more recent frames are more likely to be indicative of the field's current behavior than samples from more distant frames.

Consider the case of an object traveling linearly, and bouncing elastically off a surface at an angle. We may have three archived frames, one from before the object struck the surface, one during the object striking the surface, and one afterwards, where the object is now traveling away from the surface. A linear predictor will correctly observe the two most recent locations of the object and correctly infer the linear trajectory on which the object is now traveling. A quadratic or cubic predictor will observe the "v" shape formed by the last three positions, and approximate the motion with a parabolic function, incorrectly estimating that the object is accelerating.

In practise, different predictors wind up being useful for different fields for this exact reason. When an object has higher order dynamics, such as acceleration or jerk, or is traveling along a path defined by a higher order function, it makes sense to approximate its behavior with higher order polynomials. When an object has simpler dynamics, it makes sense to approximate its behavior with simpler functions, so as to actively deprecate older information that may not be helpful.

5.3 Making Predictions with Incomplete Data

Each predictor requires a specific number, m, of sample values from previous frames. There will always be situations in which that many frames are unavailable, such as, for instance, an object being recently created or discovered, and fewer frames being archived on the client than a predictor requires. The simplest way to handle such a scenario is to modify the predictor selection algorithm to only consider predictors which can be used with the number of sample points available. For instance, if the known best predictor for a particular field is a quadratic one, but a particular object only has two sample points available, we can fall back to a linear predictor.

However, this still does not solve the problem. A freshly created or discovered object will not have any snapshots archived on the client at all. To address this, we can add in a trivial "zero" predictor. This predictor predicts a constant value of zero at all times, and requires no samples at all to operate.

In this case, the δ terms being transmitted are simply the actual values of the field, and the distribution being learned is the distribution of values the field can take on. It is particularly useful that we can continue to learn about the distributions corresponding to predictors we are not using, as it means that over the course of the simulation, we can learn about the ranges and distributions of values for each field, within the context of the zero predictor. Whenever freshly discovered objects are encountered, we will already have a good idea of the possible ranges of values the object's fields can take on, and we can transmit them efficiently.

For instance, the field corresponding to positions can learn that positions of objects in a game that takes place on a map can range from $(0, 0)$ to (W, H). If W and H are 1024, then newly visible objects will probably never need more than 10 bits to express each coordinate of their positions, even if the distribution of objects across the map is uniform.

5.4 Justification for Learning Predictors

The reason we go to all this trouble at all is that different fields correspond to different properties of objects in the game world, and those different properties will most likely have highly different dynamics.

For instance, a field representing ammunition in a first person shooter game, a common genre of online games, will typically remain constant for large periods of time, and then change suddenly due to the player firing his weapon or picking up extra ammunition. The vast majority of the time, approximating this field with a constant function (where the constant is equal to the value in the last acknowledged frame) will produce a perfect prediction. In the few cases where ammunition does change, it does not typically make sense to extrapolate based on this change (the fact that you picked up 30 rounds this frame does not mean you will also pick up 30 rounds in the frame after that), so simply sending an error delta once and then assuming a new, greater constant makes sense.

By contrast, a character that was moving in a particular direction over the past several frames will likely continue to move in that direction, so modelling the coordinates of their position with linear functions, built from the last two acknowledged positions, will often produce highly accurate predictions. Whenever the character changes what direction they are moving, nonzero error deltas will be sent, but once they are moving in a consistent direction again (or once they have stopped), the predictor will again be accurate and the error deltas will frequently be zero.

Similarly, a projectile launched into the air will often trace a quadratic path of motion (at least in the vertical coordinate), as they are subject to a constant acceleration term. A predictor which models this quadratic path, based on the projectile's position in the last three frames, will again produce highly accurate predictions, and error deltas would generally only need to be sent for the first few frames (to get the motion started), and again when the projectile struck another object and stopped or bounced off.

The point is that, even if you know the rough dynamics of a particular field, it is difficult to say how the mechanics of the game will affect the distribution of error terms. A fast moving object whose normal kinematics are quadratic in nature, but which frequently impacts other objects and bounces around might best be handled by simply sending error deltas from the last known position, as its movement is too unpredictable within its environment. By learning which predictor to use, we can be sure that we are using the predictor that will produce the most compressible error deltas.

Furthermore, even for a relatively simple engine whose dynamics are known and well understood, the effect of different levels or areas, different scenarios, and different styles of play can change the nature of the information being synchronised at a particular point in time. During a match in a real time strategy game, the dynamics of information will be very different between a period where the players are building up their armies and exploring the map, and a period in which the players are in combat with one another, the former focussing primarily on object creation, discovery, and stable, long term movement, while the latter would have rapid status changes and quick short term movements. A learning system can handle all of this information dynamically.

Finally, we can even provide a close estimate as to how much it will cost the server to synchronise a field (or an object) to the client, which could be useful for servers attempting to manage level-of-detail to match the bandwidth available to each individual client. This information would be almost impossible to arrive at analytically, even to a programmer with access to the complete source text of a game engine.

5.5 Approximating First Derivatives

Dead reckoning, covered in Section 2.3.1, and cubic spline interpolation, covered in Section 2.3.3, are valuable techniques for improving the visual representation of game state on the client. Both of these techniques rely on having information such as velocities or rates of change available on the client for the purposes of extrapolation and smooth interpolation. With our polynomial function approximation, we can provide an approximation of the first derivative of the value of any numeric field essentially for free.

Once again, consider a field for which we have points $(t_0, f(t_0)), \ldots, (t_{n-1}, f(t_{n-1}))$. As before, we wish to model a function $g(t) \approx f(t)$. We can use the first derivative $g'(t)$ as an approximation for $f'(t)$. Recall that $g(t) = UA$, and that A is constant with respect to t. We can use the power rule to derive $g'(t)$ as follows:

$$\begin{aligned}
\frac{d}{dt}g(t) &= \frac{d}{dt}(UA) && (5.12)\\
\frac{d}{dt}g(t) &= \frac{d}{dt}(U)A && (5.13)\\
\frac{d}{dt}g(t) &= \frac{d}{dt}(\begin{pmatrix} 1 & t & t^2 & \ldots & t^{m-1} \end{pmatrix})A && (5.14)\\
&= \begin{pmatrix} \frac{d}{dt}1 & \frac{d}{dt}t & \frac{d}{dt}t^2 & \ldots & \frac{d}{dt}t^{m-1} \end{pmatrix} A && (5.15)\\
&= \begin{pmatrix} 0 & 1 & 2t & \ldots & (m-1)t^{m-2} \end{pmatrix} A && (5.16)\\
&= U'A && (5.17)
\end{aligned}$$

Now, just as before, observe that, for a given frame, being delta-encoded from a known set of frames, both the matrix T, built from the frame numbers of our sample frames, and the matrix U', built from the frame where we want to sample the derivative, are constant. Therefore, we can form a predictor for the derivative of a field as in Equation 5.19.

$$P' = U'T^{-1} \tag{5.18}$$

$$g'(t_n) = P'F \tag{5.19}$$

Note that we are using the exact same F we use when forming the prediction of our base values. Additionally, like P, P' depends only on the time values we are sampling at, and has no dependency whatsoever on any information yielded from a specific field. This means that, for the price of an extra matrix multiplication per predictor per frame, we can form a very quick and reasonably accurate approximation of the first derivative of any numeric field without having to transmit any additional information.

This method can be extended arbitrarily to deliver an approximation of any derivative of our field. We simply repeatedly differentiate the row vector P with respect to t to take successively higher derivatives. Note that, as we are approximating our function with a polynomial, we can only take as many derivatives as we have terms of our polynomial before we are down to a constant, and all subsequent derivatives will be zero.

5.5.1 Faster Derivation

The values we use for t and t_i are on their own somewhat arbitrary, only the relationship between these values is important. If we define $s_i = t_i + k$, we can instead approximate $f(t)$ with Equation 5.20

$$h(t) = \sum_{i=0}^{m-1} b_i s^i = \sum_{i=0}^{m-1} b_i (t+k)^i \tag{5.20}$$

Exactly as before, we take advantage of known values at known times to form a system of equations.

$$\begin{array}{ccc}
\sum_{i=0}^{m-1} b_i(t_0+k)^i & = & f(t_0) \\
\sum_{i=0}^{m-1} b_i(t_1+k)^i & = & f(t_1) \\
\vdots & & \vdots \\
\sum_{i=0}^{m-1} b_i(t_{n-1}+k)^i & = & f(t_{n-1})
\end{array}$$

Once more, we will write this as the matrix equation Equation 5.21, or more succinctly, Equation 5.22.

$$\begin{bmatrix} 1 & (t_0+k) & (t_0+k)^2 & \dots & (t_0+k)^{m-1} \\ 1 & (t_1+k) & (t_1+k)^2 & \dots & (t_1+k)^{m-1} \\ 1 & (t_2+k) & (t_2+k)^2 & \dots & (t_2+k)^{m-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & (t_{n-1}+k) & (t_{n-1}+k)^2 & \dots & (t_{n-1}+k)^{m-1} \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ \vdots \\ b_{m-1} \end{bmatrix} = \begin{bmatrix} f(t_0) \\ f(t_1) \\ f(t_2) \\ \vdots \\ f(t_{n-1}) \end{bmatrix} \tag{5.21}$$

$$SB = F \tag{5.22}$$

We can now use these matrices to express $h(t_n)$ as 5.26.

$$\begin{aligned}
h(t_n) &= \sum_{i=0}^{m-1} b_i s_n^i && (5.23)\\
&= \sum_{i=0}^{m-1} b_i (t_n+k)^i && (5.24)\\
&= \begin{bmatrix} 1 & (t_n+k) & (t_n+k)^2 & \dots & (t_n+k)^{m-1} \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ \vdots \\ b_{m-1} \end{bmatrix} = VB && (5.25)\\
&= VS^{-1}F && (5.26)
\end{aligned}$$

Note that we have not yet explicitly chosen the value of k, which defines matrices V and S. Let us choose $k = -t_n$. Making this substitution, $h(t_n)$ becomes 5.28.

$$h(t_n) = \begin{bmatrix} 1 & (t_n+k) & (t_n+k)^2 & \dots & (t_n+k)^{m-1} \end{bmatrix} S^{-1}F \tag{5.27}$$
$$= \begin{bmatrix} 1 & 0 & 0 & \dots & 0 \end{bmatrix} S^{-1}F \tag{5.28}$$

Note that $VS^{-1}F = h(t_n) = g(t_n) = UT^{-1}F = PF$, so VS^{-1} is simply another way to obtain our predictor P. However, V has only one nonzero element, and essentially boils down to selecting the first row from S^{-1}. Indeed, this is the case. By using not absolute time values, but instead, time offsets relative to the time we wish to approximate, we need only compute one row of our inverse matrix. Furthermore, the values inside that matrix will be smaller, because the base of every exponent will be a small negative value, which is good for avoiding overflow.

More interestingly, however, is that we can obtain the derivative $h'(t)$ in exactly the same way as we did for $g'(t)$, and it amounts to taking the derivative of matrix V instead of matrix U. For instance, see Equation 5.33.

$$\frac{d}{dt}h(t) = \frac{d}{dt}(VB) \tag{5.29}$$
$$= \frac{d}{dt}(V)B \tag{5.30}$$
$$= \frac{d}{dt}(\begin{bmatrix} 1 & (t_n+k) & (t_n+k)^2 & \dots & (t_n+k)^{m-1} \end{bmatrix})B \tag{5.31}$$
$$= \begin{bmatrix} \frac{d}{dt}1 & \frac{d}{dt}(t_n+k) & \frac{d}{dt}(t_n+k)^2 & \dots & \frac{d}{dt}(t_n+k)^{m-1} \end{bmatrix} B \tag{5.32}$$
$$= \begin{bmatrix} 0 & 1 & 2(t_n+k) & \dots & (m-1)(t_n+k)^{m-2} \end{bmatrix} B \tag{5.33}$$

When we substitute $k = -t_n$ into this equation, we get Equation 5.35.

$$h'(t_n) = \begin{bmatrix} 0 & 1 & 2(t_n+k) & \dots & (m-1)(t_n+k)^{m-2} \end{bmatrix} S^{-1}F \tag{5.34}$$
$$= \begin{bmatrix} 0 & 1 & 0 & \dots & 0 \end{bmatrix} S^{-1}F \tag{5.35}$$

So our predictor P' can now be formed by taking the second row of S^{-1}. In fact, this process can be continued. Each subsequent derivative is simply a constant factor multiplied by a row from S^{-1}, until you run out of rows, at which point, all further derivatives are zero.

5.6 Improved Derivatives

The methods outlined above approximate the derivatives of field values using the same function used to make predictions of a field, however, they leave out one very useful piece of information: the actual value of the field at the time we are approximating its derivative. Once we have made the initial prediction, received the error delta, and reconstructed the value of a field at the current frame, we can use this value to make a second, more up-to-date approximation function, from which to approximate derivatives.

There is really very little involved in doing this. Instead of using m prior values at m prior points in time, we will use $m-1$ prior values, as well as the current value at the current time. To do this, we must modify our T matrix (or its time-shifted counterpart, the S matrix), to incorporate this new set of time values. Since the frame that we are approximating is known to us ahead of time, these special derivative predictors can still be precalculated, once per frame. However, when taking the dot product between these predictor vectors and our sample vector, we must make sure to use a modified sample vector that contains the current value.

5.7 Explicit Derivatives

In some cases, instead of approximating the derivatives from our polynomial prediction functions, we may want to send an explicitly chosen derivative. This might be necessary if some aspect of our client-side animation system relies on having very accurate values for field derivatives. In this case, we have some interesting opportunities. First, we can make predictions about the derivative of a field either from previous derivatives (treating the derivative like an ordinary value in and of itself), or from previous values (making an approximation of the derivative via any of the methods outlined above), and encode the error delta between the actual derivative and whichever prediction we want. We can even learn which of these predictors is most effective, based on the distribution of error deltas they produce.

However, what is more interesting is that we can use our knowledge of the precise first derivative to approximate higher order polynomial functions from fewer data points. For instance, knowing the values and derivatives of only two points is sufficient to uniquely describe a cubic function between those two points. For fields whose behavior is best approximated with higher order polynomial functions, we can form those functions using more recent data. This helps to minimise the number of nonzero error deltas that need to be sent when something unexpected occurs.

As an aside, this principle of using more recent data when available is the reason we did not devote a significant amount of discussion to linear regression techniques earlier in the chapter. Although sample points far in the past are usually readily available, they are typically not as relevant as sample points closer to the present.

Let us assume, as before, that we are attempting to approximate our unknown function $f(t)$ with an order-m polynomial, $g(t) = \sum_{i=0}^{m-1} a_i t^i$. Consider any point t_j for which we have both $f(t_j)$ and the actual value (not an approximation) of $f'(t_j)$. This point in fact gives us two constraints which we can use to form $g(t)$.

$$g(t_j) = \sum_{i=0}^{m-1} a_i t_j^i \tag{5.36}$$

$$g'(t_j) = \sum_{i=0}^{m-1} a_i i t_j^{i-1} \tag{5.37}$$

Table 5.1: High-order derivative prediction scheme

Step	Eqn1	Eqn2	Eqn3	Eqn4	Predict
1	$g(t_0) = f(t_0)$	$g'(t_0) = f'(t_0)$	$g''(t_0) = f''(t_0)$	$g'''(t_0) = f'''(t_0)$	$f'''(t_1)$
2	$g(t_0) = f(t_0)$	$g'(t_0) = f'(t_0)$	$g''(t_0) = f''(t_0)$	$g'''(t_1) = f'''(t_1)$	$f''(t_1)$
3	$g(t_0) = f(t_0)$	$g'(t_0) = f'(t_0)$	$g''(t_1) = f''(t_1)$	$g'''(t_1) = f'''(t_1)$	$f'(t_1)$
4	$g(t_0) = f(t_0)$	$g'(t_1) = f'(t_1)$	$g''(t_1) = f''(t_1)$	$g'''(t_1) = f'''(t_1)$	$f(t_1)$

Thus, we can essentially form our T matrix as before, but this time, the rows of T can correspond either to Equation 5.36 or to Equation 5.37. Whichever equation we pick, we must make sure that our sample vector F is filled with the values or derivatives corresponding to the appropriate point in time. For instance, we could set up our matrices as in Equation 5.38.

$$\begin{bmatrix} 1 & t_0 & t_0^2 & t_0^3 \\ 1 & t_1 & t_1^2 & t_1^3 \\ 0 & 1 & 2t_0 & 3t_0^2 \\ 0 & 1 & 2t_1 & 3t_1^2 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} f(t_0) \\ f(t_1) \\ f'(t_0) \\ f'(t_1) \end{bmatrix} \tag{5.38}$$

Once more, we can solve for $A = T^{-1}F$, and thus form our predictor $P = UT^{-1}$, or our derivative predictor $P' = U'T^{-1}$. However, we must take special care to note that the format of F is specific to how we lay out our matrix.

We have plenty of options as to exactly how we can form our T matrix. If we are attempting to model a quadratic formula, we could use three values, two values and one derivative, or one value and two derivatives. We cannot use three derivatives as the leftmost column of T would be entirely zero, leading to a singular matrix (this makes sense, three past velocity readings of a vehicle would give you no idea where its current position might be).

We can also use known values of higher derivatives to make predictions. One could imagine a situation as follows, where we are synchronising a field's values at $t1$ based on known values at $t0$, and we are predicting $f(t)$ with a cubic polynomial $g(t)$. At each step, we use the most recent information about each field to make a prediction in order to delta-encode a new piece of information. One possible scheme for synchronising a field's information is shown in Table 5.1.

It is easy to verify that despite the apparent interdependencies, each step still boils down to a dot product between a vector of known sample values and a constant predictor vector, and that the predictor vector can be calculated in advance, once per frame.

The hope with such a setup as above would be that, at each step, the error values are so small that they can be transmitted cheaply, and the availability of higher order information enables subsequent predictions about lower order values to be more accurate, leading to further small errors.

In practise, it is rare for more than first order derivatives about fields to even be available on the server, much less accurate. Off-the-shelf physics engines, for instance, will usually make the velocities of objects available to the user, but acceleration is complicated, as some acceleration is handled by explicit forces such as gravity or spring forces, while other acceleration is handled by impulses (direct instantaneous additions onto momentum, such as during elastic collisions or resolution of constraints). For other types of motion, such as that driven by animations, frame-based hierarchies, or character controllers, it may simply be easier to let polynomial predictors crunch the basic values of a field, and approximate derivatives for you.

5.8 Custom Predictive Contracts

Everything discussed in this chapter so far has been completely agnostic to the actual content of the numeric fields being synchronised. This is why we have described these methods as "semi-content aware". They are designed to learn about the individual characteristics of each field in order to develop an efficient encoding scheme, but do not require any actual input on the part of the game or simulation programmer (with the obvious exception that methods for utilising explicit derivatives need to have those derivatives provided by the server).

However, it is also possible to incorporate application-specific information into this prediction algorithm. To accomplish this, rather than using values sampled directly from a past snapshot, some deterministic algorithm is provided by the game programmer to "fast forward" a past snapshot to the current time, and values are sampled from their predicted state. As an example, consider mobile objects moving around a three dimensional game environment. In most games, the majority of the environment is static. The basic terrain of an area, and many structures and objects within it, are completely static, and do not change within a single play session. It stands to reason that a significant number of collisions that affect objects will occur with the static environment.

If we know that the last acknowledged snapshot has an object at a particular position, with a particular velocity, we could integrate it forward for the amount of time that has elapsed between that snapshot and the present, handling collisions using our actual game logic. While we must be careful that this "fast forward" algorithm takes into account only static data (such as the level geometry) and data available in that snapshot (the only information we can be certain is available to the client), then we can make our predictions based not on historical values, but on the forecasted present values based on historical values.

Under this system, events such as a rocket striking a wall and exploding require no network traffic, since both server and client will predict the impact with the wall, and its consequences. The server can confirm that, in the actual simulation, the rocket behaved exactly as expected, and send a very cheap zero delta.

On the other hand, if the rocket struck a character or mobile object that the client did not expect to be there (perhaps the character's motion changed since the last acknowledged frame), then the ground truth on the server will be different from the state predicted by both client and server, and an error delta can be sent describing the differences.

There are two obvious downsides to this technique. First, it is expensive, far more so than simple polynomial predictors that only require past states to be sampled. These calculations would need to be performed for every snapshot of every field they affect, every frame, although they may partially make up for it by requiring fewer snapshots to obtain a good prediction. Second, and more importantly, the predictions only work well if they match what is actually going on on the server, which brings up issues of maintenance as well as performance.

However, this technique might be part of a larger system, akin to UnrealScript[16] (Epic Games), which allows entity logic to be written in one place, and run on both client and server when appropriate. Indeed, one of the design goals of UnrealScript is to facilitate network programming.

While extremely sophisticated predictive contracts start to venture back into co-simulation territory, with the performance and mechanical transparency pitfalls it entails, simpler predictive contracts that catch easy collisions or enforce known ranges or constraints on fields might eliminate a class of situations in which error deltas need to be sent without incurring too much runtime cost or making the application too complex.

Chapter 6

Efficient Integer Encoding

The work in Chapter 5 allows us to reduce snapshots to a series of error deltas between ground truth values and reasonably accurate predictions. These error deltas technically take up the exact same amount of space as the fields they are meant to replace. After all, the difference between two 32-bit numbers is another 32-bit number, while the difference between two 16-bit number is likewise, another 16-bit number.

However, if our predictors are at all effective, then we should be seeing some non-uniform distribution in the values these error deltas take on. Recall that we are able to form several predictors and record the distribution of error terms for each. Furthermore, in Chapter 3 we showed that entropy coding, particularly arithmetic coding, is a natural fit for encoding a series of symbols for which the conditional probability of each symbol is known.

6.1 Simple Arithmetic Coding using Frequencies

For our purposes, we will encode each error delta, which corresponds to the difference between actual and predicted values for a known field, using frequencies drawn from the recorded distribution of error deltas for that field. Recall that the optimal way of encoding symbols is to use the probability that a symbol appears at a point in the stream, conditioned on all previously encoded symbols having appeared in the order that they did. Instead, we are essentially just using the probability that the symbol in question was drawn from its alphabet according to its distribution. By doing this, we are essentially making the assumption that the error deltas corresponding to different fields and different objects are independent within a single frame. That is, we are assuming that $P(s_n|s_1, \ldots, s_{n-1}) = P(s_n)$.

The assumption that error terms corresponding to different fields and different objects are independent is obviously false. For any environment that we care about, there will be both correlations among objects (a large group of objects might start moving at the same time, in the same direction), and outright causations (two objects may collide, thus altering their positions in "unpredictable" ways relative to their prior motion).

However, arithmetic coding does not **require** us to use the correct conditional probabilities at any point during the algorithm. It merely guarantees that if we **do** use the correct conditional probabilities, the length of our encoded message will be close to the Shannon entropy of our plaintext message. The closer we can get to using exact conditional probabilities, the closer we will be to Shannon entropy.

Remember that error deltas are formed by taking the different between an actual value held by a field, observed on the server, and a value deterministically predicted on both server and client based on the past values held by a field. Thus, any given error delta will tend to fall into one of three cases:

1. $\delta = 0$: The prediction was exactly correct.
2. Small δ: The prediction modelled the correct phenomena, but was slightly off in calculation, so a small corrective term must be sent.
3. Large δ: Some event occurred which changed the value of a field in a way that could not have been predicted from past values.

Essentially, by learning the distribution of error deltas, we are learning about how often our predictions are correct, what distribution of corrective terms we might need to use to keep our view of the field accurate, and with what frequency and magnitude "unexpected events" occur.

The important thing to consider is that, by making predictions based on the past value of the field, and encoding error deltas, we are already taking into account a lot of information about the dynamics of a field. Indeed, this is where the biggest compression gains in our approach come from. Furthermore, by learning only a single distribution for each (field, predictor) pair, we ensure that every single observed value contributes to this distribution. If we were using some manner of conditional distribution, then observed values would contribute only one conditional distribution at a time, out of potentially many. This allows us to learn distributions very quickly.

6.2 Using Approximate Distributions

Consider a value x drawn from a random variable X whose probability mass function, $f_X(x)$, is unknown. Assume we have a second random variable X', whose known probability mass function, $f'_X(x)$, is similar but not necessarily equal to $f_X(x)$. What is the expected cost to encode x, via arithmetic coding, as though it were drawn from X' instead of X?

Recall from Equation 3.22 that the expected cost to represent a message $E[c(m)]$ is determined by $-\sum_{i=1}^{k} P(m_i)log_2(l_i) + \epsilon$. In this case, each m_i will correspond to a value x drawn from X. $P(m_i) = P(X = x) = f_X(x)$ and we will choose $l_i = P(X' = x) = f_{X'}(x)$. Thus, our expected cost to represent x can be derived as in Equation 6.1.

$$E[c(x)] = -\sum_{x \in X} f_X(x) log_2(f_{X'}(x)) + \epsilon \tag{6.1}$$

Recall that the actual Shannon entropy of X is given by $-\sum_{x \in X} P(X = x) log_2(P(X = x))$. Thus, the penalty, or difference between our encoding of x and the optimal encoding of x can be derived as in Equation 6.6.

$$\begin{aligned}
E[c(x) - c^*(x)] &= -\sum_{x \in X} f_X(x) log_2(f_{X'}(x)) + \epsilon - (-\sum x \in X f_X(x) log_2(f_X(x))) &(6.2)\\
&= -\sum_{x \in X} (f_X(x) log_2(f_{X'}(x)) - f_X(x) log_2(f_X(x))) + \epsilon &(6.3)\\
&= -\sum_{x \in X} f_X(x)(log_2(f_{X'}(x)) - log_2(f_X(x))) + \epsilon &(6.4)\\
&= -\sum_{x \in X} f_X(x)(log_2(\frac{f_{X'}(x)}{f_X(x)})) + \epsilon &(6.5)\\
&= \sum_{x \in X} f_X(x)(log_2(\frac{f_X(x)}{f_{X'}(x)})) + \epsilon &(6.6)
\end{aligned}$$

Observe that, for all $x > 1$, $log(x) > 0$. Therefore, $log_2(\frac{f_X(x)}{f_{X'}(x)}) > 0$ whenever $f_{X'}(x) < f_X(x)$. That is, we will incur a penalty any time we underestimate the probability of a symbol appearing.

The logarithm of the ratio between the actual probability and the assumed probability represents the number of additional bits that will be needed to describe our coded number passing through a smaller-than-appropriate subinterval. As each bit doubles the precision of our number, we only need the logarithm of the ratio in extra bits to handle this case.

This additional cost is multiplied by $f_X(x)$, which indicates that we are penalised this constant amount in bits each time the symbol arises, which it does with probability $f_X(x)$.

Observe that any time we overestimate, instead of underestimate, the probability of a symbol appearing $f_{X'}(x) > f_X(x)$, we actually receive a “discount”, because we can represent the symbol more cheaply. However, since we make gains on overestimates and losses on underestimates, we are making gains on symbols which actually appear less frequently than we believe, while incurring losses on symbols which actually appear more frequently than we believe. This is why we cannot design a distribution which outperforms the true probability distribution, and why we cannot achieve an expected cost below the Shannon entropy.

6.3 Integer Bucketisation

Most applications of arithmetic coding apply to streams of symbols (as in text compression) or to small numbers, such as bytes or byte differences (as in image and video compression). When the number of possible symbols to encode is small, such as the range of a byte ($2^8 = 256$ possible values), we can simply learn the frequencies of each symbol in a small table and make use of them during encoding.

However, the output of our prediction system is a collection of integer deltas, which can take on substantially more values. 16-bit integers can take on $2^{16} = 65536$ values, while 32-bit integers can take on $2^{32} \approx 4294967296$ values. To explicitly learn a frequency distribution for a 32-bit error delta, even if we used only one byte per frequency, would eat up 4 GB on the spot, to say nothing of how exceptionally cache-unfriendly any attempt to use the table would be.

To circumvent this problem, we can partition the range of an integer data type into a set of intervals, such that each integer value maps to exactly one interval. We can consider this scheme a bijection between $x \in \{-2^{k-1}, \ldots, 2^{k-1} - 1\}$ and a tuple $\{(i_x, j_x)\}$ where i_x is the index of an interval and j_x is the position of x within that interval.

To make this explicit, consider n intervals defined by the values $a_0, a_1, \ldots, a_n$ such that $a_0 = -2^{k-1}$, $a_n = 2^{k-1}$, and interval $I_i = [a_i, a_{i+1})$. We define i_x such that $x \in I_{i_x}$ and j_x such that $x = a_{i_x} + j_x$.

Consider x and i_x to be samples drawn from random variables X and I respectively. The the cumulative distribution function F_X be defined by Equation 6.7.

$$P(a \leq X < b) = F_X(b) - F_X(a) \tag{6.7}$$

Thus, the probability mass function f_X can be derived as in Equation 6.8.

$$f_X(x) = P(X = x) = F_X(x+1) - F_X(x) \tag{6.8}$$

Observe that $I = i$ only when $a_i \leq X < a_{i+1}$. Therefore, the probability mass function f_I can be derived as in Equation 6.9

$$f_I(i) = P(a_i \leq X < a_{i+1}) = F_X(a_{i+1}) - F_X(a_i) \tag{6.9}$$

Now consider a separate random variable X' with corresponding random variable I', such that $f_{X'}(x) = k_{i_x}$, where $\{k_0, k_1, \ldots, k_{n-1}\}$ are constants, and $f_{I'}(i) = f_I(i)$. That is, the probability distribution of values within any particular bucket is uniform, but the distribution of values falling into specific buckets is identical between X and X'. We can define $f_{I'}(i)$ as in Equation 6.14.

$$\begin{aligned} f_{I'}(i) &= \sum_{x=a_i}^{a_{i+1}-1} f_{X'}(x) && (6.10) \\ &= \sum_{x=a_i}^{a_{i+1}-1} k_{S(x)} && (6.11) \\ &= \sum_{x=a_i}^{a_{i+1}-1} k_i && (6.12) \\ &= k_i \sum_{x=a_i}^{a_{i+1}-1} 1 && (6.13) \\ &= k_i(a_{i+1} - a_i) && (6.14) \end{aligned}$$

Since we defined $f_{I'}(i) = f_I(i)$, we can combine Equation 6.9 and Equation 6.14 to yield Equation 6.17.

$$\begin{aligned} f_{I'}(i) &= f_I(i) && (6.15) \\ k_i(a_{i+1} - a_i) &= f_I(i) && (6.16) \\ k_i &= \frac{f_I(i)}{a_{i+1} - a_i} && (6.17) \end{aligned}$$

This finally gives us the probability mass function of X', as in Equation 6.18.

$$f_{X'}(x) = \frac{f_I(i)}{a_{i+1} - a_i}, i = S(x) \tag{6.18}$$

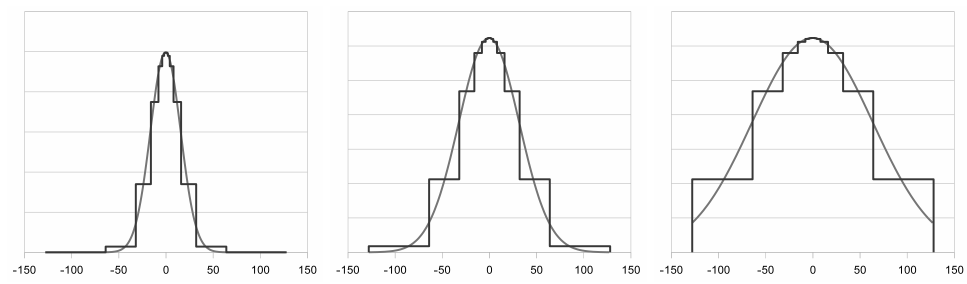
What this means is that we can learn probability mass function of I, which has only n possible values, and then use that distribution to approximate the unknown distribution of X with our simplified, bucketised distribution X'.

6.4 Bucketing Schemes

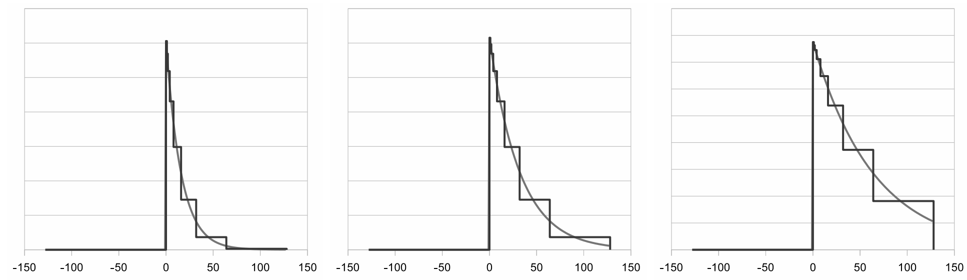
We have a number of factors to consider in designing our bucketing scheme.

1. Flexibility: The bucketing scheme should be capable of approximating a wide variety of distributions. Since we will only be learning $f_I(i)$, the probability mass function of the bucket index, we need a bucketing scheme that is flexible enough to handle whatever type of distribution we encounter.

2. Detail: The bucketing scheme should use small buckets where appropriate to be able to closely approximate areas where the original distribution is likely to contain a lot of probability mass. The terms of our approximation error, given in Equation 6.6, are proportional to the amount of probability mass in each erroneous term, so we want to be sure to accurately represent areas of high probability.



(a) $\sigma = 16$-Normal (b) $\sigma = 32$-Normal (c) $\sigma = 64$-Normal



(d) $\lambda = \frac{1}{16}$-Exponential (e) $\lambda = \frac{1}{32}$-Exponential (f) $\lambda = \frac{1}{64}$-Exponential

Figure 6.1: Bucketised normal and exponential distributions

3. Speed: We should have a fast way of computing the bucket index of a given integer. A direct lookup table is not an option, and ideally we would like something faster than a simple iteration through buckets.

4. Size: The more buckets we use, the more memory we will consume to represent a particular distribution. Consider that we must learn multiple distributions for each field, and store multiple copies of our overall dataset to handle frame-by-frame learning. Even small differences in the size of our bucketing scheme can affect memory usage and cache coherency in a big way.

In order to address these issues, we examined a particular family of bucketing schemes, in which small buckets were placed around zero, and exponentially larger buckets were placed sequentially in both directions. This allows for distributions of widely varying scales to be modelled, while ensuring that there is always extra expressiveness to approximate the distribution around zero. Furthermore, exponential patterns are self-similar across scales, which facilitates fast lookup (a compact lookup table can be used to bucketise values in the range of $[0, 256)$, and shifts and offsets can handle larger values).

Figure 6.1 show how an exponentially patterned bucketing scheme can be used to approximate a variety of normal and exponential distributions. Remember that any time the approximation underestimates the true probability distribution, it becomes more expensive to represent those values, and any time the approximation overestimates the true probability distribution, it becomes cheaper to represent those values, but that both effects are scaled by the amount of probability mass in the values in question.

Thus, the only remaining variable is size, specifically, how many buckets to use. We started with a 64 bucket scheme, 32 for negative numbers, 32 for nonnegative numbers, where each bucket roughly corresponded to all numbers whose most significant bit was in a specific location. Hence, one bucket would contain 0, one would contain 1, and then $[2, 3], [4, 7], [8, 15], \ldots$. Of note is the fact that -1, 0, and $+1$ (as well as -2) each have a bucket entirely to themselves. Thus, we can exactly represent the probabilities that a prediction is accurate, or that it is off-by-one.

Four other bucketisation schemes were developed, which made use of 124, 94, 48, and 32 buckets, respectively. The 32 bucket policy was created by simply merging every pair of adjacent buckets from the 64 bucket policy. The 124, 94, and 48 bucket policies were created by placing bucket boundaries at exponential locations. The exact boundaries for these bucketisation policies is given in Chapter B.

In order to determine how effective these bucketisation policies were at capturing various types of distributions, we simply ran them through a battery of tests in which each scheme was used to approximate a known statistical distribution. The cost of encoding a variable drawn from that distribution with the approximation produced by the bucketing scheme was computed, and compared with the calculated entropy of the original distribution, to determine the actual cost in bits incurred by approximating the distribution with each particular bucketisation schema.

The distributions examined were a family of normal and exponential distributions. The normal distributions were specified with $\mu = 0$ and varying σ. The exponential distributions were given varying λ. These parameters were varied exponentially to test the ability of each bucketing scheme to handle distributions at different scales. The results are shown in Table 6.1 and Table 6.2. Note that, across the range of distributions tested, the cost of using the 64 bucket scheme costs less than 0.1 bit on top of Shannon entropy. These data show that for well-behaved distributions, the cost of using a bucketing scheme is close to negligible.

6.5 Estimating Distribution Costs

With a known bucketing scheme, we can record the distribution of values falling into any of the different buckets, and we can quickly estimate the cost of expressing an integer via our bucketing scheme. Recall that the cost of approximating an unknown distribution X via a known, approximate distribution X' is given by Equation 6.1, and that $f_{X'}(x)$ is given by Equation 6.18. We can derive a fast cost approximation as shown by Equation 6.26.

Table 6.1: Effectiveness of Bucketing Schemes on $(\mu = 0, \sigma)$-Normal Distributions

σ	Entropy	124 Buckets	94 Buckets	64 Buckets	48 Buckets	32 Buckets
2^{-2}	1.001	1.001	1.001	1.001	1.001	2
$2^{-1.5}$	1.043	1.043	1.043	1.043	1.048	2
2^{-1}	1.268	1.268	1.268	1.268	1.312	2.001
$2^{-0.5}$	1.658	1.658	1.658	1.663	1.785	2.05
2^{0}	2.105	2.105	2.107	2.136	2.245	2.339
$2^{0.5}$	2.577	2.579	2.592	2.629	2.7	2.877
2^{1}	3.062	3.073	3.09	3.129	3.22	3.405
$2^{1.5}$	3.555	3.569	3.587	3.627	3.699	3.806
2^{2}	4.051	4.067	4.087	4.128	4.182	4.263
$2^{2.5}$	4.549	4.569	4.585	4.627	4.697	4.859
2^{3}	5.048	5.07	5.083	5.128	5.174	5.401
$2^{3.5}$	5.548	5.568	5.584	5.627	5.689	5.805
2^{4}	6.047	6.068	6.085	6.128	6.197	6.263
$2^{4.5}$	6.547	6.568	6.585	6.627	6.674	6.859
2^{5}	7.047	7.068	7.084	7.128	7.194	7.401
$2^{5.5}$	7.547	7.568	7.584	7.627	7.683	7.805
2^{6}	8.047	8.068	8.084	8.128	8.177	8.263
$2^{6.5}$	8.547	8.568	8.584	8.627	8.698	8.859
2^{7}	9.047	9.068	9.084	9.128	9.175	9.401
$2^{7.5}$	9.547	9.568	9.584	9.627	9.686	9.805
2^{8}	10.05	10.07	10.08	10.13	10.19	10.26
$2^{8.5}$	10.55	10.57	10.58	10.63	10.67	10.86
2^{9}	11.05	11.07	11.08	11.13	11.19	11.4
$2^{9.5}$	11.55	11.57	11.58	11.63	11.68	11.81
2^{10}	12.05	12.07	12.08	12.13	12.18	12.26
$2^{10.5}$	12.55	12.57	12.58	12.63	12.7	12.86
2^{11}	13.05	13.07	13.08	13.13	13.17	13.4
$2^{11.5}$	13.55	13.57	13.58	13.63	13.69	13.81
2^{12}	14.05	14.07	14.08	14.13	14.19	14.26
$2^{12.5}$	14.55	14.57	14.58	14.63	14.67	14.86
2^{13}	15.05	15.07	15.08	15.13	15.2	15.4
$2^{13.5}$	15.55	15.57	15.58	15.63	15.68	15.81
2^{14}	16.05	16.07	16.08	16.13	16.18	16.26
$2^{14.5}$	16.55	16.57	16.58	16.63	16.7	16.86
2^{15}	17.05	17.07	17.08	17.13	17.17	17.4
$2^{15.5}$	17.55	17.57	17.58	17.63	17.69	17.81
2^{16}	18.05	18.07	18.08	18.13	18.19	18.26

Table 6.2: Effectiveness of Bucketing Schemes on $(\lambda = \frac{1}{\mu})$-Exponential Distributions

μ	Entropy	124 Buckets	94 Buckets	64 Buckets	48 Buckets	32 Buckets
2^{-2}	0.1343	0.1343	0.1343	0.1346	0.1503	1.005
$2^{-1.5}$	0.3442	0.3442	0.3444	0.3467	0.3852	1.039
2^{-1}	0.6614	0.6616	0.6626	0.6704	0.7284	1.161
$2^{-0.5}$	1.057	1.058	1.062	1.077	1.141	1.418
2^{0}	1.501	1.504	1.511	1.532	1.596	1.79
$2^{0.5}$	1.972	1.978	1.987	2.012	2.073	2.219
2^{1}	2.458	2.466	2.477	2.504	2.559	2.674
$2^{1.5}$	2.95	2.96	2.972	3.001	3.051	3.161
2^{2}	3.446	3.459	3.47	3.5	3.546	3.668
$2^{2.5}$	3.945	3.958	3.969	3.999	4.044	4.165
2^{3}	4.444	4.457	4.468	4.499	4.544	4.651
$2^{3.5}$	4.943	4.957	4.968	4.999	5.043	5.152
2^{4}	5.443	5.457	5.468	5.499	5.543	5.664
$2^{4.5}$	5.943	5.957	5.968	5.999	6.042	6.163
2^{5}	6.443	6.457	6.468	6.499	6.541	6.651
$2^{5.5}$	6.943	6.957	6.968	6.999	7.042	7.151
2^{6}	7.443	7.457	7.468	7.499	7.542	7.664
$2^{6.5}$	7.943	7.957	7.968	7.999	8.042	8.163
2^{7}	8.443	8.457	8.468	8.499	8.542	8.651
$2^{7.5}$	8.943	8.957	8.968	8.999	9.041	9.151
2^{8}	9.443	9.457	9.468	9.499	9.542	9.664
$2^{8.5}$	9.943	9.957	9.968	9.999	10.04	10.16
2^{9}	10.44	10.46	10.47	10.5	10.54	10.65
$2^{9.5}$	10.94	10.96	10.97	11	11.04	11.15
2^{10}	11.44	11.46	11.47	11.5	11.54	11.66
$2^{10.5}$	11.94	11.96	11.97	12	12.04	12.16
2^{11}	12.44	12.46	12.47	12.5	12.54	12.65
$2^{11.5}$	12.94	12.96	12.97	13	13.04	13.15
2^{12}	13.44	13.46	13.47	13.5	13.54	13.66
$2^{12.5}$	13.94	13.96	13.97	14	14.04	14.16
2^{13}	14.44	14.46	14.47	14.5	14.54	14.65
$2^{13.5}$	14.94	14.96	14.97	15	15.04	15.15
2^{14}	15.44	15.46	15.47	15.5	15.54	15.66
$2^{14.5}$	15.94	15.96	15.97	16	16.04	16.16
2^{15}	16.44	16.46	16.47	16.5	16.54	16.65
$2^{15.5}$	16.94	16.96	16.97	17	17.04	17.15
2^{16}	17.44	17.46	17.47	17.5	17.54	17.66

$$
\begin{aligned}
E[c(x)] &= -\sum_{x\in X} f_X(x)log_2(f_{X'}(x)) + \epsilon && (6.19)\\
&= -\sum_{i\in I}(\sum_{x\in[a_i,a_{i+1})} f_X(x)\log_2(f_{X'}(x))) + \epsilon && (6.20)\\
&= -\sum_{i\in I}(\sum_{x\in[a_i,a_{i+1})} f_X(x)\log_2(\frac{f_I(i)}{a_{i+1}-a_i})) + \epsilon && (6.21)\\
&= -\sum_{i\in I}(\sum_{x\in[a_i,a_{i+1})} f_X(x)(\log_2(f_I(i)) - log_2(a_{i+1}-a_i))) + \epsilon && (6.22)\\
&= -\sum_{i\in I}((\log_2(f_I(i)) - \log_2(a_{i+1}-a_i))\sum_{x\in[a_i,a_{i+1}))} f_X(x)) + \epsilon && (6.23)\\
&= -\sum_{i\in I}((\log_2(f_I(i)) - \log_2(a_{i+1}-a_i))f_I(i)) + \epsilon && (6.24)\\
&= \sum_{i\in I}(-f_I(i)\log_2(f_I(i)) + f_I(i)\log_2(a_{i+1}-a_i)) + \epsilon && (6.25)\\
&= \sum_{i\in I} f_I(i)(\log_2(a_{i+1}-a_i) - \log_2(f_I(i))) + \epsilon && (6.26)
\end{aligned}
$$

Note that the first term inside the summation in Equation 6.25 represents the entropy of the variable I, and corresponds to the cost of encoding into which bucket a particular value falls. The second term corresponds to the cost to encode a value drawn from a uniform distribution whose domain is the subinterval assigned to the bucket in question. Furthermore, this cost, represented by $\log_2(a_{i+1} - a_i)$, depends only on the interval assigned to the bucket, and can in fact be precomputed. Thus, computing the expected amount of memory used to encode an integer via a bucketing scheme is only slightly more expensive than computing the entropy of the bucket index.

This is of chief important because the techniques outlined in Chapter 5 rely on learning **multiple** distributions, one for the error terms between the actual values encountered and the values predicted by each of the different polynomial predictors. What we want is to be able to frequently and cheaply evaluate which predictor is producing the distribution of error terms that is cheapest to represent, and then use that predictor when transmitting values from server to client.

By doing this separately for each field we are interested in, we can take advantage of the inherent behavior of each field to create exceptionally cheap delta encodings for each one.

6.5.1 Distributions in Practise

The actual distributions we see in practise are quite varied, and do not necessarily comply with known statistical distributions like normal or exponential distributions.

For instance, “motion” related fields should have a large amount of mass at zero, for when the prediction is correct, additional mass near zero, for handling slight round-off errors or cases where the dynamics aren’t exactly smooth, and then a small amount of mass that extends outward to certain ranges. This mass represents all of the rare events that cause an object’s motion to change quickly.

The exact amount of mass used to represent these events will roughly equal the probability of one of these events occurring between the last acknowledged frame and the current frame, and the range to which the mass extends will be related to the amount an object's motion can change in that time frame.

On the other hand, fields corresponding to "health" or "supplies" of some sort might have more interesting distributions. These fields might stay constant for long periods of time, and suddenly changed by fixed amounts, according to the rules of the game. Ammunition might decrease only by one, as individual rounds are expended, but occasionally increase by twenty as a box of ammunition is picked up. Health might decrease by specific numbers corresponding to the damage imposed by different enemy attacks, and increase by one periodically as a character healed.

It is for these reasons that it is beneficial to learn a general probability distribution, rather than learning parameters to known common distributions, such as normal or exponential. It is always particularly important to be able to represent the probability of the "zero" symbol (and perhaps a few values near zero) separately, in order to indicate correct predictions or unchanged values. Similarly, it is beneficial to learn the distribution of positive and negative numbers separately, since different dynamics may be responsible for increasing and decreasing a field.

However, even though common distributions like normal or exponential distributions are unable to fully describe most fields, it is still beneficial to know that our bucketing scheme can represent them well, as certain parts of our distributions may resemble them, such as the probability of symbols other than 0 for a motion based field (which might look like a normal distribution), or the negative numbers for a field representing health (as the distribution of damage caused by attacks might resemble an exponential distribution).

6.6 Recency Weighted Frequency Estimates

By simply recording the bucket index of each value as that value is encountered, we will build up frequency tables whose fractional probabilities (the frequency of a symbol divided by the total number of symbols recorded) will converge to an estimate of the unconditional probability distribution over bucket indices. If the distribution is actually static, then this is exactly what we want. However, in some cases, the actual probability distribution over values, and hence, over buckets, changes at runtime.

As an example, consider a real time strategy game in which we are synchronising the properties of units. At the start of the game, each player likely only has access to ground units, which move slowly, with frequent changes in direction, as they pathfind around the map. However, later on in the game, players may unlock the technology to build air units, which can travel more quickly, but with less frequent stopping and changing direction. These factors would influence the runtime distribution of error deltas in predicted positions.

It is very simple to introduce a decay effect in the stored frequencies, to treat data gathered in the past as less relevant than data gathered in the present. All we need to do is multiply every frequency in a frequency table by some factor $\gamma < 1$ at regular intervals, perhaps every frame, or every second. Since each frequency in the table is scaled by γ, the sum of all values within the table will likewise be scaled by γ, and the ratio of a particular frequency to the total number of observed values remains the same. Thus, scaling a frequency table by γ does not affect any of the actual probabilities stored in the table. However, future additions to the frequency table will now be "worth" slightly more, and it will be easier for future data to overcome trends set in the past.

It is worth noting that, if integer frequencies are used, a scalar multiplication by γ will most likely be performed via an integer multiplication followed by an integer division, representing a rational decay factor, and the latter operation will introduce some truncation error. While the effect of this cannot be outright avoided, it can be minimised by storing fixed-point frequencies. As previously noted, scaling an entire table by a constant factor does not change the probability values represented by the ratio of one frequency to the table's recorded total value. Thus, we can introduce a scaling factor when populating our frequency tables. Each observed value can be recorded by incrementing a frequency by, for instance, 256, instead of 1. This essentially gives us eight bits to use to record a "fractional" component to our frequencies, which will be useful when decaying the frequencies over time.

6.7 Distribution Dataset Archiving

If the distribution is to be learned on-the-fly, then it must be learned simultaneously and identically between server and client. This is not particularly hard to do. Simply archive the statistical dataset per-frame on both client and server in the same way that object snapshots are archived per-frame.

While encoding an update message, the server can clone the dataset from the most recently acknowledge frame, modify it by learning from the data present in the update, and then archive the new dataset alongside the snapshots for that frame. When the client decodes the update message, it simply clones its own identical dataset corresponding to the same frame, modifies it by learning from the data present in the update, and then archives the once more identical dataset alongside its own snapshots for the frame. In this way, the server and client will have identical copies of all statistical distributions at each frame which the client is able to reconstruct.

Note that, fundamentally, if learning is to be done on the fly, then the server must learn separately for each client, since clients will not be receiving the same data at the same times and their distributions will naturally go out of synch. This is not as bad as one might assume at first.

One advantage of learning separately for each client is that no information is leaked by the server about how many other clients there are, or any data corresponding to any of those clients.

Another, more significant advantage is that the distributions learned for each client can match not only that client's specific game experience, but that client's specific network connection to the server. A client with a high latency connection will see more frequent large values, as changes must be sent multiple times before they are received and acknowledged, while a client with a low latency connection will see much more probability mass clustered tightly around zero, as changes are transmitted and acknowledged quickly, and do not need to be sent as many times.

Chapter 7

Experiments

A number of experiments were performed to compare the effectiveness of my method to alternative methods for synchronising data in a server-client setting. As I did not have a robust network of computers around the world with which to test, most tests were run on simulated network environments, which introduced artificial latency and packet loss at user-specified levels.

Latency was generally assumed to be a fixed delay in the round trip time between a message being sent from one peer to another, and a reply being received from that peer. The logical behavior of the server-client system is unaffected by whether the latency is incurred by the initial message, the reply, or some combination of both, only that a specific amount of time elapses between sending a message and the earliest possible point at which the message can arrive.

Packet loss was treated as a fixed probability of any particular packet not reaching its destination. Packet loss was applied both to server-to-client update messages and to client-to-server acknowledgements, thus, a round-trip exchange actually has two opportunities to be dropped. In this case, the logical behavior of the server-client system is affected by where the drop occurs. Data transmitted from the server to the client which is dropped is lost forever, however, acknowledgements from the client to the server which are dropped can be resent the next frame, and the server can make better decisions about delta-encoding once it knows that the data arrived at the client.

The limitations of this simple parametric model are fairly clear. In actual networks, "lag" often occurs in bursts, as particular routers temporarily become congested with traffic. These "lag spikes" increase both latency and drop rates, and would cause these factors to vary over time in unpredictable ways. Furthermore, large messages, particularly messages which must be fragmented into multiple UDP packets, are more likely to be dropped, due to increased chances of transmission errors, increased chances of running out of buffer space on a particular router, or simply because they take more time for each router to checksum and retransmit.

The reason that no attempt was made to simulate these factors is simply that it would be too difficult to account for the complex interactions involved. Instead, it is simpler to test against both average and worst case conditions, and assume that actual performance will always be somewhere in between.

Table 7.1: Round-trip Latencies to Worldwide Servers

Server	Distance (km)	Avg. Latency (ms)	Standard Deviation (ms)
Grand Prairie, Alberta	350	43.6	3.38
Seattle, Washington	900	35.0	2.5
San Francisco, California	1900	58.4	5.97
Miami, Florida	4100	94.6	2.84
Manchester, United Kingdom	6500	172.0	4.22
Seoul, South Korea	8300	218.8	7.77
Tauranga, New Zealand	12100	217.2	8.30
Pretoria, South Africa	15550	368.0	6.29

7.1 Latency

Information about the general behavior of network latency was gathered from Pingtest.net, a website which provides standardised round-trip latency ("ping") and packet loss testing to a variety of servers located around the world.

The latency testing is performed by forming a TCP connection to a remote host, and after a simple authentication handshake, sending 112 short TCP messages (18 bytes long each) back and forth in alternation. Each message from a particular peer (56 from each) contains an integer timestamp of the system time when the message was sent. As each machine sends its next message immediately upon receiving a message from its peer, and the messages are short and require little processing on the part of the operating system, the intervals between two messages from a particular machine are a good estimate of the round-trip latency between the two peers.

Thus, each such test produces 55 such intervals for each machine, or 110 intervals in total, which are samples of the round trip latency between two peers. Eight servers around the world were chosen based on distance, and tests were performed between Edmonton, Alberta, Canada, and each of those servers. and performed five ping tests for each, resulting in 550 estimates. The results are shown in Table 7.1.

From this, we can infer that a latency of about 100 milliseconds is a reasonable expectation when communicating within a single continent, although latency can increase sharply when communicating across oceans.

Pingtest.net also performs packet loss testing, by sending out 250 numbered UDP packets. The server counts up how many packets arrive, and reports back. Across all tests, no packet loss was reported. Therefore, for most of my testing, a flat rate of 5% packet loss was simulated, mainly to test the robustness of the algorithms involved.

7.2 Experimental Design

For each of the testing environments, a collection of objects were simulated, with a particular subset of this collection being visible to the client. Object views entered and left the client's visible set, and snapshots were taken every frame for the views inside the client's visible set. Four different techniques were tasked with the same goal: Transmit the exact contents of the snapshots of the objects in the visible set to the client.

Each technique observes the same snapshots and produces output as a stream of byes. These techniques can optionally be run through an additional layer of ZLib[6] compression. In the event that ZLib compression is used, it is allowed to retain its internal state from frame to frame.

The four techniques were as follows:

Binary contents (BC) Encode the exact binary contents of the snapshot into the message. Used primarily to show how much information is being synchronised on each frame. This technique will not be used in every experiment, as it is highly dependent on such things as the size of integer data types used for snapshots, which is not truly indicative of the amount of information being transmitted. However, it is at least worth showing that techniques never do worse than this baseline. It will be referred to as "BC" in future experiments.

Binary difference, with Zlib (BD Zn) Encode each byte of a snapshot as the integer difference from the corresponding byte of the snapshot on the last acknowledged frame. Very fast, and produces output highly suitable for ZLib's Huffman encoding. This most resembles the system used in Quake 3 and subsequent games, in that we are transmitting changes from the last known state, and we are using a form of Huffman coding to encode those changes. Where Quake 3 used a precomputed Huffman tree designed to be effective for its specific network traffic, this technique will allow ZLib to retain its state from frame to frame, allowing it to learn whatever dynamic Huffman tree fits the data it is presented with. The n will refer to the level of ZLib compression used. "BD Z6" will correspond to binary differencing with default ZLib compression, while "BD Z9" will correspond to binary differencing with maximum ZLib compression.

Prediction difference, with ZLib (PD Zn) Predict the value of each field in each snapshot, based on the previous values of that field. Write out the integer difference between the actual and predicted values for each field. Produces output suitable for ZLib's Huffman encoding. This essentially combines the field prediction methodology outlined in Chapter 5 with standard off-the-shelf ZLib string-matching and Huffman coding. It will be referred to as "PD Zn" in future experiments, for "Prediction Difference, with ZLib", where n again refers to the level of ZLib encoding.

Prediction difference with entropy coding (PD Entropy) Encode the difference between the actual and predicted values for the field, using a form of entropy coding derived from the statistical distribution of past differences for each specific field. Output has high entropy and is incompressible. This combines the field prediction methodology outlined in Chapter 5 with the entropy coding based on learned statistical distributions outlined in Chapter 6.

Table 7.2: “Particle System” Test Results (1000 particles)

Technique	Bandwidth Usage (B/frame)		Encode Time (ms/frame)		Decode Time (ms/frame)	
	Avg.	Std.Dev.	Avg.	Std.Dev.	Avg.	Std.Dev.
BC	20179	6.003	0.345	0.055	1.069	0.093
BD Z6	5756	25.14	3.305	0.185	1.512	0.113
BD Z9	5424	25.73	32.34	1.188	1.509	0.118
PD Z6	2035	41.52	2.628	0.129	2.255	0.147
PD Z9	1857	39.36	15.60	0.776	2.258	0.157
PD Entropy	1034	22.49	1.696	0.118	2.813	0.134

Note that in order to select which polynomial field predictor should be used to make predictions for a given field, both prediction difference techniques need to learn the statistical distribution of the error terms for each predictor. While the fourth technique uses these distributions for both predictor selection and error term encoding, the third technique only uses these distributions for predictor selection.

7.3 Hypotheses

We hypothesise that binary differencing with ZLib encoding will show statistically significant reductions in space usage compared to the straight binary contents technique, as it is based off of a technique that has been used with great success in the past.

We also hypothesize that prediction difference with ZLib encoding will show further statistically significant compression gains compared to binary differencing, indicating that delta encoding from predicted values produces smaller and more densely clustered error terms than delta encoding from the most recently acknowledged value alone, and that the techniques detailed in Chapter 5 are effective at learning which predictor to use for which field.

Finally, we hypothesize that using field-specific entropy coding instead of off-the-shelf ZLib encoding will further show statistically significant compression improvements, indicating that it is possible to learn to take advantage of the differences in data observed from different fields.

7.4 Particle System

The first test environment that was used is a toy problem. A simple two dimensional particle system is simulated by the server, and the state of each particle is synchronised to every client on every frame. Particles are defined by five numeric values: A pair of coordinates for the physical location of the particle, and a triple of values corresponding to the red-green-blue encoded color of a particle. The particles are simulated inside a closed “room” with a floor, ceiling, and two walls, and can bounce off each surface with a coefficient of restitution of $\frac{1}{2}$. They are accelerated downwards by gravity, and their color values decay towards black over the course of a four second lifespan. The system simulates at sixty frames per second. The output of the simulation is shown in Figure 7.1.

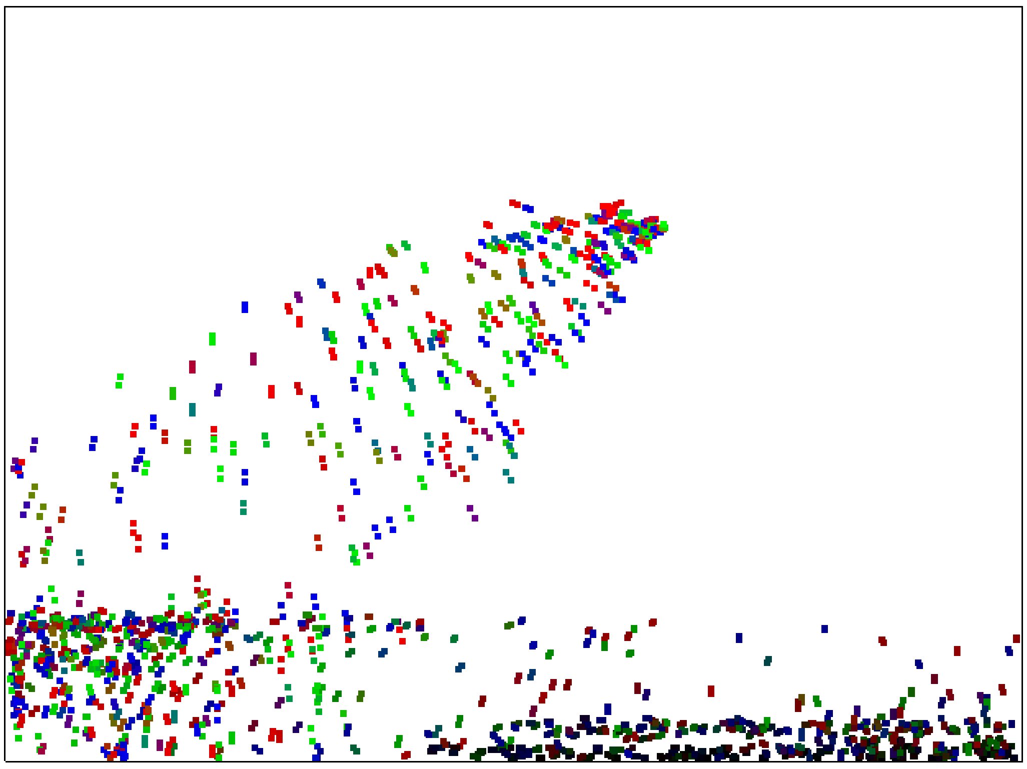


Figure 7.1: Simple particle system environment

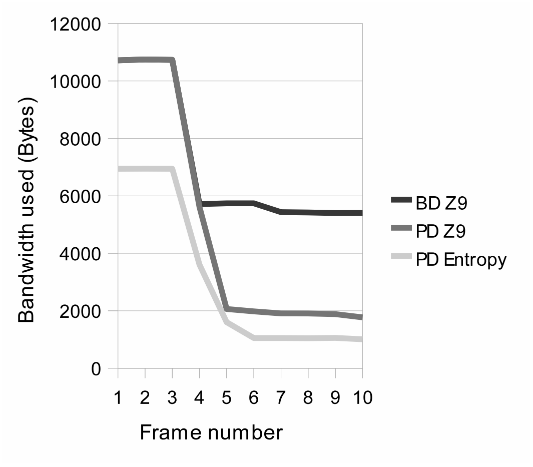


Figure 7.2: Learning in particle system environment

An experiment was conducted where the exact message lengths, encode times, and decode times of six different coding techniques were recorded for 1000 frames of simulation of approximately 1000 particles at any point in time (particles have individual lifespans, so this number varies very slightly). This test was run on one core of an Intel Q6600 Core 2 Quad processor, clocked at its standard frequency of 2.40 GHz.

The message lengths for the first ten frames are shown in Figure 7.2. This shows how, initially, it is fairly expensive to encode 1000 particles, as we must treat them as completely new objects each frame, until the first messages are acknowledged. At this point, encoding costs drop rapidly because we can start sending differences from known or predicted data. Over the next several frames, the prediction-based techniques become even cheaper as they refine their knowledge of which predictors are most efficient for each field. By about the seventh frame, all techniques have more or less stabilised, although they will continue to improve at very slight rates for a while beyond this.

The values in Table 7.2 were obtained from the exact message lengths, encode times, and decode times of frames 11 to 1000 of a trial run lasting for 1000 frames, and show the long-term properties of these synchronisation techniques on the particle system environment. On any given frame, our method was able to support 1000 particles at an overhead of roughly 1.03 bytes per particle. Using ZLib instead of our specialised entropy coder achieved 2.04 bytes per particle with standard compression, and 1.86 bytes per particle with maximum compression. Using ZLib on a binary difference required about 5.76 bytes per particle with standard compression and 5.42 bytes per particle with maximum compression. A straight binary view would use 20.2 bytes per particle (five 32-bit integers, plus some overhead when particles are created and deleted), although obviously, the full range of those integers is not utilised.

Of some note is that the baseline decoding time was fairly high compared to the baseline encoding time. This is most likely due to each client needing to marshall decoded snapshots into separate client view objects, while the server takes snapshots of each server view object on its own, and these snapshots are waiting in snapshot archives for the different encoders to make use of them. However, it is known that both Huffman Coding (used in ZLib) and Arithmetic Coding are slightly more expensive when decoding than when encoding. The former technique can encode with a lookup table but requires tree traversal to decode, while the latter can decide the interval boundaries for an encode with a lookup table but requires some sort of interval search for a decode.

Interestingly, every technique has reasonable decoding times. Even though you would almost never have 1000 objects visible to a client at once, under this system it would take less than $\frac{1}{10}$th of the processing power of a single 2.4 GHz core on an Intel Core 2 processor to decode the data for these objects using the most expensive technique.

In terms of encoding times, it is clear that ZLib should be left on a relatively fast setting. Even though compression gains on the order of 10% can be achieved by cranking compression up to maximum, the additional CPU costs make it simply too expensive for use on a server.

What is further interesting, however, is that in this environment, doing the full prediction and entropy coding is actually faster than using a simpler form of encoding, in addition to achieving better compression. This can be partially explained in that most encoding schemes partially "pay by the bit", in that there are certain operations that must be done for each bit emitted. When dealing with well-behaved data and longer messages, the spatial savings of a more efficient code can also translate into time savings.

7.5 Virtual Starcraft Server

One of the main genres which currently achieves multiplayer via peer-to-peer co-simulation is the realtime strategy game genre, of which Blizzard's Warcraft and Starcraft franchises are some of the most prominent. I was fortunate to have access to a community-developed application programming interface, known as BWAPI, whose objective was to allow Starcraft games to be observed and manipulated by C++ programs, primarily for the purpose of writing programs to play the game competitively.

Using BWAPI, I was able to run a number of experiments on how my library would handle if used to run a game of Starcraft in a client-server setting instead of simulating the game on every participating computer. A previously mentioned feature of Starcraft is its ability to save out replay files containing the exact orders and commands issued by every player over the course of the game. These can be "played back" through the game engine, for the purpose of examining and studying past matches. The Starcraft executable program in fact simulates the complete game, start to finish, treating the commands read from the replay file as though they had come from the local player or from the network, and executes the exact same instructions as though the match were being played for the first time. By obtaining and playing replays of competitive Starcraft matches, it is possible to use BWAPI to observe and learn about the conditions present during real-world Starcraft matches, making for a much more effective testing ground.

For this experiment, the complete state of a Starcraft game was treated as the state of the server, and each player was treated as a client. Each client was granted visibility over the state of its player (such as resources and upgrades) as well as the state of all units owned by that player. Our networking library would then produce packets, which would be sent over a simulated network connection, with tuneable latency and packet loss rates, to a virtual client which would receive the game state and transmit acknowledgement packets back over the same simulated network connection. In the process, we gathered information about how much network traffic would need to be sent from server to client if a game like Starcraft were being run in a server-client setting.

Via this setup, statistics were gathered about the amount of bandwidth required to keep each player's client in synch under a number of different potential scenarios with regards to network traffic. These data painted a picture not only of the potential for using our techniques to drive a professional, industry-quality real time strategy game, but also how it would perform over different types of connections.

One fundamental assumption that we are making is that replays of actual Starcraft matches are representative of what replays would look like for a server-client version of Starcraft. Specifically, we are assuming that the actions being taken in a Starcraft match would not change significantly if the game were being synchronised explicitly instead of implicitly. This is not strictly true. If Starcraft were simulated explicitly, client actions could take place the moment they reached the server, instead of having to wait for a predetermined latency period to elapse. On the other hand, events that occur on the server would need time to propagate to the client before they were made visible. It is hoped that the increased responsiveness would more-or-less cancel out the delay incurred by transmitting state explicitly instead of simulating it locally, but it is possible that high level play, in which it is common to see 200 to 300 orders issued by each player every minute, would be affected slightly.

7.5.1 Starcraft Overview

Starcraft is a real time strategy game. Games are most commonly played between two players, although larger numbers of players are not uncommon. Each player's goal is to build an army and use it to defeat his opponent or opponents. In games with more than two players, some players may choose to work together as a team, but the overall objective remains unchanged. In order to defeat one's enemies, players must maintain a robust economy (which may involve constructing additional bases to gather more resources), build production facilities to construct larger numbers of units, or more advanced units, research new upgrades and abilities to maximise the effectiveness of their armies, maintain control of key locations on the map, and acquire good intelligence on what their opponent is planning, in order to be able to execute effective countermeasures in time.

Each player begins the game with a single structure, which can train worker units, and four workers, which can harvest resources as well as construct additional structures. There are two types of resources which can be harvested, and both are required to train units, construct structures, and research upgrades. These resources are "minerals", which can be harvested from mineral fields, and "vespene gas", which can be harvested from vespene geysers. Vespene geysers are much rarer than mineral fields. Typically, low-tech units such as infantry cost only minerals, while high-tech units such as vehicles or flying units cost both minerals and vespene gas, and "spellcasters" (units with special abilities that can be invoked during battle) cost high amounts of vespene gas. For this reason, players who adopt strategies involving "gas-intensive" units must often expand aggressively to control larger numbers of vespene geysers than their opponent.

Military units are broadly divided into the categories of ground units, which cannot cross cliffs or water, and air units, which can freely fly over any part of the map. Air units obviously have greater mobility, but often pay for it with higher production costs and reduced durability. A special category of air units are "transports", flying units that cannot attack but which can carry up to eight ground units and deploy them to different parts of the map.

Additionally, a small number of unit types can “cloak”, rendering them invisible under normal circumstances. Some units expend energy to cloak, others are permanently cloaked, and still others can achieve the effects of cloaking by burrowing underground, in which case they may not be able to attack while hidden. These units can be rendered visible again by “detectors”, high tech units that make visible any cloaked units within their sight radius. Certain structures can also act as detectors.

Furthermore, each player can select one of three factions, or “races”, to play. The “terrans” are human exiles who live on hostile planets. They can construct their bases anywhere, and their structures can even lift into the air and fly around the map. Terran armies consist of a wide variety of units with varying military capabilities.

The “zerg” are a collection of constantly evolving alien species. All of their units, including their structures, are biological in nature. Zerg based can only be constructed on a biological carpet called “the creep”, and all of their production is centralised, as their units morph from larva produced at hatcheries. Zerg armies consist of large quantities of cheap units, and they rely on subversive tactics such as ambushes and highly mobile raids to achieve success.

The “protoss” are an advanced alien species with powerful technology and psychic abilities. Protoss structures require an infrastructure of power fields generated by buildings known as pylons, making them vulnerable to sabotage. Protoss armies consist of smaller numbers of specialised units, which are devastating if used effectively. The protoss have more technology choices early on, but they do not have as many “mainstay” units that are good in all situations.

Due to all of these factors, individual games of Starcraft can play out very differently from one another. Movement patterns are very different between different unit types or between ground and air based armies. Some games are decided by the use of special abilities, others by trickery and deceit, still others by brute force. Sometimes players will expand to cover the map with bases and defensive structures, while at other times players will focus more on constant combat, and prevent each other from expanding too rapidly. Additionally, each of the six one-versus-one matchups of the three races takes on a very different flavour, as each race must adopt different tactics to effectively oppose each other race.

7.5.2 Snapshots

Two different types of snapshots were used to represent the state of a Starcraft game. The first corresponded to players, and was responsible for keeping track of the faction of the player (which can be one of three alternatives), the resources owned by the player (used to purchase units and build structures), the supply count used by the player (a measure of the number of units owned, for which there is a total allowable amount), whether any of the forty-seven different types of research or sixty-three different types of upgrades were being actively researched, whether they have already been researched at the current point in time, and the levels of any upgrades that have more than

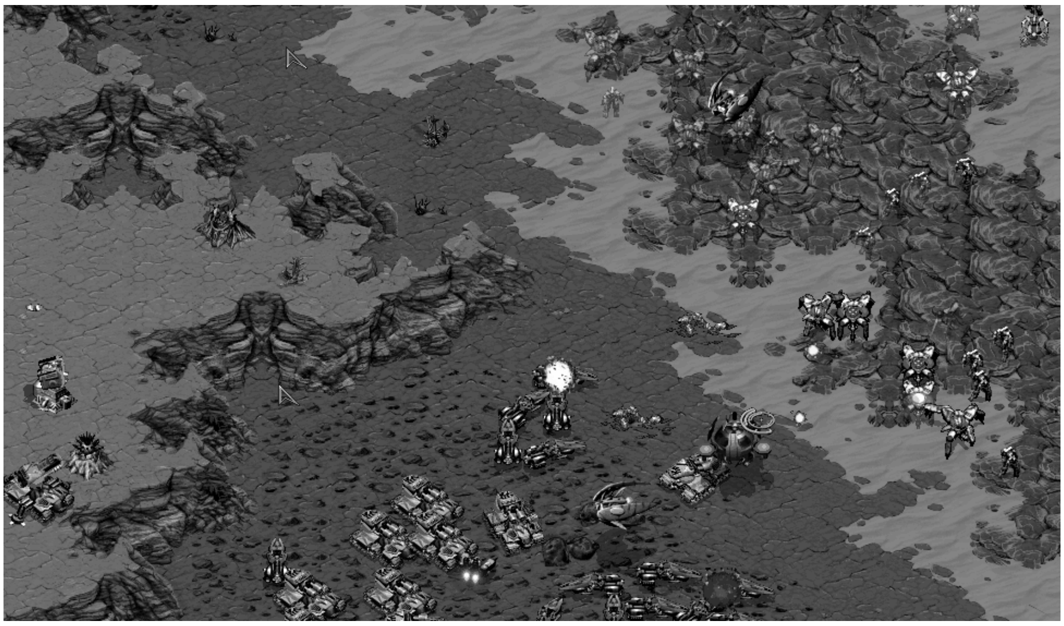


Figure 7.3: A mechanised terran army holds off a cloaked protoss assault

Table 7.3: Contents breakdown of Player snapshot

Field	Minimum	Maximum	Bits	Amount	Total Bits
Faction (Terran, Protoss, or Zerg)	1	3	2	1	2
Resources (Minerals and Gas)	0	200000	15	2	30
Supply (Used and Total)	0	400	9	2	18
Upgrade Level	0	3	2	63	126
Is Upgrading?	false	true	1	63	63
Has Researched Tech?	false	true	1	47	47
Is Researching Tech?	false	true	1	47	47
Total					333

one level (weapon and armor upgrades, for instance, can be purchased three times, with cumulative effects). Taking into account the effective ranges of all the variables involved, there are roughly 42 bytes of information associated with each player, as shown in Table 7.3.

The second snapshot type corresponded to units, and was responsible for keeping track of the type and allegiance of the unit, its position, its vital statistics (hit points, shields, energy, etc), the cooldown periods before it can use its attacks and abilities, the state of all buffs/debuffs (effects applied by the abilities of other units which alter the performance of the unit), and forty two boolean state variables corresponding to a number of special modes and states that a unit can be in. Again, taking into account the maximum ranges of all the variables involved, each unit has roughly 27 bytes of information associated with it, as shown in Table 7.4.

The initial visible set in any Starcraft game will consist of the player's starting building, four worker units, and nine visible resource locations (eight mineral clusters and one vespene geyser), for a minimum of fourteen units, as well as the player's initial state.

Table 7.4: Contents breakdown of Unit snapshot

Field	Minimum	Maximum	Bits	Amount	Total Bits
Unit Type	1	256	8	1	8
Allegiance	1	12	4	1	4
Hit Points	0	2000	11	1	11
Shields	0	750	10	1	10
Energy	0	250	8	1	8
Resources	0	5000	13	1	13
Cooldown Timers	0	255	8	3	24
Buff/Debuff Timers	0	255	8	10	80
Build Timers	0	7680	13	1	13
State Variables	false	true	1	42	42
Total					213

All things considered, the initial scene will contain at least 377 bytes of state. It is common, in the middle of an average game, for there to be about 280 units in play, resulting in about 7497 bytes of state. To send this much information every frame would require about 180 KB/s sustained upstream per client, and each individual frame would need to be split across six UDP packets. However, it is not unheard of for up to 900 units to be visible at once during normal one versus one matches, requiring about 24 KB of state information, and about 576 KB/s sustained upstream to each client, as well as having state information split across seventeen UDP packets. It should immediately be apparent that a game such as Starcraft is almost prohibitively complex to be run via a server-client model without some form of compression. However, I will show that these bandwidth requirements can be reduced by almost two orders of magnitude by using the techniques I have built into my networking library, and that even under maximum observed load, a single UDP packet can suffice to express the entire game state relative to previously observed states.

7.5.3 Technique Comparison

Twenty replays were downloaded randomly from Team Liquid's replay archives[9]. These games were simulated to completion, with three virtual clients corresponding to each player, using the BD Z6, PD Z6, and PD Entropy techniques. For every frame, information was saved out about the bandwidth usage in bytes, and encode and decode times in milliseconds, for each virtual client. This information was aggregated into a pool of 1479509 data tuples, each corresponding to a particular frame of a particular Starcraft match, and containing the compression performance and runtime costs of all three techniques. The results are summarised in Table 7.5. The "contents" field is merely an estimate of the amount of data being synchronised, based on the estimated information content of all of the views visible the client.

It is immediately apparent that the predictive techniques are only effective on the Starcraft environment when combined with our specialised entropy coder. Both "BD Z6" and "PD Entropy" fall on the pareto-optimal boundary, with the binary differencing technique being fastest, and the prediction and entropy coding technique achieving the greatest compression.

Table 7.5: "Virtual Starcraft Server" Test Results

Technique	Bandwidth Usage (B/frame)		Encode Time (ms/frame)		Decode Time (ms/frame)	
	Avg.	Std.Dev.	Avg.	Std.Dev.	Avg.	Std.Dev.
Contents	6946	6913				
PD Z6	747	541	2.14	1.98	1.61	1.44
BD Z6	340	188	1.09	0.988	0.925	0.757
PD Entropy	95.1	65.9	1.70	1.63	2.50	2.42

Table 7.6: "Virtual Starcraft Server" Test Result Ratios

Technique Ratio	Relative Bandwidth Usage		
Num. / Denom.	Avg.	Std.Dev.	Percentage of Time Better
PD Z6 / Contents	13.3%	3.45%	100%
BD Z6 / Contents	7.43%	3.61%	100%
PD Entropy / Contents	1.93%	1.39%	100%
BD Z6 / PD Z6	52.9%	14.1%	99.95%
PD Entropy / PD Z6	13.7%	5.71%	99.87%
PD Entropy / BD Z6	26.7%	8.63%	99.87%

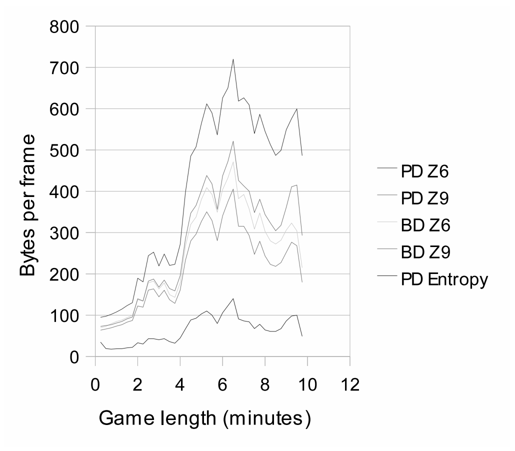
However, the size of the compressed messages are highly correlated among the different techniques. All three techniques can compress a simple early game state to a small number of bytes, similarly, all three techniques will require more memory to compress a game state corresponding to a late game scenario where many units are in play at once. Therefore, we also calculated the statistical properties of the ratios between the three techniques, resulting in Table 7.6.

This shows that all three techniques were able to synchronise the game state using messages that were a mere fraction of the rough size of the game state. PD Z6 was able to achieve about $7.53 : 1$ compression, while BD Z6 was able to achieve about $13.5 : 1$ compression. Best of all, however, our proposed technique was able to achieve about $50.7 : 1$ compression on average (though do bear in mind that this is compression exploiting frame-to-frame coherency, not general compression). Comparing our technique to the next best alternative, BD Z6, we did on average about four times better in terms of compression ratios. Furthermore, we achieved better compression than BD Z6 99.8703% of the time. In fact, it is generally only for a few frames at the start of the game (within the first couple of seconds) that our technique does worse than any of its alternatives, and in that time, we tend to use no more than 300 bytes per frame.

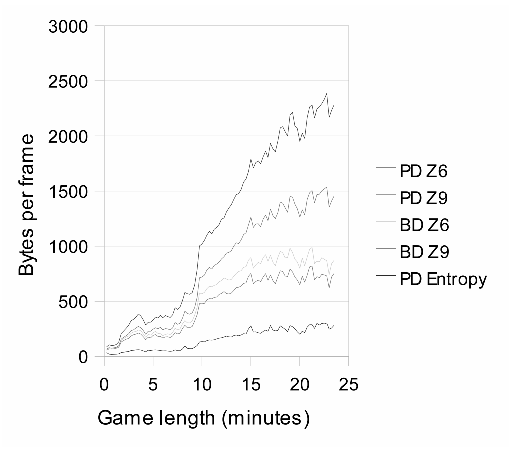
Thus, we have shown that our proposed technique uses consistently less bandwidth than any of the other synchronisation schemes we examined, even if the runtime costs are somewhat higher than the alternatives.

7.5.4 In Depth Technique Comparison

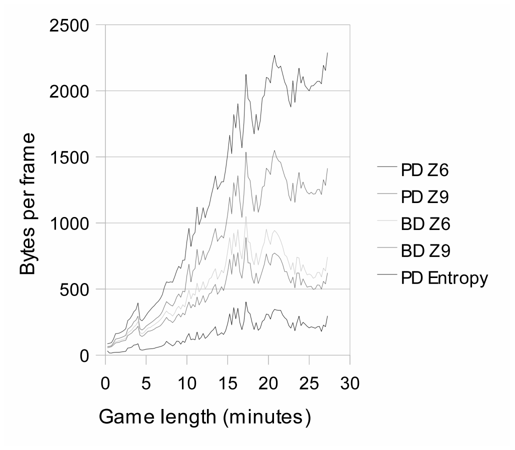
Four games were selected randomly for in depth technique comparisons. For these games, in addition to BD Z6, PD Z6, and PD Entropy, we also tested BD Z9 and PD Z9, variations in which ZLib encoding is set to maximum. We recorded average bandwidth usage, encoding time, and decoding time for all five synchronisation methods.



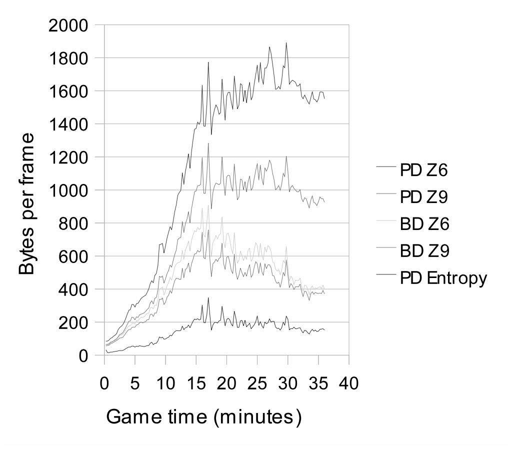
(a) Zergboy vs a0.oa



(b) Bisu vs Canata

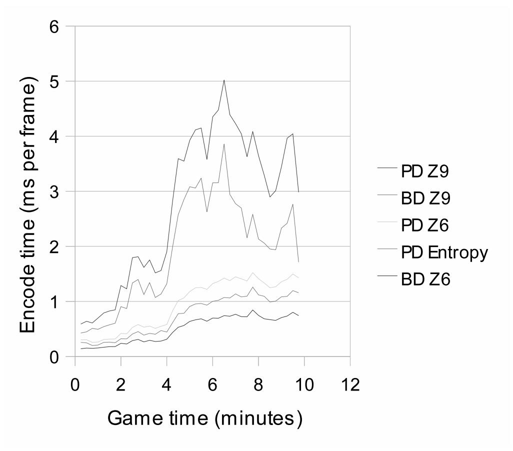


(c) Flash vs Backho

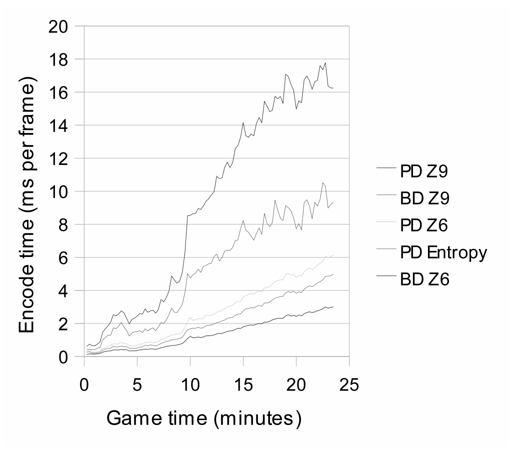


(d) WhiteRa vs. Strelok

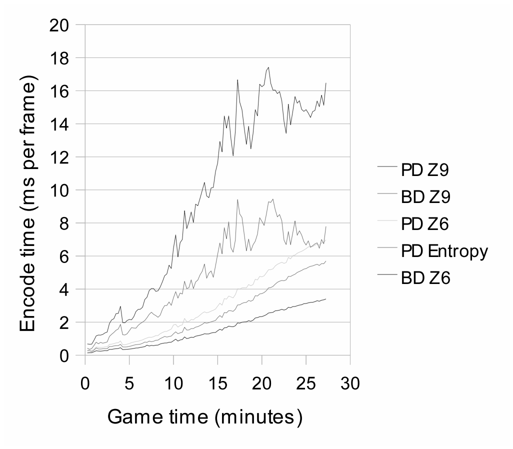
Figure 7.4: In-depth Comparison of Techniques (Message Length)



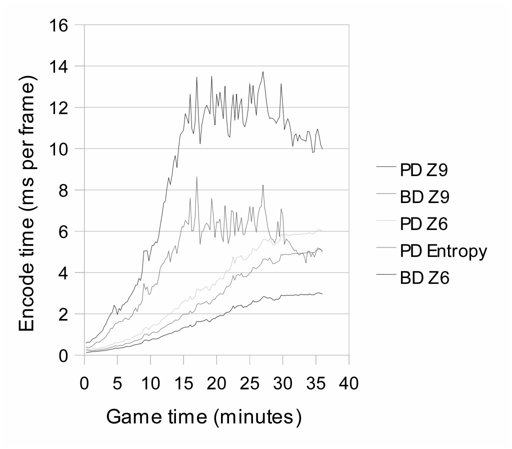
(a) Zergboy vs a0.oa



(b) Bisu vs Canata

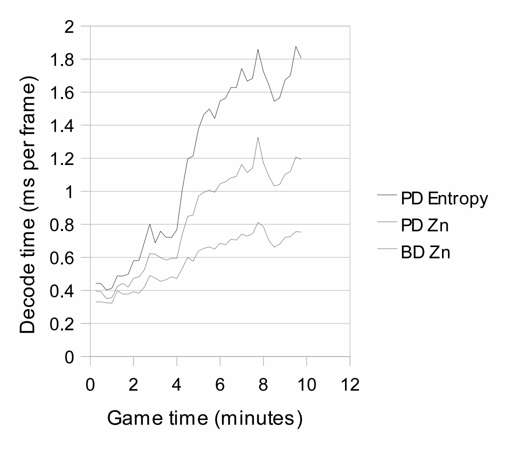


(c) Flash vs Backho

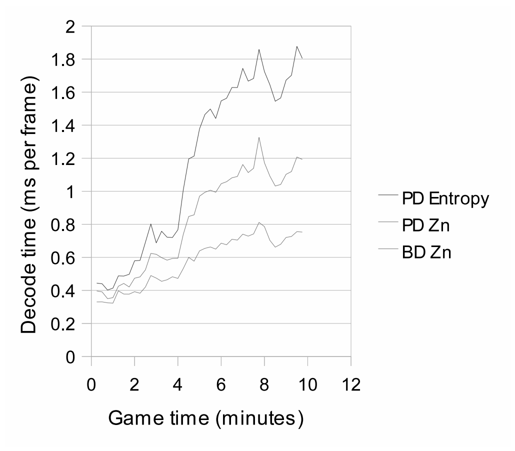


(d) WhiteRa vs. Strelok

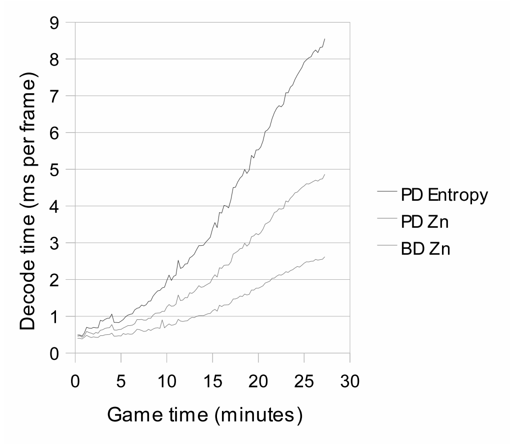
Figure 7.5: In-depth Comparison of Techniques (Encode Time)



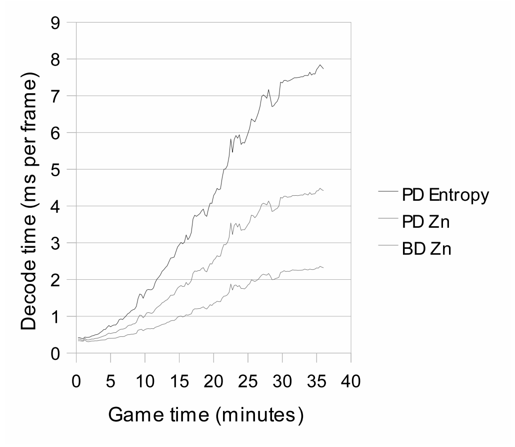
(a) Zergboy vs a0.oa



(b) Bisu vs Canata



(c) Flash vs Backho



(d) WhiteRa vs. Strelok

Figure 7.6: In-depth Comparison of Techniques (Decode Time)

The selected games were a short 10 minute game between “Zergboy” and “a0.oa”, a 23 minute game between “Bisu” and “Canata”, a 27 minute game between “Flash” and “Backho”, and a 36 minute game between “Whitera” and “Strelok”.

The overall trends depicted by these four games are quite clear. In terms of compression size, shown in Figure 7.4, our technique, combining prediction differences and custom entropy coding, consistently outperforms the other techniques at all phases of the game, often by a large margin. Once more, combining prediction differences with ZLib encoding was much poorer than using a simple binary difference combined with ZLib encoding. One potential explanation for this anomaly is that Starcraft primarily contains objects with linear dynamics, so the extra complexity of modelling quadratic or cubic predictors does not earn us very much predictive power, and a simple binary difference is feeding more consistent information to ZLib, allowing it to develop a more efficient encoding. For both techniques that use ZLib, ZLib’s maximum compression setting offers significant compression gains over its standard setting, however, neither technique is able to come close to our custom entropy coding.

In terms of encode times, depicted in Figure 7.5 we very consistently observe that the ZLib techniques on maximum compression settings (BD Z9 and PD Z9) are phenomenally expensive, perhaps prohibitively so, as they can consume between than 10 and 20 ms per frame per client, which is between 240 ms and 480 ms out of every second, per client. This could very well exceed the CPU budget available for networking, and would require parallelisation to avoid adding punishing latencies to communications with clients (If it takes 30 ms to encode updates for two clients in serial, then the second client effectively has an additional 30 ms on their latency).

Otherwise, we observe that a ZLib encoded binary difference (BD Z6) is the fastest encoding method that we have, which seems to take about 60% as much time as our prediction difference and entropy coding solution (PD Entropy). Both of these techniques use much more reasonable CPU resources, seeming to top out around 5 ms. Of some concern is the observed tendency for encode times to grow over the course of the game. Army sizes are capped at 200 “supply”, with military units typically taking up between 1 and 3 “supply”, leading to armies that tend to max out at about 100 units. However, structures are not constrained in a similar way, and players will usually construct multiple bases over the course of longer games. Even though these structures are largely static, and thus do not consume much bandwidth, they still incur costs during encoding and decoding, as under the current algorithm the same predictions and frequency table lookups must be performed for them as for anything else.

In terms of decode times, shown in Figure 7.6 the ZLib variations have very little difference from one another. In general, the entropy coding solution (PD Entropy) was the most expensive during decode, followed by PD Zn, with BD Zn being the most efficient decoding technique. The prediction-based techniques currently do a substantial amount of additional learning during decoding, as they learn not just about the distribution of error deltas from the predictor that was used, but about the distribution of error deltas from other predictors as well. The costs of Arithmetic Coding compared to Huffman Coding are also nontrivial. While this is somewhat problematic, note that each client only needs to decode messages from a single source, so even the high observations of 8 ms per frame only correspond to a cost of 192 ms per second, or $\frac{1}{5}$th of a single core on a 2.40 GHz Intel Core 2 CPU.

While game servers often require substantial CPU resources to handle game logic, collision detection, physical simulation, and artificial intelligence (especially pathfinding), game clients need only function as "dumb terminals", blending animations and marshalling rendering operations to the GPU, while recording player actions and transmitting them to the game server. Therefore, the most expensive CPU costs involved in using our technique are incurred on the machines which are most able to handle those costs. Furthermore, since the client can acknowledge receipt of a message before decoding it, the client's decode time does not contribute to the round trip latency between server and client.

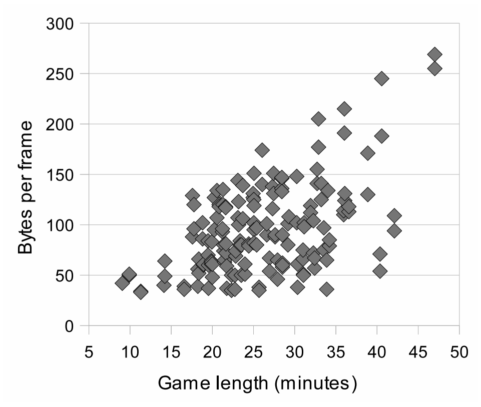
7.5.5 Thorough Stress Test

For this test, eighty replays were downloaded randomly from Team Liquid's replay archives [9], drawn from the time period from September 2002 to October 2010. For each game, three virtual clients were created corresponding to each player, using our technique (prediction differences and entropy coding). These clients were placed on virtual networks with a round-trip ping to the server of 41.7 ms, 83.3 ms, and 125 ms respectively. These figures correspond precisely to a delay of one, two, and three frames passing before a packet from the server to the client is acknowledged in a return packet from the client to the server. Furthermore, a 5% drop rate was put in place for all clients.

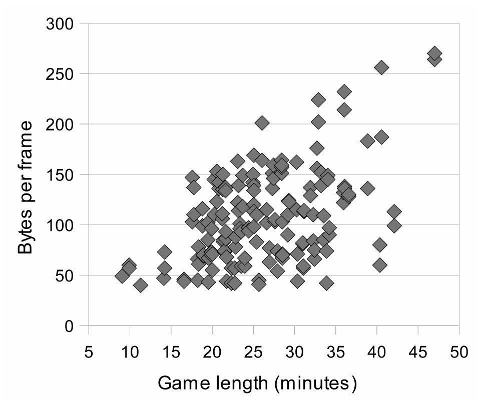
Each of the eighty games were then simulated to completion. The total game time in frames, average bandwidth usage per frame, maximum bandwidth usage per frame, and average number of units visible during the game were then recorded for each of the six virtual clients (three per player). These tests were intended to gather information about how effective our technique is across a wide variety of game types, play styles, and scenarios, as well as varying network conditions. Collectively, these games take up about 35 hours of realtime play data, or 70 hours of player time (two players per game). Individual games ranged from 9 to 47 minutes, with an average length of 26 minutes and a standard deviation of 7.3 minutes.

Figure 7.7 demonstrates the average bandwidth usage for the 160 half-games that were analysed, compared to the length of the game in question. Note that, as latency increases, the average bandwidth usage increases monotonically, but the increase is very slight.

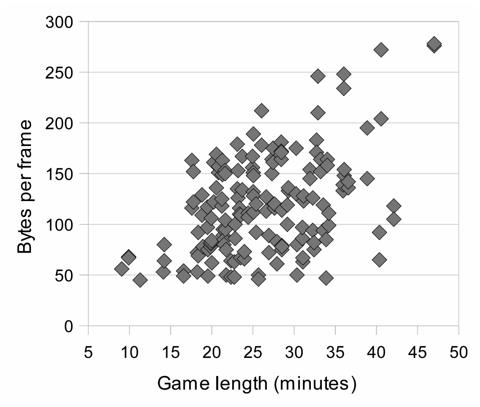
There are three apparent boundaries in the bandwidth usage. First, average usage never drops below a certain threshold, which is slightly higher for higher-latency games, seeming to range between about 35 bytes per frame under low latency conditions, and 45 bytes per frame under high



(a) 42 ms latency

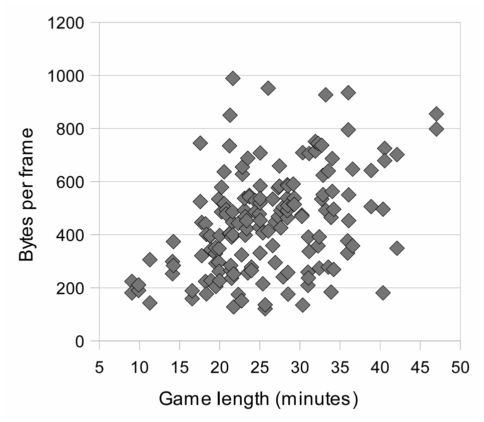


(b) 83 ms latency

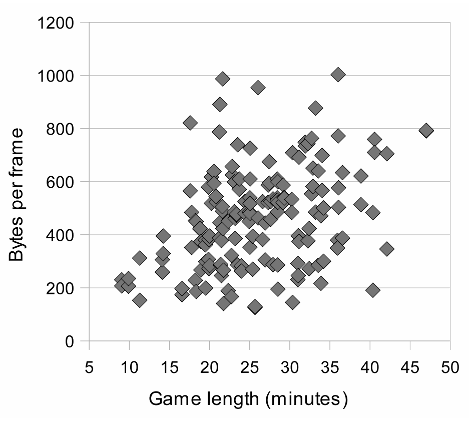


(c) 125 ms latency

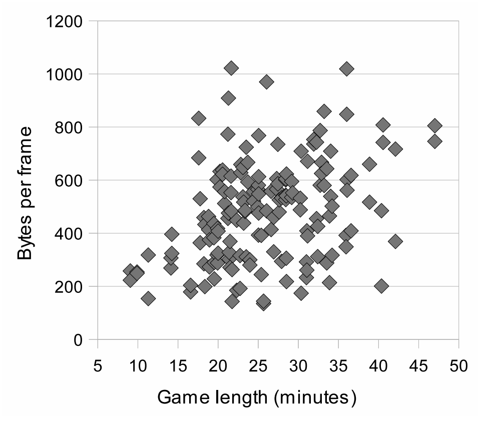
Figure 7.7: Average bandwidth usage



(a) 42 ms latency



(b) 83 ms latency



(c) 125 ms latency

Figure 7.8: Maximum bandwidth usage

latency conditions. This likely represents a certain baseline complexity of a Starcraft game, such as the movements of workers and basic military units, and the state of the buildings required to build those basic military units.

There also seems to be a sort of diagonal upper bound which is most likely due to the inherent time it takes to build up forces at the start of the game. For instance, for each unit in the game, there is a minimum amount of time that it will take to construct a single instance of that unit, derived from the build times of the prerequisite buildings, and the training time of the unit in question. Similarly, certain abilities cannot appear on the field of battle until caster units have been trained, they have had time to build up to certain energy levels, and their abilities have been researched, all of which impose time constraints. Therefore, the sort of complex behavior that will require greater overall bandwidth usage cannot appear in shorter games.

Finally, there seems to be a soft cap on average bandwidth usage which sits at about 150 bytes per frame for low latency games and 200 bytes per frame for higher latency games. Although we have outliers which exceed this, the density of games below this threshold is drastically higher than the density of games above it.

One possible explanation is that this represents some sort of bound on the number of units that can be produced on a map. In an ordinary game of Starcraft, each player starts with one base, and there are a finite number of locations to which they can expand and build secondary bases. There is also a hard cap on the number of units that can be produced by any particular player. Thus, even in laid back games where both players expand to cover half the map, and build very large armies before sending them against one another, there is still a hard limit on the complexity of a game of Starcraft.

The outliers, whose bandwidth requirements are significantly above the “soft cap” were investigated manually, and it was found that they were simply extremely high level games in which many different things were happening across the map simultaneously, and each player was switching tactics frequently. In these cases, learning done early on would not be as effective in assisting compression for later points in the game.

Figure 7.8 indicates the maximum length of any single update message at any point during the same set of 160 half-games, again, plotted against the length of the game. The reason for gathering these statistics is that the distribution of interesting events across the span of a game of Starcraft is not uniform, more state will be changed during large battles or clever gambits than during the methodical buildup of forces. Therefore, it is important to show that the maximum bandwidth consumed by a single frame still falls within a reasonable range.

The first thing to note is that at no point during any of the games did bandwidth usage of our technique exceed about 1050 bytes per frame. This is still a healthy margin below the typical MTU for UDP packets, which is 1440 bytes. This is important, because we can avoid datagram fragmentation, and the compounded drop rate of datagrams that must be send via multiple packets.

The second thing to note is that the longest games, which used the most bandwidth on average, did not have the highest peak bandwidth usage for single frames. It is likely that, as these games consisted of constant action all over the map, that we were able to learn about them more quickly, and keep peak bandwidth costs lower. Those games which did have the highest peak bandwidth likely involved lengthy buildups followed by intense battles, or other rapid changes in the nature of the game, which would cause our method to have a small hiccup in which it must learn about the new statistical distributions of various fields.

It is worth noting that unlike average bandwidth, maximum bandwidth usage does NOT increase monotonically with latency. In some matches, maximum bandwidth usage even decreases as latency increases. The reason for this is that, in higher latency matches, nonzero error deltas are observed much more commonly, as the differences due to unexpected events must be sent multiple times, and may be comparatively larger over time. Therefore, while more bandwidth is necessary on average, the particular encoding learned will be comparatively better at handling large amounts of unexpected data. Therefore, the most expensive frames over the course of the game actually become cheaper to represent, even though there will likely be more of them.

Thus, we have shown that our technique can easily handle the sort of traffic that occurs during typical complete matches of Starcraft, across a wide variety of games and circumstances. We have also shown that performance decays gracefully as latency increases.

7.6 Environment Comparison

Our methods demonstrated very strong compression capabilities on both the particle system and virtual starcraft server environments. However, the runtime performance was much better for the particle system than for the Starcraft server. Recalling that the particle system environment has only a single snapshot type, with only five fields, whereas the Starcraft environment, which has two snapshot types, dozens of numeric fields, and hundreds of boolean fields between them. This means that both the frequency datasets and snapshot archives will be larger for the Starcraft environment, and while neither constitute a serious fraction of the main memory available to personal computers in 2010, the access patterns of our technique on these datasets provide a strong incentive to try to keep them within the capacities of available L2 cache.

In spite of these potential challenges in runtime performance, which will be discussed further in Chapter 8, we have shown very strong compression of gamestate across both test environments.

Chapter 8

Conclusions

In this thesis, we have demonstrated a powerful paradigm for handling explicit synchronisation of game state in server-client multiplayer games.

Specifically, we have shown a suite of techniques that can learn highly efficient network protocols at runtime, requiring almost no effort on the part of game developers. These techniques allow for game developers to produce complicated games involving hundreds or even thousands of visible objects, moving and interacting in sophisticated ways, without having to invest precious developer hours into engineering a customised network protocol. We have further shown that these techniques are robust to changes in gameplay dynamics during rapid development, requiring almost no maintenance. The result is stable, reliable, low risk multiplayer support for a wide variety of games.

Furthermore, we have demonstrated that these techniques are efficient enough that an enterprising game developer could use them to tackle genres which are traditionally handled via implicit synchronisation, such as ever-popular real time strategy games. With the rise of e-Sports in South Korea and around the world, it is becoming more and more important for aspiring developers to secure their games, to prevent cheating and ensure a fair experience for all players. The method presented in this thesis could provide the first step towards the development of strict server-client real time strategy games.

Finally, we have shown that these properties can be achieved via a non-intrusive software library, which communicates with the game server and game client through a simple, mostly clean interface, and allows for server-client networked multiplayer to be easily added to both new and existing software projects in a matter of hours, not weeks or months.

8.1 Future Work

The techniques that have been presented thus far are inherently CPU intensive. Values must be sampled from snapshot histories, predictions must be made, error deltas must be recorded in statistical distributions, which must be formatted into intervals suitable for arithmetic coding, and the arithmetic coding itself must be performed.

There are plenty of opportunities, at multiple levels, to improve upon these techniques and create an architecture that is much faster, while retaining the bandwidth efficiency of these techniques.

8.1.1 Range Coding

Experimenting with variations on arithmetic coding, such as range coding, might yield significant speed improvements with relatively little effort.

8.1.2 Symbol Ordering

My current techniques are very naive about symbol ordering, such that the most frequently used symbols tend to be in the middle of the list of possible symbols. Arithmetic coders must be passed subintervals of $[0, 1)$, so frequency information must typically be transformed into cumulative frequencies in order to be used by arithmetic coders. Cumulative frequencies can most easily be updated if the most frequently used symbols occur near the end of the list of symbols. Whether by simply reordering the buckets in a bucketing scheme, or introducing some sort of dynamic transposition tables, it may be possible to speed up learning by simply allowing statistical distributions to be updated more easily.

8.1.3 Selective Learning

A large part of the computational cost of our technique comes from the need to be constantly learning about the statistical distribution of error terms. It may be possible to design a disciplined approach by which learning can be dynamically turned on or off as a simulation runs, based on the levels of compression being achieved.

For instance, Chapter 6 goes over how to compute the expected cost of encoding a random variable X when you only know the distribution of a random variable X' whose distribution approximates X's. It may be possible to record how many bits are being spent encoding each field in a given frame, compared to the expected number of bits being spent, and make some intelligent decision about how frequently to learn from observed error deltas. For instance, when we are paying very close to the expected costs of encoding a field, we can choose not to learn at all, but as we start to overpay, we can start learning from every tenth error delta, etc, until at some point if we are drastically overpaying we can start learning from every error delta.

Such a scheme might also be able to control the decay rate for recency-weighted learning. It might be possible to intelligently flushed previously learned information about a field if we can determine that it no longer applies, while preserving information that is serving us well for a long time. This might yield both compression gains and speed gains. It is worth remembering that a significant portion of the CPU costs of using arithmetic coding is paid by the bit (such as bounds checking and renormalisation). Fewer bits emitted or consumed translates to fewer operations and less CPU cost.

8.1.4 Cache Coherency

The greatest weakness of the architecture I have presented is its erratic access to memory. We have object views, which each have several stored snapshots, which each have some specific number and arrangement of stored field values. Each object view is an instance of a class, which has a number of fields, which have several distributions, which have several frequencies. Some operations need to iterate over fields (delta encoding or decoding), some need to iterate over snapshots (making predictions), some need to iterate over distributions (selecting the best predictor), and some need to iterate over frequencies (decoding a value, recording a new value). The end result is a library that is highly sensitive to available cache, and whose performance drops dramatically once the manipulated objects no longer fit into cache.

It might be possible to design a better architecture, such that information is separated out by, for instance, fields. The delta encodings for a particular field for all visible objects could be written out at once, allowing the statistical distributions for the error terms corresponding to that field to be loaded into cache only once, instead of being cycled in and out as different fields are considered. If the values for a particular field were separated out of each snapshot and stored in separate archives, this could be a particularly fast way of doing things, without changing the actual semantic behavior of the library.

8.1.5 Additional Data Types

Some additional policies for supporting different types of non-numeric data would be helpful to end users. For instance, containers are currently difficult to represent, as you need to have a separate view for each element, and a field indicating the container to which they belong. This requires assigning network IDs and some marshalling work on both server and client which could ideally be taken care of by the library itself.

For that matter, a standardised and disciplined way of handling references between objects would be of use to network programmers. In addition to conveying information about object relationships from the server to the client, it would be beneficial to have a standardised way that the client can refer to serverside objects when transmitting user input and commands back to the server. Object references, typically handled via integer IDs, are trickier to make good predictions about than numeric fields, but perhaps something can be done in this regard.

Bibliography

[1] Jesse Aronson. Dead reckoning: Latency hiding for networked games. *Gamasutra*, 1997.

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

[3] Paul Bettner and Mark Terrano. 1500 archers on a 28.8: Network programming in Age of Empires and beyond. In *Game Developers Conference*, 2001.

[4] Michael Buro. ORTS: A hack-free RTS game environment. In *International Computers and Games Conferencee*, 2002.

[5] Nick Caldwell. Defeating lag with cubic splines. *gamedev.net*, 2000.

[6] Jean-loup Gailly and Mark Adler. ZLib. `http://www.zlib.net/`, 1995-2010.

[7] Jr. Glenn Langdon and Jorma Rissanen. Method and means for arithmetic string coding. `http://www.google.com/patents?vid=4122440`, 1977-1978.

[8] David Huffman. A method for the construction of minimum-redundancy codes. In *I.R.E.*, volume 40, 1952.

[9] Team Liquid. Starcraft replay archive. `http://www.teamliquid.net/replay/`.

[10] Matt Mahoney. Large text compression benchmark. `http://mattmahoney.net/dc/text.html`, November 2010.

[11] G. Nigel N. Martin. Range encoding: An algorithm for removing redundancy from a digitized message. In *Video and Data Recording Conference*, July 1979.

[12] Mark Nelson. Arithmetic coding + statistical modeling = data compression. *Dr. Dobb's Journal*, February 1991.

[13] Video Game Content Extraction Project. Starcraft commands. `http://code.google.com/p/vgce/wiki/starcraftCommands`.

[14] Jorma Rissanen. Generalized Kraft inequality and arithmetic coding. *IBM Journal of Research and Development*, 20(3):198–203, May 1976.

[15] Claude Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 1948.

[16] Tim Sweeney. Unreal network architecture. 1999.

[17] The Doom Wiki. Doom networking component. `http://doom.wikia.com/wiki/Doom_networking_component`.

Appendix A

Library Architecture

This appendix gives a description of the classes and components built into our networking library, both to assist interested end-users, and to provide inside to anyone seeking to adapt these ideas into their own systems. This library is somewhat complex, but I have aimed to provide a very simple interface to the end user.

As shown in Figure A.1, there are three main classes to deal with.

- Schema: Encapsulates data about the layout of snapshot objects. This forms the basis for the statistical datasets learned by the server and client, as well as enabling a form of library-side reflection on the snapshot objects you provide on the server, and receive on the client. The clients should use a Schema object constructed in the exact same way as the one used on the server.
- ServerViewManager: Instantiated with a particular Schema, and used to manage a collection of RemotePeers, with which all ServerViews should be registered. Used to capture snapshots of the game state and transmit them to clients.
- ClientViewManager: Instantiated with a particular Schema, and used to receive a visible set of views from the server, and expose that state to clientside objects.

A.1 Schema

Instantiated with spooky::CreateSchema(), the Schema object consists primarily of a collection of Class objects. Each Class object, created by Schema::CreateClass(), is used to define the reflection properties of a particular type of snapshot. This includes registering the constructor, destructor, and static size of the snapshot, as well as registering pointers to the snapshot's various fields. This information is used internally by the engine to manage archives of snapshots correctly. Currently, two major types of fields are supported.

Numeric These fields can take on any integral or floating point type, and represent values that exist

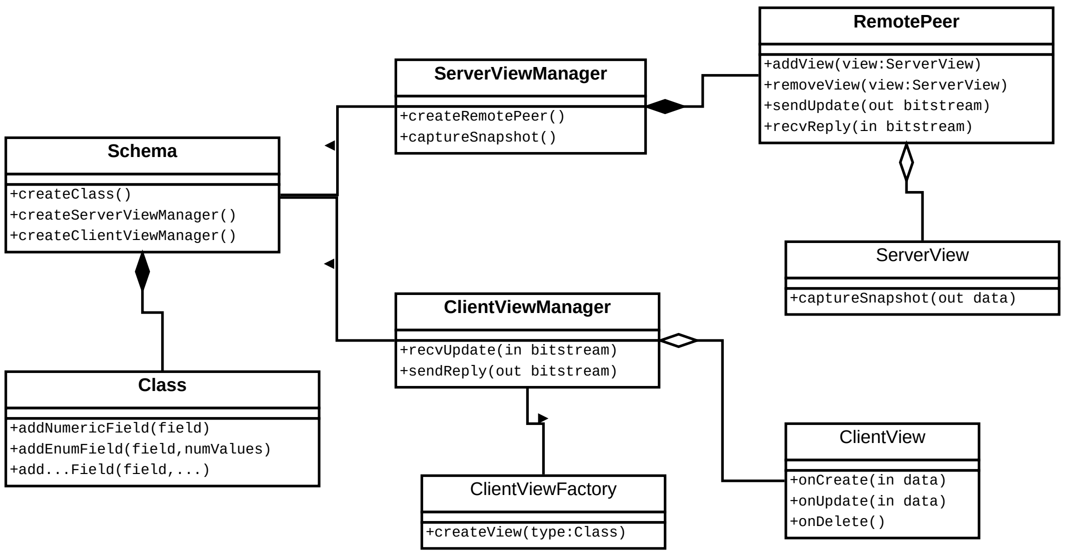


Figure A.1: UML Class Diagram

on a continuum. The library automatically makes predictions about these fields as described in Chapter 5, and the error deltas are transmitted as in Chapter 6.

Enumeration These fields represent specific states in a finite state machine. The library learns about the Markov transition matrix between states, and transmits each value using the conditional probability of a transition from the last acknowledged state to the present state. Boolean fields are handled as a simplified version of an enumeration, having only two values.

Once each class has been allocated and initialised, Schema::Finalise() should be called. The Schema is now ready to be used to instantiate ServerViewManager or ClientViewManager objects.

A.2 ServerViewManager

Instantiated with Schema::CreateServerViewManager(), the ServerViewManager object manages the task of archiving game state from frame to frame, as well as a collection of RemotePeer objects, created by ServerViewManager::CreateRemotePeer().

Each RemotePeer object represents a separate client's visible set. Implementations of ServerView should be added and removed from these visible sets separately via calls to RemotePeer::AddView() and RemotePeer::RemoveView(). Once each RemotePeer's set of views is correctly configured, ServerViewManager::CaptureSnapshot() should be called. This will record snapshots of all ServerViews visible to at least one RemotePeer, as well as snapshots of the visible set of each RemotePeer. There is no need to explicitly register or remove ServerViews from the ServerViewManager itself.

After this call, RemotePeer::SendUpdate() can be called for each peer, producing a binary message to be sent to the actual client in question. These operations can be done in parallel for each RemotePeer object, and can further be performed while the game is simulating, as all information

```
1. Instantiate a ServerViewManager via Schema::CreateServerViewManager()
2. While the simulation has not finished:
    (a) For each received network message:
        • If message from new client:
            – Instantiate a RemotePeer object
        • Else message from existing client:
            – Call RemotePeer::RecvReply() to handle reply from client
            – Optional: Handle user input, actions, messages, etc.
    (b) Parallel for each connected client:
        i. Add newly revealed/created views with RemotePeer::AddView()
        ii. Remove newly hidden/destroyed views with RemotePeer::RemoveView()
    (c) Call ServerViewManager::CaptureSnapshot() to capture the state of all visible views
    (d) Parallel operations:
        • Advance the simulation
        • Parallel for each connected client:
            i. Call RemotePeer::SendUpdate() to produce a binary message
            ii. Transmit this message over the network
```

Figure A.2: Main loop for server program (with multiple clients)

needed to perform the task of encoding updates has been recorded and archived by the ServerViewManager. The server's rough logical flow should match Figure A.2.

A.2.1 ServerView

Views are objects which inherit from the ServerView class. The only requirement of a view is that it implement the ServerView::CaptureSnapshot() method, which will capture some aspect of the server simulation in a data structure of the user's choosing.

Typically, a view will reference some particular object on the server, and it will capture some properties of that object, such as position or orientation. It is possible to have as many different types of views as you want, and different views could conceivably expose different properties of the same object. For instance, in video games, players are often permitted to know more about their own characters or units than they are about those of their enemies. This could be accomplished by having two views per object, one for "public" properties and one for "private" properties, with the latter only being exposed to the client who "owns" the object.

Simply put, a view is the means by which the network library is permitted to access the state of the simulation, by capturing snapshots to be archived and replicated to the clients. This particular abstraction was chosen to provide a clear layer of separation between game or simulation code and the library-friendly snapshot data structures used for synchronisation. The actual data stored in game objects can be of any type, use any sort of data structures, and can be kept private and only exposed

1. Connect to server
2. Create implementation of ClientViewFactory
3. Instantiate a ClientViewManager via Schema::CreateClientViewManager()
4. While connected to server:
   (a) Receive message from network
   (b) Call ClientViewManager::RecvUpdate() to interpret the update message
   (c) Call ClientViewManager::SendReply() to produce the reply message
      - Optional: Append user input, actions, messages, etc to reply message
   (d) Transmit reply packet to server
   (e) Update the client (render scene, capture input, etc)

Figure A.3: Main loop for client program

by observer methods. Under most circumstances, it should not be necessary for any game code to be modified to work with the library. It should be possible for implementations of ServerView to simply call existing methods on a pointer or reference to some game class, in order to fill out a snapshot structure.

Note that, if the user wishes for a more compact implementation, they can choose to have game objects themselves inherit from ServerView, and implement the ServerView::CaptureSnapshot() method to pull data directly from member variables. While this approach pollutes game logic slightly, it does allow game objects to be directly added and removed from the visible set of a RemotePeer object with RemotePeer::AddView() and RemotePeer::RemoveView().

A.3 ClientViewManager

A ClientViewManager is comparatively simpler than a ServerViewManager, because while the latter must serve many potential clients, the former interacts solely with a single server. It operates by receiving update packets, and generating ClientView objects corresponding to the ServerView objects on the remote machine. The rough logical flow of a client program is demonstrated in Figure A.3.

Where a ServerView observes simulation state and produces snapshots, a ClientView must simply receive snapshots and handle them in whatever way is appropriate. In the case of a graphical simulation, a ClientView might use snapshots to update the position of graphical elements such as sprites or 3D models, or it might change the text of GUI elements such as statistics, leaderboards, or chat messages.

In order to facilitate type safety, an implementation of the IClientViewFactory interface must be provided to ClientViewManager on construction. This factory will be used to instantiate the ClientView objects, and gives the library user a chance to construct and record ClientView objects however he pleases. For instance, he can insert newly created views into a list, and iterate through that list to render the objects the views represent.

A.3.1 ClientView

Classes which inherit from ClientView can choose to implement the OnCreate(), OnUpdate(), and OnDelete() callbacks, which are means by which the ClientViewManager communicates changes to a view to the user's application.

ClientView::OnCreate() and ClientView::OnUpdate() are called identically, with the current, most up-to-date snapshot data when a new frame has been received from the server. OnCreate() is called with the first snapshot received for a newly created ClientView, and OnUpdate() is called for all subsequent snapshots. This is to allow OnCreate() to perform additional setup work such as allocating scene graph objects and user interface elements corresponding to the visible object. OnUpdate() could then modify the positions, animations, or contents of existing graphics objects.

If schemes such as dead reckoning or cubic spline interpolation, from Chapter 2 are being used, then the user should incorporate their particular integration scheme into the ClientView object. Then, whenever OnUpdate() is called, the data received from the server can be used to introduce corrections to the integration process.

OnDelete() is called once the server indicates that a particular view is no longer in the client's visible set. It will only ever be called once, and once it is called, no further OnUpdate() calls will be made. It accepts no data, because the server stops sending data for removed views immediately.

Note that the ClientViewManager will not actually delete ClientView objects. As it instantiates views via the user-provided ClientViewFactory, it makes no assumptions about whether they came from the heap or from custom allocators, etc. Furthermore, it is likely that any real world client program will store references to ClientView objects in several places, which need to be cleaned up. For this reason, the user should use the ClientView::OnDelete() call to handle cleanup of a particular view, whether by setting a flag, inserting the object into a deletion queue, or simply invoking "delete this;".

A.4 Snapshots

Snapshots are the glue which binds the ServerViewManager and the ClientViewManager together, the means by which a ServerView communicates with its corresponding ClientView. Having this abstraction means that the actual ServerView and ClientView can be built around wildly different functionality, such as simulation objects and graphical representations. No attempt is made to replicate the actual server-side objects to the client, only the snapshots.

An actual snapshot is typically provided via a struct or class. Its fields must be registered with a Class object, so that the library knows how to interpret them.

1. ServerView::CaptureSnapshot() is called to take a snapshot

2. The snapshot is placed into an archive of snapshots corresponding to specific frames
3. A delta-representation of the snapshot is created and added to the next outgoing packet
4. The packet is transmitted over the network
5. The client decodes the snapshot from its delta-representation, and places it into its own archive
6. The snapshot is passed to ClientView::OnUpdate()

It is worth noting that the actual method of packet delivery is left up to the application programmer. The library is designed to function correctly when the data is transmitted over an unreliable protocol like UDP. It does not require any particular packet to arrive, packets to arrive in the correct order, or packets to contain valid data, and will gracefully handle packet loss, reordering, or corruption, though compression rates will slowly decrease if communication is cut off for an extended period of time (as the "shared context" becomes less and less relevant).

That being said, nothing prevents this library from being used over reliable protocols like TCP, or even over pipes or shared memory on a single machine, or any other communication mechanism. This allows the library to be layered into or combined with other networking libraries designed to handle specific functions such as VoIP, autopatching, or encrypted authentication protocols.

A.5 Licensing and Availability

While this library is still under development, it will be released shortly on Google Code as source code under a nonrestrictive BSD-style license. The library is written in standard C++ and makes limited use of the Boost family of libraries. It should function correctly on any platform with 32-bit integer support, and should produce endian-neutral messages that can be transmitted via the platform's native sockets API without further transformation. These properties will be verified and thoroughly tested prior to public release. In the mean time, anyone interested in receiving advance copies of the source code, or in continuing research in this area, should contact the author at sgorsten@gmail.com.

Appendix B

Bucketisation Policy Tables

The following tables define the five integer bucketisation policies mentioned in Chapter 6. The "Interval" column contains the interval of numbers which fall within a particular bucket. The intervals are left-inclusive and right-exclusive. The "Range" column indicates the quantity of integers which fall into a particular bucket.

The "Bits" field indicates the cost, in bits, to represent a number from this bucket via an arithmetic coder, assuming a uniform distribution, calculated as the base two logarithm of the range. Note that the 64-bucket policy has buckets whose lengths are a power of two, and whose cost of representation is a whole number of bits. These costs do not include the cost to indicate which bucket an integer falls into, which varies based on the specific distribution being approximated.

The reason the 94 and 124 bucket tables do not contain 96 or 128 buckets is that, under the exponential schemes by which they were laid out, some of their buckets would have contained less than one element. In those cases, the buckets in question were merged.

Table B.1: Bucketing Scheme (32 buckets)

Interval	Range	Bits	Interval	Range	Bits
[−2147483648, −536870912)	1610612736	30.58	[−536870912, −134217728)	402653184	28.58
[−134217728, −33554432)	100663296	26.58	[−33554432, −8388608)	25165824	24.58
[−8388608, −2097152)	6291456	22.58	[−2097152, −524288)	1572864	20.58
[−524288, −131072)	393216	18.58	[−131072, −32768)	98304	16.58
[−32768, −8192)	24576	14.58	[−8192, −2048)	6144	12.58
[−2048, −512)	1536	10.58	[−512, −128)	384	8.58
[−128, −32)	96	6.58	[−32, −8)	24	4.58
[−8, −2)	6	2.58	[−2, 0)	2	1.0
[0, 2)	2	1.0	[2, 8)	6	2.58
[8, 32)	24	4.58	[32, 128)	96	6.58
[128, 512)	384	8.58	[512, 2048)	1536	10.58
[2048, 8192)	6144	12.58	[8192, 32768)	24576	14.58
[32768, 131072)	98304	16.58	[131072, 524288)	393216	18.58
[524288, 2097152)	1572864	20.58	[2097152, 8388608)	6291456	22.58
[8388608, 33554432)	25165824	24.58	[33554432, 134217728)	100663296	26.58
[134217728, 536870912)	402653184	28.58	[536870912, 2147483648)	1610612736	30.58

Table B.2: Bucketing Scheme (48 buckets)

Interval	Range	Bits	Interval	Range	Bits
[−2147483648, −852229450)	1295254198	30.27	[−852229450, −338207482)	514021968	28.94
[−338207482, −134217728)	203989754	27.6	[−134217728, −53264341)	80953387	26.27
[−53264341, −21137968)	32126373	24.94	[−21137968, −8388608)	12749360	23.6
[−8388608, −3329021)	5059587	22.27	[−3329021, −1321123)	2007898	20.94
[−1321123, −524288)	796835	19.6	[−524288, −208064)	316224	18.27
[−208064, −82570)	125494	16.94	[−82570, −32768)	49802	15.6
[−32768, −13004)	19764	14.27	[−13004, −5160)	7844	12.94
[−5160, −2048)	3112	11.6	[−2048, −813)	1235	10.27
[−813, −323)	490	8.94	[−323, −128)	195	7.61
[−128, −51)	77	6.27	[−51, −20)	31	4.95
[−20, −8)	12	3.59	[−8, −3)	5	2.32
[−3, −1)	2	1.0	[−1, 0)	1	0.0
[0, 1)	1	0.0	[1, 3)	2	1.0
[3, 8)	5	2.32	[8, 20)	12	3.59
[20, 51)	31	4.95	[51, 128)	77	6.27
[128, 323)	195	7.61	[323, 813)	490	8.94
[813, 2048)	1235	10.27	[2048, 5160)	3112	11.6
[5160, 13004)	7844	12.94	[13004, 32768)	19764	14.27
[32768, 82570)	49802	15.6	[82570, 208064)	125494	16.94
[208064, 524288)	316224	18.27	[524288, 1321123)	796835	19.6
[1321123, 3329021)	2007898	20.94	[3329021, 8388608)	5059587	22.27
[8388608, 21137968)	12749360	23.6	[21137968, 53264341)	32126373	24.94
[53264341, 134217728)	80953387	26.27	[134217728, 338207482)	203989754	27.6
[338207482, 852229450)	514021968	28.94	[852229450, 2147483648)	1295254198	30.27

Table B.3: Bucketing Scheme (64 buckets)

Interval	Range	Bits	Interval	Range	Bits
[−2147483648, −1073741824)	1073741824	30	[−1073741824, −536870912)	536870912	29
[−536870912, −268435456)	268435456	28	[−268435456, −134217728)	134217728	27
[−134217728, −67108864)	67108864	26	[−67108864, −33554432)	33554432	25
[−33554432, −16777216)	16777216	24	[−16777216, −8388608)	8388608	23
[−8388608, −4194304)	4194304	22	[−4194304, −2097152)	2097152	21
[−2097152, −1048576)	1048576	20	[−1048576, −524288)	524288	19
[−524288, −262144)	262144	18	[−262144, −131072)	131072	17
[−131072, −65536)	65536	16	[−65536, −32768)	32768	15
[−32768, −16384)	16384	14	[−16384, −8192)	8192	13
[−8192, −4096)	4096	12	[−4096, −2048)	2048	11
[−2048, −1024)	1024	10	[−1024, −512)	512	9
[−512, −256)	256	8	[−256, −128)	128	7
[−128, −64)	64	6	[−64, −32)	32	5
[−32, −16)	16	4	[−16, −8)	8	3
[−8, −4)	4	2	[−4, −2)	2	1
[−2, −1)	1	0	[−1, 0)	1	0
[0, 1)	1	0	[1, 2)	1	0
[2, 4)	2	1	[4, 8)	4	2
[8, 16)	8	3	[16, 32)	16	4
[32, 64)	32	5	[64, 128)	64	6
[128, 256)	128	7	[256, 512)	256	8
[512, 1024)	512	9	[1024, 2048)	1024	10
[2048, 4096)	2048	11	[4096, 8192)	4096	12
[8192, 16384)	8192	13	[16384, 32768)	16384	14
[32768, 65536)	32768	15	[65536, 131072)	65536	16
[131072, 262144)	131072	17	[262144, 524288)	262144	18
[524288, 1048576)	524288	19	[1048576, 2097152)	1048576	20
[2097152, 4194304)	2097152	21	[4194304, 8388608)	4194304	22
[8388608, 16777216)	8388608	23	[16777216, 33554432)	16777216	24
[33554432, 67108864)	33554432	25	[67108864, 134217728)	67108864	26
[134217728, 268435456)	134217728	27	[268435456, 536870912)	268435456	28
[536870912, 1073741824)	536870912	29	[1073741824, 2147483648)	1073741824	30

Table B.4: Bucketing Scheme (94 buckets)

Interval	Range	Bits	Interval	Range	Bits
[−2147483648, −1352829926)	794653722	29.57	[−1352829926, −852229450)	500600476	28.9
[−852229450, −536870912)	315358538	28.23	[−536870912, −338207482)	198663430	27.57
[−338207482, −213057362)	125150120	26.9	[−213057362, −134217728)	78839634	26.23
[−134217728, −84551871)	49665857	25.57	[−84551871, −53264341)	31287530	24.9
[−53264341, −33554432)	19709909	24.23	[−33554432, −21137968)	12416464	23.57
[−21137968, −13316085)	7821883	22.9	[−13316085, −8388608)	4927477	22.23
[−8388608, −5284492)	3104116	21.57	[−5284492, −3329021)	1955471	20.9
[−3329021, −2097152)	1231869	20.23	[−2097152, −1321123)	776029	19.57
[−1321123, −832255)	488868	18.9	[−832255, −524288)	307967	18.23
[−524288, −330281)	194007	17.57	[−330281, −208064)	122217	16.9
[−208064, −131072)	76992	16.23	[−131072, −82570)	48502	15.57
[−82570, −52016)	30554	14.9	[−52016, −32768)	19248	14.23
[−32768, −20643)	12125	13.57	[−20643, −13004)	7639	12.9
[−13004, −8192)	4812	12.23	[−8192, −5160)	3032	11.57
[−5160, −3251)	1909	10.9	[−3251, −2048)	1203	10.23
[−2048, −1290)	758	9.57	[−1290, −813)	477	8.90
[−813, −512)	301	8.23	[−512, −323)	189	7.56
[−323, −203)	120	6.91	[−203, −128)	75	6.23
[−128, −81)	47	5.56	[−81, −51)	30	4.91
[−51, −32)	19	4.25	[−32, −20)	12	3.59
[−20, −13)	7	2.81	[−13, −8)	5	2.32
[−8, −5)	3	1.59	[−5, −3)	2	1.0
[−3, −2)	1	0.0	[−2, −1)	1	0.0
[−1, 0)	1	0.0	[0, 1)	1	0.0
[1, 2)	1	0.0	[2, 3)	1	0.0
[3, 5)	2	1.0	[5, 8)	3	1.59
[8, 13)	5	2.32	[13, 20)	7	2.81
[20, 32)	12	3.59	[32, 51)	19	4.25
[51, 81)	30	4.91	[81, 128)	47	5.56
[128, 203)	75	6.23	[203, 323)	120	6.91
[323, 512)	189	7.56	[512, 813)	301	8.23
[813, 1290)	477	8.90	[1290, 2048)	758	9.57
[2048, 3251)	1203	10.23	[3251, 5160)	1909	10.9
[5160, 8192)	3032	11.57	[8192, 13004)	4812	12.23
[13004, 20643)	7639	12.9	[20643, 32768)	12125	13.57
[32768, 52016)	19248	14.23	[52016, 82570)	30554	14.9
[82570, 131072)	48502	15.57	[131072, 208064)	76992	16.23
[208064, 330281)	122217	16.9	[330281, 524288)	194007	17.57
[524288, 832255)	307967	18.23	[832255, 1321123)	488868	18.9
[1321123, 2097152)	776029	19.57	[2097152, 3329021)	1231869	20.23
[3329021, 5284492)	1955471	20.9	[5284492, 8388608)	3104116	21.57
[8388608, 13316085)	4927477	22.23	[13316085, 21137968)	7821883	22.9
[21137968, 33554432)	12416464	23.57	[33554432, 53264341)	19709909	24.23
[53264341, 84551871)	31287530	24.9	[84551871, 134217728)	49665857	25.57
[134217728, 213057362)	78839634	26.23	[213057362, 338207482)	125150120	26.9
[338207482, 536870912)	198663430	27.57	[536870912, 852229450)	315358538	28.23
[852229450, 1352829926)	500600476	28.9	[1352829926, 2147483648)	794653722	29.57

Table B.5: Bucketing Scheme (124 buckets)

Interval	Range	Bits	Interval	Range	Bits
[−2147483648, −1518500250)	628983398	29.23	[−1518500250, −1073741824)	444758426	28.73
[−1073741824, −759250125)	314491699	28.23	[−759250125, −536870912)	222379213	27.73
[−536870912, −379625063)	157245849	27.23	[−379625063, −268435456)	111189607	26.73
[−268435456, −189812531)	78622925	26.23	[−189812531, −134217728)	55594803	25.73
[−134217728, −94906266)	39311462	25.23	[−94906266, −67108864)	27797402	24.73
[−67108864, −47453133)	19655731	24.23	[−47453133, −33554432)	13898701	23.73
[−33554432, −23726566)	9827866	23.23	[−23726566, −16777216)	6949350	22.73
[−16777216, −11863283)	4913933	22.23	[−11863283, −8388608)	3474675	21.73
[−8388608, −5931642)	2456966	21.23	[−5931642, −4194304)	1737338	20.73
[−4194304, −2965821)	1228483	20.23	[−2965821, −2097152)	868669	19.73
[−2097152, −1482910)	614242	19.23	[−1482910, −1048576)	434334	18.73
[−1048576, −741455)	307121	18.23	[−741455, −524288)	217167	17.73
[−524288, −370728)	153560	17.23	[−370728, −262144)	108584	16.73
[−262144, −185364)	76780	16.23	[−185364, −131072)	54292	15.73
[−131072, −92682)	38390	15.23	[−92682, −65536)	27146	14.73
[−65536, −46341)	19195	14.23	[−46341, −32768)	13573	13.73
[−32768, −23170)	9598	13.23	[−23170, −16384)	6786	12.73
[−16384, −11585)	4799	12.23	[−11585, −8192)	3393	11.73
[−8192, −5793)	2399	11.23	[−5793, −4096)	1697	10.73
[−4096, −2896)	1200	10.23	[−2896, −2048)	848	9.73
[−2048, −1448)	600	9.23	[−1448, −1024)	424	8.73
[−1024, −724)	300	8.23	[−724, −512)	212	7.73
[−512, −362)	150	7.23	[−362, −256)	106	6.73
[−256, −181)	75	6.23	[−181, −128)	53	5.73
[−128, −91)	37	5.21	[−91, −64)	27	4.75
[−64, −45)	19	4.25	[−45, −32)	13	3.7
[−32, −23)	9	3.17	[−23, −16)	7	2.81
[−16, −11)	5	2.32	[−11, −8)	3	1.59
[−8, −6)	2	1.0	[−6, −4)	2	1.0
[−4, −3)	1	0.0	[−3, −2)	1	0.0
[−2, −1)	1	0.0	[−1, 0)	1	0.0
[0, 1)	1	0.0	[1, 2)	1	0.0
[2, 3)	1	0.0	[3, 4)	1	0.0
[4, 6)	2	1.0	[6, 8)	2	1.0
[8, 11)	3	1.59	[11, 16)	5	2.32
[16, 23)	7	2.81	[23, 32)	9	3.17
[32, 45)	13	3.7	[45, 64)	19	4.25
[64, 90)	26	4.7	[90, 128)	38	5.25
[128, 181)	53	5.73	[181, 256)	75	6.23
[256, 362)	106	6.73	[362, 512)	150	7.23
[512, 724)	212	7.73	[724, 1024)	300	8.23
[1024, 1448)	424	8.73	[1448, 2048)	600	9.23
[2048, 2896)	848	9.73	[2896, 4096)	1200	10.23
[4096, 5793)	1697	10.73	[5793, 8192)	2399	11.23
[8192, 11585)	3393	11.73	[11585, 16384)	4799	12.23
[16384, 23170)	6786	12.73	[23170, 32768)	9598	13.23
[32768, 46341)	13573	13.73	[46341, 65536)	19195	14.23
[65536, 92682)	27146	14.73	[92682, 131072)	38390	15.23
[131072, 185364)	54292	15.73	[185364, 262144)	76780	16.23
[262144, 370727)	108583	16.73	[370727, 524288)	153561	17.23
[524288, 741455)	217167	17.73	[741455, 1048576)	307121	18.23
[1048576, 1482910)	434334	18.73	[1482910, 2097152)	614242	19.23
[2097152, 2965821)	868669	19.73	[2965821, 4194304)	1228483	20.23
[4194304, 5931642)	1737338	20.73	[5931642, 8388608)	2456966	21.23
[8388608, 11863283)	3474675	21.73	[11863283, 16777216)	4913933	22.23
[16777216, 23726566)	6949350	22.73	[23726566, 33554432)	9827866	23.23
[33554432, 47453133)	13898701	23.73	[47453133, 67108864)	19655731	24.23
[67108864, 94906266)	27797402	24.73	[94906266, 134217728)	39311462	25.23
[134217728, 189812531)	55594803	25.73	[189812531, 268435456)	78622925	26.23
[268435456, 379625063)	111189607	26.73	[379625063, 536870912)	157245849	27.23
[536870912, 759250125)	222379213	27.73	[759250125, 1073741824)	314491699	28.23
[1073741824, 1518500250)	444758426	28.73	[1518500250, 2147483648)	628983398	29.23