Expertise
and insight
for the future

Ramon Brand

Defining a Keyframe Based Protocol for Optimising Network Updates of Large Sets of Entities

Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Bachelor's Thesis

07 March 2019

Author	Ramon Brand
Title	Defining a Keyframe Based Protocol for Optimising Network Updates of Large Sets of Entities
Number of Pages Date	52 pages + 2 appendices 26 November 2018
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Professional Major	Software Engineering
Instructors	Peter Hjort, Supervisor

Large sets of entities often need to be synchronised through a network in real time, requiring a high data rate in order to update the entities' properties in a tight update cycle. This thesis aims to propose and formalise a Open Systems Interconnection layer 6 network protocol as a potential solution to reduce the required data rate when updating large sets of entities in real time, by exploring the advantages and disadvantages of using keyframes as a means to encode value change over time.

The main use case explored is that of Real Time Strategy games, and the need to synchronise the game state to all players on the network, where each player possesses a high number of individual entities. Due to the nature of Real Time Strategy games, the proposed protocol possesses attributes which apply very specifically to the said case.

The proposed protocol is compared to existing solutions in the specified domain, followed by a technical specification of the protocol. Theoretical data transfer calculations are conducted, comparing the proposed protocol to a brute force approach, from which a breakeven formula is derived. Furthermore, additional considerations and benefits are determined, all of which could then be used to determine the suitability of the protocol in any target system.

It was found that the implementation of the protocol is significantly more complicated than a brute force implementation. In addition, the protocol is only more efficient than a brute force approach when very specific characteristics are present in the target system, including linearity and predictability in value change. It was, however, discovered that the protocol has some very useful attributes, such as the ability of sampling and rewinding state on the client, and due to persistent historical state on the client, the client is able to perform heuristic analysis on the state, including future client-side predictions. In addition, the protocol's ability to send data in advance has the added benefit of mitigating network latency in many cases.

It was concluded that the proposed protocol can be used to efficiently reduce network traffic in very specific use cases, with the downside of increasing computation time on the client. The protocol is thus recommenced for any situation where the breakeven formula indicates a favourable efficiency and the bottleneck of the target system is the network traffic and not the processing time.

Keywords	network keyframe protocol data transmission

Contents

List of Abbreviations

ARPA	Advanced Research Projects Agency
ASCII	American Standard Code for Information Interchange
AVL	Adelson-Velsky and Landis
BSD	Berkeley Software Distribution
CoH2	Company of Heroes 2
CPU	Central Processing Unit
DDoS	Distributed Denial of Service
FPS	Frames Per Second
KP	Keyframe Protocol
KPF	Keyframe Protocol Frame
MTU	Maximum Transmission Unit
NPM	Node Package Manager
OSI	Open Systems Interconnection
PLP	Pipelined Lockstep Protocol
RFC	Request for Comment
RLE	Run-Length Encoding
TCP	Transmission Control Protocol
UDP	User Datagram Protocol

1. Introduction

Transferring real time data through a network is inherently limited by the bandwidth the network is able to support, and as such, the ability to sufficiently reduce the needed data transfer over the network is critical. Striving for the reduction of network traffic is a task often considered to be of high priority. In many real life systems, namely systems in which data is constantly changing, such as multiplayer video games, the ability to reduce the network traffic generally coincides with the ability to enhance the user experience. As an example, multiplayer games need to share data between multiple hosts in the system in order for all players to view and experience the changes present in the game. Due to the ever-growing complexity of video games, the amount of information that needs to be synchronised between players is constantly growing, and is increasingly pushing the limits of what most consumer level networks can handle. For this reason, game functionality is often sacrificed in order to reduce the needed data transfer between players, often negatively impacting the game itself.

Reducing the amount of data that needs to be transferred over the network while maintaining the ability to deduce the desired result, allows lower bandwidth networks to be used, or systems to be expanded while remaining within the bandwidth limitations of a given environment. If multiplayer, network based games are able to more efficiently send the needed information to all players, the games will be given room to add complexity and functionality, which could lead to a richer gameplay experience for players. Over the years of network based game development, developers have been trying to find ways of compressing and reducing the needed data. A selection of these historical solutions are explored in this thesis, and the advantages and disadvantages of each analysed.

The aim of this thesis is to define a keyframe based network protocol to reduce the needed data transfer in the specific domain of high volume entity property changes present in Real Time Strategy (RTS) games and to evaluate its performance against a common baseline implementation. Initially starting with the explaining of the exact domain and the concepts of the protocol, the thesis proceeds to define the protocol specification itself, followed by calculations of data transfer and compare the keyframe protocol to a brute force implementation. Critically, a breakeven formula is derived which will serve as an indicator of applicability for any project considering the usage of the protocol. Following this, a Typescript implementation of both the client and server is developed and studied. Finally, the thesis delves into the testing of the protocol implementation.

2. Theoretical Background

In some cases more than a single entity needs to be updated through a network. In a common situation where a server needs to update many entities on a client, it is often not of utmost importance to optimise the transfer of data, because the additional computation time is of little consequence in a static system. However, in a system where the entities are constantly changing and are required to stay in sync on all clients, the optimisation of the update process becomes very important, because if the update process takes too much time, the process may not be completed in sufficient time for the system to be able to achieve it's required functionality, such as the real time streaming of the data in question.

The general requirement for real time updating of data is present in a multitude of computation systems; however, one specific system which will be focused on in this thesis, and forms a practical example, would be a multiplayer game which contains real time updates that need to be applied to all hosts. It is important that the time it takes to apply the update does not exceed the time between the updates, which would cause the system to run at a pace slower than real time, resulting in a cascading delay effect, common in ill performing multiplayer games. What this means in practice is that players are not able to see the changes in the game as they are happening. In the case of a game where small increments in time are crucial, such as in RTS games, the delayed data, or in extreme cases, indefinitely incorrect data, could result in the player making gameplay decisions on incorrect data, or is even unable to play entirely.

2.1. Specific Case

The specific case of focus will be a client-server network based game system in which the players are clients on the network, each managing a collection of entities, which respond to player control, and are able to communicate with the server, but not directly with each other.

Note that the server is authoritative and is the sole host which is able to make a change to the entities in all cases, such that the client will simply send a request to the server, asking for a change to be made, as opposed to the client directly manipulating the entities itself. The actual data traffic of the 'change request' will be considered negligible since the size, frequency and complexity of the change requests sent from the client to the server is dependent specifically on the system model, and it is theoretically possible for a system to have an entirely server controlled model, where the client nev-

er sends data to the server, and the server is the sole controller of all logic, thus eliminating the need for any client to server communication with regards to entity manipulation.

Furthermore, the case in question specifically contains a large number of entities, of which many possess multiple properties, which change over time, often in a somewhat predictable fashion. Although this may be a subjectively vague criteria, it fits the description of RTS games perfectly, and as such, RTS games will be the primary candidate for which this thesis is attempting to develop a solution with regards to network traffic.

In order to further understand why this case matches with the use case of RTS games, one needs to consider the context of the data RTS games generally transfer. RTS games generally allow players to control a large number of entities, such as units, pawns or board pieces, each containing properties determining their state, such as position, health etc. Due to the nature of play, which includes movement along paths, long lasting actions such as building and demolishing structures, and the overall change of gameplay entities over time, the properties of the entities in an RTS game are often quite predictable and linear in nature. This predictability and linearity is a key aspect as to why RTS games stand to benefit from the proposed solution, and will be clarified extensively further on in this thesis.

2.2. Historical Solutions

The specific case described in the previous section will also be used as the basis for exploring historical solutions. It is important to remember that not all solutions explored were intended to solve the specific case the solution is being evaluated against, and may thus be an unfair representation of the usefulness or effectiveness of the solution as a whole. Rather, the solutions explored are methods which have been attempted to be used in the specific case, and are presented in order to show the shortcomings, and need for a more domain specific solution.

In the early days of the consumer internet, around 1998, networks had much tighter bandwidth constraints, with the common, consumer level V.90 modem being able to download at a maximum rate of 56kbit/s and upload at a maximum rate of 33.6kbit/s (International Telecommunications Union 2008). In order to further put these numbers into perspective, it can be mathematically deduced that only 1,750 32 bit floating point numbers could be downloaded in a second.

It quickly became apparent to game developers, that an efficient solution for updating large sets of entities will have to be found if RTS games were to successfully exist. As such, a multitude of methods were devised over time for compressing, packing and cutting down on data, some of which still form the basis of the methods used in today's RTS games. As is common with software development, these methods have evolved, improved and specialised over time and have often gotten increasingly complex, making them less and less accessible to smaller game development studios.

As a baseline, a brute force solution to the problem would be to send the state of each entity to each client from the authoritative server at as quick a rate as possible. Implementing a brute force synchronisation model is generally very simple, since the property values are simply sent over the network as is, making it a very attractive, first choice for game developers needing an easy solution.

The problem with the brute force approach becomes apparent as the number of entities grow for two major reasons. Firstly, the amount of data that needs to be transferred increases linearly with respect to the volume of the data. This means that updating two 32 bit floats will require the transfer of twice the network data as compared to a single 32 bit float, which can be said to have a time complexity of O(n) in Big O notation (Das 2006, p. 8-9). At first this may seem obvious and reasonable, and in some applications it may be reasonable, such as the protocol presented in this thesis, however, it does not necessarily need to be the case.

To further illustrate this point, Run-Length Encoding (RLE) can be explored, which is a very simple form of lossless compression. Let the assumption be made that a system contains 8 values of 8 bits each, which could for example be a string containing 8 American Standard Code for Information Interchange (ASCII) characters. Using a brute force approach, the 8 values would be transferred as is, which would mean that the total data transfer volume of the 8 values is 8 times the size of a single value. However, if the values are somewhat nonrandom, it could be more efficient to encode the number of occurrences of each value, rather than just repeating the value itself. For example the string 'AAAABBBB' would be sent as 'AAAABBBB' using a brute force approach, but as '4A4B' when encoded with RLE. Note that 8 bits can encode a number from 0 to 255, meaning up to 256 occurrences could be encoded into a single value's size in this specific example. (Pu 2005, p. 49-56.)

What is important to note, is not the specific use of RLE, but the concept behind compression, and the relation of compression to data volume. As the volume of data becomes larger, the possibility for patterns become greater, thus decreasing the entropy,

and so the compression factor improves (Pu 2005, p. 34-38). This results in a better than O(n) time complexity (Das 2006, p. 8-9).

Entropy is an important factor when dealing with compression, and is a way of determining the lack of predictability in a stream of data. Entropy is always at its maximum, when all possible outcomes in a data stream are equally likely. Therefore, when the predictability of data increases, the entropy goes down, thus forming the fundamental idea of what entropy represents, which is the number of questions that needs to be asked to be able to predict the information contained in the data. Entropy is calculable from the following formula: (Pu 2005, p. 34-38.)

1. $$H = \sum_{i=1}^{n} p_i \times log_2\left(\frac{1}{p_i}\right)$$

This formula shows that the entropy **H** is the sum of the probability of all symbols multiplied by the base 2 log of the number of outcomes, which is $\frac{1}{p_i}$.

The brute force approach could potentially benefit from a form of compression such as RLE, since in the use case of RTS games the behaviour of the entities are predictable by nature, due to the repeating values of stationary entities, multiple entities changing in the same manner and the linear change present in many entity properties, which would mean the entropy is low, and would translate to a high compression factor (Pu 2005, p. 34-38). However, for the purposes of this thesis, the brute force implementation will be explored without any compression.

The second major reason the brute force approach becomes problematic, is that the brute force algorithm does not take change into account, and transmits the value of a property, regardless of the rate of change of the property. In practice, any value which needs to be synchronised in the game, even if the value is only changed once during the lifetime of the system, takes up its value size in network bandwidth on each update. Often times in RTS games, players possess many entities which store a position, consisting of x, y and z coordinates, each of 32 bits. In many cases, the majority of time is spent with these entities not moving in the game world, meaning these values do not change. When using the brute force approach, these values will be sent to all players, on every game tick, resulting in essentially redundant information being sent over the network.

The game MiMaze attempted to use a distributed architecture to reduce the load on the network (Diot 1999, p. 3). Distributing refers to the data being sent not only from the server to the clients, but from clients to each other, reducing the load on a single network connection, namely the connection to the server (Diot 1999, p. 3). Although removing the central point through which all data must flow, the server, does remove the bottleneck effect with the server, it does introduce the problem of authority. Since clients send data to each other, they are able to alter the data being sent further in any way, which could invalidate the system. This is particularly problematic as players are able to cheat by simple sending favourable data to other clients. The problem is not present when a central server is used, such as in the study case presented, since the server can simply ignore any invalid data sent to it from the client, and not pass the data along to other clients, thus the server processes full authority.

Another possibility would be to send an initial state to the clients, and have the clients derive the following state from actions performed on the initial state, where only the actions are sent to the clients after the initial state. This is known as a lockstep system and has proven to be a viable solution in many cases, since the actions are often discreet, and can be encoded onto a small data footprint. As an example, a movement action can be examined. Instead of sending the position of an entity to the client each frame, the server can simply send an action containing the task, such as 'move', and any needed arguments, such as the destination coordinates. The action is only required to be sent to the client at the onset of the movement, and the client is able to calculate the position of the entity knowing its behaviour.

However, lockstep systems have some major disadvantages, one being that the computation occurs on the client, which means all the clients need to ensure they are able to compute values exactly the same, or the system would fall out of sync over time (Liljekvist 2016, p. 12). Although it may seem like a trivial issue, since computers are able to calculate without errors, the issue arrises when specific Central Processing Units (CPUs) deal with arithmetic differently, such as with decimals in floating point numbers. These errors might be very small at first, but compound and exponentially grow to the point where the system fails. This phenomenon is known in the field of physics as chaos theory. (Oestreicher 2007.)

A practical example of this can be seen in a game called Company of Heroes 2 (CoH2), where players on a specific operating system are only able to play with other players using the same operating system (Ellie 2015). This is due to the fact that some internal algorithms will be calculated differently depending on the operating system (Ellie 2015). As an example, it is possible for both players to have the same state, an

entity at a specific position in the game world. One player decides to issue the 'move' command to move the entity to another location in the game world. The player does this by sending the requested command to the server which determines that the move is valid, and the server responds with the 'ok' to all clients containing the command that is needed to move the entity to the requested positions. At this point all players' clients are computing the path to move the entity from the start to end position, as they have been given the 'ok' from the server. No cheating has been involved in the process, but ultimately the 'path finding' is down to the client, which means nothing is stopping the client from finding a different path for the entity to move along compared to other clients or server. Taking it further, it is possible that midway during the transition, the player issues the command to stop the movement of the entity. At this point the question can be asked where the entity is located. This is because mid-way through two different paths are two different positions. At this point the game is fully desynchronised, and will not return to synchronisation.

Another, similar on the surface, but fundamentally different issue, is that in order for the client to be able to calculate the state of an entity given an action performed on the entity, the client needs to understand the behaviour of the entity. At first this may seem the same as the previous issue discussed, but although the result may be the same, the underlying architectural cause is not. The previous issue looked at the differences in calculation on different clients, whereas this issue focuses on the fact that the client needs to have an understanding of the game itself. This may seem trivial, but further in the thesis it will be explored how and why a 'dumb' client, which acts purely as a 'viewer of state' has benefits.

In addition to falling out of synchronisation, the lockstep architecture still allows the client to cheat, since they are in theory able to make a decision after receiving the opposing player's move (Liljekvist 2016, p. 11). As explained by Liljekvist, the cheating problem can be solved at the cost of a delay, which was formalised with the introduction of the Pipelined Lockstep Protocol (PLP), which is an optimisation of the basic lockstep implementation.

2.3. Proposed Solution

Originally appearing in a blog from a developer on the video game Planetary Annihilation, by the name of Forrest Smith, the idea arose to use keyframes to represent a value over time (Smith, 2013). The idea would be to plot the values of entity properties at a specific time, and later interpolate between the values with a lookup operation.

This concept of keyframe based interpolation and its respective data transfer can be wrapped into a clearly defined protocol, which will be called the keyframe protocol (KP) and is bespoke to this thesis and is not in reference to an existing protocol.

In overview, this solution has its own advantages and disadvantages. One aspect to remember is that the KP intends to solve a very specific problem, leading to one of its disadvantages, which is its narrow scope of use. During the exploration of this solution, throughout this thesis, including the network data calculations section, it is mentioned that the property being synchronised will only benefit from this method in very specific cases, and is not at all a method of general data compression, such as the previously explored RLE.

At the centre of the KP is the idea of storing a value in the client not as an instant value which is applied to the entity, but rather as a value with its corresponding time, being the time at which the property is the given value, and thus the lookup of a value at a given time is done, rather than an application of a value at an update interval. It should be noted here that the client is in control on what is presented to the user, since the client can lookup the values of the properties at any point in time. This is in contrast to the brute force approach where the client is required to apply the change immediately, since it is not known which time the value is valid for.

This lookup can be visualised as a chart, with the values plotted at their corresponding times. The introduction of the chart adds a very important aspect to the system, which is a second dimension, and allows the client to intemperate the values over time. In a system where the values are applied to the entities as soon as they are received contains only one dimension in practice, because the client is unable to make any judgement as to which point in time the value is valid for, other than for the immediate moment when the value is received.

As previously mentioned, in the specific domain of exploration, namely RTS games, it is often the case where the properties of the entities are changing in a very predictable manner, meaning the server very often might know the future values of a property, such as entity positions when the entity is moving. In a brute force implementation of the system, the server will need to hold on to the information it already knows, and is only able to send the information to the client at the moment the client should apply the values. However, in the case of including the time dimension with the value, the server is able to send values to the client at any point, which will then only be applied at a later time by the client itself.

Values are represented in a two dimensional space, meaning with each value comes an associated time, forming a tuple, or set of coordinates. This tuple of time and value can be though of as a keyframe, representing a given value at a given time.

2.4. Spread

Note that the simplest solution would be to send a keyframe every update cycle, which is placed at the current time, thus having the client always doing a lookup after the most recent keyframe, resulting in the same brute force behaviour described earlier, only with the additional overhead of the keyframe time data included in the transmission, which is clearly not beneficial. In order to extract value from the KP, the keyframes will have to occur less frequently to maintain a constant data rate, since the keyframes are larger in data size than the values themselves.

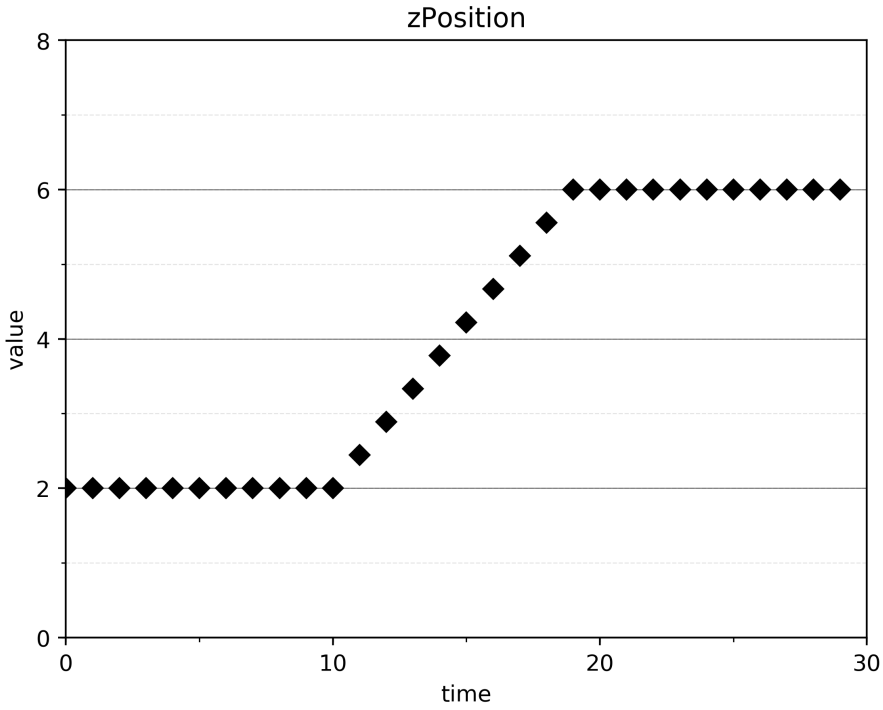

Figure 1. zPosition of entity over time

It needs to be determined how large a time gap there should be between keyframes, since increasing the gap will reduce the data rate, but potentially reduce the resolution of change, and decreasing the gap will increase the data rate, but potentially improve the resolution of change. Replicating the brute force approach would see a keyframe every tenth of a second, which will of course be grossly inefficient since the values which were present in the brute force approach are still present, and are being transferred at the same rate, but now also include the data related to their time. However, for the sake of clarity, Figure 1 shows the chart representation using keyframes of a property, in this example the zPosition of an entity.

Let the assumption be made that the x axis of the chart is time, and the y axis is the value of the property. Note that the plot is representative of a single property of a single entity over time, rather than in the brute force approach where an update is representative of all the properties of all entities at a single point in time.

As is explained in the protocol specification section of this thesis, the KP specifies that the client interpolate between the keyframes, however, for this example, let the assumption be made that the client will update the value of the property only when there is a keyframe present at the specific time, thus an update will occur every tenth of a second, and result in exactly the same behaviour as the brute force approach, only

with an increase in the data sent to the client. The only purpose of this example is to show that it is possible to achieve the same behaviour as present in the brute force approach. Although it might seem as though the use of the KP has just reduced the efficiency and added unnecessary complexity, the benefit arrises when interpolation is introduced.

2.5. Interpolation

Since the values are represented with their specific associated time, it is possible to interpolate between them. Interpolation provides multiple benefits, of which the first is the ability to ignore the keyframes which lye on the linear interpolation line between other keyframes, thus reducing the number of keyframes which need to be sent to the client. The second benefit is that the client is free to update the property value as often as it wants, and will be a smooth transition between the values based on the precise time, improving even on the tenth of a second increments from a brute force implementation. This also allows different clients to sample the data at different rates, while maintaining absolute accuracy with the server simulation, irrespective of the number of updates being performed client side. Client side update rate could vary due to rendering ability or client preference, but as long as the client receives all the keyframes, the rate at which the values are sampled has no bearing on the accuracy of the values.

It needs to be considered that interpolation is not always the desired behaviour. For example, discreet values, such as a boolean indicating if a specific player is able to control an entity might change over time, but can only exist in two discreet states, namely on or off. Interpolating between these states does not make logical sense. The ability to control the client interpolation, or lack there of, is explained in the optimisations section of this thesis.

In general, interpolation can be done using various methods, including linear, cosine and cubic. Linear interpolation produces a straight line between two points, and is the most computationally lightweight method of interpolation (Bourke 1999). Cubic interpolation allows smoother interpolation where sharp corners are not present on the keyframe joins, however, requires additional computational power to evaluate, which is of utmost importance since the evaluation will need to take place on each update cycle on the client for each property. An additional concern with cubic interpolation is that the absolute interpolation curve can not be deduced by just two points, but rather depends on surrounding keyframes, which increase complexity, and reduces versatility. Going back to the specific use case of RTS games, it can be said that much of the behaviour

of change is linear in nature, be it constant entity movement speed or constant change in health, and thus linear interpolation lends itself well to the use case in question.

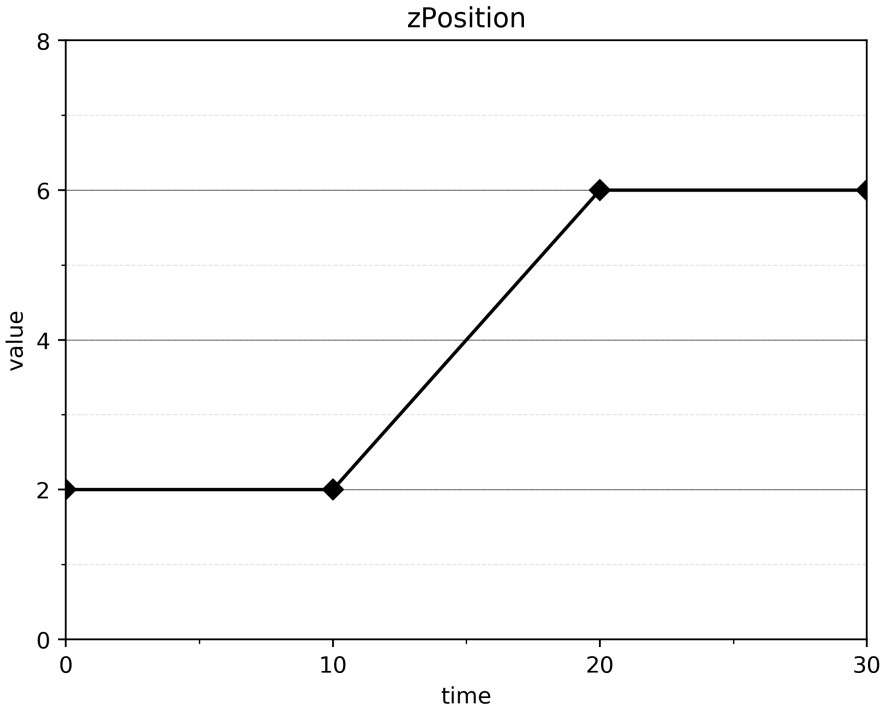

Figure 2. zPosition of entity over time with interpolation

The example in Figure 2 demonstrates the use of linear interpolation between the keyframes. All the needed information for the time period is present in the four keyframes, each with a time and value. It is important to note that the four keyframes contain all and more information than the previous example in Figure 1 since the KP will result in the identical values at the given times in Figure 1 but will also smoothly interpolate between the keyframe times present in Figure 1. This all goes to say that Figure 1 does not contain any more information than Figure 2, but Figure 2 requires significantly less data to communicate the same information.

It is important to note, that the only data that now needs to be sent to the client are the four keyframes, since the client will be able to interpolate between them, given the time. Having more keyframes as in Figure 1 does not improve the interpolation at all, because the keyframes lie on the interpolated line; thus they provide no additional information.

Commonly when using a brute force update cycle, in order to increase data fidelity, the number of updates per second is simply increased, and this would make logical sense since the client would then update the target value more often. However, in the case of the KP, the purpose of the keyframe is different and should not be thought of as a drop

in replacement for an update, but rather as a tool to manipulate the shape of a curve, which represents the value of the property over time. The target property should no longer be thought of as having an absolute value, but rather as a curve whose shape is determined by the placement of keyframes.

Keyframes have been used as a method of denoting points for interpolation in a variety of sectors, including web development and Computer Generated Imagery (CGI) animation (Jackson et al 2018, s. 4). In some cases, parallels can be drawn to how keyframes are used in the KP, such as with Cascading Style Sheets (CSS) where keyframes are used to denote the stages of style animations, such as button colour and movements, and can be thought of as representing properties in the KP system.

2.6. Considerations

Querying the value of a property at a given time is the entire purpose of the KP system, and as such the KP system needs to ensure that a lookup for a value of a property succeeds in all cases. In an ideal situation, the lookup is able to interpolate between two keyframes, resulting in a smooth transitional value for the property. However, it is possible that the property does not contain keyframes on both sides of the lookup time, in which case the interpolation will fail.

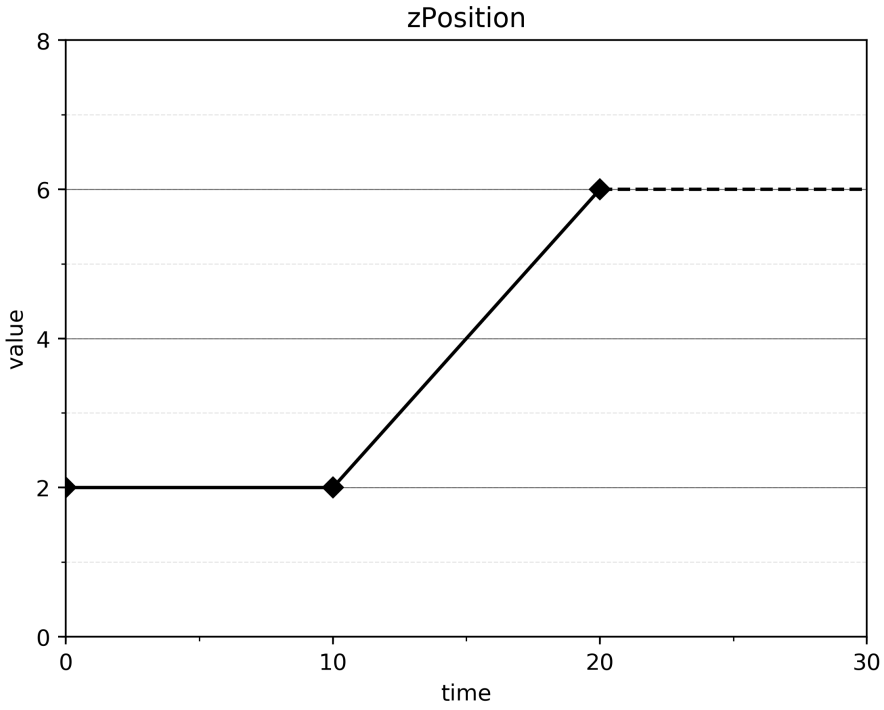

Figure 3. zPosition of entity over time without last keyframe

2.6.1. Lookup Failure

In order to ensure a valid value is returned, the KP system needs to consider the given situation and define a specification as to how it is handled. Possible options include a 'nearest' value, which simply uses the value which is closest to the lookup time, of which an example can be seen in Figure 3.

Although it may seem unintuitive, flatlining the value of the most recent keyframe if no further keyframe exists is a viable solution in most cases, since this would mean if there is no update, the last known value would be assumed, which is the exact same behaviour as with the brute force approach, where the client value is never updated unless an update is received from the server. The last keyframe would simply serve as the last update served in the brute force implementation.

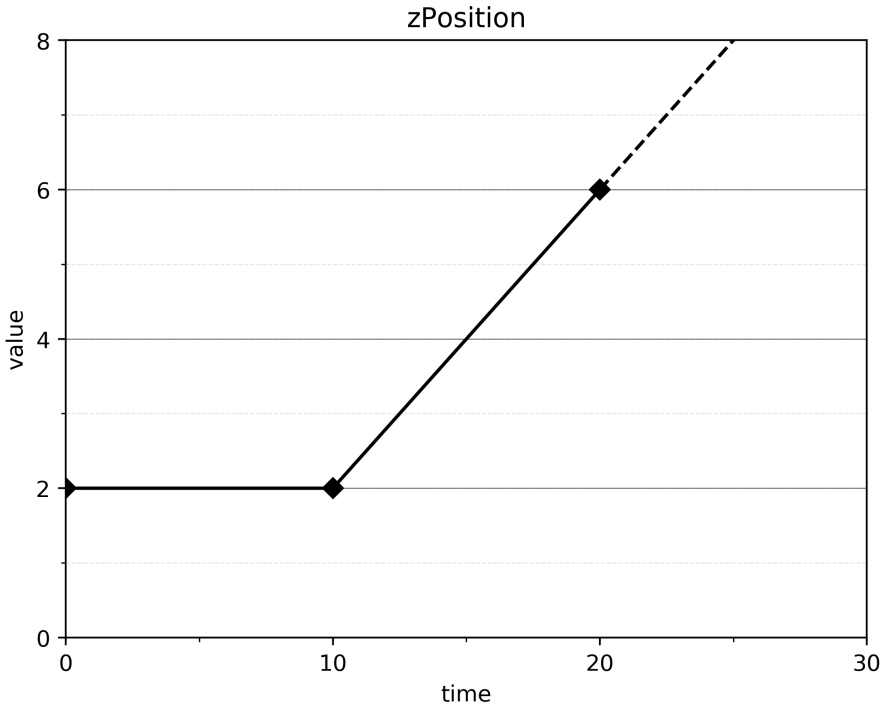

Figure 4. Extrapolated value after last keyframe

Another possibility is to sample past the last keyframe, and extrapolate the value rather than flatlining it; however, this is usually less useful since it is possible for the value to extrapolate outside of a valid range. If the value is flatlined, it can be assumed that at the very least, the value was once valid (Bourke 1999). This can be seen in Figure 3, which shows the same deduced values as Figure 2, while only using three keyframes, and assuming the value of the last keyframe when no further keyframes exist, where Figure 4 shows the last value being extrapolated past the last keyframe. Note that the

same logic can be applied when no preceding keyframe exists; thus the value of the nearest keyframe after the evaluated time is used.

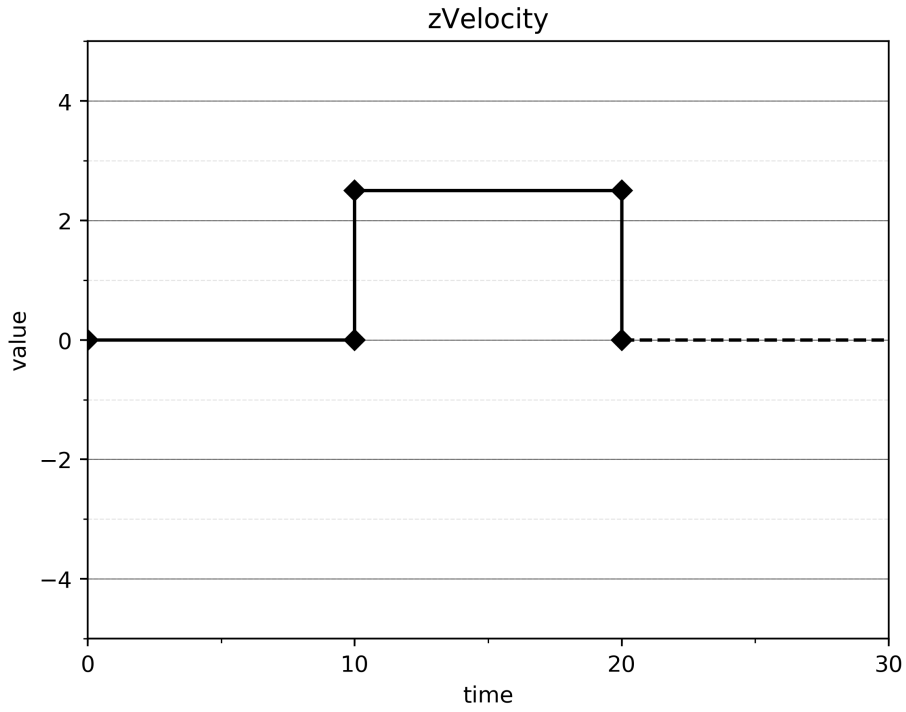

Figure 5. zVelocity instead of zPosition

2.6.2. Smoothing Value Change

A further consideration would be smoothing the change of the values, such as the position not changing at a constant velocity, but rather the property accelerating to a specific rate of change. Given the example in the Figure 3, it would require many keyframes be added to round the corners. However, a potential solution to this problem would be to simply store the first derivative of the desired property instead, such as in Figure 5, which in this case would be velocity.

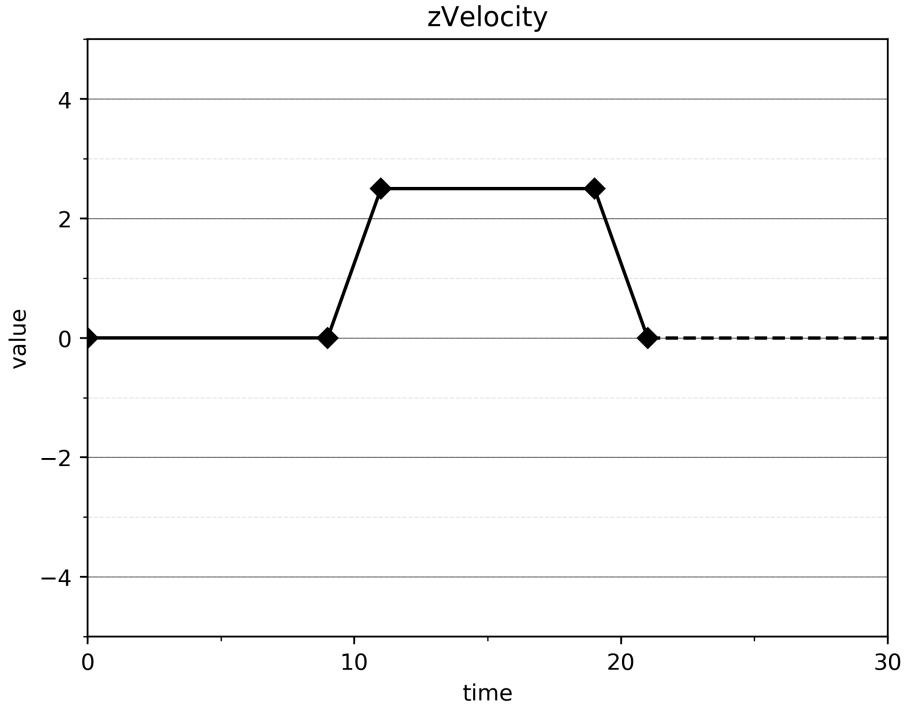

Figure 6. zVelocity with smooth acceleration

The slope of the velocity curve, which is the change in velocity, can then be used to control the acceleration, such as in Figure 6. This also presents a viable alternative to the issue previously presented in which cubic interpolation would be used to smooth the transitions of values.

2.6.3. CPU Time

In addition to the smoothing of the value curve, the CPU time needs to be considered.

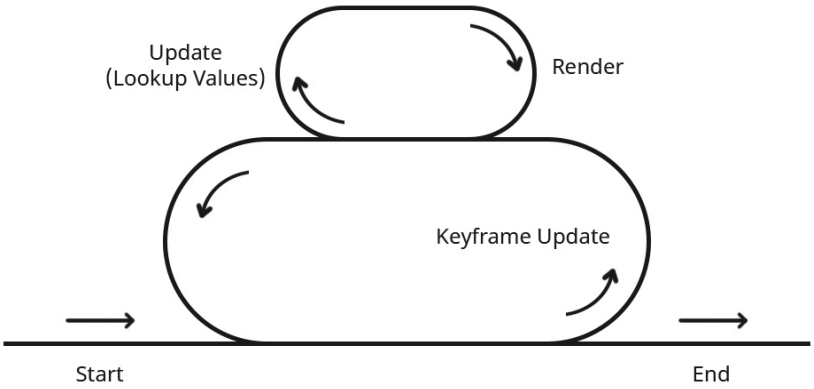

Figure 7. Game loop within game lifecycle

Taking a look at the game loop presented in Figure 7, it can be seen that for each property, the value needs to be queried on each rendered frame. This means that in a common client game environment, which runs at 60 Frames Per Second (FPS), each property needs to have its value queried, which includes an interpolation between two keyframes, which each need to be found. This could turn out to be very costly for the client CPU. Further exploration on lookup performance is done later in this thesis, including the introduction of specific data structures that could be used to improve lookup time.

The very principle of needing a lookup on each rendered frame introduces an important aspect of the KP system, which is that the proposed system intends to reduce the required network data transfer, but does so at the expense of CPU time on both the server and client, namely creating packets and looking up values. It is often the case that the network does form the bottleneck in the system, in which case the KP system addresses the problem, however, if the system is CPU time constrained, and the network does not form the bottleneck, the use of the KP system is nonsensical.

The same holds true for high frequency properties that behave in an erratic, unpredictable and non-linear fashion. The use of the KP system needs to be reconsidered if it would be logically difficult to create a curve of value over time with keyframes. The same applies to a system in which the number of properties is excessively massive, since the CPU overhead for each property using the KP is significantly higher than a simple brute force approach in which the values are simply assigned directly.

2.6.4. Latency

Lastly, the latency of the network needs to be considered and understood. It is important to understand that the KP will not improve the network latency, but merely attempt to reduce the amount of data being transferred over the network. Latency can be described as the delays present in a network, and could result from a wide variety of causes, including computation time in network nodes, links and proxy servers. (Sterbenz 2001, p. 2-4.)

In an ideal situation the properties all contain keyframes well ahead of the desired query time, thus the queries on the properties can be said to be valid. However, if the latest keyframe is positioned before the query time, the sampled value will be that of the previous keyframe as explained earlier. Under normal behaviour the property curve will be updated with keyframes so that the current query time does not 'jump' to a different value due to interpolation between the previous keyframe and the newly added keyframe. This may, however, not be the case if the lack of future keyframes is due to excessive latency in the network, causing the keyframes to arrive later than the query time of the previous keyframe.

Notice that when the brute force approach is used, network latency always adversely affects the client, since the server will send out the data for the current time, at the current time, resulting in the client updating as late as the latency. In the case of the KP, the server is able to send data ahead of time, meaning the latency does not affect the experience of the client at all, assuming the server is able to send data which is ahead of its time by more than the latency.

2.7. Theoretical Benefits

Given the outlined concept of how the KP functions, it is possible to make predictions on possible benefits the KP could provide, and how the KP compares to other implementations in aspects other than the size of data transferred over the network.

Due to the fact that the KP encodes the values at a specific time and the fact that the client maintains a full list of all keyframes, including past keyframes, it is possible for the client to sample any point in time and is thus not limited to only retrieving the state at the current time. At first this may not seem useful, but its is important to notice that if the client is able to access the state at any point in time, it is even possible for the client to rewind and play the state backwards. The game Planetary Annihilation, which

uses a keyframe based approach similar to the KP being formalised in this thesis, makes use of this in a feature called Chronocam, which allows the player to rewind and view the state of the world at any point in time (Smith, 2013). This adds an incredible level of immersion for the player and especially spectators viewing the game, allowing them to analyse the strategies of the other players.

Furthermore, the history storing ability garnered from maintaining the keyframes in memory gives the client the ability to perform actions and calculations based on client-side analysis of the historical data. This can range from a simple computation such as computing the duration that a property has existed in the game world, to performing a mathematical analysis on the behaviour of the property in the past, and doing a client-side prediction for the property in the event that a keyframe is not present.

2.8. Example

As a full example for a specific entity, the movement of the entity can be used. The server may decide to move an entity from position (4, 7) on the two dimensional cartesian plane to (6, 12) over a period of 20 time units. Let the assumption be made that the acceleration is instant, and the movement linear. It can be concluded that given the starting and ending point, and the times related to them, any intermediate value can be deduced by interpolation as explained earlier.

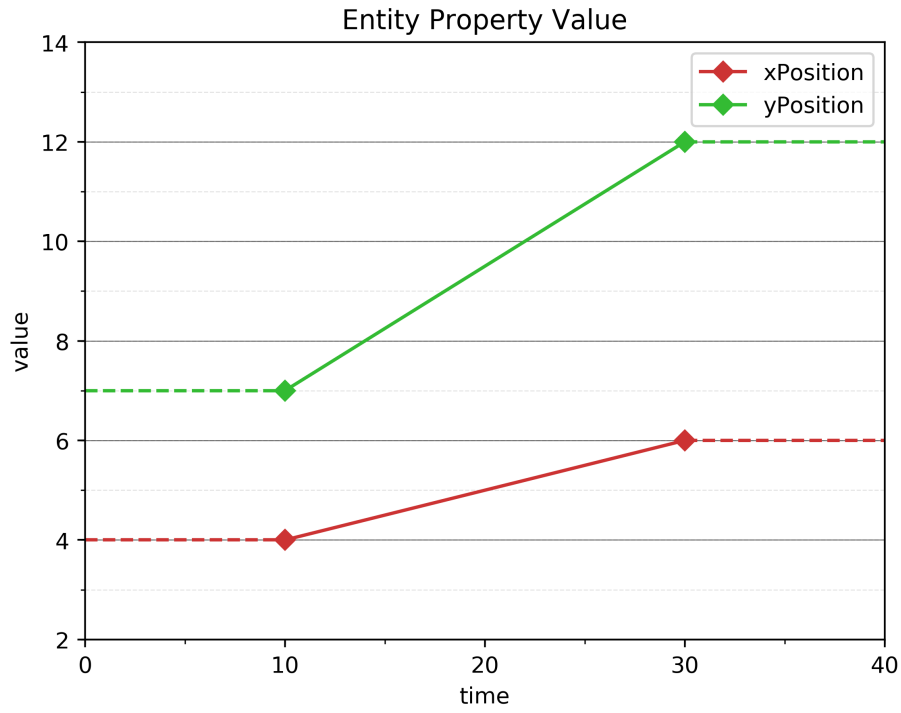

Figure 8. Entity property values

Figure 8 shows how keyframes can be used to describe the change in property value. Note that the position prediction can be made ahead of time, since the server knows the speed of the entity, and the position the entity will move from and the path the entity will take to reach its destination. The client is able to interpolate between the given keyframes, or assume the closest value if the evaluated time is outside the given keyframes. The client is also able to update the its values as rapidly as needed or as quickly as it is able to render, which is often the case in an RTS game client.

In summary, with the use of the brute force implementation, each update cycle within the time period between the keyframes would have required the server to send a value to the client, whereas sending the values with their respective times attached in the form of a keyframe allows the values in between to be deduced through interpolation, removing the need to send the data over the network.

A key aspect to notice with regards to the KP, is that the KP 'explains' the state over time to the client, thus the client may know the future state of some properties in the system but not all. The KP allows the server to build up the state in the client to the extent that both the server and client have a mutual understanding of the state over time. This is in stark contrast to the brute force approach where the server possesses the

understanding of the state relative to time, and just sends the instantaneous values to the client, discarding the previous values. The values often need to be discarded due to the simple fact that the volume of data is simply too much, since all the data from all the properties on each update need to be stored in order to maintain history, unlike only storing the keyframes with the KP.

3. Keyframe Protocol Specification

In order to be a formal protocol, a specification needs to be provided from which implementations can be implemented, thus formalising the specification of the KP is important as both server and client needs to be in agreement on the functionality of the protocol and how to interpret and create packets of data that the other will understand (Blank 2004, p. 14). Once the specification is established, independent implementations can be developed using any method desired, including any desired computer language, as long as the interface matches the specification. This decouples the implementation, and allows the implementation to evolve over time and be implemented specifically and in an optimised manner for any target system, while maintaining the ability for systems to communicate with one another.

The KP is an Open Systems Interconnection (OSI) layer 6 protocol built on top of the Transmission Control Protocol (TCP), also identifiable by its Request for Comment (RFC) reference of 793. The KP is considered an OSI layer 6 protocol because it deals with the syntax and structure of the data being sent from host to host. It would not be considered an OSI layer 7 protocol, because the protocol does not specify the application usage itself, which for example the File Transfer Protocol (FTP) does. (Blank 2004, p. 21.)

It is important to clearly note the mandate of the KP and not to overextend functionality to exceed the boundaries of the KP realm. The KP is only responsible for the exchange and lookup of keyframes which are related to properties. The KP is not responsible for prediction, compression, smoothing or optimisation of the placement of the keyframes themselves, as these tasks are system dependent and could be achieved by a future protocol which is a superset of the KP.

3.1. Lifecycle

The client initiates a TCP connection with the server, following which the client sends a ‘Sync’ type packet to the server, with the ‘Method’ field set to ‘Meta’ and the ‘BaseTime’ field set to zero.

The server will receive the request and send a similarly typed ‘Sync’ packet to the client with the ‘BaseTime’ field set to the desired server base time. This will allow the client and server to agree upon a base time. Upon receiving keyframe data, the client will sum the keyframe time increment with the stored base time it received from the server

during the initial handshake, thus allowing the client and server to agree on a specific absolute unix time, while sending a smaller byte count number with each keyframe.

Following the initial handshake, the server is able to start sending keyframes to the client. The server will continue to include the client in the pool of active clients until the client or server closes the TCP connection.

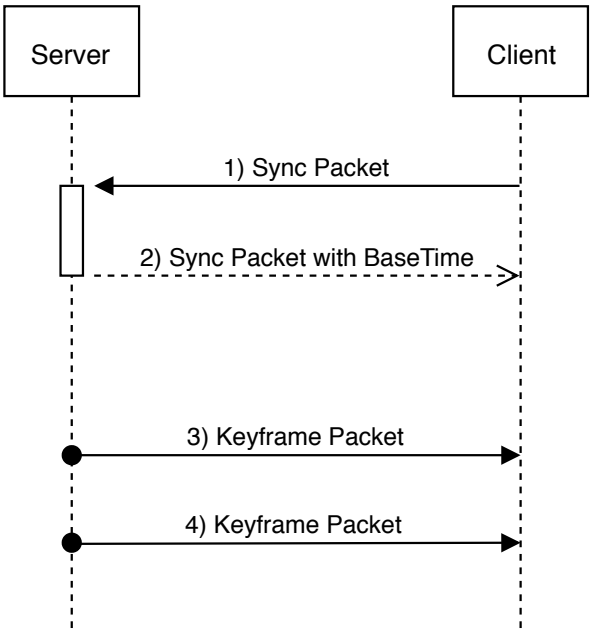

Figure 9. KP Handshake and Lifecycle

In Figure 9, it can be seen that the handshake is unlike the TCP handshake, because no acknowledgement is needed after the server sends the base time to the client (Information Sciences Institute, University of Southern California 1981, p. 31). This is intentional as there is no need for the KP to reimplement the checked delivery ability of TCP, which is already implemented in the underlying TCP connection.

3.1.1. Protocol State

The KP is a stateless protocol, meaning there is no agreed upon state between the client and the server. Either side of the connection can ask for missing information from the other if needed, but the opposing side of the connection does not keep track of the information the other side possesses.

In addition to not knowing the state of the other side of the connection, the KP does not track the sequence of events either. This is due to the fact that just as TCP already im-

plements a connection handshake, so does it already sufficiently ensure the sequence is maintained by the use of the SEQ field (Information Sciences Institute, University of Southern California 1981, p. 30-32). Due to the nature of the KP protocol, the sequence of sending keyframes to the client does not change the functionality as long as the keyframes are added to the client state before the time of the keyframe previous to the keyframe being sent. This is to ensure that the client can interpolate correctly between the keyframe being sent and the keyframe before the keyframe being sent.

3.2. Data Fields

The protocol calls for the data to be sent in a very specific layout. The data is packed into a buffer, which is 14 bytes in size, carefully split into fields, which are read on the opposing end of the connection. Note that not all fields are present in all packets, and the exact combination of fields and the layout of the buffer is explained in more detail later in this thesis. All possible fields are as follows:

- Type
- Method
- Base Time
- Time Increment
- Property
- Value
- Time Offset
- Time Offset Negation

Note that fields other than the value were not applicable in the brute force implementation, since all data is sent on every update and the relative position of the data could be used to identify the property whose value needs to be set simply by indexing into the buffer by multiplying the index by the the four bytes for the float payload.

3.2.1. Type

In order to determine the initial interpretation of the message, the first field is the type of the message. The type can either be ‘Sync’ or ‘Open’, thus requires only one bit in the buffer. However, it does make sense to allocate two bits to the type, as it allows

easy expansion of the protocol in the future, and negligibly affects the overall size requirement.

As seen in the lifecycle, the ‘Sync’ type is used when the client requests the base time from the server, or when the server sends the base time to the client. For all cases where the server is sending keyframe manipulation data to the client, the type is ‘Open’.

3.2.2. Method

In order to manipulate the keyframes, a finite set of operations are defined, which are implemented on both the server and client.

Keyframes can be manipulated using the following methods:

- Add
- Remove
- Move
- Meta

Note the meta method which is used for syncing, is also used to indicate actions other than direct keyframe manipulation. This is important to ensure the possibility of adding additional complexity to the protocol in the future, without the need to change the structure, thus ensuring backwards compatibility.

Since there are a total of four possible methods, two bits are sufficient to encode which method is desired. Just as with the ‘Type’, extra bits will be allocated to ensure future proofing of the protocol, bringing the ‘Method’ field up to a total of four bits, thus allowing an expansion to a total of 16 methods.

3.2.3. Base Time and Time Increment

A logical assumption can be made that all clients can agree on a global start time, then use the value encoded in the ‘TimeIncrement’ field as an offset from the global time, thus allowing the values not to be tied to global time and reduce the size of the needed time data type.

2. $$time_{value} = time_{basetime} + time_{increment}$$

Both the ‘BaseTime’ and the ‘TimeIncrement’ is stored in milliseconds, where the 'BaseTime' is the number of milliseconds since January 1, 1970, which is also known as Unix Time. In order to determine a suitable data size for the ‘BaseTime’ and ‘TimeIncrement’ it first needs to be determined what are the largest reasonable numbers which will be required to be stored in the fields.

It should be noted, that if the ‘TimeIncrement’ field reaches its maximum value, the protocol will fail. A solution to this would be to simple set the ‘BaseTime’ of the server to a more recent time, and inform all the clients of the change. However, doing so will break the absolute time value for the keyframes before the more recent ‘BaseTime’, since they will now be added to a more recent time. This can be avoided in multiple ways, including ‘baking’ in the times of all the keyframes which have already passed into absolute times, or simply removing them and loosing the ability to rewind, which may be a viable option depending on the system.

With the considerations in mind, and taking into account the original problem at hand, it would be reasonable to estimate a maximum duration of 12 hours.

In order to determine the field bit size, the calculation can be done in reverse as follows:

3. $$\begin{aligned} 12_{hours} &= 720_{minutes} \\ 720_{minutes} &= 43{,}200_{seconds} \\ 43{,}200_{seconds} &= 43{,}200{,}000_{milliseconds} \end{aligned}$$

At which point the number of bits can be calculated:

4. $$\begin{aligned} bits &= \log_2(43{,}200{,}000) \\ &= 25.36452797660028_{bits} \\ &\approx 26_{bits} \end{aligned}$$

Thus it can be seen that 26 bits of data is sufficient for the ‘TimeIncrement’ field.

For the purposes of the ‘BaseTime’, the size is less important since the value will only be sent when the server sends a ‘Sync’ packet. It is, therefore, not important to optimise the size of the field and it can be set at 12 bytes to ensure sufficient capacity.

3.2.4. Property

Since the assumption is made that the system contains 1,000 entities with three properties each, any unique property can be identified by a number with a minimum of 3,000 unique values. Twelve bits of data would allow for 4,096 unique values, which is sufficiently above the 3,000 unique values requirement. However, as previously mentioned, more space will be allocated to ensure future expansion is possible. The ‘Property’ field will, therefore, be allocated 24 bits, resulting in a total of 16,777,216 uniquely identifiable properties.

5. $$properties = 2^{24_{bits}} = 16{,}777{,}216$$

3.2.5. Value

Given the initial assumption that the only values to be sent to the client will be floats, the value can be a fixed four bytes.

3.2.6. Time Offset and Time Offset Negation

The 'TimeOffset' is only applicable when the method is 'Move', since it is used to denote the time shift of the target keyframe. The 'N' bit is used to indicate a positive or negative offset and is needed because the buffer is always read as an unsigned integer. When the 'N' bit is set the 'TimeOffset' is simply multiplied by negative one.

3.3. Keyframe Protocol Frame

As all the individual required data elements for KP have been identified, the data can be packed into a data frame, which will be called a Keyframe Protocol Frame (KPF). The frame consists of a 14 byte buffer, which is of fixed length, resulting in multiple unused bytes of buffer space. However, the unused space does allow for future expansion of the protocol. It is also possible to develop the protocol to use variable size buffers with are dependent on the type identified in bits 0 and 1.

Tables 1, 2, 3 and 4 show the different packing schemes of the packet types and associated method variations. It is important to note the drastic data size saving due to the ‘BaseTime’ being sent in the ‘Sync’ packet instead of along with each 'Open' packet. This can be seen by comparing the size of the 'BaseTime' in Table 1 with the size of the 'TimeIncrement' present in tables 2, 3 and 4.

Table 1. Data Packing for Keyframe Protocol when type is ‘Sync’

Byte	0								1							
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	Type		Method				xx		BaseTime							
2	BaseTime															
4	BaseTime															
6	BaseTime															
8	BaseTime															
10	BaseTime															
12	BaseTime								xx							

Table 2. Data Packing for Keyframe Protocol when type is ‘Open’ and method is 'Add'

Byte	0								1							
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	Type		Method				TimeIncrement									
2	TimeIncrement															
4	Property															
6	Property								Value							
8	Value															
10	Value								xx							
12	xx															

Table 3. Data Packing for Keyframe Protocol when type is ‘Open’ and method is 'Remove'

Byte	0								1							
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	Type			Method					TimeIncrement							
2	TimeIncrement															
4	Property															
6	Property								xx							
8	xx															
10	xx															
12	xx															

Table 4. Data Packing for Keyframe Protocol when type is ‘Open’ and method is 'Move'

Byte	0								1							
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	Type			Method					TimeIncrement							
2	TimeIncrement															
4	Property															
6	Property								TimeOffset							
8	TimeOffset															
10	TO			xx					N		xx					
12	xx															

Table 3 illustrates that the 'Remove' method does not need any additional information besides the keyframe to remove, which is identified by the property and the time at which the target keyframe exists.

Table 4 illustrates that the 'Move' method targets the keyframe in the same manner as the 'Remove' method, followed by a time offset which indicates how much the keyframes should be moved on the timeline. Note the 'N' bit used for negation of the time offset as explained earlier.

3.4. Lookup

Properties need to be able to be queried at a specific time. The returned value needs to satisfy the following rules:

- A time lookup before the first keyframe needs to return the value of the first keyframe.
- A time lookup after the last keyframe needs to return the value of the last keyframe.
- A time lookup between two keyframes needs to return the value resulting from a linear interpolation between the two keyframes.
- A time lookup on a property with no keyframes should return the equivalent of null

4. Network Data Calculations

Calculating the data rate through a network is a complex process, since there are many aspects of the network which alter the data as it moves through the network, thus changing the total size of data moving through the network depending on which point in the network is being measured.

In is not realistically possible to calculate the exact amount of data which will be transferred over the wire, since the size of the data changes as the data is transferred, caused by the wrapping and unwrapping, and fragmentation and reassembly of the packets by the specific hardware in the network. Nevertheless, it is possible to calculate an approximate theoretical data size, which assumes reasonable averages, given current standards, such as known limitations of IPv4 and Ethernet II.

In addition, the commonly used terms need to be clarified. Bandwidth refers to the speed at which a specific node in the network can forward data, and does not depend on the protocol type. The network throughput refers to the useful data which is sent from one host to another. (Guojun 2002, p. 4-5.)

Building on this, the Maximum Transmission Unit (MTU) of the specific network needs to be considered. The MTU is the maximum packet size supported on the IP layer, and thus will force packets to be below this maximum size by fragmenting larger packets into multiple smaller packets. This becomes a problem due to the additional overhead which accompanies the added packets, and is explained clearly in the exact data rate calculation section of this thesis. It is possible to increase the MTU from the default 1,500 bytes to 9,000 bytes, since IPv4 supports it, however, not all networking devices support the larger MTU and thus can not be considered standard. (Langer 2010.)

4.1. Measurement

Optimisations to any system can only be made when it is possible to measure the performance of the system before and after the change, thus indicating an improvement or a regression in system performance. It has been established that measuring directly on the wire will be an inaccurate measurement since the data will vary from network to network, thus a better approach would be calculating a theoretical data size, and attempting to minimise the theoretical size using the proposed solution.

It should be kept in mind, that when calculating a theoretical data size, any additional data that is added outside of the theoretical context would not distort the comparative results, assuming the external data is linearly relative to the theoretical data, since less input data would still produce less output data. However, this assumption can only be made if the additional overhead data is indeed linearly relative to the theoretical data, which is not the case with MTU, and as such needs to be taken into account in the theoretical calculation.

This can be demonstrated with an example where the MTU is set to 1,500 bytes. In the case where the packets over the network fit into a single MTU, such as an example packet of size 1,490 bytes, would only carry the overhead of a single frame, while a n example packet of size 1,510 bytes would need to be split into two frames, resulting in double the overhead, but not double the payload, hence a non-linear relation. Therefore, when calculating the theoretical data size transferred over the network, the MTU needs to be considered.

4.2. Network Failures

Networks are unreliable, and are prone to failures beyond the control of the user or the application developer. It must always be assumed that a network can fail at any moment, and thus it is important to design the system with this in mind. Mitigations can be made on a lower level, such as relying on ensured delivery and sequencing that is provided by TCP (Marin 2005).

Failures can range from hardware failures, software failures and even deliberate attacks, including Distributed Denial of Service (DDoS) attacks which aim to deny service from the system. (Marin 2005.)

In some cases, failures might be acceptable, having no or an acceptable level of negative impact on the system. However, identification and consideration are important none the less, and therefore, when choosing underlying protocols, the handling of network errors will be considered.

4.3. Assumptions

In order to ensure that the minimum viable system is achieved using various methods, a variety of assumptions need to be made regarding the requirement and performance

expectations, such that all proposed systems satisfy the requirements. This is to ensure that the calculations of various solutions are fair, and that all proposed solutions are indeed capable of reaching a minimum requirement, thus ensuring that the comparison is balanced.

Only a single client can be assumed, since the amount of data that needs to be uploaded from the server will scale linearly as the number of clients increase. Computation for each client will be done on the server before the data is sent, therefore, the time due to calculation of keyframes or updates will be ignored, since it falls outside the scope of the protocol.

Let the assumption be made that the client only needs to have a data accuracy within a tenth of a second, and the system will never run for longer than a day, which is common for multiplayer video games. Furthermore, the server will need to sync one thousand entities to the client. For calculation purposes, it can be assumed that each entity contains three properties of type 'float', which are four bytes each.

It will also be assumed that the server makes 100 percent accurate predictions with regards to future keyframes in the keyframe based system, thus no alteration is needed to existing keyframes due to incorrect prediction. Keep in mind, this does not include alterations made due to shape constraints, such as when adding a keyframe at a future time provides an undesired interpolation line, requiring additional keyframes to fix, which is normal behaviour.

4.4. Brute Force

In addition to requiring the ability to measure the system performance, an optimisation can only be made if there is a baseline to measure against, thus allowing an improvement to be seen. The baseline can be established by calculating the data rate when the brute force solution to the problem is used.

Since the assumption for the system is such that updates are required with no more than a tenth of a second of inaccuracy, the brute force solution needs to incorporate the ability to update the entities on the client at a minimum of ten times a second, or a single update every 100 milliseconds.

Each entity will be updated in full on each update cycle, meaning all properties will be updated on every update cycle, regardless of wether or not the property value has

changed since the previous update cycle. As such, the payload data size of a single entity can be given as follows.

6.
$$bytes_{entity} = 3_{floats} \times 4_{bytes_{float}} = 12_{bytes}$$

The total raw entity data size in bytes per update using the brute force solution can then be calculated with:

7.
$$bytes_{entities} = bytes_{entity} \times count_{entities}$$

Substitution then yields:

8.
$$12_{bytes_{entity}} \times 1{,}000 = 12{,}000_{bytes} = 12_{KB}$$

4.4.1. Data Transfer Overhead

Transferring data end-to-end over a network can be done by following the conventions specified in the TCP/IP internet protocol suite (Simoneau 2006). Forming the basis of these conventions is the Advanced Research Projects Agency (ARPA) core protocols set, and amongst them are the most widely used OSI layer 4 protocols, which are specifically of interest to the target use case. Namely, these are User Datagram Protocol (UDP) (RFC 768) and TCP (RFC 793) (Miller 2004, p. 8-11). These are also the only two, OSI layer 4 protocols in the TCP/IP suite (Blank 2004, p. 46-47).

4.4.1.1. Protocol Comparison

In order to ensure that an appropriate underlying OSI layer 4 protocol is chosen, it is important to compare the advantages and disadvantages of each. The UDP protocol (RFC 768), has an average packet overhead of only eight bytes, containing the source port, destination port, length and checksum, making UDP a very lightweight OSI layer 4 protocol, with minimal overhead. (Blank 2004, p. 52.)

With the small overhead comes the disadvantage of not having a delivery confirmation or 'ACK', returned from the destination. UDP packets are sent and are not checked for reaching their destination, in a 'fire and forget' fashion. Considerations need to be taken due to the lack of delivery acknowledgement. When using UDP it must always be

assumed that the packets could potentially not be delivered, and if a missed packet would result in an error in the application, a layer of error checking needs to be built on top of the protocol, or a different protocol such as TCP needs to be considered. (Blank 2004, p. 45.)

TCP has a higher overhead than UDP, ranging from 20 to 60 bytes, with the average TCP packet having a header of 32 bytes (Blank 2004, p. 52). In the additional bytes, the packet header also contains information on the sequence and acknowledgement number of the packet. With the added header information contained in a TCP packet, the packets can be guaranteed to be delivered in order, with the knowledge of whether or not the delivery was successful given by the 'ACK' which is returned from the destination (Marin 2005). Consideration should also be given to the size of the data packets containing the 'ACK', and a decision needs to be made whether or not the 'ACK' packets should be included in the data transfer calculations.

4.4.1.2. Overhead Comparison

In the specific case of using a brute force approach, the data will be overwritten on each update, so it can, therefore, be assumed that in most cases the missed packet would not effect the client in a significant enough manner to warrant the added overhead of ensuring the delivery with an error checked protocol, such as TCP, since the correct value will simply be sampled when the next packet arrives on the next successful update. It is important to note at this point that this exact behaviour demonstrates the 'statelessness' of the brute force approach, by which the client can gain a 100 percent accurate understanding of the entities' states in a single update and does not rely on historical data to build the state.

Given that UDP is an OSI layer 4 protocol, the packets will be wrapped with an Internet Protocol (IP) packet header, specifically IPv4, being OSI layer 3 as well as an ethernet packet header, specifically Ethernet II, being OSI layer 2, of sizes 20 bytes and 14 bytes respectively. (Simoneau 2006.)

As such, in the case of UDP, the total packet size can then be calculated as follows:

9.
$$\begin{aligned} bytes_{packet} &= 14_{bytes_{ethernet}} + 20_{bytes_{ip}} + 8_{bytes_{udp}} + bytes_{payload} \\ &= 42_{bytes_{overhead}} + bytes_{payload} \end{aligned}$$

It can, therefore, be concluded that each packet sent, will contain at minimum an overhead of 42 bytes. It can be logically deduced that minimising the number of packets will minimise the overhead. It can then also be logically determined that it would be ideal to send all the data in a single packet, resulting in only one multiple of the packet header overhead, however, packets do have a maximum total size. The IP packet contains a header of 20 bytes as determined earlier, of which 2 bytes specify the length of the IP header and its data in bytes, which includes the UDP header. Given 2 bytes at 8 bits per byte, the 16 bit unsigned integer can give a maximum value of 65,535. Thus, the maximum payload size can be calculated as:

10.
$$\begin{aligned} bytes_{payload} &= 65{,}535_{bytes_{total}} - 20_{bytes_{ip}} - 8_{bytes_{udp}} \\ &= 65{,}507_{bytes} \end{aligned}$$

Most of the internet is based on the ethernet protocol, thus the assumption will be made that the OSI layer 2 protocol is indeed ethernet, specifically Ethernet II. Since ethernet is OSI layer 2, and the IP packet above is OSI layer 3, the IP packets will be wrapped with an ethernet header, in the same way the UDP packet is wrapped with an IP header. This means that if the IP packet has a payload size which is less than the packet to be wrapped, the packet will need to be split. Given that the maximum size of an ethernet packet is 1,500 bytes due to the MTU excluding the ethernet header, it can be concluded that the maximum payload of the original UDP packet is: (Langer 2010.)

11.
$$\begin{aligned} bytes_{payload} &= 1{,}500_{bytes_{ethernet}} - 20_{bytes_{ip}} - 8_{bytes_{udp}} \\ &= 1{,}472_{bytes} \end{aligned}$$

4.4.2. Minimum Required Data Transmission Rate

As previously calculated, 12,000 bytes of value data needs to be sent to the client to update the state of all entities. Since the specification requires no more than 100ms delay between updates, it can be concluded that 10 updates need to be sent to the client each second.

12.
$$\begin{aligned} bytes_{total} &= 12{,}000_{bytes_{update}} \times 10_{updates} \\ &= 120{,}000_{bytes} \end{aligned}$$

Given the conclusion that during normal network operations, packets will effectively be limited to a payload of only 1,472 bytes, a calculation can be made to determine the number of packets needed, thus allowing the size of the overhead to be calculated.

13.
$$\begin{aligned} packets &= \frac{120{,}000_{bytes}}{1{,}472_{bytes_{packet}}} \\ &= 81.521_{packets} \\ &\approx 82_{packets} \end{aligned}$$

Rounding up the number of calculated packets is needed, since half a packet is not possible, and even though the last packet may not be full, the overhead size will remain the same, since the packet headers are still present, regardless of the payload size, and as such, the overhead can be calculated as:

14.
$$\begin{aligned} bytes_{overhead} &= 82_{packets} \times 42_{bytes_{packet}} \\ &= 3{,}444_{bytes} \end{aligned}$$

Since the total payload size and the total overhead is known, it can be concluded by addition, that a total of 123,444 bytes would need to be transmitted each second to the client in order to meet the application requirement. Thus the data rate can be calculated as follows:

15.
$$\begin{aligned} bits_{second} &= 123{,}444_{bytes_{second}} \times 8_{bits_{byte}} \\ &= 987{,}552_{bits_{second}} \\ &\approx 988_{kbit_s} \end{aligned}$$

It has thus been deduced that the use of the brute force approach would yield a data rate of approximately 988kbit/s.

4.5. Keyframe Protocol

Calculation of the data rate when using the keyframe model is significant more complicated, since the data required by the client varies depending on the change of the property values. Since data can no longer just be sent as a single array, more complicated logic is needed to organise the data and tell the client how to use the sent data, which is the job of the keyframe protocol itself and has already been explored earlier.

Even though the data might not be simply calculable, it is possible to derive a formula for determining relative performance compared to the brute force approach. This can be done by calculating the breakeven point at which the KP becomes more efficient.

4.5.1. Keyframe Breakeven

It needs to be understood that the date rate will be highly dependable on the nature of the property itself, and the predictability and linearity of the value over time, thus the following calculations hold true only for comparing a single property at a time. However, the breakeven point can be deduced by comparing the total bytes per second required by the specific property, from both the KP and the brute force implementations.

The breakeven point will be the number of keyframes per second which will match the data rate of the brute force approach, thus if the property requires less keyframes per second in practice than the theoretical breakeven point, the property will be more efficiently synchronised with the KP.

16. $$KP_{frames_{second_{breakeven}}} = \frac{bytes_{second_{bruteforce}}}{bytes_{second_{KP}}}$$

In order to calculate the breakeven keyframe rate, the bytes per second first needs to be calculated for the KP and brute force implementations for the specific property in question.

In order to calculate the date rate for the KP, the total size of the KP packet is first required and can be calculated as follows:

17. $$\begin{aligned} KP_{bytes_{packet}} &= 14_{bytes_{ethernet}} + 20_{bytes_{ip}} + 32_{bytes_{tcp}} + 14_{bytes_{KPpayload}} \\ &= 80_{bytes} \end{aligned}$$

It is important to note that the overhead from the wrapping headers can be linearly added in the case of KP, since the KP payload will always be sent in its own ethernet frame. This is a point mentioned in the optimisations section of this thesis. Another important aspect to note, is that the TCP 'ACK' response from the client is ignored in the calculation, since the data is moving in the opposite direction through the network and should be dealt with separately.

At this point it needs to be determined how many updates are needed per second for the brute force implementation of the specific property to reach the 'equivalent' level of accepted precision as with the KP implementation.

Thus, determining the density for a specific property would require careful analysis and testing of the specific use of the property, including the subjective analysis on what

would be considered 'equivalent precision', since the KP system would by nature be smoother due to interpolation.

This phenomenon can be explained using a simple example. In the case of a property smoothly changing from value ***a*** to ***b*** over a time ***t***. Using the KP, the item would be moving with 100 percent precision between ***a*** and ***b*** due to the interpolation. However, in order to achieve the 'equivalent precision' with a brute force approach, there would need to be an infinite number of updates between t_a and t_b, thus the choice needs to be made how many updates would suffice to achieve a near similar precision in the specific system. By nature, this is subjective, and introduces a subjective variable to the calculation of the breakeven value.

It was earlier assumed in this example that 10 updates a second would be an acceptable level of accuracy for all properties in the system; thus the conclusion can be drawn that for any given property in the system, 10 updates per second will be sufficient. Due to this, it will be assumed that 10 brute force updates per second will give 'equivalent' precision.

The calculation now needs to be made to determine the size of a brute force packet. Previously it was concluded that in the brute force implementation, multiple values are packed into a single packet, thus sharing the overhead of only a single packet. This needs to be factored into the calculation when determining the average data size of only a single value.

As previously stated, the example calls for a property to be 4 bytes, and as previously seen, the UDP packet's effective payload is limited to 1,472 bytes, due to its headers and the MTU. (Langer 2010.)

If the assumption is made that in a best case scenario for the brute force implementation, the UDP payload is filled, and the overhead is the previously calculated value of 42 bytes, the overhead percentage can be calculated as follows:

18.
$$overhead_{bruteforce} = (\frac{42}{1{,}472}) = 2.85\,\%$$

As an average, this percentage of the overhead will be added to each value in the brute force calculation to compensate for the collective overhead from packet headers.

It can now be calculated how many bytes per second the brute force implementation would average for the specific property.

19.
$$\begin{aligned} bruteforce_{bytes_{second}} &= (4_{bytes_{value}} * 10_{updates_{second}}) * 1.0285 \\ &= 41.14_{bytes_{second}} \end{aligned}$$

The original breakeven formula can now be completed and rearranged with the substituted values as follows:

20.
$$\begin{aligned} KP_{frames_{second_{breakeven}}} &= \frac{41.14_{bytes_{second_{bruteforce}}}}{80_{bytes_{second_{KP}}}} \\ &= 0.51425_{KP_{frames_{second}}} \end{aligned}$$

It can thus be determined that in order for the KP to be more efficient for the specific property, the KP keyframe rate should be no more than 0.51425 keyframes per second, compared to the 10 updates per second when using the brute force approach. Clearly it can be seen that the brute force approach would solve rapidly changing properties better, since those properties will most likely require much more than 0.51425 keyframes per second, however, the slower, more predictable properties may benefit from the use of the KP.

It may seem that the reduction from 10 updates per second in a brute force approach to a mere keyframe every two seconds in the KP is a significant drop in data resolution, and this may be the case, however, it should be remembered, that as previously noted, properties often do not need to be updated as frequently with the KP, due to the major advantage of interpolation between the keyframes, which is particularly effective when dealing with rarely changing properties or properties with constant linear change.

5. Implementation

The KP implementation does not need to be the same for all systems. The only requirement for a system is to meet the KP specification, thus it is possible for implementations to be written in any language, with varying efficiencies. As a proof of concept, a Typescript implementation satisfying the KP specification was developed. The structure of the code was evaluated, and the shortcomings of the specific implementation was explored, including the possible optimisations that could be made.

5.1. Structure

At the root of the implementation is the KPStore, which maintains a list of all the properties, each having a list of keyframes. The KPStore is responsible for ‘applying’ a packet, meaning any manipulation to the store can only be done through packets, acting in a similar manner as 'actions' in Redux (Redux Documentation 2018). This separation of state management to logic is purposefully done to minimise the coupling between the state of the properties and the keyframe logic.

Furthermore, this allows the KPServer and KPClient to be a simple extension of the KPStore itself, where the client simply passes the received packets to the store’s ‘apply’ method. The server system uses the interface exposed by the KPServer to build a specific 'manipulation' object, from which a packet is created and sent to all the clients as well as the server’s own ‘apply’ method, which is available since the KPServer is an extension of the KPStore.

In practice, clients simply create instances of KPClient, and servers create instances of KPServer.

5.2. Usage

Listing 1 shows the example instantiation of a KPServer in Typescript. First the KPServer object is created, followed by a delay emulating the passing of time in the system. At some point in the future the system decides to add a keyframe to property 7 at a time 30 seconds ahead of the current time, with the keyframe having a value of 5.

```
const kpServer= new KPServer()

// -- Game starts
console.log('Game: Started')

// -- Emulate time passing in game
delay(5 * 1000)

// -- Apply a needed manipulation
kpServer.add(7, Date.now() + 30 * 1000, 5)
```

Listing 1. Instance of KPServer

```
const kpClient = new KPClient()

// -- Game starts
console.log('Game: Started')

// -- Emulate time passing in game
delay(10 * 1000)

// Read all properties
JSON.stringify(kpClient.getProperties())

/*

[null, null, null, null, null, null, null,
   {
         "id": 7,
         "keyframes": [
               {
                     "timeIncrement": 35012,
                     "value": 5
               }
         ]
   }
]

*/
```

Listing 2. Instance of KPClient

The KPServer then creates a packet from the construction and sends it to all the connected clients, as well as to its own underlying KPStore's apply function.

It can be seen from Listing 2 that the KPClient is created, at which point it does a request to the server for the base time. Since the server and client were started at the same time in this example, delaying the client 10 seconds is sufficient to ensure the packet which is sent from the server after 5 seconds has indeed been sent. Thus, reading the properties in the client will produce the result visible in listing 2.

Note that the property is in zero indexed position 7, hence the 7 ‘null’ values before the target property. The ‘TimeIncrement’ is set at 35,012, which is just more than 35 seconds. The initial 5 second delay before creating the manipulation and the 30 second advance time which the server specified contribute to the 35 total seconds. The slight fluctuation of 12 milliseconds is a result of internal inaccuracies in the Javascript event loop with regards to the delay. In a non-simulated environment, this will not occur. Note that even with the 12 millisecond fluctuation, both the server and client have an identical fluctuation, and this is due to the fact that the issue is not in the protocol, but rather in Javascript while creating the initial packet. Thus the server and client are still both 100 percent synchronised after the packet is applied on both the server and the target client.

5.3. Shortcomings and Optimisations

Although the implementation is fully functional and complies with the protocol specification, there are significant optimisations that can be made.

5.3.1. Step Interpolation

The only form of interpolation in the implementation is linear, and no extra methods were included. This is very limiting as the ability to encode discreet values will be hindered by the interpolation and will need to be done by the use of keyframes which create a vertical line, representing the step in value. This is inefficient, since a single keyframe would be all that is needed if the client is able to simply read the nearest previous keyframe value at a given time, instead of interpolating.

5.3.2. Frame Packing

The implementation presented sends a single change packet as soon as a change is created by the server, this means that each update contains the network frame overhead as explained earlier. An optimisation can be made to where the server still creates change packets at any time, but only sends them at an interval, allowing the server to pack multiple change packets into a single network frame, getting the most out of the 1,460 byte payload limit set by the network MTU.

It is also possible to send a packet outside of the interval if the keyframe time is before the next interval, thus getting the immediate effect of the keyframe when needed, and the reduction in overhead when possible.

5.3.3. Keyframe Lookup

A key component to the KP is the the lookup of a value at a given time on a specific property. As it is the most executed function, presumably many times per second per property, it is essential that the function be as optimised as possible.

```
at(timeIncrement: number): number {
    const lastIndex = this.keyframes.length - 1
    if (timeIncrement <= this.keyframes[0].timeIncrement) {
      return this.keyframes[0].value
    }
    if (timeIncrement >= this.keyframes[lastIndex].timeIncrement) {
      return this.keyframes[lastIndex].value
    }
    for (let i = 0; i < lastIndex; i++) {
      if (
        this.keyframes[i].timeIncrement < timeIncrement &&
        this.keyframes[i + 1].timeIncrement > timeIncrement
      ) {
        const x0 = this.keyframes[i].timeIncrement
        const x1 = this.keyframes[i + 1].timeIncrement
        const y0 = this.keyframes[i].value
        const y1 = this.keyframes[i + 1].value
        return interpolate(x0, y0, x1, y1, timeIncrement)
      }
    }
  }
```

Listing 3. Lookup function calling interpolation

By looking at the implementation of the 'at' function in listing 3, it can be seen that before the first keyframe the value of the first keyframe is returned, and after the last keyframe the value of the last keyframe is returned, as specified in the protocol specification. After the edge cases are handled, the actual interpolation is calculated.

It is important to notice that in order for this function to work, the keyframes need to be sorted by 'timeIncrement' since the keyframe objects are stored in a sequential array. If a new keyframe is added to the array with a 'timeIncrement' value less than the 'timeIncrement' value of the last keyframe, the array needs to be sorted, which at worst can require all the elements to be shifted, which is an expensive operation with a time complexity of O(n) in Big O notation (Das 2006, p. 8-9). Due to the implementation of the array structure in the underlying Javascript, the array could be either sparse or

dense. In the case where the array is sparse, the array functions in the same manner as a map with keys, which will make the insertion O(1). (Kiran 2013.)

Another point is that on each lookup, the entire 'keyframes' array is iterated over from the start, since there is no way to know which index to start at given the lookup time. This means that as the number of keyframes increase, and as the system progresses, the lookup function becomes linearly slower to execute, which is very undesirable.

A potential solution to this problem is to use a binary tree data structure instead of a sequential array to store the keyframes. The binary tree can be indexed by the time itself, which means a lookup has a time complexity of O(log n). Since the keyframes will be stored as nodes on the tree, the 'insert' and 'remove' operations will also only have a time complexity of O(log n). (Das 2006 p. 229-273.)

In order to maintain a time complexity of O(log n), a binary tree needs to be balanced. Binary trees can be balanced in various ways after a tree is created or altered, or the constraints on the data structure implementation itself can ensure a degree of balance at all times. One example of a self-balancing tree is a Red-Black tree, which is able to ensure a maximum tree height of no more than 2log(n+1), with an insertion time of O(1), which ensures a maximum search time of O(log n). Note that the nodes of the tree contain additional information, in this case the colour of the node, and thus the Red-Black tree uses more memory than a standard binary tree. Another type of self-balancing binary tree is the Adelson-Velsky and Landis (AVL) tree, which is always perfectly balanced, ensuring a maximum height of 1.44log(n+1). The AVL tree does, however, require more rotations on insertion and deletion in order to maintain balance. (Das 2006 p. 229-273; Morin 2013 p. 185-204; Rajput.)

Given these possibilities and the knowledge that lookups will occur much more often than manipulations, the additional time of rotating the AVL tree might be an acceptable compromise to ensure the fastest possible O(log n) search, since the tree height is guaranteed to be less than 1.44log(n+1) compared to the 2log(n+1) of a Red-Black tree (Das 2006 p. 229-273).

6. Testing

The implementation needs to be tested, and is done using a variety of methods. Each specific problem can be tested separately, ensuring all parts of the implementation functions as expected, and functionality can be ensured when changes to the implementation are made later on.

6.1. Bit Manipulation

A important aspect of complying with the protocol is to ensure that the packet's buffer is packed correctly, containing the correct fields at the correct offsets. Packing the buffer requires bit manipulation which could become quite tedious and creates many edge cases which are often overlooked, resulting in errors at runtime.

In order to minimise errors all the bit manipulation is done through pure functions, which have no side effects, and always return the same value given the same input, which is the cornerstone of functional programming (Atencio 2016, p. 3-8). The major advantage is the ability to test the functions easily, since they can be called with test parameters, and the returned value can be asserted, without a global state needing to be set before, or to be cleaned after, the function is called, since the promise is given that the function does not create any side effects. These types of function tests are called Unit Tests (Eugene 2010, p. 163-164).

All bit manipulating functions are contained in the 'Utilities' file and are tested by the unit tests specified in the 'Tests' file. Testing is done in such a manner as to attempt to cover as many potential edge cases as possible, often with erroneous parameters, to ensure the function returns correctly, which can even be throwing a specific error in some cases. Unit tests are implemented with the help of the Chai unit testing Node Package Manager (NPM) package, which facilitates the assertion flow of 'expect' and 'toBe', making the testing simpler and the test code easier to read.

Bit manipulation initially appeared to be quite simple when dealing with a single byte, since a bit can be flipped using simple binary operations, such as 'or' or 'xor', but became very difficult when the manipulation occurs over the boundary of multiple bytes. An example of this phenomenon can be seen with the 'bufferSet' function, where the complexity grows and the solution becomes more complicated than simply setting the value of one byte to an integer. This is due to the shifting of bits over the boundary and the masking of bits which need to be ignored when copying bits to and from buffers.

When bits in a byte are shifted in a standard manner, the bits do not roll over, which means the bits being shifted off the one side do not appear on the other side, but rather the side from which the bits are shifted gets filled with '0's as can be seen in Table 5. In most cases this is the desired behaviour for single byte manipulation since the boundaries of the value being read are within the byte itself. However, when the value being read crosses a byte boundary, and a bit shift occurs, the desired behaviour would be that the bits falling off the preceding byte be placed onto the following byte, rather than be dropped entirely.

Table 5. Two byte, bit shift 4 right

Initial	1	0	1	1	1	1	1	0	1	1	1	1	0	0	0	1
Shift >> 4	0	0	0	0	1	0	1	1	0	0	0	0	1	1	1	1

This problem could be solved by copying the buffer before the bits were shifted, and then using the original copy to copy back the 'missing' bits, which were dropped off from the shift, as can be seen in Table 6. It is important to note, that this only works if the shift is less than 8 bits, since a shift of 8 or more bits, will result in all bytes' bits being '0', since the whole byte has been shifted off. In this case, the modulus of the bits being shifted over 8 needs to be taken, and the padding bytes need to be added to the buffer.

Table 6. Two byte, bit shift 4 right with overflow copy

Initial	1	0	1	1	1	1	1	0	1	1	1	1	0	0	0	1
Shift >> 4	0	0	0	0	1	0	1	1	0	0	0	0	1	1	1	1
Copy	0	0	0	0	1	0	1	1	1	1	1	0	1	1	1	1

The copying of bits is indicated by the dark purple blocks, where the 4 least significant bits in the most significant byte in the initial row is copied to the 4 most significant bits of the least significant byte in the copy row.

6.2. Wire Capture

It is also possible to check the actual bytes on the wire with a tool such as WireShark, to ensure that the traffic is as expected. Table 7 shows the bytes on the wire as cap-

tured by WireShark. The backgrounds are colour grouped, where the green is the Ethernet frame header, the orange is the IP header, and the light blue is the TCP header. The dark blue at the end is the TCP payload, which is the KP data.

Note that in this case, since the test was carried out on the local loopback on a Berkeley Software Distribution (BSD) system instead of over a router, the Ethernet header is only 4 bytes, instead of the usual Ethernet II's 14 bytes. (Harris 2005.)

Table 7. WireShark capture of Server to Client packet

0000	02	00	00	00	45	00	00	42	00	00	40	00	40	06	00	00
0010	7F	00	00	011	7F	00	00	01	0D	05	DF	FD	92	CB	21	B2
0020	F3	03	52	72	80	18	18	EB	FE	36	00	00	01	01	08	0A
0030	31	95	6A	90	31	95	5D	E9	40	00	88	C2	00	00	07	00
0040	00	00	05	00	00	00										

It can be seen from the wire check shown in Table 7, that the sizes of the fields are as expected, and the data present, specifically in the KP data field is as expected.

6.3. Simulation

It is important to test that while the system is running, the server and client are indeed synchronised, since the entire purpose of the protocol is to ensure both the server and client stay synchronised. The 'Simulate' file can be run for both the server and client, at which point the server will start sending packets to the client with simulated time delays and manipulations. Once the simulation is complete, the state of properties on both the server and client is compared to ensure equality.

Note it is important to run the 'Simulate' file as a server before running the file as a client, but the client must be initiated within five seconds of the server, to ensure the client is connected before the server starts sending packets.

7. Conclusion

The aim of this thesis project was to present and define a keyframe based protocol to reduce the needed data transfer in the specific domain of high volume entity property changes present in RTS games and evaluate its performance against a common baseline implementation. From the results presented in the theoretical data rate calculation section of this thesis, it can be seen that the proposed KP can be used to reduce the amount of data sent over the network in very specific cases, namely with very linear, predictable and slow changing properties. The additional advantages of the KP has been shown, such as the ability to sample the world state at any point in time, including in the past, and even the ability to rewind and play the state backwards. It can be seen from the implementation section of this thesis that the KP does present some technical difficulties, such as the complexity of bit shifting over multiple bytes and the data management required to ensure sufficient query performance.

A breakeven formula was derived to determine when the KP would become more efficient than the brute force baseline, and thus providing a method of analysis for potential adopters of the protocol to make a clear judgement on the suitability of using the KP in the intended system. It can, therefore, be said that the KP should be considered in any situation where entities need to be updated over time, and fully recommended in a situation when the target properties have a predicted breakeven point under then calculated value obtained from the breakeven formula. In addition to the breakeven formula, further considerations were mentioned in the theoretical background section of this thesis, prompting caution when the number of properties is excessively massive and that the KP should only be considered in a system where the network is the bottleneck in performance. This is due to the additional CPU overhead for KP properties compared to the studied brute force approach.

As mentioned in the theoretical background section, the KP is able to express the server state to the client with respect to time, giving the client a multi dimensional understanding of the state, building up over time as the server sends additional keyframe data. As explained earlier, this is in stark contrast to the brute force approach which simply sends the instantaneous values to the client, and then in most cases discards the historical state data. In addition, it has been shown that due to this multi dimensional understanding, the client is able to take the historical information into account to calculate additional information, such as the duration that the property has existed or even to predict the behaviour of the property in the event that future keyframes do not exist. Furthermore, it has been shown that although the KP does not directly reduce the network latency, the KP may mitigate the effects of the network latency by sending the

data in advance, meaning the clients are still all able to sample the values at the correct time, regardless of their varying latency.

The KP explored the concept of encoding the value of a property over time, rather than encoding the absolute value of a property in a more compressed format which is common in some of the historical alternatives explored. Even if the protocol presented in this thesis is not successful in practice, the protocol presented provides a first step into the concept of time based encoding and given that the aim of this thesis was to define a keyframe based protocol and compare it to a baseline implementation, it can be said that the initially set objectives, have indeed been achieved.

Although the project presented in this thesis constitutes a complete and functional protocol, the protocol contains many shortcomings and areas of possible improvement for which additional research can be done. From the client's point of view, improvements could be made by, for example, interpolating using various different methods, as opposed to the fixed linear interpolation in the presented specification. Adding more specific behavioural information to the keyframes could be done to tell the client how the keyframe should be read, including spline, polynomial or even piecewise interpolation, giving the ability to encode the change of the value in different ways. This could be particularly useful by rounding off corners, which could encode acceleration without the use of a specific 'first derivative' property as explained in the theoretical background section of this thesis. Furthermore, from the server's point of view, more advanced manipulation methods could reduce the needed instructions that need to be sent to the client, such as a 'move within' method, which moves all keyframes within a given time interval a given offset, thus eliminating the need for a packet to be sent to the client for each 'move' operation of each keyframe in the time interval.

References

Atencio, Luis. 2016. *Functional Programming in Javascript*, Manning Publications, Viewed 22 March 3019, ISBN: 9781617292828

Blank, Andrew G. 2004. *TCP/IP Foundations*, Hoboken: John Wiley & Sons, Inc, Viewed 18 March 2019, Available from: ProQuest Ebook Central

Bourke, Paul. 1999. *Interpolation Methods, Viewed 15 March 2019*, Available from: http://paulbourke.net/miscellaneous/interpolation/

Das, Vinu V. 2006. *Principles of Data Structures Using C and C++*, New Age International, Daryaganj. Viewed 13 April 2019, Available from: ProQuest Ebook Central

Diot, Christophe. 1999. *A distributed architecture for multiplayer interactive applications on the Internet,* IEEE, Viewed 4 April 2019, Available from: https://ieeexplore.ieee.org/abstract/document/777437

Ellie, [FERAL]. 2015. *This is why CoH2 will have per platform multiplayer*, Steam Community, 19 August, Viewed 12 April 2019, Available from: https://steamcommunity.com/groups/maclinux/discussions/0/528398719787800607/

Eugene, Liang Yuxian. 2010. *Javascript Testing Beginner's Guide: Test and Debug Javascript the Easy Way,* Packt Publishing Ltd, Olton. Viewed 2 April 2019, Available from ProQuest Ebook Central

Guojun, Jin. 2002. *Algorithms and Requirements for Measuring Network Bandwidth*, Distributed Systems Department, Lawrence Berkeley National Laboratory,Viewed 1 April 2019, Available from: https://pdfs.semanticscholar.org/02ed/90da10b4aef07f176760d4d4cc45a8a1a66a.pdf

Harris, Guy. 2005. *WireShark Documentation Wiki*, Viewed 11 April 2019, Available from: https://wiki.wireshark.org/NullLoopback?action=recall&rev=2

Information Sciences Institute, University of Southern California. 1981. *Transmission Control Protocol, Darpa Internet Program, Protocol Specification*, IETF*, Viewed 3 February 2019*, Available from: https://tools.ietf.org/html/rfc793

International Telecommunications Union. 2008. *Data communication over telephone network.* Viewed 17 April 2019, Available from: https://www.itu.int/rec/T-REC-V/en

Jackson, Dean; Baron, David L; Atkins, Tab Jr; Birtles, Brian. 2018. *CSS Animations Level 1*, W3C*, Viewed 22 March 2019*, Available from: https://www.w3.org/TR/css-animations-1/

Kiran. 2013. *How are Javascript arrays implemented internally?* Quora, 2 June, Viewed 21 April 2019, Available from: https://www.quora.com/How-are-javascript-arrays-implemented-internally

Langer, Steve G, Todd French, Colin Segovis. 2010. *TCP/IP Optimization over Wide Area Networks: Implications for Teleradiology*, Viewed 7 April 2019, Available from: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3056981/

Liljekvist, Hampus. 2016. *Detecting Synchronisation Problems in Networked Lockstep Games*, Viewed 6 March 2019, Available from: http://www.diva-portal.org/smash/get/diva2:947287/FULLTEXT01.pdf

Marin, G.A. 2005. *Network security basics*, IEEE, https://ieeexplore.ieee.org/document/1556540, 68-72, Viewed 14 April 2019, Available from: doi:10.1109/MSP.2005.153

Miller, Mark A. 2004. *Internet Technologies Handbook: Optimizing the IP Network*, John Wiley & Sons, Incorporated, Hoboken. Viewed 14 April 2019, Available from: ProQuest Ebook Central

Morin, P. 2013. *Open Data Structures: An Introduction,* Atabasca University Press, Edmonton. Viewed 23 April 2019, Available from: ProQuest Ebook Central

Oestreicher, Christian. 2007. *A history of Chaos Theory*, Viewed 21 April 2019, Available from: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3202497/

Pu, Ida Mengyi. 2005. *Fundamental Data Compression,* Elsevier Science & Technology, Oxford. Viewed 21 April 2019, Available from: ProQuest Ebook Central

Rajput, Abhishek. *Self-Balancing Binary Search Trees (Comparisons),* GeeksForGeeks, Viewed 23 April 2019, Available from: https://www.geeksforgeeks.org/self-balancing-binary-search-trees-comparisons/

Redux Documentation. 2018. *Actions*, Viewed 11 April 2019, Available from: https://redux.js.org/basics/actions

Simoneau, Paul. 2006. *The OSI Model: Understanding the Seven Layers of Computer Networks*, Viewed 4 April 2019, Available from: http://ru6.cti.gr/bouras-old/WP_Simoneau_OSIModel.pdf

Smith, Forrest. 2013. *The Tech of Planetary Annihilation: ChronoCam, Viewed 26 November 2018*, Available from: https://www.forrestthewoods.com/blog/tech_of_planetary_annihilation_chrono_cam/

Sterbenz, JPG; Touch, JD; Escobar, J; Krishnan, R; Qiao, C; Sterbenz, J. 2001. *High-Speed Networking: A Systematic Approach to High-Bandwidth Low-Latency Communication,* John Wiley & Sons, Incorporated, New York, Viewed 24 April 2019, Available from: ProQuest Ebook Central

Appendix 1: Typescript Implementation

```
import { Socket, createServer, Server } from 'net'
import {
  bufferAlign,
  bufferExtract,
  formatBuffer,
  bufferSet,
  bufferFromNumber,
  propReduce,
  interpolate,
} from './utils'

const PACKET_SIZE = 14
const TYPE_SYNC = 0
const TYPE_OPEN = 1
const METHOD_ADD = 0
const METHOD_REMOVE = 1
const METHOD_MOVE = 2
const METHOD_META = 3

const packetFieldLayouts: { [key: string]: PacketFieldLayout } = {
  type: {
    bitStart: 0,
    bitCount: 2,
  },
  method: {
    bitStart: 2,
    bitCount: 4,
  },
  baseTime: {
    bitStart: 8,
    bitCount: 8 * 12,
  },
  timeIncrement: {
    bitStart: 6,
    bitCount: 26,
  },
  property: {
    bitStart: 32,
    bitCount: 24,
  },
  value: {
    bitStart: 56,
    bitCount: 32,
  },
  timeOffset: {
    bitStart: 56,
    bitCount: 26,
  },
  timeOffsetNegation: {
    bitStart: 88,
    bitCount: 1,
  },
}

const sendPacket = (connection: Socket, packet: Packet) => {
  const buffer = propReduce(
    (acc, prop) => {
      return bufferSet(
        bufferFromNumber(packet[prop], 12),
        acc,
        packetFieldLayouts[prop].bitCount,
        packetFieldLayouts[prop].bitStart,
      )
    },
```

```
    packet,
    new Buffer(PACKET_SIZE),
  )
  connection.write(buffer)
}

const cleanBuffer = (buffer: Buffer): Buffer => {
  return buffer.slice(0, PACKET_SIZE)
}

const validateBuffer = (buffer: Buffer): boolean => {
  return buffer.length == PACKET_SIZE
}

const fieldValue = (buffer: Buffer, field: PacketFieldLayout): number => {
  const byteCount = Math.ceil(field.bitCount / 8.0)
  return bufferAlign(bufferExtract(buffer, field.bitStart, field.bitCount),
byteCount).readUIntBE(
    0,
    byteCount,
  )
}

interface PacketFieldLayout {
  bitStart: number
  bitCount: number
}

interface Packet {
  type: number
  method: number
}

interface PacketSync extends Packet {
  baseTime: number
}

interface PacketOpen extends Packet {
  timeIncrement: number
  property: number
  value?: number
  timeOffset?: number
  timeOffsetNegation?: number
}

class Property {
  private id: number
  private keyframes: Keyframe[]

  constructor(id: number, keyframes: Keyframe[]) {
    this.id = id
    this.keyframes = keyframes
  }

  private sort() {
    this.keyframes = this.keyframes.sort((a, b) => a.timeIncrement - b.timeIn-
crement)
  }

  at(timeIncrement: number): number {
    const lastIndex = this.keyframes.length - 1
    if (timeIncrement <= this.keyframes[0].timeIncrement) {
      return this.keyframes[0].value
    }
    if (timeIncrement >= this.keyframes[lastIndex].timeIncrement) {
      return this.keyframes[lastIndex].value
    }
    for (let i = 0; i < lastIndex; i++) {
```

```
      if (
        this.keyframes[i].timeIncrement < timeIncrement &&
        this.keyframes[i + 1].timeIncrement > timeIncrement
      ) {
        const x0 = this.keyframes[i].timeIncrement
        const x1 = this.keyframes[i + 1].timeIncrement
        const y0 = this.keyframes[i].value
        const y1 = this.keyframes[i + 1].value
        return interpolate(x0, y0, x1, y1, timeIncrement)
      }
    }
  }

  add(timeIncrement: number, value: number): Property {
    this.keyframes.push({ timeIncrement: timeIncrement, value: value })
    this.sort()
    return this
  }

  remove(timeIncrement: number): Property {
    for (var i = this.keyframes.length - 1; i >= 0; --i) {
      if (this.keyframes[i].timeIncrement === timeIncrement) {
        this.keyframes.splice(i, 1)
      }
    }
    return this
  }

  move(timeIncrement: number, timeOffset: number, timeOffsetNegation: number):
Property {
    const keyframe = this.keyframes.find(keyframe => keyframe.timeIncrement
=== timeIncrement)
    const correctedOffset = timeOffsetNegation === 0 ? timeOffset : -timeOff-
set
    keyframe.timeIncrement = keyframe.timeIncrement + correctedOffset
    this.sort()
    return this
  }
}

interface Keyframe {
  timeIncrement: number
  value: number
}

export abstract class KPStore {
  protected baseTime: number = 0
  protected properties: Property[] = []

  getProperties(): Property[] {
    return this.properties
  }

  private get(id: number): Property {
    const property = this.properties[id]
    if (!property) {
      return (this.properties[id] = new Property(id, []))
    }
    return property
  }

  protected apply(packet: PacketOpen) {
    console.log(`Store: Applying:
      Method: ${packet.method}
      TimeIncrement: ${packet.timeIncrement}
      AbsoluteTime: ${this.baseTime + packet.timeIncrement}
      Property: ${packet.property}
      Value: ${packet.value}
```

```
      TimeOffset: ${packet.timeOffset}
      TimeOffsetNegation: ${packet.timeOffsetNegation}`)
    switch (packet.method) {
      case METHOD_ADD:
        this.get(packet.property).add(packet.timeIncrement, packet.value)
        break
      case METHOD_REMOVE:
        this.get(packet.property).remove(packet.timeIncrement)
        break
      case METHOD_MOVE:
        this.get(packet.property).move(packet.timeIncrement, packet.timeOff-
set, packet.timeOffsetNegation)
        break
      case METHOD_META:
        break
      default:
        break
    }
  }
}

export class KPServer extends KPStore {
  private server: Server
  private connections: Socket[] = []

  constructor() {
    super()
    console.log('Server: Started')
    this.baseTime = Date.now()
    this.server = createServer((connection: Socket) => {
      console.log('Server: Client connected')
      this.connections.push(connection)
      connection.on('data', (buffer: Buffer) => this.onData(connection, buf-
fer))
      connection.on('end', () => this.onEnd(connection))
      connection.on('error', (error: Error) => this.onError(error))
    })
    this.server.listen(3333)
  }

  add(property: number, absoluteTime: number, value: number) {
    if (absoluteTime < this.baseTime) {
      throw Error('Absolute time before base time')
    }
    const timeIncrement = absoluteTime - this.baseTime
    const packet: PacketOpen = {
      type: TYPE_OPEN,
      method: METHOD_ADD,
      timeIncrement: timeIncrement,
      property: property,
      value: value,
    }
    this.send(packet)
    this.apply(packet)
  }

  remove(property: number, absoluteTime: number) {
    if (absoluteTime < this.baseTime) {
      throw Error('Absolute time before base time')
    }
    const timeIncrement = absoluteTime - this.baseTime
    const packet: PacketOpen = {
      type: TYPE_OPEN,
      method: METHOD_REMOVE,
      timeIncrement: timeIncrement,
      property: property,
    }
    this.send(packet)
```

```
    this.apply(packet)
  }

  move(property: number, absoluteTime: number, timeOffset: number) {
    if (absoluteTime < this.baseTime) {
      throw Error('Absolute time before base time')
    }
    if (absoluteTime + timeOffset < this.baseTime) {
      throw Error('Time offset results is absolute time before base time')
    }
    const timeIncrement = absoluteTime - this.baseTime
    const correctedOffset = Math.abs(timeOffset)
    const negation = timeOffset >= 0 ? 0 : 1
    const packet: PacketOpen = {
      type: TYPE_OPEN,
      method: METHOD_MOVE,
      timeIncrement: timeIncrement,
      property: property,
      timeOffset: correctedOffset,
      timeOffsetNegation: negation,
    }
    this.send(packet)
    this.apply(packet)
  }

  private send(packet: Packet) {
    console.log('Server: Sending to clients')
    console.log(packet)
    this.connections.map(connection => {
      sendPacket(connection, packet)
    })
  }

  private onData(connection: Socket, buffer: Buffer) {
    console.log(`Server: Received ${formatBuffer(buffer)}`)
    const a = cleanBuffer(buffer)
    const isValid = validateBuffer(a)

    if (!isValid) {
      console.log('Server: Malformed packet, ignoring...')
      return
    }

    const type = fieldValue(a, packetFieldLayouts['type'])

    switch (type) {
      case TYPE_SYNC:
        this.onTypeSync(connection, a)
        break
      default:
        console.log('Server: Unknown type, ignoring...')
        break
    }
  }

  private onTypeSync(connection: Socket, buffer: Buffer) {
    console.log(`Server: Received sync -> Sending baseTime of ${this.baseTime}
to client`)
    const packet: PacketSync = {
      type: TYPE_SYNC,
      baseTime: this.baseTime,
      method: METHOD_META,
    }
    sendPacket(connection, packet)
  }

  private onEnd(connection: Socket) {
    console.log('Server: Client disconnected')
```

```
    const connectionIndex = this.connections.indexOf(connection)
    if (connectionIndex === -1) {
      console.warn('Server: Could not find connection to remove')
      return
    }
    this.connections.splice(connectionIndex, 1)
  }

  private onError(error: Error) {
    console.log('Server: Error ' + error)
  }
}

export class KPClient extends KPStore {
  private socket: Socket

  constructor() {
    super()
    console.log('Client: Starting')
    this.socket = new Socket()
    this.socket.connect(3333, 'localhost', () => {
      console.log('Client: Connected to server')
      const packet: PacketSync = {
        type: TYPE_SYNC,
        baseTime: 0,
        method: METHOD_META,
      }
      sendPacket(this.socket, packet)
    })
    this.socket.on('data', (buffer: Buffer) => this.onData(this.socket, buf-
fer))
    this.socket.on('end', () => this.onEnd(this.socket))
    this.socket.on('error', (error: Error) => this.onError(error))
  }

  private onData(connection: Socket, buffer: Buffer) {
    console.log(`Client: Received ${formatBuffer(buffer)}`)
    const a = cleanBuffer(buffer)
    const isValid = validateBuffer(a)

    if (!isValid) {
      console.log('Server: Malformed packet, ignoring...')
      return
    }

    const type = fieldValue(a, packetFieldLayouts['type'])

    switch (type) {
      case TYPE_SYNC:
        this.onTypeSync(connection, a)
        break
      case TYPE_OPEN:
        this.onTypeOpen(connection, a)
        break
      default:
        console.log('Server: Unknown type, ignoring...')
        break
    }
  }

  private onTypeSync(connection: Socket, buffer: Buffer) {
    this.baseTime = fieldValue(buffer, packetFieldLayouts['baseTime'])
    console.log(`Client: Received sync -> Setting baseTime to ${this.baseTime}
`)
  }

  private onTypeOpen(connection: Socket, buffer: Buffer) {
    const method = fieldValue(buffer, packetFieldLayouts['method'])
```

```
    const timeIncrement = fieldValue(buffer, packetFieldLayouts['timeIncre-
ment'])
    const property = fieldValue(buffer, packetFieldLayouts['property'])
    const value = fieldValue(buffer, packetFieldLayouts['value'])
    const timeOffset = fieldValue(buffer, packetFieldLayouts['timeOffset'])
    const timeOffsetNegation = fieldValue(buffer, packetFieldLayouts['timeOff-
setNegation'])
    const packet: PacketOpen = {
      type: TYPE_OPEN,
      method: method,
      timeIncrement: timeIncrement,
      property: property,
      value: value,
      timeOffset: timeOffset,
      timeOffsetNegation: timeOffsetNegation,
    }
    this.apply(packet)
  }

  private onEnd(connection: Socket) {
    console.log('Client: Disconnected from server')
  }

  private onError(error: Error) {
    console.log('Client: Error ' + error)
  }
}
```

Appendix 2: Typescript Utilities, Tests and Simulation

```
//// Utils

export const appendString = (baseString: string, appendString: string) =>
baseString + appendString

export const repeatString = (repeatString, count) => {
  let finalString = ''
  for (let i = 0; i < count; i++) finalString = appendString(finalString, re-
peatString)
  return finalString
}

export const formatByte = (byteString: string) => {
  const paddedByte = repeatString('0', 8 - byteString.length) + byteString
  const splitByte = paddedByte.match(/.{1,4}/g).join(' ')
  return splitByte
}

export const formatBuffer = (buffer: Buffer): string => {
  const byteArray = Array.prototype.slice.call(buffer, 0)
  const formattedByteArray = byteArray.map(byte =>
formatByte(byte.toString(2)))
  return formattedByteArray.join(' | ')
}

export const byteMask = (bitStart: number, bitCount: number): number => {
  const endBit = bitStart + (bitCount - 1)
  if (endBit > 7 || bitStart < 0) {
    throw Error('Reading or writing out of byte range')
  }
  return (Math.pow(2, bitCount) - 1) << (8 - bitStart) - bitCount
}

export const bufferMask = (byteCount: number, bitStart: number, bitCount: num-
ber): Buffer => {
  const fullBytes = Math.floor(bitCount / 8.0)
  const bitMod = bitCount % 8
  const shiftLeft = (byteCount * 8) - bitStart - bitCount
  const a = new Buffer(byteCount - fullBytes - 1)
  const b = new Buffer([byteMask(8 - bitMod, bitMod)])
  const c = Buffer.alloc(fullBytes, 0xFF)
  const d = Buffer.concat([a, b, c])
  const e = bufferShiftLeft(d, shiftLeft)
  return e as Buffer
}

export const bufferAlign = (buffer: Buffer, desiredByteCount: number): Buffer
=> {
  if (buffer.length > desiredByteCount) {
    const excessCount = buffer.length - desiredByteCount
    return buffer.slice(excessCount, buffer.length)
  }
  const paddingBuffer = new Buffer(desiredByteCount - buffer.length)
  const bufferArray = [paddingBuffer, buffer]
  return Buffer.concat(bufferArray)
}

export const bufferClearLeft = (buffer: Buffer, bitClearCount: number): Buffer
=> {
  const a = new Buffer(buffer)
  a[0] = a[0] & (Math.pow(2, 8 - bitClearCount) - 1)
  return a
}
```

```
export const byteSet = (source: number, destination: number, bitStartSource:
number, bitCount: number, bitStartDestination: number): number => {
  const bitEndSource = bitStartSource + (bitCount - 1)
  const bitEndDestination = bitStartDestination + (bitCount - 1)
  if (bitEndSource > 7 || bitEndDestination > 7 || bitStartSource < 0 || bit-
StartDestination < 0) {
    throw Error('Reading or writing out of byte range')
  }
  const maskSource = byteMask(bitStartSource, bitCount)
  const maskDestination = byteMask(bitStartDestination, bitCount)
  const shift = bitStartDestination - bitStartSource
  const a = source & maskSource
  const b = shift > 0 ? a >> shift : a << shift * -1
  const c = destination & ~maskDestination
  const d = b | c
  return d
}

export const bufferShiftLeft = (buffer: Buffer, shift: number) => {
  const fullShifts = Math.floor(shift / 8.0)
  const shiftRemainder = shift - (fullShifts * 8)
  const a = buffer.slice(fullShifts)
  const b = a.map(byte => byte << shiftRemainder)
  if (buffer.length === 1) {
    return b
  }
  const c = b.map((byte, i) => i < buffer.length - 1 ? byteSet(a[i+1], byte,
0, shiftRemainder, 8 - shiftRemainder) : byte)
  const d = Buffer.concat([c, new Buffer(fullShifts)])
  return d
}

export const bufferExtract = (buffer: Buffer, bitStart: number, bitCount: num-
ber): Buffer => {
  if (bitCount > buffer.length * 8 - bitStart || bitCount <= 0 || bitStart < 0
|| bitStart >= buffer.length * 8) {
    throw Error('Reading or writing out of byte range')
  }
  const bitEnd = bitStart + bitCount
  const byteStart = Math.floor(bitStart / 8.0)
  const byteEnd = Math.floor((bitStart + bitCount - 1) / 8.0)
  const byteCountSource = byteEnd - byteStart + 1
  const byteCountDestination = Math.ceil(bitCount / 8.0)
  const offsetLeft = bitStart % 8
  const offsetRight = bitEnd % 8 == 0 ? 0 : 8 - (bitEnd % 8)
  const leftClear = (offsetLeft + offsetRight) % 8
  if (byteCountSource == 1) {
    const a = new Buffer([buffer[byteStart] >> offsetRight])
    const b = bufferAlign(a, 1)
    const c = bufferClearLeft(b, leftClear)
    return c
  }
  const a = buffer.slice(byteStart, byteEnd + 1)
  const b = a.map(byte => byte >> offsetRight)
  const c = b.map((byte, i) => (i > 0 ? byte | (a[i - 1] << (8 - offsetRight))
: byte))
  const d = bufferAlign(c as Buffer, byteCountDestination)
  const e = leftClear == 8 ? d : bufferClearLeft(d, leftClear)
  return e
}

export const bufferSet = (source: Buffer, destination: Buffer, bitCount: num-
ber, bitStartDestination: number): Buffer => {
  if (bitCount > destination.length * 8 - bitStartDestination || bitCount <= 0
|| bitStartDestination < 0 || bitStartDestination >= destination.length * 8) {
    throw Error('Reading or writing out of byte range')
  }
  const byteCount = destination.length
```

```
  const sourceAligned = bufferAlign(source, byteCount)
  const sourceBitLength = byteCount * 8
  const leftShift = sourceBitLength - (bitCount + bitStartDestination)
  const maskSourceBitStart = sourceBitLength - bitCount
  const maskSource = bufferMask(byteCount, maskSourceBitStart, bitCount)
  const maskDestination = bufferMask(byteCount, bitStartDestination, bitCount)
  const maskedSource = sourceAligned.map((byte, i) => byte & maskSource[i])
  const a = bufferShiftLeft(maskedSource as Buffer, leftShift)
  const maskedDestination = destination.map((byte, i) => byte & ~maskDestina-
tion[i])
  const c = a.map((byte, i) => byte | maskedDestination[i])
  return c as Buffer
}

export const bufferFromNumber = (value: number, byteCount: number) => {
  const buffer = new Buffer(byteCount)
  buffer.writeUIntBE(value, 0, byteCount)
  return buffer
}

export const propMap = (callbackfn: (prop: string) => void, object: Object) =>
{
  for(const prop in object) {
    callbackfn(prop)
  }
}

export const propReduce = (callbackfn: (accumulator: any, prop: string) =>
any, object: Object, accumulator: any) => {
  for(const prop in object) {
    accumulator = callbackfn(accumulator, prop)
  }
  return accumulator
}

export const interpolate = (x0: number, y0: number, x1: number, y1: number,
tx: number): number => {
  const tr = (tx - x0) / (x1 - x0)
  const ty = ((1 - tr) * y0) + (tr * y1)
  return ty
}
```

```
//// Tests

import { expect } from 'chai'

import * as Utils from '../src/utils'

describe('String Utils', () => {
  describe('appendString', () => {
    it('should append string to end of base string', () => {
      expect(Utils.appendString('hello', 'world')).to.equal('helloworld')
    })
  })
  describe('repeatString', () => {
    it('should repeat string specified number of times', () => {
      expect(Utils.repeatString('0', 4)).to.equal('0000')
    })
    it('should return empty when count is 0', () => {
      expect(Utils.repeatString('0', 0)).to.equal('')
    })
    it('should return base string when count is 1', () => {
      expect(Utils.repeatString('0', 1)).to.equal('0')
    })
  })
})

describe('Buffer Utils', () => {
  describe('formatByte', () => {
    it('should split binary byte string into 4 bit binary groups', () => {
      expect(Utils.formatByte('10010111')).to.equal('1001 0111')
    })
    it('should pad binary string to 8 bits', () => {
      expect(Utils.formatByte('010111')).to.equal('0001 0111')
    })
  })
  describe('formatBuffer', () => {
    it('should split buffer into binary byte strings', () => {
      expect(Utils.formatBuffer(new Buffer([0b11001100, 0b01110010]))).to.e-
qual(
        '1100 1100 | 0111 0010',
      )
    })
  })
  describe('byteMask', () => {
    it('should throw error if reading or writing outside bounds', () => {
      expect(() => Utils.byteMask(0, 9)).to.throw('Reading or writing out of
byte range')
      expect(() => Utils.byteMask(-4, 6)).to.throw('Reading or writing out of
byte range')
      expect(() => Utils.byteMask(6, 6)).to.throw('Reading or writing out of
byte range')
    })
    it('should create mask at specified start bit for specified bit count', ()
=> {
      expect(Utils.byteMask(0, 8)).to.equal(255)
      expect(Utils.byteMask(4, 3)).to.equal(14)
      expect(Utils.byteMask(2, 5)).to.equal(62)
      expect(Utils.byteMask(2, 4)).to.equal(60)
      expect(Utils.byteMask(4, 4)).to.equal(15)
      expect(Utils.byteMask(0, 4)).to.equal(240)
      expect(Utils.byteMask(1, 3)).to.equal(112)
    })
  })
  describe('bufferMask', () => {
    it('should create mask in buffer with the specified length and start bit',
() => {
      expect(Utils.bufferMask(1, 0, 4)).to.deep.equal(new
Buffer([0b11110000]))
```

```
      expect(Utils.bufferMask(2, 0, 4)).to.deep.equal(new Buffer([0b11110000,
0b00000000]))
      expect(Utils.bufferMask(2, 5, 3)).to.deep.equal(new Buffer([0b00000111,
0b00000000]))
      expect(Utils.bufferMask(2, 5, 4)).to.deep.equal(new Buffer([0b00000111,
0b10000000]))
      expect(Utils.bufferMask(3, 5, 4)).to.deep.equal(new Buffer([0b00000111,
0b10000000, 0b00000000]))
      expect(Utils.bufferMask(3, 9, 4)).to.deep.equal(new Buffer([0b00000000,
0b01111000, 0b00000000]))
      expect(Utils.bufferMask(3, 9, 10)).to.deep.equal(new Buffer([0b00000000,
0b01111111, 0b11100000]))
    })
  })
  describe('bufferAlign', () => {
    it('should pad the buffer to the specified number of bytes', () => {
      expect(Utils.bufferAlign(new Buffer([0xaf, 0x3e]), 4)).to.deep.equal(
        new Buffer([0x00, 0x00, 0xaf, 0x3e]),
      )
    })
    it('should chop preceeding bytes over the specified size', () => {
      expect(Utils.bufferAlign(new Buffer([0xa2, 0x55, 0x3e]), 2)).to.deep.e-
qual(
        new Buffer([0x55, 0x3e]),
      )
    })
  })
  describe('bufferClearLeft', () => {
    it('should clear bits in the first byte', () => {
      expect(Utils.bufferClearLeft(new Buffer([0xaf]), 4)).to.deep.equal(
        new Buffer([0x0f]),
      )
    })
    it('should clear bits in the first byte when buffer contains multiple
bytes', () => {
      expect(Utils.bufferClearLeft(new Buffer([0xaf, 0x3e, 0x4d]),
2)).to.deep.equal(
        new Buffer([0x2f, 0x3e, 0x4d]),
      )
    })
    it('should clear the whole first byte', () => {
      expect(Utils.bufferClearLeft(new Buffer([0xaf, 0x3e, 0x4d]),
8)).to.deep.equal(
        new Buffer([0x00, 0x3e, 0x4d]),
      )
    })
  })
  describe('byteSet', () => {
    it('should throw error if reading or writing outside bounds', () => {
      expect(() => Utils.byteSet(0, 0, 0, 9, 0)).to.throw('Reading or writing
out of byte range')
      expect(() => Utils.byteSet(0, 0, -2, 4, 0)).to.throw('Reading or writing
out of byte range')
      expect(() => Utils.byteSet(0, 0, 6, 3, 0)).to.throw('Reading or writing
out of byte range')
    })
    it('should set destination byte bits to the same as source', () => {
      expect(Utils.byteSet(0xff, 0x00, 6, 2, 6)).to.equal(0x03)
      expect(Utils.byteSet(0xff, 0x00, 0, 8, 0)).to.equal(0xff)
      expect(Utils.byteSet(0xaf, 0xf5, 4, 4, 4)).to.equal(0xff)
    })
  })
  describe('bufferShiftLeft', () => {
    it('should shift single byte', () => {
      expect(Utils.bufferShiftLeft(new Buffer([0b00110000]), 2)).to.deep.e-
qual(
        new Buffer([0b11000000]),
      )
```

```
    })
    it('should shift single byte out of byte edge', () => {
      expect(Utils.bufferShiftLeft(new Buffer([0b00110000]), 3)).to.deep.e-
qual(
        new Buffer([0b10000000]),
      )
    })
    it('should shift muiltiple bytes', () => {
      expect(Utils.bufferShiftLeft(new Buffer([0b00000110, 0b01100000]),
1)).to.deep.equal(
        new Buffer([0b00001100, 0b11000000]),
      )
    })
    it('should shift muiltiple bytes and roll bits over to next byte', () => {
      expect(Utils.bufferShiftLeft(new Buffer([0b00000110, 0b01100000]),
2)).to.deep.equal(
        new Buffer([0b00011001, 0b10000000]),
      )
      expect(Utils.bufferShiftLeft(new Buffer([0b00000110, 0b01100000,
0b11001100]), 2)).to.deep.equal(
        new Buffer([0b00011001, 0b10000011, 0b00110000]),
      )
    })
    it('should shift muiltiple full bytes', () => {
      expect(Utils.bufferShiftLeft(new Buffer([0b00000110, 0b01100000,
0b11001100]), 8)).to.deep.equal(
        new Buffer([0b01100000, 0b11001100, 0b00000000]),
      )
    })
    it('should shift muiltiple full bytes and remainder', () => {
      expect(Utils.bufferShiftLeft(new Buffer([0b00000110, 0b01100000,
0b11001100]), 9)).to.deep.equal(
        new Buffer([0b11000001, 0b10011000, 0b00000000]),
      )
      expect(Utils.bufferShiftLeft(new Buffer([0b00000110, 0b01100000,
0b11001100, 0b11111111]), 17)).to.deep.equal(
        new Buffer([0b10011001, 0b11111110, 0b00000000, 0b00000000]),
      )
    })
  })
  describe('bufferExtract', () => {
    it('should throw error if reading or writing outside bounds', () => {
      expect(() => Utils.bufferExtract(
        new Buffer([0b11111111, 0b00110011, 0b00001111]),
        -1,
        2,
      )).to.throw('Reading or writing out of byte range')
      expect(() => Utils.bufferExtract(
        new Buffer([0b11111111, 0b00110011, 0b00001111]),
        28,
        2,
      )).to.throw('Reading or writing out of byte range')
      expect(() => Utils.bufferExtract(
        new Buffer([0b11111111, 0b00110011, 0b00001111]),
        16,
        10,
      )).to.throw('Reading or writing out of byte range')
    })
    it('should extract and align from single byte', () => {
      expect(Utils.bufferExtract(new Buffer([0xaf, 0x3e]), 0, 4)).to.deep.e-
qual(new Buffer([0x0a]))
      expect(Utils.bufferExtract(new Buffer([0xaf, 0x3e]), 4, 4)).to.deep.e-
qual(new Buffer([0x0f]))
    })
    it('should extract and align over byte boundary', () => {
      expect(Utils.bufferExtract(new Buffer([0xaf, 0x3e, 0x44]), 4,
8)).to.deep.equal(
        new Buffer([0xf3]),
```

```
      )
    })
    it('should extract and align more than a single byte', () => {
      expect(Utils.bufferExtract(new Buffer([0xaf, 0x3e, 0x44]), 6,
10)).to.deep.equal(
        new Buffer([0x03, 0x3e]),
      )
    })
    it('should extract and align and pack less than a byte', () => {
      expect(Utils.bufferExtract(new Buffer([0xaf, 0x3e]), 7, 2)).to.deep.e-
qual(new Buffer([0x02]))
      expect(Utils.bufferExtract(new Buffer([0xaf, 0x3e]), 7, 5)).to.deep.e-
qual(new Buffer([0x13]))
      expect(Utils.bufferExtract(new Buffer([0xaf, 0x3e]), 14, 2)).to.deep.e-
qual(new Buffer([0x02]))
    })
  })
  describe('bufferSet', () => {
    it('should throw error if reading or writing outside bounds', () => {
      expect(() => Utils.bufferSet(
        new Buffer([0b11111111, 0b00110011, 0b00001111]),
        new Buffer([0b00000000, 0b00000000, 0b00000000]),
        -1,
        2,
      )).to.throw('Reading or writing out of byte range')
      expect(() => Utils.bufferSet(
        new Buffer([0b11111111, 0b00110011, 0b00001111]),
        new Buffer([0b00000000, 0b00000000, 0b00000000]),
        0,
        26,
      )).to.throw('Reading or writing out of byte range')
      expect(() => Utils.bufferSet(
        new Buffer([0b11111111, 0b00110011, 0b00001111]),
        new Buffer([0b00000000, 0b00000000, 0b00000000]),
        27,
        4,
      )).to.throw('Reading or writing out of byte range')
    })
    it('should copy to 0x00 set buffer', () => {
      expect(Utils.bufferSet(
        new Buffer([0b11111111, 0b00110011, 0b00001111]),
        new Buffer([0b00000000, 0b00000000, 0b00000000]),
        4,
        2,
      )).to.deep.equal(new Buffer([0b00111100, 0b00000000, 0b00000000]))
    })
    it('should mix with destination when not overlapping', () => {
      expect(Utils.bufferSet(
        new Buffer([0b11111111, 0b00110011, 0b00001111]),
        new Buffer([0b00000000, 0b00000000, 0b11000000]),
        4,
        2,
      )).to.deep.equal(new Buffer([0b00111100, 0b00000000, 0b11000000]))
    })
    it('should copy more than one byte', () => {
      expect(Utils.bufferSet(
        new Buffer([0b11111111, 0b00110011, 0b00001111]),
        new Buffer([0b00000000, 0b00000000, 0b11000000]),
        16,
        0,
      )).to.deep.equal(new Buffer([0b00110011, 0b00001111, 0b11000000]))
    })
    it('should copy accross byte boundaries', () => {
      expect(Utils.bufferSet(
        new Buffer([0b11111111, 0b00110011, 0b00001111]),
        new Buffer([0b00000000, 0b00000000, 0b11000000]),
        16,
        4,
```

```
      )).to.deep.equal(new Buffer([0b00000011, 0b00110000, 0b11110000]))
    })
  })
  describe('bufferFromNumber', () => {
    it('should return a buffer from given value', () => {
      expect(Utils.bufferFromNumber(4, 2)).to.deep.equal(new Buffer([0x00,
0x04]))
      expect(Utils.bufferFromNumber(256, 2)).to.deep.equal(new Buffer([0x01,
0x00]))
    })
  })
  describe('interpolate', () => {
    it('should interpolate linearly between the parameters', () => {
      expect(Utils.interpolate(10, 5, 20, 10, 15)).to.equal(7.5)
      expect(Utils.interpolate(10, 10, 20, 5, 15)).to.equal(7.5)
      expect(Utils.interpolate(10, 10, 20, 5, 20)).to.equal(5)
      expect(Utils.interpolate(10, 10, 20, 5, 10)).to.equal(10)
    })
  })
})
```

```
//// Simulation

import { KPServer, KPClient } from './KPNet'

const delay = milliseconds => {
  return new Promise(resolve => setTimeout(resolve, milliseconds))
}

const IS_SERVER = process.argv[2] === '--server'

if (IS_SERVER) {
  const kpServer = new KPServer()
  const gameLogic = async () => {
    await delay(5000)

    kpServer.add(7, Date.now() + 1000 * 30, 5)

    await delay(1000)

    kpServer.add(2, Date.now() + 1000 * 4, 9)
    kpServer.add(7, Date.now() + 1000 * 14, 6)

    await delay(9000)
    const tenSecondsFromNow = Date.now() + 1000 * 10
    const fifteenSecondsFromNow = Date.now() + 1000 * 15

    kpServer.add(2, Date.now() + 1000 * 6, 22)
    kpServer.add(7, tenSecondsFromNow, 3)
    kpServer.add(7, fifteenSecondsFromNow, 3)

    await delay(2000)

    kpServer.remove(7, tenSecondsFromNow)

    await delay(1000)

    kpServer.move(7, fifteenSecondsFromNow, -15000)

    await delay(1000)

    console.log(JSON.stringify(kpServer.getProperties()))
  }
  gameLogic()
} else {
  const kpClient = new KPClient()

  const gameLogic = async () => {
    await delay(20000)

    console.log(JSON.stringify(kpClient.getProperties()))
  }
  gameLogic()
}
```