# Networked Simulation

## Opportunity and Architecture

Jad Nohra, with contributions by Teodor Cioacă

# I Introduction

## I.1 An Ocean of Servers

Along with XboxOne's arrival, Microsoft announced free dedicated servers to any developer wanting to use them. These are allocated from a pool of 300,000 servers which is powered by the larger Azure cloud. As of October 2014, Azure touts a total of 11 million servers[1], recently revamped to try to accommodate for high performance computing[2], despite the challenges[10]. At least one known case of a game that relies on the XboxOne's dedicated servers is 'Titanfall'. When Titanfall evaluated its options[3], it found that it could not get reasonably priced deals with cloud computing providers like Amazon and Rackspace, but this may change in the future. On the PS4 side, 30,000 servers are available, with different conditions. Of course, all major game publishers have their own private data centers for which we do not have numbers.

Video game streaming is another facet of the networked simulation trend. It is relevant to us because games that already run in the data center, even though on one server, are more readily extendable to multiple. Video game streaming is not at all without technical challenges not to mention commercial ones. To summarize the former, one could boil it down to the technical battle of latency[20], with bandwidth almost solved due to its prime importance to the movie publishing industry. When it comes to streaming games as video, Crytek already considered this in 2005 and paused it two years later because the internet infrastructure was not ready. In 2013, the company declared that this is changing[19], mainly with the push from the aforementioned movie publishing industry.

Despite the difficulties, many companies that consider themselves industry leaders already take networked simulation seriously. Most of our references for this section date from 2013, while NVidia's GRID was already fully operational in 2012.

This ocean of servers is a reality, along with the gargantuan financial investment already made in it and its infrastructure, pressuring to be capitalized on. In the meantime products such as our physics do not run on two processors connected by a network cable[1]. While it is true that the demand for such a feature is not already hitting us from all sides, there are reasons to be proactive about it, at least the fact that once this happens, it will be too late to start development and opportunities might get lost.

[1] Granted, neither does NVidia's PhysX

## I.2 Wide but Manageable

Networked simulation, even restricted to rigid body dynamics, remains quite an ill defined concept. It often happens that two people discussing it without explicit agreement on terminology are talking about two different problems, requiring two different solutions. In fact, the scope of the whole topic is quite appreciably large once one seriously considers all the relevant factors.

The effect of this will be visible multiple times in this document. This is not lack of focus but a purposeful act aiming at the seeding of ideas and the creation of a terminology easing effective communication. Another goal of this wide exposition is to show that quickly jumping in and starting an implementation with no detailed plan will be of very limited utility because so many variables have to be fixed at the onset.

We will mainly try to show that this width need to startle us. There is a non-trivial and well defined technological common denominator that is worth investing in. This investment is not an all or nothing bet on the success of cloud gaming or video game streaming. Instead, it will help us handle with more confidence, more ease, less reluctance and less panic the unknowns to face whatever they are. This 'investment' is by all measures microscopic compared to the investments made by hardware oriented companies. Finally and more technically, we note that this is the natural step after instruction and thread level parallelism, and one of the very few options left for more performance ever since the frequencies of conventional processors stopped following Moore's law.

## I.3 Opportunity and Difficulty

What is shared between all use cases of switching to networked simulation is that a 'traditional' more or less monolithic application is subjected to splitting into parts, and potentially to the parallelization of some of these parts over a network. Both the splitting and the parallelization provide opportunities for technological features not yet present in our products.

Common parallelization obstacles are amenability to parallelization, added complexity, overhead and scaling caps as well as technical skill[2]. Since the parallelization happens over a physical network, the overhead in terms of latency and the caps in terms of bandwidth are in the general case quite considerable and constitute a large part of the technical challenge if not almost all of it. We say this to stress that a design or even a prototype that deals lightly with these two issues will fail irreparably on real hardware.

## I.4 Terminology and Classification

We found no better way to present the classifications and various terms we need other than simply putting them all in this single large list.

[2] See Appendix: Parallel Computing Skills

1. Compute Class. Since the application is split, different roles can reside on computers of different computational capability.
    1. **Mobile**. A mid-end smartphone or tablet.
    2. **Personal**. The kind of computer an average PC gamer is expected to possess.
    3. **VM**. Usually in a computing cloud, but also potentially in a dedicated server farm for load balancing. When cost is critical, and VM instances of low performance are chosen (they also usually come with low bandwidth)[5], this must be taken into account in the design. We note that in general, instances with a certain spec perform surprisingly badly compared to a dedicated computer with the same spec. The reasons for this are the hypervisor, load balancing and other details.[12]
    4. **Power**. The kind of server hardware used in a high performance data center. We do not call this 'server' for disambiguation with the software location class below, and because in some applications[3], one could have power clients.
2. Bandwidth Class. For client communication, we use current video streaming bandwidth demands as a reference because it is the video-on-demand industry that is shaping the internet's infrastructure. For communication within the data center, it is Eth1 and above that are relevant.
    1. **SD**. Taken from Netflix recommendations, this is the bandwidth recommended for stream SD video (720x576) and is around **3 Mbps**.
    2. **HD**. Around **5 Mbps**
    3. **Eth1**. The ethernet **1 Gbps** standard, now mostly overhauled, but possibly in use in a low budget data center.
    4. **Eth10**. The ethernet **10 Gbps** standard. This is what will mostly be found as a standard[6,16] in either a dedicated data center or a cloud computing service, between two compute nodes. The connectivity of multiple nodes is a complicated IT matter. For example,
        - Azure totally revamped its internal design to what it calls the 'quantum 10' architecture in 2012, with a total bandwidth of 30,000 Gbps.[4]
        - The total bandwidth for a supercomputer cluster in a rack can range from 48 to 160 Gbps.[18]
    5. **Infini40**. Not only in supercomputers where it is by now standard, but also in networks now available to game developers, infiniband, also called RDMA, provides **40 Gbps**[8,2]. When heterogeneous computing is considered with two processors or more sharing the same mainboard, the communication between them can use 'QuickPath Interconnect' (**QPI**) if they are Intel based. Since the bandwidth of QPI is also around 40 Gbps, we include it with Infini40.[17]
3. Latency Class.
    1. **Micro**. The latest switches have the capability of providing a latency of around **3 $\mu$s** but can even go down to an impressive 90 nanoseconds, but this

[3] See: MilSim Opportunities

only holds for the smallest packet sizes, and could potentially be exploited for very fine grained synchronisation. [11,13]

2. **Milli**. A latency of around **1 ms** should be the reference for calculations within the data center.[9]
3. **Internet**. A latency of around **15 ms** should be the reference for calculations involving an internet client and a web facing machine in the data center. This also holds for machines within the same cloud but in different data centers. [15]
4. **High**. Anything higher than a **15 ms** latency, that is, half a frame in a 30 fps simulation.

4. Scaling Class. A very rough terminology for the number of instances, be it clients servers, compute nodes, etc.
    1. **Single**.
    2. **Multiple**. More than one.
    3. **Many**. Multiples instances reaching the usual maximum depending on the context. Per example, in the context of a multi player shooter, this could be 24 for number of clients.
    4. **Large**. More than many, hinting to a technically impressive scale.
5. Software Location. By software we mean either simulation or application code. In ambitious scenarios, both application and simulation code are spread amongst all locations.
    1. **Client**. The client is the end user of the simulation.
    2. **Server**. The server is anything that is not a client.
        1. **Compute Node**. A node where the bulk of the computations are done.
        2. **Aggregator**. A node that aggregates results from compute nodes for redistribution or other purposes. Should application code reside in the server, an aggregator would be a good place for it because at the aggregator the full view of the simulation is available, which is not true on individual compute nodes.
        3. **Orchestrator**. Should compute nodes be in any way orchestrated, or share data in a central entity, this entity would be called an orchestrator.
6. Hardware Location.
    1. **Internet**. This is where **client** computers will be located in general.
    2. **Cloud**. Despite the extreme fuzziness of the term 'cloud', we here mean a computing cloud service that is public and hence has a specific fixed api not under the control of the application or simulation.
    3. **Dedicated Data Center**. A private data center, with machines dedicated to the application and simulation, but possibly sharing network infrastructure. In a dream scenario, even the hardware network infrastructure could be available for tweaking.[14]

4. **Shared Data Center**. A private data center, but with computers being shared by multiple applications. In this scenario, the simulation must be ready to handle additional unpredictability in processor performance and bandwidth.

7. Compute Role.
    1. **Physics**. **hknpPhysics**.
    2. **Fx**. **hknpFX**.
    3. **Destruction**. **hknDestruction**.
    4. **Query**. Querying the world but not interacting with it.
    5. **Application**. Application specific code that interacts with the simulation using its API, along with any networking extensions to it.
    6. **Render**.
8. Client Class.
    1. **Active**. A client that interacts with the simulation and hence needs low latency, or at least the illusion of it.
    2. **Passive**. An observing client. Here latency is less of an issue.
9. Usage Class.
    1. **Permanent**. The application uses the simulation permanently.
    2. **Burst**. The application uses the simulation permanently, but in addition, scales from time to time to use some simulation extensions that are very heavy on processing.

## I.5 Scenarios

We shall describe scenarios by mixing and combining items from the above list. Clearly we have a combinatorial explosion of possibilities not present in the case of a non-networked application. Despite this, there does exist a non-trivial technological denominator common to all of them. This common denominator should be the target for the first technology prototypes.

## I.6 Client Opportunities

Even though we are trying to look into the future, a future in which the clients themselves will be looking for ways to create more immersive, more detailed worlds, possibly by reaching for server or cloud computing, it is more immediate to focus on the ones that already use networking to begin with. We shall without proper research list some basic opportunities that come to mind, in the hope they are corrected and refined by more knowledgable colleagues.

### I.6.1 Bubble Simulation

Since our physics has no built-in networking support, some clients already use a local simulation around the 'player' on the client to hide latency. We call this 'bubble simulation'. Our defunct Copenhagen team had developed a prototype for such a feature but it has not been pursued. This could now be continued and made into a built-in feature.

### I.6.2 Compression

The minimal multi player client-server setup using our terminology is:

1. Multiple active personal computer clients with SD bandwidth.
2. Single power computer server which is dedicated and has internet latency to the client.
3. Physics, destruction and querying are located at the server.
4. Rendering is located at the client.

In this case, the opportunity is adding high quality and tunable compression schemes to the physics and destruction products for transferring their data to the client for rendering. These schemes should be on par or better than the clients' existing implementations.

**I.6.2.1 Video Streaming** If video game streaming becomes mainstream[4], the need for server to client compression will dwindle and the opportunity might become outdated. We think that latency is the only thing that could prevent this from happening in the longterm because other than that, this kind of networked solution removes a large amount of complexity from a networked application and is thus technically very attractive. However, if latency problems are not completely overcome, there will always be clients that are not dumb terminals, and where 'bubble simulation' can be performed as 3D data is streamed instead of video data that is only usable for display.

### I.6.3 Permanent Distributed World

The difference from the pervious case is point 2.

- Multiple dedicated servers in a data center with milli latency and Eth10 bandwidth between them.

In this case, the opportunity arises if the physics simulation exceeds the frame budget when run on a single server. This leads to the feature that comes most naturally to mind when one mentions networked physics: built-in support in the product to seemlessly simulate one 'world' across multiple networked computers.

[4] Currently, 32 simultaneous users per GPU are possible with one Nvidia K1 card[21].

Currently, the physics density [5] of common applications is quite low. If the distribution of density is homogenous, we have two cases:

1. The client really does not need more density, at least not as a luxury he is willing to pay any price for.
2. The client would like more density as long as the frame budget is kept fixed. Given that the simulation is already multi-server and hence the additional development cost small, the opportunity is real.

### I.6.4 Burst Distributed World

As opposed to the previous scenario, the density distribution is not homogeneous and while the world is mostly 'physics sparse', very high usage of physics is concentrated at certain locations and/or points in time (e.g 'boss level'). In this case, the opportunity does arise in two forms.

1. The client is willing to use permanent physics simulation on multiple servers to keep things simple.
2. The client is not willing to do so, but would like to handle the dense spots and moments smoothly, or make them even more dense and 'impressive'. In this case, the opportunity for ***burst*** mode arises. The physics world runs on a single server, but at certain locations or times known in advance, other computations are reduced and a portion of the physics world is run on multiple servers, distributing the cost. This might be even more relevant for destruction, where the computational cost can is higher than physics.

In a more ambitious use case, one could dream of ***burst*** mode on demand, the additional challenge being that starting the ***burst*** mode will probably results in an unacceptable bandwidth and computation spike if implemented naively.

### I.6.5 Distributed Querying

We have yet to hear of a game AI programmer who does not crave more raycasts. In general, physics clients do usually have concerns about the number of possible raycasts per second. For other query types, Havok Fx is for example a heavy user of closest point queries with a cost of … per query, measured in a specific test on a specific processor. As opposed to dynamics, a lucky feature of queries is that the application can, and for many clients already does, live with deferred queries. If in addition the client's setup in networked one might envisage the following scenario, differing from the distributed world one by the following:

- Physics (and destruction) running on only one of the servers.

[5] See Appendix: Physics Density

- A mirroring feature of only the broad and narrow phase on other servers, for the sole purpose of processing queries.

If bandwidth is a problem, and queries are allowed to be off by a frame or two, the mirroring could use a UDP variant. We will not discuss the architecture of such mirroring in this document, however, it would share a good part of the distributed world infrastructure and might be a more realistic opportunity because gameplay and AI can make qualitative jumps if the number of raycasts is increased by orders of magnitude. This should be possible by simply adding more query processing servers. One might even envisage a cluster of servers dedicated for query processing for multiple games, maybe even combined with ***burst*** mode.

### I.6.6 Exotic Opportunities

It does not to hurt to mention quite roughly more exotic or downright ridiculous 'opportunities' for the sake of documentation.

1. Accelerated Physics on a Mobile. Using the computing power of the server for either an online single player, or a multiplayer mobile game.
2. Large Scale Passive on Mobile. A very large number of passive mobile clients join in to watch a twitch.tv kind of viewing. This is possible if our compression efforts turn out to be quite impressive and video streaming is not an option.
3. Video Streaming Acceleration. An even more dreamy vision is using the rendering capabilities of the viewers as an alternative to video streaming, weighing on a compression that is heavily tuned towards physical behavior (e.g: compressing a ballistic trajectory needs only the initial velocity for the whole trajectory), offering HD rendering on network connections that are sub-HD.
4. Culling Compression. Performing some sort of software culling in order to save bandwidth.

## I.7 Internal Opportunities

One could argue that these are not even opportunities but necessities, since their abscence can block whole teams and products from succesfully testing the waters of networked simulation.

### I.7.1 A Networking Framework

Should any of our product teams decide to experiment with networked simulation, they would find themselves missing quite an amount of infrastructure. The minimum benefit from taking networked physics even half seriously would be the creation of such an infrastructure. Comprised of a solid low level networking library, a performance and networking oriented serialization library and a discovery library. From our experience we currently have none of these.

### I.7.2 A Cloud Computing Framework

When not using dedicated servers, some aspects of the software to be deployed might vary, at least the method of deployment and scaling management. To accelerate experimentation and testing, a basic infrastructure that support all interesting cloud providers could be implemented.

## I.8 MilSim Opportunities

Unlike gaming, MilSim might be especially attractive because the trend is larger and larger simulations and more fidelity demand. Furthermore, the hardware is much more configurable and the cost is less of an issue when it gets the job done. In fact, we have already had very recently concrete interest from on client, but this is beyond the scope of this document.

## I.9 Prestige Opportunities

Having a networked physics and/or destruction prototype that scales reasonably well, one could give it as much computing resources and as much bandwidth as it can take using bought hardware, borrowed hardware, rented hardware, or a high performance cloud computing service and create a sort of **'superdemo'** with extremely dense physics. This would showcase the technical achievement and open the door to interest in more mundane applications such as the mentioned client opportunities.

If the design also supports scaling to a large number of clients, one could modify the previous showcase, trading a bit of world scale for bandwidth to take the number of active and/or passive clients to the limit. This would create another flavor of 'superdemo'.

## I.10 Introduction References


1. Microsoft "loves Linux" as it makes Azure bigger, better
2. New High Performance Capabilities for Windows Azure
3. Let's talk about the Xbox Live Cloud
4. Windows Azure Storage - Speed and Scale in the Cloud
5. Azure Cloud Services Pricing
6. A Guided Tour through Data-center Networking
7. High Performance Datacenter Networks Architectures, Algorithms, and Opportunities
8. MPI Latency on Google Compute Engine
9. Azure Network Latency & SQL Server Optimization
10. What's Killing Cloud Interconnect Performance
11. Myri-10G 10-Gigabit Ethernet Performance
12. Comparing Windows Azure VM Performance, Part 2

13. Mellanox Delivers the World's Fastest EDR 100Gb/s InfiniBand Switch With Latency Less Than 90 Nanoseconds
14. Latency in the Data Center Matters
15. Microsoft Azure Speed Test
16. Data-Center Networking: What's Next Beyond 10 Gigabit Ethernet?
17. Intel QuickPath Interconnect
18. Performance Guide for HPC Applications on iDataPlex dx360 M4 systems
19. Crytek: Streaming games service viable in 2013
20. NVIDIA GeForce GRID—A Glimpse at the Future of Gaming
21. Virtual GPU Technology

# II The TCP/IP Protocol

We provide a very short overview of the TCP/IP protocol. Knowing at least the key aspects presented below if important, for the following reasons:

1. The choice of protocol and variant have decisive influence on the design.
2. Bandwidth and latency calculation cannot be done without such understanding.
3. Real life testing and performance tuning.

TCP/IP encompasses two protocol classes: TCP and UDP. These are fundamentally serving different purposes, with TCP aiming for reliability, congestion control and UDP aiming for speed at the cost of having no means of controlling neither reliability nor traffic.

## II.1 TCP

Bellow, we summarize the mechanisms that help TCP accomplish such goals:

- **Slow start**: a sender mechanism used to control transmission rate by monitoring the return rate of receiver acknowledgements. Two different windows are used behind the curtains to limit the output flow: ***congestion window*** and ***advertised window***. The congestion window is controlled by the sender (i.e. perceived network congestion), while the advertised window is imposed by the receiver as to estimate the amount of data it estimates it can handle (i.e. available buffer space at the receiver). When the sender transmits, it does so within the minimum of these two windows, which is called the ***transmission window***. After the ACK for the first sent segment is received, the congestion window size is incremented, thus two segments could be sent over the network. If these two segments are also acknowledged, the congestion window is expanded to four. This style is akin to an exponential growth, but it's slightly inaccurate since the receiver might send only one ACK for every two received segments.

- **Congestion avoidance**: a way to cope with lost packages. Congestion can occur whenever a faster network pumps out messages into a slower one but also when multiple when a router is overwhelmed with streams of data. Thus, congestion occurs not only at the border of a network, but also inside it. The sender is made aware of such a situation through the use of retransmission timers or by receiving duplicate ACKs. The immediate response to a potential bottleneck is the sender halving its transmission window size. Two situations are possible:

- Timeout: the window is reset to a size of segment (back to slow start mode)

- Duplicate ACKs: two algorithms are used - ***Fast Retransmit*** and ***Fast Recovery***.

  - ***Fast retransmit***. Duplicate ACKs might happen because of out of order reception. If more than two duplicate ACKs are received by the sender, this can indicate that a segment was lost. If more than three ACKs are received, the sender does not wait for a retransmission timer and it immediately resends the data.
  - ***Fast recovery***. This algorithm allows for higher throughput for large windows under mild congestion. The details will not be provided here.

Several variations on these mechanism further determine what is here termed as ***TCP flavors***:

- ***TCP Tahoe*** this is the first widely used implementation which implements slow start, congestion avoidance and fast retransmit. The basic trait of this flavor is the fast retransmit: does not wait for a timer to expire if several duplicate ACKs are received.
- ***TCP New Reno*** has become the standard TCP implementation. It is predated by the Reno implementation. This variation employs ***partial acknowledgements*** to further optimize the congestion handling. We also do not provide its technical details.
- ***TCP Vegas*** is said to achieve between 37 and 71% higher throughput on the Internet with 1/5 to 1/2 the losses of the Reno implementation. TCP Vegas measures the RTT (round trip time) for every sent segment and computes individual timeout periods. Upon receiving a duplicate ACK, if the timeout has expired, the segment is resent. If other, non-duplicate but close ACKs are received, an unacknowledged segment gets resent if its timer expires.

A lot of research effort has been invested into the analysis and betterment of the TCP implementation and we recommend consulting the available Internet resources on this topic. The de-facto tool for comapring TCP implementations seems to be ns-3: https://www.nsnam.org/.

**REMARKS**: On Linux, TCP is more configurable. See https://drupal.star.bnl.gov/STAR/blog-entry/jeromel/2009/feb/18/tcp-parameters-linux-kernel for a list of parameters and their meaning. On Windows, the socket options are perhaps the only way of changing behavior: https://msdn.microsoft.com/en-us/library/windows/desktop/ms738596(v=vs.85).aspx .

## II.2 UDP

UDP is basically just a wrapper around the IP protocol, adding a pair of ports for transport identification and a checksum field for potential error detection. Given the many drawbacks and the one key advantage, i.e. **speed**, that UDP has, two situations in favor of UDP are identifiable:

- when performance is more important than completeness (e.g. video streaming where lost packages have no significant effect)
- when the transmitted information is very short (e.g. short request/reply exchanges which can be typically carried in a single IP datagram, hence no need to worry about data arriving in the wrong order. Also, the application layer can handle losses by using timers and resend messages when no ACKs arrive).

Another advantage of UDP is that it supports **multicast**.

Application-layer implementations over UDP that offer some of the benefits of TCP should consider these factors:

- congestion control: this requires embedding some kind of a numbering mechanism and tag individual packets with a sequence number in order to have a receiver notice that it was supposed to either assemble more packets than it successfully received or such that it notices that they are not received in order. This way, the sender could reduce the flux at its end.
- reliability: if the sender needs to make sure that all packages of a message are eventually received, some sort of an acknowledgement needs to be sent by the receiver. To maximize throughput, pipelining is used. This means that a window of messages is used to send messages without waiting for individual ACKs. This way the sender does not have to wait for each packet's ACK, but it can continue to send while it receives ACKs for presumably other packages sent in the near past.
- integrity: data corruption can occur due to a number of reasons. Having a CRC-like mechanism helps identify invalid packets. This mechanism should be coupled with the reliability one and made to work on top of ACKs.

It is recommendable NOT to attempt and mix this application-layer logic with that used by the distributed simulation itself. More clearly, whenever a certain type of message between two known entity types is sent, an underlying network interface software entity with no simulation responsibilities should address either of the previously mentioned issues.

As a reference implementation to an in-house UDP-based protocol, we should consider UDT (UDP-based Data Transfer protocol http://udt.sourceforge.net/) which seems to have become the norm in HP computing scenarios. It is worth nothing that probably many reliable UDP modifications already exist. The following is just a short list of a few of them:

- reliable UDP http://sourceforge.net/projects/rudp/
- ENET http://enet.bespin.org/ (which could even replace TCP message handling if found to perform better)

As a final remark, a versatile network implementation could and should mix UDP and TCP for delivering different types of data, depending on the requirements on the delivery and traffic control.

## II.3 The State of Data in Transit

In order to add support for traffic and reliability support, a networking application API needs to distinguish between four category states that characterize the data stream it has to send.

- ***data sent and acknowledged***: category #1
- ***data sent but not acknowledged***: category #2
- ***data not yet sent for which recipient is ready***: category #3
- ***data not yet sent for which the recipient is not ready***: category #4

## II.4 Measuring Performance

The purpose of ascertaining the performance of different network software and hardware set-ups is to identify the most promising strategies to adopt in the higher-level design of the actual application. As previously stated, the two transportation strategies based on TCP and UDP behave in fundamentally different ways. Nevertheless, a quantitative understanding of these differences is required prior to simply using a library to deliver data.

In this sense, starting with a pure TCP vs UDP set of comparisons is the launchpad for more elaborated tests. TCP is the result of decades of dedicated research, so immediately tuning it or either trying to reinvent it might lead to wasted time. The default TCP parameters are tuned towards a fair sharing of bandwidth instead towards maximizing the throughput. Literature suggests that the more immediate tuning strategy for TCP is to adjust the size of its buffers.

## II.5 Methodology

The metrics that should be considered are:

- **mean** and **instant throughput**

- **mean**, **maximum** and **minimum end-to-end delay** For such metrics, existing tools can provide a fast solution. For example: iperf (https://iperf.fr/), D-ITG(http://traffic.comics.unina.it/software/ITG/). A dated but still useful document on tuning TCP can be found at http://dst.lbl.gov/publications/usenix-login.pdf .

Suggested reading:

- http://bkarak.wizhut.com/www/lectures/networks-07/TCP-Tuning-Tutorial.pdf talks about the terminology, parameters and tools involved in performance measurements, providing a quick practical guide.
- http://www-spires.slac.stanford.edu/cgi-wrap/getdoc/slac-pub-10996.pdf although dated, describes the methodology used in comparing TCP to UDT (the UDP based transport protocol)
- https://blogs.oracle.com/mandalika/entry/measuring_network_bandwidth_using_iperf offers a short example on how iperf could be used
- http://www.techrepublic.com/blog/linux-and-open-source/iperf-a-simple-but-powerful-tool-for-troubleshooting-networks/ offers a case study on how to compare UDP and TCP using iperf

### II.6 Caveats

TCP is a reliable protocol, but this is an unfortunate formulation since it is understood as a guarantee for data delivery. Applications themselves cannot rely on TCP solely for higher level guarantees regarding the delivery of their messages. That means it is required to have an extra mechanism that acknowledges correct message reception. It is not only recommendable but necessary to account for the following situations:

1. permanent or temporary network outages
2. the failure of a remote application to which data is being sent to
3. the failure of the hardware host of that application The failure of a network segment can be detected, but it takes a considerable amount of time, which might not be acceptable. The failure of an application is not as detrimental as that of a machine, since the machine can send reset or finish messages to the connected peer announcing that the connection was ended. In this case, a timeout can close the connection at the peer or, if the machine is reset, it will signal a connection reset since it no longer is aware of it having been created.

## III Serialization (WIP)

[TODO] Should be minimalistic, fast, extremely lightweight

## III.1 Desired serialization functionality

- message `m` gets serialized into a preallocated, pool-based buffer `b` of known size = `m.serializedSize`
- every serialized message comes with a minimalistic header which helps identify its type
- after `b` is sent over the network, that memory should become available for other serialization calls

The simulation relies on a relatively small set of predefined messages to be seamlessly carried over the network. In essence, each of the messages will be serialized inplace, without dynamically allocating or deallocating memory. For this to work, each message could use a pool of raw memory which should be threadsafe or even lock-free.

## III.2 Message categories

There are two main message categories:

- The ***command message***, which is simply a lightweight template for describing a predefined command that a ***Sender*** entity requires a ***Receiver*** entity to execute;
- A ***data message***, which can be a much more massive structure containing not only the ***Sender*** and ***Receiver*** IDs, but also array members that hold the actual information which needs to be transferred.

The **command message**:

```
CommandMessage
{
    SenderID;   // the sender GUID
    ReceiverID; // the receiver GUID
CommandID; // number that encodes a conceptual command such as an ACK or a REQUEST
}
```

The **data message**:

```
DataMessage
 {
     SenderID;   // the sender GUID
     ReceiverID; // the receiver GUID
     Array<T1>;  // an array of T1 objects
     ...
     Array<Tk>;  // array of Tk objects
 }
```

### III.2.1 Deserialization and Message Handling

When a buffer `b` is received, its header is used to instantiate a message with the correct type. For ***command messages***, a dedicated pool can be an efficient solution since they always have the same size and structure. ***Data messages***, on the other hand, require array pools for each of their `Array` members. All object or array pools need to be thread safe and, if possible, lock-free.

Once a message `m` is assembled, the network entity that produced this instance needs to forward it to a set of ***handlers*** that will consume the information. This means that a network entity acts as a dispatcher for a list of observer entities that subscribe themselves to this entity.

# IV Compression

Since bandwidth is so precious, compression plays a pivotal role in networked simulation. The number of compression techniques is large to begin with, and some of them can be mixed and matched. This makes it difficult to provide a comprehensive analysis, or to determine a best of all.

We content ourselves to a listing of techniques, and recommendations for first implementations. In a full blown product, a library of compression techniques should be available for testing, comparison and tuning.

When comparing the techniques for the purpose of newtorked simulation, we shall use the following terms:

1. Reduction (***red***): The maximum rate of compression possible.
2. Variance (***var***): The amount of possible variation in compression. When the variance is not negligible, ***red*** must be taken with a grain of salt, because bandwidth spikes can occur, and those can be deal breakers depending on the application.
3. Computation (***comp***): The computational cost. When the computational cost is not negligible, we enter the situation where we are trading bandwidth with processor cycles, both on the server side if there is more than one, and at the client side. For some applications, this must be taken into consideration.
4. Fidelity (***fid***): The resulting fidelity of which we distinguish two types.
    1. Visual Fidelity. This is mostly relevant on the client side and not necessarily easy to quantify as we shall explain in the visual fidelity section.
    2. Simulation Fidelity. This is relevant both for client interaction and for the application if deferred querying or simulation are used.

To ease analysis, we classify the techniques into two large classes: static and dynamic. We call static the techniques that do not require any temporal state, and dynamic those that do. All the static techniques we mention are semantically agnostic, in the sense that they can be applied to any kind of data. This makes them ideal for a two-stage compression pipeline, where they are applied to the output of a dynamic technique for additional compression. Despite scoring low on reduction, static techniques tend to score higher in all other metrics, sometimes much higher. Because of that, there could be applications in which only a static technique is used. Additionally, they are usually very easy to implement, and are hence ideal for reference implementations.

The static techniques of truncation, quantization and Huffman Coding are too common, so we forgo describing their implementations, and list a rough table of their metrics:

| Technique | ***red*** | ***var*** | ***fid*** | ***cmp*** |
|---|---|---|---|---|
| Slight Truncation | low | none | high | low |
| Truncation | mid | none | low | low |
| Quantisation | low | none | low | low |
| Adaptive Quantisation | low | none | mid | mid |
| Huffman Coding | low | low | high | mid |

## IV.1 Dynamic Compression Techniques

## IV.2 Linear Prediction

[TODO] What Oliver's compressor-decompressor implements is actually a basic version of linear prediction.

## IV.3 Adaptive Techniques

[TODO]

## IV.4 Visual Fidelity

The Human visual motion perception system is a complex one. Very sensitive to some types of motions and easily tricked by large perturbations for other types. It has very specialized detectors, honed by natural evolution. Any compression scheme that aims beyond very conservative limits (a necessity for competitiveness) must take the details of this system and its idiosyncrasies into account. As a result, quantitative calculations must be at the center of decision making about the choice of compression technique or the parameters thereof. Quantitative calculations can capture a large number of cases all at once. They can be turned into question answering formulas for questions like: What is the best value for a certain parameter to achieve a certain fidelity. These formulas can

be used both at design and at runtime. For validation, carefully crafted experiments can be used to implement some of the cases.

### IV.4.1 Truncation and Position Change Perception

As any developer of numerical simulations knows, one can never have enough bits for calculations. When it comes to transmitting information for the sole purpose of display, and not for further calculation, one might be for once having more bits than necessary. Truncation is the most direct way of fixing this and saving bandwidth.

Given a distance to viewer ***d***, a field of view ***f***, and a horizontal screen resolution ***r***, the formula determining the amount of meters per pixel ***ppm*** is

$$ppm(f, r, d) = \frac{2.d.tan(f/2)}{r}.$$

From this, The number of mantissa bits in position change (assuming an exponent of one ($2^0$), the object observed being near the origin) that correspond to an unnoticeable difference of a quarter of a screen pixel is

$$qbits(f, r, d) = -ln_2(\frac{d.tan(f/2)}{2.r}).$$

Finally, the formula for the viewer distance needed to require a number of mantissa bits ***n*** is

$$ndist(f, r, n) = \frac{2.r.(2^{-n})}{tan(f/2)}.$$

Using these formulas we can calculate the following:

$$qbits(\pi/2, 1280, 4) \approx 9 \text{ bits},$$

and

$$ndist(\pi/2, 1280, 23) \approx 0.003 \text{ mm}.$$

Clearly, in this setup, the 23 bits of mantissa as available in the ***c*** language's ***float*** data type are never all needed when it comes to position display. In the less accommodating general case, the maximum distance from the origin must be taken into account when deciding how many bits to truncate. This is so because one additional bit is needed for every additional power of two displacement from the origin. Hence, assuming a sector diameter of a kilometer, up to $ln_2(1000/2) \approx 9$ bits, might additionally have to be kept, leaving a meagre number of $23 - 9 - 9 = 5$ bits for truncation.[6]

[6] ***Cells*** as described in Arch1, can be used to alleviate this problem, recycling the processing cycles that went into grouping to the advantage of bandwidth saving.

#### IV.4.2 Group Perception

[TODO] Discuss the shortcuts that can be taken for dense groups to save bandwidth.

# V Arch1, a Proposed Architecture for a Networked *hknpPhysics*

[TODO] Describe the functioning prototype that was created before the GDC. [TODO] Minimalistic, decentralized, desynchronized, core-balancing, border-hiding.

## V.1 The Simulation's Network Entities

- **Sector** - this is a networked unit of simulation which corresponds to a Physics world instance. Besides holding this instance, the sector has all the necessary network details in place for it to establish connections with other sectors or network entities that might be interested in querying its internal Physics world. The sector neighbourhood does not refer to the network topology or hardware design, but to the fact that its internal world might share a geometric boundary with another one from a different sector. The sector itself, being a single unit, could reside on its own dedicated machine, but, where hardware allows it, several sectors can share the same local hardware. Even if certain sectors are deemed as geometrical neighbours, it can very well happen that they are at a more considerable distance from the network's point of view.
- **Viewer** - in its simplest form, the viewer entity is concerned with the gathering the most basic information needed to render objects. This means that it could connect to another entity which can supply information regarding object positions, shapes or dynamical states. As detailed below, to deal with the problem of many-to-many in terms of connection channels that viewers and other entities could require, a mediator entity that aggregates relevant, commond data needs to be employed.
- **Coordinator** - in a scenario where a traffic regulator entity needs to individually exist and serve such purposes, a coordinator entity can serve this exact purpose. It can also hold meta-information regarding the body ownership and characteristics, should these details be required by another network entity such as a sector, viewer, etc.
- **Proxy** - this entity acts as an aggregator for geometric and dynamic body attributes and can propagate these further to more passive entities (such as the viewers). The reason behind this is to decouple the sector logic from the one concerning render and representation activities. In a nutshell, the proxy represents a network interface between some consumers (e.g. viewers) and the actual active simulation entities (the sectors). The design should support as many viewers as necessary so that a communication balance is achieved.

## V.2 The Simluation and the Application

[TODO] discuss the interaction between the two, also in terms of synchronization needs.

## V.3 Synchronisation

The architecture provides a number of network synchronization schemes, but is designed to work perfectly well with even the loosest of them. The leitmotif being the avoidance of networked 'conversations' that require strict response timings of any sort, or that break down irrecoverably if a message is lost. The price that we deem acceptable to pay for this looseness in communication is physical artefacts. All schemes should be implemented and coexist as simulation options in a full blown product. As we shall see, it is even expected that at times, some entities will be synchronised using one scheme and other with a different one, all within the scope of one simulation. The schemes are designed as more or less subsets of each other. The rationale is that the implementation should tackle the simplest first, with the highest probability of artefacts but already showing the maximum attainable simulation performance. In order, the more evolved schemes should be added, and not much if anything at all is lost in implementation effort because of the mentioned subset nature of the scheme relations. This is not only a technical advantage, but also a business one since it allows the timely creation of prototypes that show the full expected performance and scaling.

### V.3.1 Unified Frame Rate (*ufr*)

This is the loosest and simplest scheme. All sectors have the same target frame rate and hence frame time (***dt***). When a sector finishes its computation before ***dt*** elapses, it waits or busy spins. Otherwise, it immediately starts processing the next frame. An architecture that can successfully work under such a loose synchronization can trivially work under stricter ones. In the optimistic case of load balancing working properly, and all sectors keeping their frame rate targets, the amount of desynchronization between any two of them is two frames at most in the short term. The worst case happens between sectors that happen to start their frames around the time other sectors end them. In the long term, the number of total frames each sector will have simulated after a certain amount of global has elapsed time will be arbitrarily divergent.

### V.3.2 Frame Locked

In this scheme, additionally to a unified frame rate, sectors join each other into a synchronisation point at the end of their frames. This keeps all sectors running at the same global time, and constrains the global simulation to run at the pace of the slowest and most loaded sector.

**V.3.2.1 Instability of Frame Locking** This attractiveness of this scheme is its conceptual closeness to non-networked simulation. However, the hard reality of simulation and of networking tells another story. Running at the pace of the slowest sector also means running at the mercy of any performance spikes, and any networking hiccups, the occurrence of which is probably higher if the simulation is not run on dedicated servers, but in a load-balanced cloud. As the ambitions for scaling increase, and the number of sectors with it, the chances of such problems increases. Additionally, as is very well known from multi-threading experience, as soon as such a strict waiting design is implemented, the task becomes optimizing it by filling the idle waiting time, which is exactly what the looser scheme already does to some extent. Alternatively, one might think of 'timing-out' while waiting for troubled sectors, but this brings us once more into a hybrid of frame locked and ***ufr*** locked schemes that we predict to be too complicated, and eventually boiling down to pure ***ufr***.

Furthermore, no matter what scheme is implemented, viewers have to be designed to able to extrapolate motion no matter what.

[TODO] discuss the effect and way of handling in different applications and scenarios, because it depends.

### V.3.3 Sub-Frame Locked

[TODO] Lock at multiple stages within a single frame. [TODO] Discuss Infiniband and possibilities in a high performance computing data center.

## V.4 Network Simulation Build

[TODO] Discuss a special network simulation build, where some structures needed for networking are built-in into the body, motion, broadphase, etc. Also discuss optimised add body, remove body, set motion path.

## V.5 Adding bodies

## V.6 Body IDs

In a universe of coupled simulation worlds, each body must have a unique ID that is not specific to only one world, but to all of them. As detailed in a later section, a sector is the simulation entity which wraps a Physics world. It is natural to incorporate in the global body ID a mark of the sector where that body was created. Almost irrespective to the chosen simulation scheme, a body will be owned by the sector in which it was created, even though that body might end up in other sectors. Thus, a global body id or ***network identifier*** (short `nid`) is a bitwise join between a sector id (`sid`) and a local ID (denoted as `bser`). This means that `nid = [sid,bser]`, where `sid` occupies the leftmost bits of `nid` and `bser` the remaining.

## V.7 Networking Considerations

### V.7.1 Sending Messages

[TODO] Zero packet loss within the data center, but not outside of it. [TODO] Zero packet loss does not mean timely reception when ack is not used. [TODO] spin wait, linux and windows [TODO] Use the TCP/IP overview to discuss where in the network what should be used, possibly using different channels even between the same two entities.

## V.8 Boot Configuration

[TODO] describe the bootin configuration, which is shared between all network entities. This could be a pasive file or a shared library, maybe also including application code.

## V.9 Body Network Identifiers

### V.9.1 Creation

Each body possesses a network identifier (***nid***) that is unique in its universe. The sector that creates a body assigns its ***nid***, using a scheme that is simple and decentralized. A local, serial, hence strictly monotonous number (***bser***) is incremented at each local body creation event. The size of ***bser***'s type should be less than that of ***nid***'s. The value of ***bser*** before the creation of a body is used to fill the rightmost bits of ***nid***, while the sector's index is inserted in the remaining leftmost bits.

### V.9.2 Life and Death

A body keeps its ***nid*** until its death viz. removal from the universe. This holds even when the is transfered to another sector. The meaning and rules of transfer are explained elsewhere in this document. The death of a body, an event that is arguably less frequent than the creation of one, maybe even very rare, triggers a communication between the body's current sector and its origin sector if they differ. The communication signals that the body's original ***bser*** can be recycled for other bodies, using a free list or some alternative. This communication is not urgent, since delaying it only keeps ***bser*** locked.

### V.9.3 Body Count

The size of ***nid***'s type should be less of equal to the size of ***hknpBody::userData***'s type, so that it can be held in it. Currently, that type of is … in all builds configurations. Assuming a maximum simulation size of … sectors, the maximum number of bodies simultaneously existing in the universe that originated (but not necessarily currently belonging) from a single sector is … . The maximum number of bodies in the simulation is the maximum number that the type of ***nid*** can attain. This means that only after … bodies have been

created in a sector, that ***bser*** recycling and management becomes necessary. If it is known by the application that no more bodies than that are ever created in any sector, the simulation can be configured to turn ***bser*** recycling off, eschewing the need for death signaling, which is light and lazy to begin with.

## V.10 Sector Body Control

By body control we mean the control of the physical behaviour of a body. That is, changing its position every frame, either directly, or by stepping of velocities or accelerations, and changing those by the application of impulses or equivalent.

The schemes presented can be roughly split into exclusive and cooperative. In exclusive schemes, only the owning sector of a body controls it. In cooperative schemes, this is not necessarily so. Naturally, cooperative schemes are more complicated than exclusive ones, but result in potentially better fidelity for sector border straddling bodies (***straddlers***).

### V.10.1 Scheme Commonalities

The schemes use sector and body volumes for decision making. In the simplest case, sector volumes are neighbouring but non-intersecting axis-aligned bounding boxes, fixed by the boot configuration. Body volumes are the axis-aligned bounding boxes of the world's broad-phase.

It is assumed that communication is only necessary with immediate neighbours. This means at most eight sectors in a two dimensional configuration, and twenty-six in a three dimensional one. Bodies that are larger than a sector are not supported in this architecture.

Most schemes will allow for leeway in how long to wait before sending a message of a certain kind, and if applicable, how long to wait for a reply. The word used for this is ***urgency***. The architecture is designed such no disaster happens, no matter how lazily this urgency is interpreted in the implementation. Usually, the result will be a larger frequency of artefacts. In the same spirit, the choice of when to execute the sending and receiving of messages in relation to the physics steps is left unspecified.

### V.10.2 Scheme 1

In this exclusive scheme, body control belongs to the owning sector, and ownership is determined by the following exhaustive set of rules.

1. The body has just been created. The creating sector is the owner.
2. The body's volume has an intersection with its owner's volume. Nothing changes.
3. The body's volume has no intersection with its owner's volume but intersections with other volumes. Of all intersecting sectors, the one with least index, (or alternatively largest intersection), becomes the new ownership candidate. A non-urgent

message is sent to the candidate sector. If sending succeeds, the body is removed from the current sector's world, otherwise, it is kept until the next frame.

Clearly, this scheme breaks down for constrained bodies, and exhibits artefacts for ***straddlers***. Additionally, bodies might get lost, disappearing from the simulation, if the sent message is not received. Nevertheless it is a good starting scheme that should be kept as a reference implementation because it is the lightest possible.

### V.10.3 Scheme 1.b

In this scheme, bodies never get lost. A transferring body is only removed from the world once a (lazy) acknowledgment is received back. In the meantime, Proxies will receive updates for the same body from multiple sectors. The way they handle this is described in a later section.

### V.10.4 Scheme 2

As in scheme 1, control is exclusive. However, the basic unit of control is a body group and not an individual body. We will first discuss the scheme's body grouping strategy, then list the schemes rules.

Bodies can be grouped in multiple ways, because grouping does not have to be exact. We mean exact in the sense that for an artefact free result, two bodies that are constrained together by a joint or a contact, should end up in the same group, while at the same time, there is no harm done for two independent bodies ending up together. To fully exploit the information already provided by the physics engine, and minimize group management overhead, two kinds of groups are to be implemented: ***linkage*** and ***cell***.

**V.10.4.1 Linkages** A group of ***linkage*** kind is an arbitrary but long lived list of bodies grouped together. It should be used for all scenarios for which it is known that a set of bodies should always be simulated together. Good candidate of ***linkages*** are:

1. Mechanical systems, such as ragdolls.
2. Groups containing multiple individual bodies and/or mechanical systems, turned into a ***linkage*** by application specific logic, exploiting meta information not known to the physics simulation.
3. The physic's engine deactivated islands. These provide multiple opportunities in the case in which they need to be transferred to another sector: Their velocities are known to be zero, their positions are known to be close-by, it is quite probable that they will deactivate together again.

In this scheme, each physics body must maintain its ***linkage*** id. If space allows, this can be inserted into the bits of its ***knpBody::userData***. If not, ***knpBody::userData*** must be used as a pointer to a dynamically allocated structure containing all that the needed information.

**V.10.4.2 Cells** ***Linkages*** are a good solution for long lived groups. This leaves out the situation of independent bodies, interacting through contacts. To handle such bodies, each sector is statically and non-hierarchically subdivided into either a two or three dimensional grid, spanning the sector's volume.

The granularity of the subdivision is dictated by the configuration. Finer subdivision incurs no performance penalty because the grid is not hierarchical, and hence the main operation on the grid viz. determining grid occupancy, is a cheap O(1) calculation. Subdivisions that are too fine, such that they cannot contain a fair amount of bodies, let us say 32 or more, are also counter-productive, since they defeat the purpose of grouping multiple bodies for exclusive control. Each subdivision, including the bodies that occupy it, is called a ***cell***.

The grid occupancy of a volume (that of a body or a group) is the determination of a list of ***cells*** that the volume intersects. For a convex volumes, the occupancy is not an arbitrary list of ***cells***, but a rectangle or cuboid of them. Such an occupancy can therefore be perfectly represented by a minimum and maximum tuple, with no need for a list of dynamic size. For this reason, we only admit convex volumes for ***linkages***, even if they lead to large over-estimation.

**V.10.4.3 Joins** A ***linkage***, controlled by a sector, has its occupancy continuously updated by that sector. All ***cells*** that are touched by the occupancy enter a ***join***, which is a list of ***cells***. A sector can efficiently check the occupancy of a ***linkage*** into any neighbour's grid because the grids have the same simple structure in all sectors. A ***join*** has a set of sectors associated with it, the list of sectors that control its ***cells***.

When this list does not consist of only the sector itself, The whole ***join*** should preferably be simulated exclusively, to minimize the chance of artefacts in the behaviour of its ***linkage***. This list is processed with the following

**V.10.4.4 Control Ownership** The control ownership for groups is subject to the following rules:

At the beginning of the simulation, each sector controls all the ***cells*** of its ***grid***. Additionally, whenever a sector creates a ***linkage***, it owns it for that frame. As the simulation progresses, this ownership of these groups continuously renegotiated, using the following rules:

1. At the beginning of the simulation, all ***cells*** are controlled by their original sector.
2. At creation, a ***linkage*** is controlled by its creating sector.
3. At the update of a ***linkage's join***, the ***join's*** sector list is found to contain other sectors than the ***linkage's*** sector. A message is sent to all these sectors, asking them to transfer any of their ***joins*** that contain any ***cell*** in the list to the sector of least index.

Note that a sector could receive conflicting ***cell*** transfer signals. In this case, the sector can simply choose one and ignore the rest. Any missed joins will be necessarily detected in the next frame by some sector, and renegotiated.

**V.10.4.5 Linkage Volumes** The volume of a linkage can simply be an axis-aligned bounding box. However, this might lead to large over-estimations, creating ***joins*** that are unnecessarily large. The implementation should provide the choice, per ***linkage***, of additional volumes structures, such as oriented bounding boxes, sphere-lists, etc., trading memory and/or processor cycles for bandwidth.

**V.10.4.6 Bandwidth Calculations** To calculate a very rough number for the number of transfers possible per second, we consider an average transfer made up of the following: 1. A cell of 32 bodies. 2. A body transfer of full position and velocity. 3. A linkage of 16 bodies, subject to 16 constraints. 4. A constraint transfer requiring 16 floats.

We have a total transfer size of 2688 bytes. For a 10 Gbps socket bandwidth between any two sectors, this amounts to 465,000 transfers per second, and 8,000 transfers per frame at 60 fps.

### V.10.5 Scheme 2.b

As a refinement on Scheme 2, the full and unconditional control transfer of rule 3 is optimized. Instead of simply using the sector of least index, a local utility function can use the current load of each sector, and the number of ***cells*** it controls, among other metrics, to choose a better candidate. The latter metric is important because, even if a sector with a high number of ***cells*** has a low load, it has the potential of moving to a high load very fast. The utility function could also use hard rules, not permitting sectors to control ***cells*** that are in the center of another sector's grid. Additionally, it could also force the splitting of ***joins***, at the risk of more artefacts, to keep processing loads manageable. The utility function could also be supplied with 'oracle' knowledge coming from the application. All the metrics that such a utility function would use, would be available locally within each sector, updated using lazy signaling.

Load balancing by moving sectors between machines is not considered in this architecture for three reasons.

1. The high costs of full world transfers.
2. The long delays introduced by the need for disconnecting and reconnecting communication channels.
3. The added complexity.

On the other hand, processor load balancing plays well with this scheme. Assuming sectors are assigned to cores using affinity or some some other scheme. When the load is too high in a sector, it can query its machine neighbours - which are not necessarily the same as its universe neighbours - and ask to either share or fully take over a machine core.

### V.10.6 Cooperative Variants

A variant of any of the above schemes can be obtained by marking bodies (or groups where it applies) as cooperative rather than exclusive. Cooperative control mirrors the multi-threaded solving technique used in physics. The cooperative entity is simulated in all cooperating sectors, and the resulting final impulse of each sector (linear and rotational) is signaled by it to the others. For a cooperative entity, it is not necessary to negotiate which sector should send proxy updates, for this is both cumbersome and unnecessary. Proxies are designed to handle multiple updates for the same entity. There are at least three possible sub-schemes for cooperative control.

**V.10.6.1 Free** Impulse signaling is simply done asynchronously, and impulses are applied at arrival of their messages. This can be enough especially in the cases where a linkage is interacting with cells, but not with other linkages.

**V.10.6.2 Locked** The cooperating sectors enter a frame synchronisation, and impulses are exchanged at the end of each frame. When sectors have similar processing times, the synchronisation overhead can be negligible.

**V.10.6.3 Super-Locked** This sub-scheme mirrors most closely the multi-threaded solving technique. At every ***ith*** solving ***substep***, the impulses are synchronised. With the most demanding setting of $i$ being equal to one the full quality of a non-networked simulation is attained. Clearly, this scheme requires synchronisation that is quite intrusive. Nevertheless, it must be implemented, at lease because the application might want to demand such quality for specific bodies. In the data center where network latency can be in the other of a few microseconds, it might even be possible without too much overhead.

## V.11 Load Awareness

[TODO] Bandwidth consumption and load lazily monitored, broadcasted, decisions are based on that, proactively, before congestion happens, at least congestion one can predict.

## V.12 Sectors and Proxies

## V.13 Body Creation and Death

Body creation and death must not be lost because this introduces quite noticeable artefacts. Nevertheless, a scheme that is both light and robust is possible. Once a message of this type is sent, it is assumed it was received. This is safe because body update messages will follow, and is handled with simple rules on the side of the update receiver's side. 1. A body update is received for a body that does not exists. A message is sent asking to resend the creation signal. 2. No body update has been received for a certain specific

time, not even the tiny empty update message. A message is sent checking for either a body death or an empty update message.

## V.14 Body Update

### V.14.1 Scheme 1

In this scheme, a sector sends a body update containing its position and orientation quaternion, at the end of each physics frame. Being the simplest, this scheme must be implemented for reference.

### V.14.2 Update Type

### V.14.3 Multiple Updates

A proxy has multiple options when it comes to handling multiples updates to the same body, during the same frame (however interpreted) from different sectors. The simplest and not necessarily bad option, is to simply take the first and discard the rest. For more evolved options, the decision must depend on whether the updates are homogeneous (of the same sort), or not. The list below is a non-exhaustive set of possible rules.

1. Homogeneous of non-incremental sort.
    a. Apply all the updates.
    b. Sequentially blend the updates and apply the result.
2. Homogeneous of incremental sort. Apply all of them, which has a similar effect to temporal block solving of constraints.

### V.14.4 Bandwidth Calculations

[TODO] Table, Scheme 1: SD:400 updates, Eth10 14k updates

## V.15 Viewer Control

### V.15.1 Channels

[TODO] Data1, Data2, Control, Immediate (thread-blocking, async)

## V.16 Query Mirroring

[TODO] Proxy casting, double buffered Immediate broad-phase. etc.

## V.17 Background I/O

## V.18 Shape Management

## V.19 Architecturally Relevant Tests

### V.19.1 The Domino-verse

[TODO] A domino level that links cells together as the pieces fall, in multiple variants.

### V.19.2 The Pendulo-verse

[TODO] A pendulum level where pendulums resides in different sectors but interact without artefacts.

# VI Appendix

## VI.1 Physics Density

It is not a secret that the use of 'dynamics' is kept to a bare minimum by most clients. To quantify this we use a simple but suggestive metric that we call physics density (***pd***). The formula for ***pd*** of a physics world is simple:

$$pd = (b + c)/a,$$

where ***a*** is the world area, ***b*** is the number of active dynamic bodies and ***c*** is the number of constraints. If any of these quantities is variable, some meaningful average of it must be used instead. The unit of ***pd*** is then ***'physics' per square meter***.

With the help of our developer relations team, we have compiled a list of client cases from which we calculated that almost all of themstay below a ***pd*** of $10^{-3}$, and can safely be called sparse. In contrast, our internal benchmarks and demos are mostly dense.

## VI.2 Approximate World Cost

To allow for approximate calculations for the computational cost of a world, we divide the bodies into four types.

1. Single body (***sb***). An active dynamic body that mostly interacts by contact with static bodies.
2. Pile body (***pb***). Mostly interacts by contact with other dynamics bodies
3. Constraint body (***cb***). Is constrained to other bodies.

4. Ignorable body. Anything else.

Given the number of such bodies in a world, we then extract a rough average total cost for simulating each type using a single thread. From Havok FX's presentations, we have the following example on a Haswell processor: ***sb***: 5 $\mu$s, ***pb***: 8 $\mu$s. We additionally extracted: ***rb*** : 1 $\mu$s.

[TODO] overhead, example calculation.

## VI.3 Parallel Computing Skills

We put this section in the appendix not because it is not important, but to prevent us from fall into digression. The point of it is that networked simulation is a form of parallel computing and so is GPU computing. Hence, there is an intersection in the skill sets they require. Unfortunately, one could say that parallel computing is not our forte, backed by the following observations:

Firstly, while Havok products have successfully accompanied instruction level parallelism (SIMD) and thread level parallelism they have not moved further despite the fact that parallel computing is the only reliable way for more performance since the frequencies of conventional processors stopped following Moore's law.

Secondly, it used to be the case that Havok's offering was a superset of Nvidia's in terms of simulation but this is no longer the true. Per example, products such as FLEX, HairWorks and PhysX Fluids have no counterpart. This seemingly ignored portfolio gap may be equal in gravity to the internally known fact that the margins of single processor performance superiority of our physics are thinning naturally and unavoidably. We do agree that the aforementioned products serve to boost Nvidia's GPU agenda. We can even, with some effort, refrain from objecting to the claim that they are commercial failures and technical trivialities. However all this is besides the point and does not mean that they should be left unanswered.

At the minimum, closing this gap prevents the engineering team from slipping in terms of parallel computing skill and experience, none of which is trivial to obtain. The parallel computing paradigm has nowadays a much broader set of requirements than single-machine, multi-core applications. This is due to the fact that practical expertise in other fields (medium-scale high performance networking, GPGPU best practices) is also required before one can successfully implement a competitive, market-relevant solution. The action to take is thus to understand and acquire these skills prior to starting development.