Internet Based Client/Server Network Traffic Reduction

Bernt Habermeier   bernt@bolt-action.com   http://www.bolt-action.com

NOTE:
See these concepts in action in our latest release of the Wulfram engine. Look for Shock Force on the bolt action web page: www.bolt-action.com.

Target Audience

Developers who are interested in the reduction of bandwidth requirements for client/server based games.

Abstract

Given today's modem speeds, it's important to keep the amount of information a server sends to game clients as low as possible. This paper explores methods to reduce IP-based modem bandwidth requirements for fast action 3D client-server games. The performance of these methods is analyzed from real traffic profiles of a 3D game that is currently in development.

1 Introduction

Present modem speeds make it a challenge to implement multi-player action games over the Internet, especially when there are many fast moving objects in the game. We have developed several network traffic reduction techniques for client/server architectures, where the server is authoritative and is situated at the middle node of a star network. Furthermore the clients are updated asynchronously with respect to one another, and the client's frame rate is independent from network traffic.

In our model, the server has authority over the entire game state. The clients must request object manipulation, creation and deletion directly from the server, which is responsible for changing object state and maintaining coherency, and for communicating such changes to the clients if and when it deems such communication to be worthwhile. No clients are allowed to broadcast state changes to other clients -- it all happens over the server. The clients are allowed to perform local state predictions and extrapolations. However, if there are any conflicts between a client's state and the state of the server, the server is the one that is correct.

When the number of dynamic objects is quite large, the server cannot be expected to maintain errorless clients. That is not possible and, fortunately, not strictly necessary, because in a 3D game no single player can be aware of all objects at the same time with the same fidelity. Close dynamic objects are much more likely to require frequent and

precise updates than far away objects. It therefore makes sense for the server to keep a priority queue of objects per connected game client. Objects at the top of the priority queue are more important to update than objects lower on the priority list. Each time the server allows itself to send more data to a specific client, it grabs as many priority items from the top of the priority queue as will fit into the allowed bandwidth allocation, and updates the corresponding objects.

In order for the server to maintain such priority queues for every connected game client, it needs to know both the true state of the game, and the state of the game on each client. The server knows the true state of the game because it has authority over the game state. However the server doesn't know the state of the game on each client, which is something it needs to know if it is going to make good judgements about which objects need to be updated on which clients. In reality, it is not necessary for the server to be completely right about its view of any particular game client, as long as it is mostly right about the objects that are important to any one particular player. In order for the server to approximate the relevant game state of each client, the server needs to mimic what each client does with the received information. This complicates things a bit, especially when we implement bandwidth saving measures that happen on the client side.

2 Calculating the Error

In estimating the error of an object on a particular client, we want to acknowledge the fact that what we are really interested in is the perceived error, and not the actual numeric error of the object in question. For example, a small position error may be quite noticeable when an object is close, and not even worth mentioning when that same object is further away. The perceived error, however, is not only a function of distance, but also of object size.
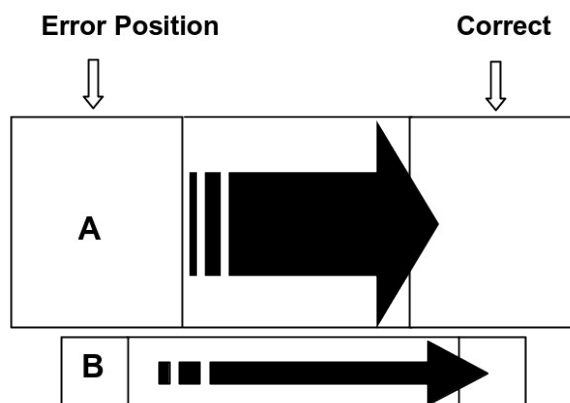

Given two objects in the above figure with similar positional errors at the same distance, it should be clear that object **A**, which is the larger of the two, has a much higher perceived error associated with it than object **B**. Then again, object **B**'s perceived error should be greater than that of an object with a similar numeric error that is not even displayed on the client's monitor.

All such considerations do not require a high degree of accuracy, and we can make many simplifying assumptions that do not unreasonably degrade the estimated perceived error, which we are trying to calculate. Because we are interested in the approximate projected area of an object on the client's screen, we do not use the actual shape, but rather a bounding sphere. The textures of the objects are ignored altogether. Furthermore, we will assume that the object in question is directly in front of the viewpoint, and not off to the side. This assumption implies that we will not calculate the usual planar projectional distortion the object otherwise might have on the client. We make other approximations below, all of which should become apparent as I describe our perceived error estimation techniques.

2.1 Position Error

An error vector defined by the two alternate positions of the object describes the position error. The tail of the vector is the object's current position, whereas the head is where the object should be. In order to calculate the perceived error of an object we compute the object's planar motion pixel error as well as its depth motion pixel error. The former error estimates how much screen area is swept over when the error is corrected, and the latter measures the difference in the projected size of the object before and after the error is taken care of.
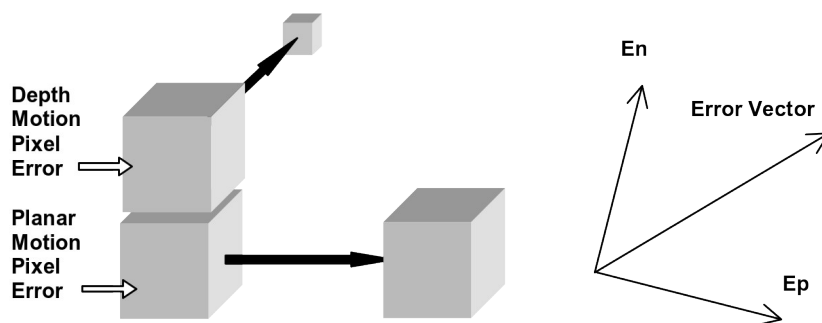
The planar motion pixel error uses the part of the error vector which is parallel to the view plane (E_p), whereas the depth motion pixel error uses the part of the error vector which is perpendicular, or normal (E_n), to the viewing plane.

Because we are interested in the perceived error, we need to find out approximately how large the objects seem to a person playing the 3D game. The figure below shows how this can be done while ignoring planar projectional distortion.
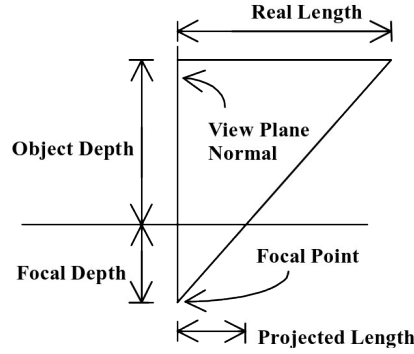
We can solve for $Xprojected$ by using similar triangles, given that we know $DEBTHfocal,$ the focal depth, $DEPTHobject,$ the object depth, and $Xoriginal,$ the length we are trying to project:

$$\frac{Xoriginal}{DEBPTHobject + DEPTHfocal} = \frac{Xprojected}{DEPTHfocal}$$

Therefore, solving for $Xprojected$

$$Xprojected = \frac{Xoriginal \cdot DEPTHfocal}{DEPTHobject + DEBTHfocal}$$

The values for $DEBTHfocal$ and $Xoriginal$ are known in advance. Below, **ObjVec** is the vector pointing from the viewpoint object to the object of interest; **ViewNormal** is the unit vector normal to the view plane of the viewpoint, and ● is the dot product.

$$DEPTHobject = \textbf{ViewNormal} \bullet \textbf{ObjVec}$$

Thus, the projected length of a line segment is known and so is the projected area of a sphere. In order to estimate the projected area of an object, we calculate how large its projected radius length would be, and use that length to compute the area.

2.1.a Planar Motion Error

In estimating this error component we first find the length of Ep and figure out what that length would be when projected onto the view plane from the object's distance, resulting in $Ep_{projected.}$ That length, multiplied by the projected area of the object, gives us the estimated planar motion error:

$$Error_{planar\_motion} = Ep_{projected} \cdot Area_{projected}$$

2.1.b Depth Motion Error

To estimate this error, we find the projected area before the error fix ($Area0_{projected}$), and the projected area after the error fix ($Area1_{projected}$), and take the difference:

$$Error_{debth\_motion} = |Area0_{projected} - Area1_{projected}|$$

In fact, because the projected area does vary when computing the planar motion error, we actually take the average projected area instead of either $Area0_{projected}$ or $Area1_{projected}$. Finally the position error can be calculated as:

$$Area_{ave} = \frac{Area0_{projected} + Area1_{projected}}{2.0}$$

$$Error_{position} = Ep_{projected} \cdot Area_{ave} + |Area0_{projected} - Area1_{projected}|$$

2.2 Orientation Error

To approximate the client's perceived orientation error of a given object, we first calculate the numeric orientation error, and weigh that numeric error by the average projected area of the object.

There probably are faster and more accurate ways to find the approximate numeric orientation error than the method we currently use. But, such as it is, our method works as follows. We define unit x, y and z vectors relative to one of the orientations (x_0, y_0, z_0), and dot product them to the unit x, y and z vectors relative to the other orientation (x_1, y_1, z_1):

The largest orientation error component occurs when any of these dot products reach -1.0. The smallest error occurs when all components are close to 1.0. Finally, we compute the numeric orientation error:

$$Error_{numeric} = \frac{\arccos(\min(Xdot, Ydot, Zdot))}{\Pi}$$

whose range is [0,1]. By inspection, it should be clear that the numeric error increases with the orientation error.

To estimate the perceived orientation error, we simply multiply the numeric error by the average projection area. When the object takes up a large part of the screen, any orientation error will become much more noticeable than when the projected area of the object is small.

$$Error_{percieved} = Area_{ave} \cdot Error_{numeric}$$

Other methods could certainly be used to find the perceived orientation error; however, for our purposes we have found this method to be adequate.

2.3 Linear and Angular Velocity Error

Perceived linear and angular velocity errors are computed in the same spirit as the orientation error. We first calculate a numeric error that is divorced from projection

information, and later multiply by the average projection area ($Area_{ave}$) to regain some sense of the perceived error.

2.4 Non-spatial Errors

Non-spatial errors need to be accounted for just as much as spatial errors. It is difficult, however, to measure the perceived error of non-spatial errors -- clearly the importance of non-spatial data depends heavily on the game itself. Because it is convenient to sum up all errors, spatial or otherwise, one needs to be aware of the numeric range the spatial errors exhibit, and to adjust some numeric multiplier constant of the non-spatial error such that the non-spatial errors do not drown out the spatial errors or vice-versa. Exactly how one is to balance spatial and non-spatial errors, depends on the game and should be thought of as performance tuning of the error estimator. If there are non-spatial components associated with 3D objects, it may make sense to weigh these in proportion to the average projection area, but again, it largely depends on the game specifics.

Now that the server can estimate the perceived error for each object, we could just update the objects with the largest errors. While that would certainly work better than updating all the objects at the same frequency, we can further reduce bandwidth requirements with the methods discussed in the next section.

3 Sending Less Data

There are two complementary fundamental ways to reduce bandwidth requirements:

1. Reduce the frequency of updates needed per object
2. Reduce the size of the object update

An authoritative server has the ability to individualize the traffic stream for each client, and can thereby easily use methods from both of the above categories to reduce bandwidth requirements. Much of the following certainly applies to the viewpoint itself, however there are some complications that need to be addressed, and so we treat the viewpoint in a later section.

3.1 Reduction of object update frequency

By reducing object update frequency, the server can use the bandwidth more effectively. Here are the techniques we have used to that end.

3.1.a Dead Reckoning (Object extrapolation)

The client can predict how objects will move based on previously acquired information about these objects. The rules used for the prediction can vary from object to object, or even be dependent on the object's state.

The most simple dead reckoning method, which works quite well for our game, is position-velocity based extrapolation:

$$Position_{guess} = Position_{last} + Velocity_{last} \cdot dt$$

One may think it would be better to include acceleration into the object extrapolation; however, for our objects, acceleration can fluctuate rapidly in ways the client could not easily predict. The server would then be required to send acceleration updates quite often so that the clients could forward extrapolate more precisely. That would defeat the purpose of the entire exercise -- we want to receive less-frequent updates, after all.

As mentioned above, the server needs to mimic what each client does with received object information, so that it can better estimate the individual object error for each client. Therefore, when the server estimates client error, it will try to use the exact same object extrapolation technique used by the client. Currently, we are using only the simple position-velocity based extrapolation. However, there is no reason why in the future we wouldn't add more elaborate extrapolation techniques that are better suited for particular objects.

If your game has goal-oriented directives, where the computer auto-pilots game units, a special form of object extrapolation would be to communicate such higher-level directives to the clients instead of the low-level dynamics-related information. For example, instead of sending frequent updates about an object navigating a winding road, the server could just tell the clients that the object is in fact going to navigate the winding road. The server would send corrections for the object's movement whenever the object were to deviate from the road.

More advanced schemes employing table lookups of frequent maneuvers or perhaps even adaptive learning prediction schemes could potentially be realized.

3.1.b Smooth Error Correction

Given severe bandwidth limitations, relatively large error can accumulate on the client side for some or perhaps most objects. If the client were to correct its state instantaneously for such errors, to the player the rendered objects would appear to jump. An easy way to avoid this behavior is to correct the state over some time-span that is in line with the magnitude of the error.

The rate of error interpolation could be constant, or it could depend on the severity of the error. Although a constant error fix rate may be smoother, it could potentially take a very long time to interpolate over a large error. Thus it's probably better to use a variable error fix rate that depends on the severity of the remaining error. We have not yet experimented with a variable error fix rate, but we will most likely use such a scheme in the future.

3.1.c Selective Object Updates

When a game contains several hundred objects, one certainly would not want to update every object with the same frequency. Rather, one would try to prioritize important objects over less important objects. In indoor maze type games where objects are spread out over many small rooms, such prioritization is quite easy. However, in open-space 3D games where there are large open areas, some care must be taken to prioritize objects reasonably. This method computes the importance of an object, which can then be

combined with the perceived error estimation to obtain the object priority. There are two components that we use to classify the relative importance of objects. The first component is based strictly on the spatial location with respect to the view point object, whereas the second component consists of non-spatial attributes of objects.

3.1.c.1 Spatial Aspects

Just as it makes sense to consider closer objects as being more important than far away objects, it makes sense to consider objects that are centered inside a player's view cone as being more important than objects that are out of that view cone.

For the following formula, let ViewVec be the unit vector that describes the direction the viewpoint object is facing, and let ViewObj be the unit vector pointing from the viewpoint object to the object whose importance we are trying to gauge. Then,

$$Importance_{spatial} = K1 \frac{1}{\text{distance}(Object_{view}, Object_{other})} \cdot K2 \left(2 + \mathbf{View}_{vec} \cdot \mathbf{View}_{obj} \right)$$

In the above equation, K1 and K2 are constants which can be tuned to vary the relative importance of the distance and the relative angle between the objects. This specific formula would make objects that are directly in front of the viewpoint three times as important as objects that are behind it. Clearly many variations on the theme of this formula can work.

Although these computations may in fact look like a partial duplication of some of the perceived error computation described in section two, that is not actually the case. Assuming that we had the same perceived error for two objects but that one of them was closer than the other, this equation specifies that the error of the closer object is more important to correct than that of the more distant one. Without this component, both objects would have had similar priorities. However, it's clearly preferable to fix errors of objects that are closer rather than further away, even if the perceived error is the same.

3.1.c.2 Non Spatial Aspects

In our game, a player can target a given object of interest. It would make sense to naturally assign higher importance to targeted objects than to untargeted ones. This is especially true in a dogfight, where it may be a lot more important to pick up the nuances of the enemy's motion than it is to receive updates for some other surrounding objects in the game. This is just an example of the type of aspects one can include into evaluating the importance of an object, and I would expect each game to have different features that would benefit from a slightly different importance classification.

3.2 Reduction of the object update packet size

It is not enough to only reduce the required frequency of updates per object. By reducing packet sizes, we allow more objects to be updated per unit time, which in turn allows the server to reduce the overall error a given client is experiencing. In order to squeeze the most out of the available bandwidth, we moved from a character or word length minimal

unit encoding to a bit encoding, where every sent bit communicates useful information. Minimal bit encoding is used for both floating-point and integer values.

3.2.a 3D Sensitive Encoding Accuracy

Given that 3D floating point data will be encoded using a fixed-point representation, it does not really make sense to require all spatial data to be sent out with the same accuracy. This is especially true for objects of varying distance with respect to the viewpoint of the client. In many cases, far away objects can be updated with much more coarse-grained fixed-point accuracy than nearby objects. How much more, depends on the game, and the graphics engine. It's relatively easy to precompute what representational accuracies are required ahead of time and use these values when encoding updates during game play. Thus, instead of always using 32 bits to encode the x, y, and z components of an object, it may be enough to use 10 or 11 bits for some objects. Below Figure 4 correlates object distance to position encoding size for our game. Clearly these values will differ for different games according to, among other things, the basic unit length definition.

Distance of Object in Game Units	Bits used for position encoding
0	23
3	22
4	21
6	20
10	19
18	18
33	17
64	16
125	15
248	14
495	13
987	12
1972	11
3940	10
7878	9
15753	8

How far one is willing to trade off specific object update accuracy for a higher rate of object updates certainly depends on the game and the available bandwidth for each client.

It would make sense to implement dynamic decision making about how many bits to use for a specific situation that not only depends on how far away the object is, but also on the overall perceived error for the client. If many objects where starting to accumulate significant error, and the current throughput to the client was too low to make significant headway in correcting such errors, it would probably help to reduce the accuracy of each update until the problem subsided. However, the scheme we currently use does not change the update accuracy based on the overall accumulating perceived error.

3.2.b Frame of Reference Encoding

Some objects will be updated much more frequently than others, and the positions of such objects probably do not vary all too dramatically on the scale of the entire game arena. Thus, instead of encoding position values with respect to the entire map of the game, one can use a much smaller frame of reference to encode the objects' position, requiring fewer

bits for the position encoding. When using an unreliable datagram service, the server would be allowed to use this frame of reference encoding only if the client acknowledges the server's suggestion of using this method. We have not implemented this method of packet size reduction for our product as of the writing of this paper.

3.2.c Selective Field Encoding

Whenever an object needs updating, it's not the case that all possible data attributes have changed or need updating. When using an unreliable datagram protocol, it is useful for the client to acknowledge some data for each object that does not change rapidly, and once these acknowledgements arrive at the server, the server can stop sending the less frequently changing data whenever a packet is sent. Such acknowledgements can also be useful for giving the server some idea about whether or not a datagram was actually received by a client, allowing the server to better estimate the client's state.

3.2.d Tag Aliasing

In our game, each object is identified with a unique 32 Bit ID tag. Obviously some objects will be resent a lot more than others, and so it would make sense for the server and client to agree on an alias for the most heavily updated objects.

There are most probably other such optimizations one might think about that could potentially reduce packet. Some of these optimizations will work better in some cases than in others, and so such packet optimizations will be the most fruitful when each game is considered on an individual basis.

4 View Point Considerations

The view point object is rather special, because unlike other objects in the game, any slight jump of the view point object is quite irritating to the player. A highly variable latency connection or even a temporary lack of throughput could have drastic effects on the viewpoint. Thus the viewpoint object requires special handling, which we will develop in this section.

It is impractical to require the server to update the viewpoint without having the client help out -- especially because the client can predict what the server is going to allow it to do. By allowing the client to predict how the server will respond to its queries, the client can run independently for some time without deviating too much from the server. When a viewpoint update does arrive from the server, the client needs to smoothly correct the error that was introduced while it was running independently. We have found that this approach works quite well when the server sends viewpoint updates at regular intervals whenever a highly sensitized version of the perceived error estimator would require the view point object to be sent. So when the view object does not change state, no viewpoint updates will be sent even if it was time to send an update in accordance to the specified viewpoint refresh rate.

To handle highly varying latency times, we smooth all arrival times out to the time-weighted average of current experienced latency. Where the time-averaged latency can be computed by updating the average with the rule:

$$Latency_{average}(t = 0) = 0$$

$$Latency_{average}(t) = \beta \cdot Latency_{average}(t - 1) + (1 - \beta) \cdot Latency_{curent}$$

$$\beta \in [0,1]$$

When beta is large more emphasis is placed on historic network latency, whereas when it is small more emphasis is placed on the currently experienced latency. In order to predict the data and the time at which the server would communicate that data to the client, the client needs to simulate the expected reactionary lag it would experience. Thus the client needs to incorporate the time-weighted average latency into the clients queries before it actually reacts. Otherwise the clients predictions would consistently be early with respect to the data the server would be sending to the client. In essence, whenever the client sends key-presses to the server, it queues the same key-presses up locally and waits to process them by whatever the current average latency is. We have found that this produces quite favorable results.

5 IP Protocol Considerations

I assume the obvious TCP/UDP tradeoffs need not be mentioned. Perhaps the one thing that does need mention is that currently over a PPP-link, UDP headers are not compressed with a Van Jacobson like TCP header compression algorithm. Thus, over a PPP link one sends about 34 bytes of overhead for UDP packets, and only 6 bytes for TCP packets.

When we use UDP, we try to send large-sized packets at a low frequently. Clearly this results in more latency, and currently it is unclear to us what a good header to data ratio is for our game. It is frustrating to think how much bandwidth is going to waste because of the 34 byte per packet overhead for UDP packets. Supposedly there is a UDP header compression proposed for ipv6, so in a few years this won't be an issue anymore.

6 Network Traffic Profiles

Below are a few preliminary traffic profiles for our game. The numbers will most likely change in some way, however, we would be surprised would they not stay in the same ballpark.

6.1 Single Object Position Accuracy profile

The profile on the next page shows how distance and bit representational accuracy relate to each other as I described above. The plot of distance versus time shows how one single object is moving closer and further away from the viewpoint object, whereas the plot of position encoding bits versus time shows the bit encoding size used for the corresponding distance.

Distance versus Time

Encoding Bits versus Time
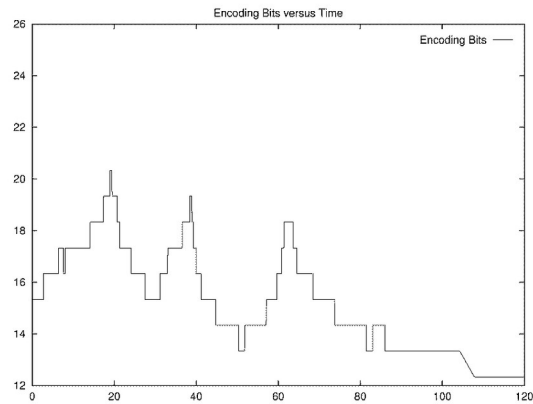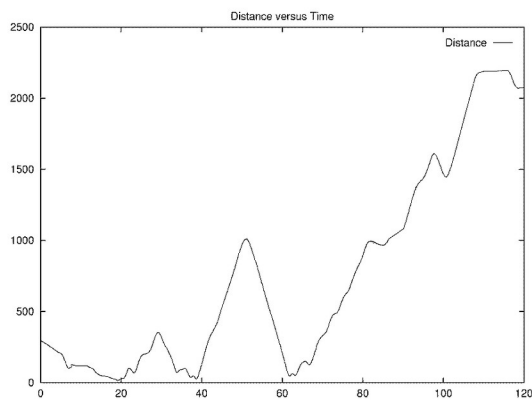
6.2 General Traffic Profile

TCP Traffic			
Packet Type	**Packets/Sec**	**Bytes/Sec**	**Bytes/Pac**
DELETE_OBJECT	1.0723	16.0849	15
WORLD_STATS	0.0076	0.2265	30
TANK	0.0076	0.1586	21
ADD_TO_ROSTER	0.0076	0.4531	60
UPDATE_STATS	0.0151	0.287	19
BIRTH_NOTICE	0.0076	0.0831	11
COMM_MESSAGE	0.0227	0.5437	24
ENTER_GAME_NOTICE	0.0076	0.1133	15
LOGIN_STATUS	0.0151	0.151	10
MOTD	0.0076	1.6387	217
BEHAVIOR	0.0302	1.9634	65
REINCARNATE	0.0076	0.6192	82
TEAM_INFO	0.0076	0.8533	113
SPACE_MAP_UPDATE	0.0076	0.6419	85
ALL_STATS	0.0076	1.0874	144
GAME_CLOCK	0.0151	0.3323	22
WARP_STATUS	0.0378	2.1522	57
TRANSLATION	0.0076	13.389	1773
JET_INFO	0.0076	1.1554	153

6.2.b UDP Traffic

UDP Traffic			
Packet Type	**Packets/Sec**	**Bytes/Sec**	**Bytes/Pac**
PING	0.8835	37.9922	43
TRANSIENT_GRAPHIC	0.4682	25.751	55
SOUND_EFFECT	1.6765	93.8817	56
UPDATE_ARRAY	8.1935	969.0761	118.2737

This traffic profile shows information on all of our packet types during a small game where we had on the average 40 objects in the world. About half of the objects where in constant motion. Note that most of our TCP packets are used only during initial client/server connections, and need not be optimized.

Clearly the UPDATE_ARRAY is the packet type that is used the most in the game, and is in some sense the most important of all. We have spent most of our time optimizing the UPDATE_ARRAY packet in terms of required size per object, and required frequency per object. Because it is the most bandwidth intensive packet, we analyzed it in greater detail which will be looked at in the next section.

6.3 UPDATE_ARRAY Profile Detail

When looking at the UPDATE_ARRAY packet more closely, we find that most of the bandwidth unsurprisingly comes from sending three-space variables. Because we send several object updates per UDP packet, we distinguish between object update traffic and UDP packet traffic.

6.3.a Per Object Update Traffic: 569.59 bytes/sec

Contents	% of Object Update	Bytes/Sec	Bytes/Object
Id	0.15	86.86	4
Content	0.05	27.14	1.25
Type	0.01	4.76	0.219
Position	0.24	135.51	6.241
Velocity	0.16	90.29	4.158
Orientation	0.16	93.11	4.288
Angular_velocity	0.17	98.5	4.536
Encoding	0.02	12.78	0.588
Health	0.02	8.98	0.414
Missile	0	0.65	0.03
Fuel	0	2.69	0.124
Jet	0.01	8.32	0.383
Total	1.00	569.59	26.231

Note that not all optimizations have been performed. I would think that frame relative position updates should cut down on the position update size considerably (at least 25%). Currently our orientation, velocity, and angular velocity are not highly optimized. Calculating the required accuracy for velocities is trickier than calculating this for the position and the orientation. The reason is that if the velocity updates are too imprecise for an object, its position or orientation error will accumulate rapidly, requiring the server to update the object at a higher frequency. Currently, it is not clear to us what a good measure of allowed approximation should be for the velocity components of the object update.

6.3.b Per UDP Packet Traffic: 335.94 bytes/sec

Classification	Bytes/Sec	Bytes/Pac
PPP/UDP header	278.8	34
Packet Type	8.2	1
Time Stamp	32.8	4
Object Count	8.2	1
Last Byte Buffer	8.2	1
Total	335.94	41

Painfully clear is what portion of the overhead comes from the PPP/UDP header versus the per packet overhead the game imposes. Ignoring the PPP/UDP overhead, we can further reduce the per packet overhead by optimizing the Time Stamp variable. Currently we use the full 32 bit encoding for the time (which is in milliseconds) -- there is no need to do that. The server and the client can regularly agree on a time stamp offset and send the time with respect to that offset. Increasing the needed frequency by which the server and client communicate the time offset becomes counterproductive at some point, as such communication takes bandwidth just as well. It is clear by looking at these profiles that there is room for improvement, however we can realize many improvements without much redesign of our networking layer -- the tools are in place.

7 Post Mortem

We are clearly trading server load, in terms of CPU and memory usage, for lowering network bandwidth requirements. There is little room for alternatives when network bandwidth is at a premium, though there is no reason why the process the server goes through to evaluate what to send to whom cannot be optimized on an algorithmic level. Though this paper does not go into the optimization issues, there are ways of trivially reducing the server load without much difficulty. One of the primary ways of doing this is to use spatial-temporal coherency of the client's viewpoint object with respect to all other objects in the world.

Assuming that the economic pressures of server load are met, one might still be worried about general scalability of the proposed design. What happens if we want to have a single game that can support several hundreds of players at the same time in the same game? There is no reason why the authoritative server cannot be made to run distributed several high-end machines connected together via a fast network backbone. There will certainly be many challenges to solve when implementing such a scheme, nonetheless the challenges do not seem insurmountable on cursory inspection.