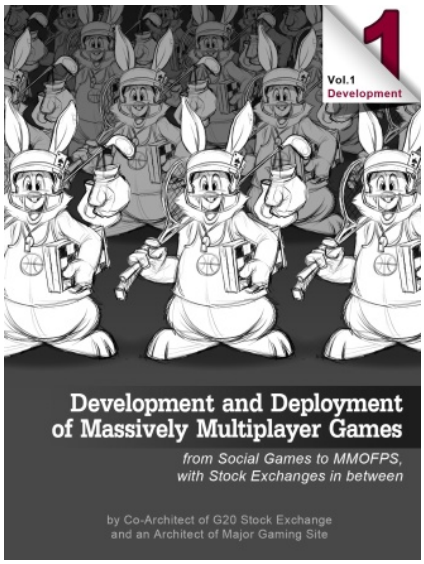
MMOG: World States and Reducing Traffic

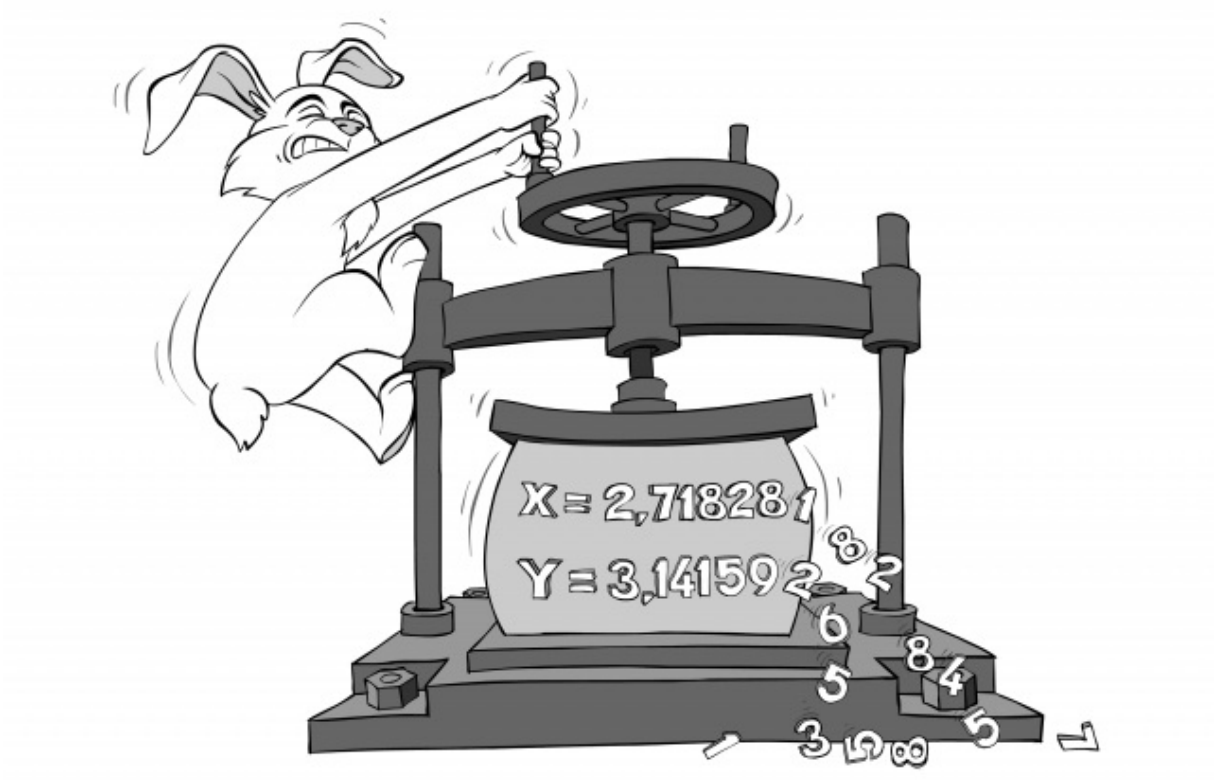
posted February 1, 2016 by "No Bugs" Hare, translated by Sergey Ignatchenko



[[*This is Chapter VII(b) from the upcoming book "Development&Deployment of Massively Multiplayer Online Games", which is currently being beta-tested. Beta-testing is intended to improve the quality of the book, and provides free e-copy of the "release" book to those who help with improving; for further details see "Book Beta Testing". All the content published during Beta Testing, is subject to change before the book is published.*

To navigate through the book, you may want to use Development&Deployment of MMOG: Table of Contents.]]

Ok, so we've finished describing data flows which may apply to your game, and can now go one level deeper, looking into specifics of those messages going between client and server. First, let's take a closer look at the message that tends to cause most of trouble at least for fast-paced simulation-based games. This is "World Update" message from Fig. VII.1-Fig. VII.3, which in turn is closely related to Publishable World State.



Server-Side, Publishable, and Client-Side Game World States

Among aspiring simulation-based game developers, there is often a misunderstanding about Game World State – "Why we need to care about different states for our Game

World, and cannot have only one state, so that Server-Side State is the same as Client-Side One?"

The answer is that "Well, depending on your game, you MIGHT be able to have one state, but for simulation games, in many cases you won't". The problem here is purely technical, but very annoying – it is a problem of bandwidth.

"The most important reason to minimize bandwidth is that traffic is expensive.

The most important reason to minimize bandwidth is that traffic is expensive. While traffic prices go down and, as of beginning of 2016, you can get unmetered 1Gbit/s for around $500-1000/month, and unmetered 10GBit/s for around $3-5K/month, it is still far from being free. On the other hand, if you're too wasteful with your traffic (like trying to send all the updates to all the meshes over the network), it simply won't fit into your player's "last mile" (connection from him to his ISP). And if you've done a good job already, still keep in mind that, as an additional incentive, making your Publishable State smaller tends to make updates to it smaller, and the smaller updates are – the less chances you have to overload your player's "last mile" (and overloading "last mile" inevitably leads to latencies going through the roof for that specific player).[1]

Note that here we're not discussing systems based on so-called "deterministic lockstep" network models; with all their simplicity, they don't work well for MMOs (in fact, [GafferOnGames.SnapshotsAndInterpolation] doesn't recommend it for Internet games with over 4 players).

---

[1] interestingly, sometimes reducing server packet size MAY help even if client's "last mile" overload is caused by a concurrent download, as there are *some* routers out there configured to give preference to smaller packets; unfortunately, I don't have stats on how widespread this effect is in practice

Client-Side State

Let's consider an example MMORPG game, OurRPG. Let's assume that our players can move within some 3D world, talk, fight, gain experience, and so on. Physics-wise, let's assume that we want to have rigid body physics and ragdoll animations, but our fights are very simple and don't really simulate physics and have fight movements animated instead (think "Skyrim").

If we have our game as a single-player, the only thing we will need, would be a Client-Side State – complete with all the meshes (with thousands of triangles per character), textures, and so on.

Server-Side State

Now, as we're speaking about MMOs, we need a Server-Side State. And one thing we can notice about this Server-Side State is that it doesn't need to be as detailed as Client-Side

State.

As we don't need to render anything on the server side, we usually can (and SHOULD) use much more low-poly 3D models on the server side

"In practice, for most classical RPGs you can get away with simulating each of your PCs and NPCs as a box (parallelepiped), or as a prism (say, hexagonal or octagonal one).

Actually, to keep the number of our servers within reason, we need to leave only the absolute minimum of processing on the server side, and this absolute minimum is defined as "drop everything which doesn't affect gameplay". In practice, for most classical RPGs (those without karate-like fights where limb positions are essential for gameplay) you can get away with simulating each of your PCs and NPCs as a box (parallelepiped), or as a prism (say, hexagonal or octagonal one). Cylinders are also possible, though these usually apply if you don't really make a classical polygon-based 3D simulation. Models of your server-side rooms can (and SHOULD) also be simplified greatly – while you do need to know that there is a wall there with a lever to be pulled in the middle, in most cases you don't need to know the exact shape of the lever.

In extreme cases, you won't even need 3D on server side at all. While this is not guaranteed, start your analysis from checking if you can get away with 2D server-side simulation – even if you will need 3D, such analysis will help you to drop quite a few things which are unnecessary on the server side.

For OurRPG, however, we do need 3D on the server side (well, we want to simulate rigid body stuff and ragdolls, not to mention multi-level houses). On the other hand, we don't need more than hexagonal prism (with additional attributes such as "attacking/crouching/..." and things such as "animation frame number") to represent our PCs/NPCs; when it comes to rigid objects simulated on the server-side – they also can be represented using only a few dozens triangles each.

When we need to simulate ragdoll on the server-side – we won't even try to simulate movements of all the limbs. What we will do, however, is calculate movement of the center of mass of the dying character. While for some games this may happen to result in too-unrealistic movements, for some other games we might be able to get away with it (and doing it this way will save lots of CPU power on the server-side), so that's what we'll try first.

This polygon reduction will lead to a drastic simplification from the classical Client-Side State (the one we'll need to render the game).

Publishable State

Now, as we've got Server-Side State and Client-Side State, we need to pass the data from the Server-Side to the Client-Side. To do it, we'll need another state – let's name it Publishable State.

And one thing to note about Publishable State is that it usually SHOULD be simpler than Server-Side state. When discussing simplifications of the Publishable State compared to Server-Side state, let's observe the following:

- As a rule of thumb, Publishable State MAY be simplified compared to Server-Side State. For example, for OurRPG the following simplifications are possible:
  - to represent PCs/NPCs, we usually can (and therefore SHOULD) throw away all the meshes, and use only a tuple of (x,y,z,x-y-angle,animation-state,animation-frame)[23]
    - in addition to the tuple required for rendering, there likely to be dozens of fields such as "inventory", "relationships with the others", and so on; whether they need to be published, depends on your client-side logic.
      - By default (and until proven that you need specific field for the client side), avoid publishing these things. The smaller your publishable state is – the better.
      - In some cases, however, you may need them. For example, if your game allows to steal something from PC/NPC, then your client's UI will likely want to show other character's inventory to find out what can be stolen. This information about the other character's inventory MAY be obtained by request, or MAY be published. In the latter case, it becomes a part of publishable state. Note that making inventory publishable won't have too bad effect on the update size, as it will be optimized via delta compression (see "Delta Compression" subsection below); on the other hand, it will increase traffic during initializations/transitions, so depending on your game it still MAY make sense to exclude inventory from publishable state and make this information available on request from a client.
      - Even if you need such rarely-changing fields as a part of your publishable state, you usually SHOULD separate them from the frequently-changed ones (for example, into separate structs or something). This is related to them having different timing requirements, which potentially lead to different retransmit policies, and it is simpler to express these policies when you have separate structs. For example, inventory is updated rarely, and is usually quite tolerant to delays of the order of 3*RTT or so; as a result, it is usually unwise to be too aggressive with re-sending it (in other words, it is usually ok to send it once and to wait for 2*RTT for confirmation before re-sending it). On the other hand, coordinates and other rendering-related stuff do need to be updated in real-time, so you should be quite aggressive with re-sending them. More discussion on re-transmission policies will be provided in Chapter [[TODO]].

"to represent PCs/NPCs, we usually can (and therefore SHOULD) throw away all the meshes, and use only a tuple of (x,y,z,x-y-angle,animation-state,animation-frame)

  - to represent rigid objects, we again SHOULD throw away all the meshes and use

only (x,y,z,x-y-angle,x-z-angle,y-z-angle) tuple.

- Whenever we CAN make Publishable State smaller, we SHOULD do it (see reasoning about reducing bandwidth above).

---

[2] actually, we can use this representation for Server-Side too, but it may or may not be convenient for the Server-Side. On the other hand, removing meshes is an almost-must for Publishable State

[3] whether we need velocities to be published is not that obvious, see "Dead reckoning" section below

Why Not Keep them The Same?

"If we'd try to transfer all the thousands of triangles every "network tick" while our character is moving, we'd need to send around 100Kbytes per "network tick" per character, and if our PC can see 20 characters at the same time, and we're using 20 "network ticks" per second, we'll end up with 40MBytes/second/

Now let's go back to the question – why not use the very same Client-Side State as Server-Side State and as Publishable State? The answer is simple – because of bandwidth. Just compare – if we'd try to transfer all the thousands of triangles every "network tick" while our character is moving, we'd need to send around 100Kbytes per "network tick" per character, and if our PC can see 20 characters at the same time,[4] and we're using 20 "network ticks" per second, we'll end up with 100KBytes/tick/character * 20characters/player * 20ticks/second = 40MBytes/second/player; this would in turn mean that we can fit only 30 players in that $5K/month 10Gbit/s channel (not to mention that only a few people will be able to play the game), Big Ouch! With our Publishable State (and even before any compression techniques are used) it is more like 50 bytes/tick/character, or (with the same assumptions) is much more manageable 50bytes/tick/character * 20characters/player * 20ticks/second = 20KBytes/second/player.[5]

Throw in the reduced Server-Side CPU load (which you will be paying for) for simplified Server-Side State, and the need to have simplified Server-Side State and Publishable State becomes obvious.

---

[4] here we're implying that we've implemented "interest management" to avoid sending unnecessary stuff, see "Interest Management" section below for further discussion

[5] it can be reduced further (see "Compression" section below), but for the moment this 3+ orders of magnitude improvement will suffice.

Non-Sim Games and Summary

For non-simulation games (such as social games or blackjack), the difference between different States is much less pronounced, and in many cases Server-Side State MAY be the same as Publishable State (though Client-Side State often will still be different). For example, whenever a card is

dealt for a blackjack game, usually it is represented as an immediate update of the Server-Side State to reflect that the card is already dealt, and update to Server-Side State is immediately pushed to the Client. All the animation of the card being dealt, is processed purely on the Client-Side (so that Client-Side State is updated without any input from the Server while the card is flying across the table).

On the other hand, even if we try to generalize our findings over the whole spectrum of the MMO games (from social ones to MMOFPS), two observations can be made. First of all, whatever our game is, the following inequation should stand:

Publishable State <= Server-Side State <= Client-Side State

The second observation is the following:

we SHOULD work hard on reducing the size of Publishable State

Publishable State: Delivery, Updates, Interest Management, and Compression

Ok, so we've decided on our Publishable State (and have done it in a reasonably optimal way), and know how to update it on the server side. The next question we face is "How to deliver this Publishable State (including updates) from Server to Client?"

Of course, the most obvious way of doing it would be just to transfer the whole state once (when the client is connected), and then to transfer updates whenever the update of the Game World occurs (which may be "each network tick" for quite a few simulation-based games out there).

However, very often we can do better than that traffic-wise. And as reducing traffic is a Good Thing(tm) both for the reducing server costs and player's latencies, let's take a closer look at these optimizations.

Interest Management: Traffic Optimization AND Preventing Cheating

Interest Management deals with sending each client only those updates within the Game World, which it needs to render the scene. It is very important for quite a few games out there.

> "Mathematically speaking, without Interest Management, the amount of data on our servers will need to send (to all players combined), is O(N^2). Interest Management reduces this number to O(N).

Let's consider OurRPG mentioned above, and the Publishable State which needs to transfer 50 bytes/network-tick/character. Now let's assume that OurRPG is a big world with 10000 players. Transferring all the data about all the players to all the players would mean transferring 10000characters*50bytes/tick/character*20ticks/second = 10MBytes/second to each player, and 100GBytes/second total (and that's with our Publishable State being reasonably optimal, i.e. without transferring meshes). However, if we notice that out of that 10000 players each given player can see only 20 other players (which is the case most of the time for most of the more-or-less realistic scenes) – then we can implement "Interest Management" and send each player only those updates-which-are-of-interest-to-her (in other words, sending only those things which are needed for rendering). Then, we need to send only 20characters*50bytes/tick/character*20ticks/second = 20KByte/second to each player (200MBytes/second total), MUCH better.

Mathematically speaking, without Interest Management, the amount of data on our servers will need to send (to all players combined), is O(N^2). Interest Management (if properly implemented) reduces this number to O(N). The same thing from a bit different perspective can be stated as

Interest Management normally allows to establish a capping on amount of traffic sent to each player, regardless of total number of players in the game.

In practice, implementations of the Interest Management can vary significantly. In the simplest form, it can be a sending only information of those characters which are currently within certain radius from the target player (or even "send updates only to players within the same "zone"). In more complicated implementations, we can take into account walls etc. between players. The latter approach will also help to address "see-through-walls" cheating.

This also leads us to a second advantage of Interest Management:

Interest Management (if properly implemented) MAY allow you to address "lifting fog-of-war" and "see-through-walls" cheats

The logic here is simple: if the client doesn't receive information on what is going on in "fog-of-war" areas or behind the wall, then no possible hacking of the client will allow to reveal this information, making this kind of attacks pretty much hopeless.

An extreme case of this class of cheats would be for an (incredibly stupid) poker site which has pocket cards data as a part of Publishable State and doesn't implement any Interest Management. It would mean that such an implementation will send pocket cards to all the clients (and then clients won't show other players' cards until the flag *show_all_cards* is sent from the server). DON'T DO THIS – the client will be hacked very soon, with pocket card revealed to cheaters from the very beginning of the hand (ruining the whole game). Interest Management (or even better – excluding pocket cards from Publishable State altogether, with, say, point-to-point delivery of pocket cards) is THE ABSOLUTE MUST for this kind of games. More or less the same stands for quite a few MMORTS out there, where lifting "fog of war" via cheating would give way too much unfair advantage.

"An extreme case of this class of cheats would be for an (incredibly stupid) poker site which has pocket cards data as a part of Publishable State and doesn't implement any Interest Management.

Note that when choosing you Interest Management algorithm, you need to think about worst-case scenarios when a large chunk of your players gather in the same place (what about that wedding ceremony which everybody will want to attend?). This can be really unpleasant, and you do need to think how to handle it well in advance. If going beyond the most obvious (and BTW working pretty well) solution of "we don't have any Big Events, so it won't be a problem" – things may become complicated (and if your game is a 3D one – the same scenarios can easily bring the number of triangles to be rendered on the client-side, beyond any reasonable limits, bringing any graphics card to its knees). One of the ways to deal with it – is to limit the number of transferred-characters to a constant limit (ensuring that O(N) thing), and when this limit is exceeded – to render the rest as a "generic crowd" simulated purely by client-side and wandering by some simple rules (and the same "generic crowd" people can be rendered as really-low-poly models to deal with polygon numbers issue).

Before Compression: Minimizing Data

One thing which needs to be mentioned even before we start to compress our Publishable State, is that most of the time we can (and SHOULD) minimize the amount of data we want to include into our Publishable State. Way too often it happens that we're publishing data field in an exactly the same form as it is available on the Server-Side, and this form is usually redundant, leading to unnecessary data being transferred over the network. A few common minimization rules of thumb:

- DON'T transfer doubles; while double operations are cheap (at least on x86/x64), transferring them is not. In 99% of cases, transferring a float instead won't lead to any noticeable changes.
- DO think about replacing floats with fixed-point numerics (in fact, an integer with an understanding where the point is, or more precisely – what is the multiplier to be used to convert from Server-State data to Publishable State and vice versa)
    - one pretty bad example of float being obviously too much, is transferring angle for an RPG. In most cases, having it transferred as 2-byte fixed-point with lower

7 bits being fraction, will cover all your rendering needs with an ample reserve

- for coordinates, calculations are more complicated, but as long as we need a fixed spatial resolution (and for rendering this is exactly what we need), fixed-point encodings are inherently more efficient than floating-point ones, as we don't need to transfer exponent for fixed-point. In addition, with standard floats it is more difficult to use non-standard number of bits. For example, if we have a 10000m by 10000m RPG world, and want to have positioning with a precision of 1cm, then we need 1e6 possible values for each coordinate. With fixed-point numerics, we can encode each coordinate with 20 bits, for 40 bits (5 bytes) total. With floats, it will take 2*32 bits = 8 bytes (that's while having comparable spatial resolution(!)), or 60% more (and if we'd transfer doubles – it would go up to 16 bytes, over 3x loss compared to fixed-point encoding).
- yet another case for transferring fixed-point numerics is all kinds of currencies (actually, it is cents which are transferred, and the rest is just interpretation)

Compression

Now we have our Publishable State with a proper Interest Management, and want to reduce our traffic further. Let's name those techniques which help us to take whatever-we-want-to-publish (after Interest Management has filtered out whatever is not necessary for the specific client), and to deliver it to the client in an optimized way, "Compression Techniques". Note that we'll interpret "Compression" much broader than usual ZIP or JPEG compression (and it will have quite a few things which are not typically used for compression), but essentially all of "Compression Techniques" are still following exactly the same pattern:

- take some data on the source side of things (server-side in our case)
- "compress" it into some kind of "compressed data"
- transfer the compressed data over the Internet
- "decompress" it back on the receiving side (with or without data loss, see on "lossless" vs "lossy" compression below)
- to get more-or-less-the-same data on the target side of things.

Also let's note that some of the techniques described below, while being well-known, are usually not named "compression"; still, I think naming them "Compression Techniques" (as a kind of "umbrella" term) makes a lot of sense and provides quite useful classification.

To make our compression practical and limited (in particular, to avoid using global states), let's define more strictly what "Compression Techniques" can and cannot do:

- "Compression Techniques" are allowed to keep a buffer (of limited size) of past values on both sides (just like ZIP/LZ77 does)
  - we MAY refer to the buffer (explicitly or implicitly) to reduce the amount of data sent
  - using this buffer creates complications when working over UDP, but there are known ways of handling it which will be discussed in Chapter [[TODO]]

- "Compression Techniques" are allowed to know about the nature of specific fields we're transferring; these specifics can be described, for example, in IDL (see Chapter [[TODO]] for more details)
    - in particular, if we have two fields, one of which is coordinate, and another one is velocity along the same coordinate, this relation MAY be used by our "Compression Technique"
- "Compression Techniques" are allowed to rely on game-specific constants, as long as they're game-wide
    - for example, if we know that for OurRPG the usual pattern when user presses "forward" button, is "linear acceleration of A m/s^2 until speed reaches V, then constant speed" – we ARE allowed to use this knowledge (alongside with A and V constants) to reduce traffic
- "Compression Techniques" are NOT allowed to use anything else. In other words, we won't consider things like client-side-extrapolation-which-takes-into-account-running-into-the-wall, as "Compression" (doing it would require "Compression" to know wall positions, and we want to keep our "Compression" within certain practical limits).
- "Compression Techniques" can be either "lossless" or "lossy". If compression is "lossy", we MUST be able to put some limits on the maximum possible "loss" (for example, if our compression transfers "x" coordinate in a lossy manner, so that client_x MAY differ from server_x, we MUST be able to limit maximum possible (server_x – client_x)). In the sections below, all the compression techniques are lossless unless stated otherwise.

Lossy compression

Lossy compression (irreversible compression) is the class of data encoding methods that uses inexact approximations (or partial data discarding) to represent the content.

— Wikipedia —

Now let's start discussing various flavours of compression.

Delta Compression

Arguably the most well-known compression is so-called "delta compression". Actually, there are two subtly different things known under this name in the context of games.

The first flavour of "delta compression" is about skipping those fields of the game state which exist in the publishable state, but which didn't change since the last update (usually, you're just transferring a bit saying "these field didn't change" instead). This kind of "delta compression" is an extremely common technique (known at the very least since Quake) which is applicable to *any* type of field, whether it is numerical or not. This, in turn, allows publishing such rarely changing things as player's inventories (though see note in "Publishable State" section above about omitting inventory from publishable state completely, or about making it available on demand; while not always possible, this is generally preferable).

VLQ

A variable-length quantity

The second flavour of "delta compression" is a close cousin of the first one, but is still a bit different. The idea here is to deal with situations when a *numerical* field changes (so skipping the field completely is not really an option), but instead of transferring new

value, to transfer a difference between "new value" and "old value" (pretty much like (A)DPCM is doing for audio signals). For example, if the field is an x coordinate, and has had an "old value" of 293.87, "new value" is likely to be "293.88", and is unlikely to be 0, so spectrum of differences becomes strongly skewed towards values with smaller absolute value, that enables further optimizations. The gain here can be obtained by either simply using less bits to encode the difference, or to play around with variable-length encodings such as VLQ, or to rely on running another layer of compression (such as Huffman compression, see "Classical Compression" subsection below) which will generally encode more-frequently-occurring-bytes (in this case – zeros) with less bits.

(VLQ) is a universal code that uses an arbitrary number of binary octets (eight-bit bytes) to represent an arbitrarily large integer.

— Wikipedia —

Let's note that the second flavour of "delta compression" can also be made "lossy": we MAY round the delta transferred, as long as we're sure that pre-defined loss limits are not exceeded. Note that ensuring of loss limits usually requires server to keep track of the current value on the client side, so that rounding errors, while accumulating, still remain below the loss limit (and are corrected when the limit is about to be exceeded).

Dead Reckoning as Compression

Dead reckoning

In navigation, dead reckoning or dead-reckoning (also ded for deduced reckoning or DR) is the process of calculating one's current position by using a previously determined position, or fix, and advancing that position based upon known or estimated speeds over

Another big chunk of simulation-related "Compression Techniques" is known as "dead reckoning". Note that despite obvious similarities, use of "dead reckoning" for the purposes of compression is subtly different from it's use for client-side extrapolation (see "Client-Side Extrapolation a.k.a. Dead Reckoning" section above)[6] When using dead reckoning for client-side extrapolation purposes we're trying to deal with latency: we don't have information on the client-side (yet), and trying to predict the movement instead, reducing perceivable latency; to do it, no server-side processing is required, and there is no precision loss. When using dead reckoning for compression purposes, we do know exact movement, and know on the server side how exactly the client will behave, so we can use this knowledge as a Compression Technique to reduce traffic (normally – as a "lossy" compression); for compression purposes, we do need server-side processing and "data loss" threshold.

The idea with a classical "dead reckoning" is to use velocities to "predict" the next value of the coordinate, while putting a limit on maximum deviation of the server-side coordinate from the client-side coordinate, so from "Compression" point of view it is a "lossy" technique with a pre-defined limit on data loss.[7]

Let's consider an example. Let's say that we have tuple (x,vx) as a part of our publishable state, and that at a certain moment client has it as (x0,vx0), and that server knows this (x0,vx0) for this specific client.[8] Now, an update comes in to the server side, which needs to make it (x1,vx1). Server calculates (x0+vx0,vx0) as a "predicted" state, and sees

elapsed time and course
— Wikipedia —

if it is "too different" from (x1,vx1).[9] If it is not too different – server can skip sending any update for this coordinate (and if it is too different – the second flavour of "delta compression" can be used to send a message fixing the difference).

For further discussion of the classical "dead reckoning" as compression (with a discussion of associated visual effects), see, for example, [Gamasutra.DeadReckoning].[10]

One last note – not only coordinates can be compressed using dead reckoning-like compression; actually, pretty much anything which can be predicted with high probability, can benefit from it. One practical example of such non-coordinates compressable by dead reckoning, is animation frame number (that is, if you need to transfer it).

"Not only coordinates can be compressed using dead reckoning-like compression; actually, pretty much anything which can be predicted with high probability, can benefit from it.

---

[6] in literature, it is usually considered to be one single "Dead Reckoning" algorithm (part of "DIS" a.k.a. IEEE1278) which reduces both perceivable latency and traffic. However, due to differences in both the effects and implementation, I prefer to consider these two uses of Dead Reckoning separately
[7] while it can be made lossless, it won't get much in terms of compression, so the lossless variation is almost-never used
[8] as noted above, when using UDP, this is tricky, but doable, see Chapter [[TODO]] for further details
[9] "too different" here is the same as "exceeding pre-defined loss limit"
[10] despite the title, most of the discussion within is not about latency, but about reducing traffic with a pre-defined threshold, which we refer to as one of "Compression Techniques"

Dead Reckoning as Compression: Variations

"Dead reckoning" as described above, is certainly not the only way to use kinematic equations to optimize traffic. Possible variations include such things as:

- using "delta compression" (the second variety described above) to encode data when the "loss limit" is exceeded
- using accelerations in addition to velocities (and predicting velocities based on accelerations)
- calculating velocities/accelerations (using previous values in the buffer) instead of transferring them
- use of smoothing algorithms to avoid sharp change of coordinates when the correction is issued. These are similar to the smoothing algorithms used for server reconciliation (see "Running into the Wall, and Server Reconciliation" section above), and the same smoothing algorithm can be used for both purposes. Whether to consider smoothing a part of compression (or a post-compression handling) – is not that important and it depends.
- using knowledge about the game mechanics to reduce traffic further.

- As one example, if in OurRPG velocity of PC always grows in a linear manner with fixed acceleration until it reaches a well-defined limit – this can be used to calculate "predicted speed" and to avoid sending updates along this typical pattern.

Classical Compression

LZ77

LZ77 is the lossless data compression algorithm published by Abraham Lempel and Jacob Ziv in 1977.

— Wikipedia —

Classical lossless compression (such as ZIP/deflate) usually uses two rather basic algorithms. The first one usually revolves around LZ77[11] (with the idea being to find similar stuff in the earlier buffer and to transfer a reference instead of verbatim stream). The second algorithm is usually related to so-called Huffman coding,[12] with the idea being to find out what symbols occur in the stream more frequently than the others, and to use less bits to encode these more-frequently-used symbols. Of course, there are lots of further variations around these techniques, but the idea stays pretty much the same. ZIP's deflate is basically a combination of LZ77 and Huffman.

Unfortunately, classical compression algorithms, such as deflate, are not well-suited for game-related compression. One of the reasons behind is that (as it was shown for deflate in [DrDobbs.OnlineCompression], these algorithms are usually not optimized to handle small updates (in other words, "flush" operation, which is required to send an update, is expensive for ZIP and other traditional stream-oriented algorithms).

On the other hand, it is possible to have a compression algorithm optimized for small updates; one example of such an algorithm is an "LZHL" algorithm in the very same [DrDobbs.OnlineCompression] by my esteemed translator. Like deflate, it is a combination of LZ77-like and Huffman-like compression, unlike deflate, it is optimized for small updates.[13]

"It is possible to have a compression algorithm optimized for small updates; one example of such an algorithm is an "LZHL" algorithm

If nothing else, you can always try to use Huffman (or Huffman-like, as described in [DrDobbs.OnlineCompression]) coding for your packets. I won't go into too much details of Huffman algorithm as such here (it is described very well in [Wiki.Huffman]), but one trick which may help here with regards to games, is the following. Usually, implementations of Huffman algorithm transfer "character frequency tables" as a part of compressed data; this leads to the complications in case of lost packets (or, if you transfer the table for each packet, they will become huge). For games, it is often possible to pre-calculate character frequency table (for example, by gathering statistics in a real game session) and to hardcode this frequency table both into the server and into the client. In this case, lost packets won't affect frequency tables at all, and this variation of Huffman will work trivially over both TCP and UDP. Note though that usually gains from Huffman are rather limited (even if your data has lots of redundancies, don't expect to gain more than 20% compression from pure Huffman), but it is usually better than nothing.

[11] and its close cousins such as LZ78 and LZW
[12] or a bit more efficient but much slower arithmetic coding
[13] note that LZHL as such won't work directly over UDP, and some significant adaptation will be necessary to make it work there; for TCP and TCP-like streams, however, it has been seen to work very well

Combining Different Compression Mechanisms and Law of Diminishing Returns

It is perfectly possible to use different compression mechanisms together. For example:

- for relatively static data (such as inventory), delta compression (1st variation), followed by classical compression, can be used
- for very dynamic coordinate-like data – dead reckoning (as a lossy compression), with dead reckoning using delta compression (2nd variation), using VLQ to encode differences, can be used

Note that the examples above are just that – examples, and optimal case for your game may vary greatly.

One further thing to note when combining different compression mechanisms, is that all of them are merely reducing redundancy in your data, so even if they're not conflicting directly,[14] traffic reduction from applying two of them simultaneously, will almost universally be less than the sum of reductions from each of them separately. In other words, if one compression gives you 20% traffic reduction and another one – another 20%, don't expect two of them combined to give you 20%+20%=40% or 1-(0.8*0.8)=36% reduction – most likely, it will be less than that 🙁.[15]

[14] examples of such direct conflicts would be trying to use dead reckoning *after* classical compression, or using LZ77 compression *after* Huffman compression
[15] while there are known synergies between different compression algorithms, notably for LZ77 *followed by* Huffman, they're very few and far between

Traffic Optimization: Recommendations

When speaking about optimizing traffic, I usually recommend the following order of doing it:

- minimize your Server-Side State. It is important not only to minimize traffic, but also to minimize server-side CPU load

- minimize your Publishable State. Be aggressive: throw away everything, and add fields to your Publishable State only when you cannot live without them
- split your Publishable State into several groups with different timing requirements
- make sure to use “Delta Compression” (the first variation above) to allow skipping updates for non-changing objects
  - treat “non-changing objects” broadly; for example, for many games out there an object which keeps moving with the same speed in the same direction, can be treated as “non-changing” (alternatively, you can handle it via “dead reckoning”)
- think about “Dead Reckoning” compression, keeping adverse visual effects in check (and reducing threshold if necessary)
  - don’t forget about variations, they may make significant difference depending on specifics of your game
- think about running Classical Compression on top of the data compressed by previous techniques, but don’t hold your breath over it
  - deflate as such won’t work for most of the games (due to the cost of “flush”, see above)
  - LZHL works for TCP, but adapting it for UDP will require an additional effort (and will hurt efficiency too)
  - Huffman with pre-populated frequency tables (see above) will work for UDP, but the gains are limited
- when combining different compression techniques, keep in mind that their order is very important
- I strongly suggest to separate all types of compression from the rest of your code (including simulation code)
  - moreover, I strongly suggest to say that compression code should be generated by your IDL compiler based on specifications in IDL, instead of writing compression ad-hoc. More on IDL in Chapter [[TODO]].

“Minimize your Publishable State. Be aggressive: throw away everything, and add fields to your Publishable State only when you cannot live without them

[[To Be Continued…

This concludes beta Chapter VII(b) from the upcoming book “Development and Deployment of Massively Multiplayer Games (from social games to MMOFPS, with social games in between)”. Stay tuned for beta Chapter VII(c), “Point-to-Point Communications”]]

[–] References

[GafferOnGames.SnapshotsAndInterpolation] Glenn Fiedler, “Snapshots and interpolation”, Gaffer on Games
[Gamasutra.DeadReckoning] Jesse Aronson, “Dead Reckoning: Latency Hiding for

Networked Games", Gamasutra
[DrDobbs.OnlineCompression] Sergey Ignatchenko, "An Algorithm for Online Data Compression"
[Wiki.Huffman] "Huffman coding", Wikipedia

Acknowledgement

Cartoons by Sergey Gordeev from Gordeev Animation Graphics, Prague.

« *MMOG. RTT, Input Lag, and How to Mitigate Them*

MMOG. Point-to-Point Communications and non-blocking RPCs »

Filed Under: Distributed Systems, Network Programming, Programming, System Architecture
Tagged With: client, compression, game, multi-player, network, protocol, server