A LOOK AT LATENCY IN NETWORKED GAMES

BY JONATHAN BLOW

Those of us developing networked games are less conscious of latency issues than we should be. Often, this is because common knowledge has already provided us with convenient excuses for our problems. When a game feels laggy or behaves unreliably, well, everybody knows that modems are slow, and everybody knows the Internet is unreliable, so obviously the game will suffer.

Here, we will look at the major sources of latency in a networked game. We will show that large portions of this latency are caused by the game itself or by the nature of serial communication in a way that is heavily influenced by the game's behavior. In

Jonathan Blow comes from the West country where the birds sing bass. He does not spell "mipmap" with a hyphen. Contact him at jon@bolt-action.com.

other words, much of the latency is our own fault — but that fact gives us the power to find solutions.

The Method

In order to analyze latency in a systematic and concrete way, we will observe the workings of a specific system: a client/server architecture, where the server is authoritative over the state of the world. The clients act as windows for viewing that world. The server frequently tells the clients about the state of entities in the world (their positions, velocities, and whatever else); each client tells the server what actions its player would like to take, and the server computes the effects of these actions on the world. Each client runs asynchronously from the server and all other clients; its frame rate is not locked to the network in any way (perhaps it uses dead reckoning to extrapolate moving objects).

Many games' architectures don't quite resemble this scheme (for example, peer-to-peer games in which clients can have authority over world state), but most of the concepts explored in this article still apply.

Some calculations made in this article are frame-rate dependent, so we must pick a typical frame rate for a game client. We will use 20 frames per second as our typical frame rate. At the time of this writing, 20 FPS is considered a reasonable frame rate for a 3D game. With news of the Voodoo2 running QUAKE at 120 FPS, it's evident that a year or two from now, 20 FPS may be considered poor. However, this is not inevitable since, in the past, developers of PC games have chosen to increase a game's features and graphical punch to the detriment of frame rate (QUAKE II is slower than its predecessor). Also, 3D games tend to follow the technology curve very closely, so whereas a game may run at 30 FPS on high-end hardware, it may run at only 12 FPS on the machines of half the people actually playing the game. Lastly, we've seen that the conditions of a multiplayer game, during the times when the user desires the most responsiveness (such as in a heavy firefight during a death match game) tend to be much more stressful than the conditions during a single-player game; therefore, the frame rates that matter will be substantially lower than the figures reported in benchmarks. For now, we will stick with the 20 FPS figure; however, we will be careful to spell out all the equations we use to compute lag so that the computations can be made for any client speed.

Variance

Besides latency, from time to time we'll also look at variance, the amount by which latencies fluctuate. Having a lot of variance in the system is bad for several reasons; it makes dead reckoning more difficult for the computer to perform, and it tends to confuse human reflexes (it's not too hard for a person to adapt to a 200ms lag between action and consequence, but it is much harder — and more frustrating — to deal with latencies that fluctuate between 50ms and 300ms).

The statistical notion of variance is not very intuitive, so we'll be looking at the standard deviation of latency, which is the square root of its variance. The standard deviation of a variable is how far away we can expect one sampling of the variable to be from the mean. We'll encounter latencies that fluctuate between two values, l_{low} and l_{high}, but can adopt any value within that range with equal probability. In this case, the mean latency is $0.5*(l_{high} + l_{low})$. The standard deviation is

$$\left(\frac{\sqrt{3}}{6}\right)*\left(l_{high} - l_{low}\right).$$

So if our system's latencies fluctuate between 50ms and 300ms, the mean is 175ms, and the standard deviation is about 72ms.

Now that we've covered the introductory material, we'll proceed in our analysis of networked games by first looking at the lag suffered during a single-player, un-networked game.

A Single-Player Game

How can a single-player game suffer lag? If we think only in terms of modems and networks, then the idea makes no sense. But in order to get a comprehensive look at the concept of lag, we must look carefully at the way a single-player game operates.

We're very familiar with the concept of frame rate: it takes a game some amount of time to draw its graphics; the faster it can do this, the higher its frame rate. Let's look at frame rate from a different angle: if a game is running at 20 frames per second, it takes one twentieth of a second (50ms) to draw each frame. When it's done drawing the frame, the player can see the new state of the world. So, at 20 FPS, once the game decides what the state of the world should be (as in, where the player is and in what direction he's looking), it takes 50ms to communicate this decision to the player. That 50ms is lag; but it's not the only kind of lag we'll see in a single-player game.

A typical game might have a loop structure that looks something like Listing 1. It's important to note that, with respect to the client's cycle time, the rendering and movement cycles (`move_objects()` and `draw_scene()`) represent an all-consuming atomic void during which no input events can be meaningfully processed. If an input event occurs (the user hits a key, for example), then we must wait for movement and rendering to complete before we can get back to `read_input()` and process the event. (We could do something tricky and have `read_input()` occur much more frequently than once per cycle; this would change the flavor of the lag, but wouldn't reduce its overall magnitude. We discuss ideas such as this in the conclusion to this article.)

Our game's intended audience, game players, are individuals with free will and human spirit and all that stuff. When a player presses a key, it's an act

LISTING 1. *A typical game loop.*

```
while (1) {
      read_input();         // keyboard, joystick or whatever;
                            // change object movement parameters based on input
      move_objects();       // change objects´ positions based on movement parameters
      draw_scene();         // all the k-rad graphics, d00dz!
   }
```

of unpredictable free will; the time of the keystroke is not related to the internal operations of the game program. So if we ask, in which phase of the client cycle, and when during that phase, does the keystroke event happen, the answer is (to a first approximation) that it can happen at any time with equal probability.

Now for simplicity, we'll assume that the calls to `draw_scene()` take 100 percent of the CPU time on the client and that each call to `draw_scene()` takes an equal amount of time. This means that incoming keystrokes will be evenly distributed across the execution of `draw_scene()`. On average, a keystroke will occur smack in the middle of `draw_scene()`. So when a keystroke occurs, we have to wait half a cycle until we can process the keystroke. Now we need to move our viewpoint in response to the input and draw the new frame, which takes a cycle. That's a total of 1.5 cycles of lag in the average case, though the amount varies between 1 and 2 cycles.

What does this mean in concrete terms? When we're playing a single-player game, strutting down hallways blasting Stroggs at 20 frames per second, that's 1,000/20 = 50ms per frame, which means that it takes the game 50*1.5 = 75ms to visually respond to our keystrokes, fluctuating between 50ms and 100ms, with a standard deviation of about 14ms.

These numbers should already be setting off warning bells in the analyst's mind. An "acceptable" 28.8Kbps modem connection has a ping time of about 150ms — that's the round-trip time for a ping packet to go to the machine at the other end of the modem and then come back. But what we're seeing is that, at 20 FPS, which is typically considered a "responsive" frame rate, we are faced with 75ms of lag. So why do modem games feel so much worse than single-player games that seem to provide instantaneous feedback? Several factors contribute to this discrepancy, but a big component of the answer is that a networked game running over a modem with 150ms ping time will suffer a real latency much higher than 150ms.

Just for kicks we'll consider the case of a single-player game running at 12 FPS. Twelve FPS is not "smooth" animation, but it's still a high enough frame rate to feel responsive. Each frame takes 1,000/12 = 83.3ms, with a typical latency of 83.3*1.5 = 125ms, which is getting pretty darn close to that 150ms of raw ping time.

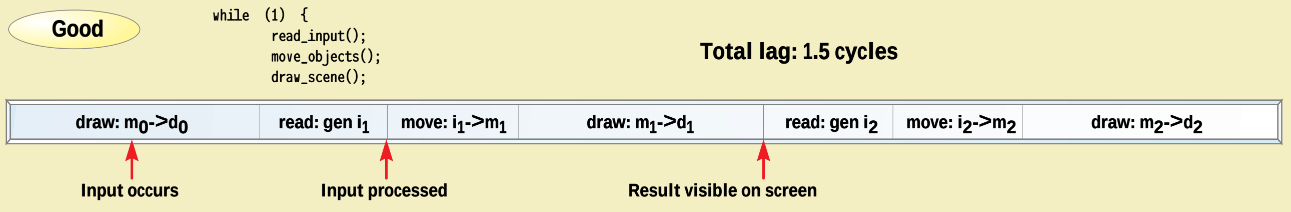
So what we're seeing is that a game runs in discrete cycles, and those cycles can cause lag in two different ways. We'll call that first half-cycle of waiting an influence lag, because it's the delay between our attempt to influence the world (by pressing a key) and the time the influence can occur. We will call the cycle required to draw the scene an

Loop Structure

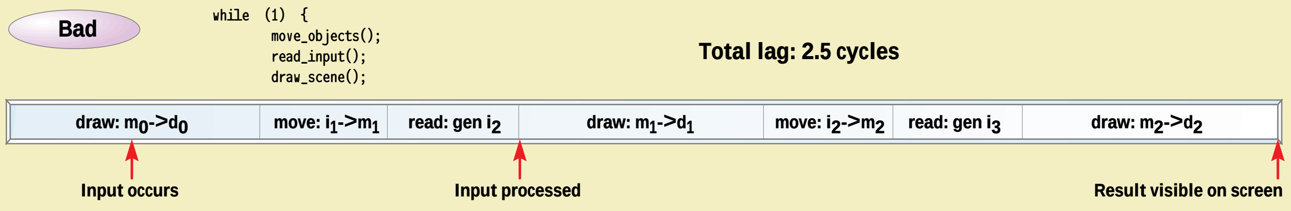
The diagnoses of latency and variance covered in this article consider the optimal case to be one in which the applications process information in the most efficient order. However, simple mistakes in the organization of a loop structure can make the situation much worse than we consider.

There are at least three basic operations the game loop has to perform: it must read input from the user, move objects in the world (including the viewpoint) based on internal simulation (which is influenced by that input), and draw the current state of the world to the screen.

The order in which we perform these steps will have an effect on the latency in the system. For simplicity, we will assume that the drawing step takes almost all of the application's CPU time; the read and move steps will be negligible in comparison.



```
while (1) {
        read_input();
        move_objects();
        draw_scene();
```



```
while (1) {
        move_objects();
        read_input();
        draw_scene();
```

The extra lag induced in the "Bad" example is obvious in some cases; many people who write single-player games see the problem and get it right. However, we present this simple case as an illustration of a phenomenon that can occur in a complex system in ways that are much subtler.

observation lag, since it's the delay between an event's occurrence and its display. These two fundamental types of lag have different effects on game play; later, we'll see that we can sometimes trade one kind of lag for another.

Besides the client frame time, some other factors can introduce lag, such as the monitor's refresh time and the time that it takes the player's brain to process the new information. These are gray and sticky areas however, and we'll avoid them. We'll be content to say that our computations yield a conservative estimate of lag, and that actual experienced values will be higher.

Multiplayer, Ideal Communications

Now we'll consider the case of a client/server game, but one with "ideal communications": in other words, a communications link of infinite speed and perfect accuracy. In this case, the only latencies introduced will be of the cycle-induced type that we've seen for the single-player game; however, the problem is now compounded because of the two communicating entities.

When a player causes input events, the client must communicate to the server in order for that player's input to affect the world. The server must communicate the resulting changes in world state (due to that player's actions, as well as those of other players) to other clients for display (Figure 1).

In the simplest version of this scheme, the client would listen to its own effects on the world in the same way it would listen to other players' effects on the world: by hearing the results from the server. This requires the client to wait for a full round-trip before seeing the results of its actions.

FIGURE 1. *A client sends requested actions to the server; the server computes the results of those actions and sends out the results for all clients to observe.*

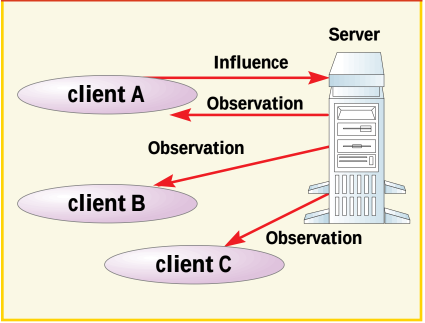
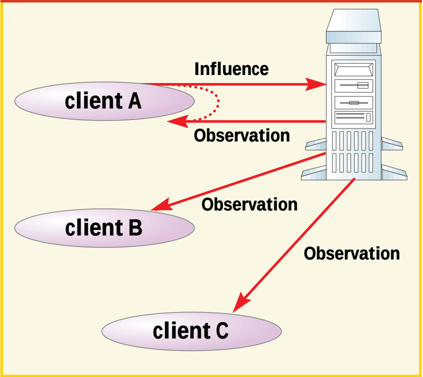


FIGURE 2. *Like Figure 1, but the client predicts the results of its own influence without hearing from the server (dotted arrow at client A).*



To reduce perceived latency of the player's own actions, we can have the client observe its own state requests and predict their results on the world without waiting for the server to process them (Figure 2). Thus, if we wish, we can make the client respond to its own events with the same latency characteristics as in the single-player game. However, we should be cautious about this because, as we will see, all other events in the game are subject to higher latencies.

Given this client/server game structure, we can calculate the amount of latency induced in the system by first looking at both the client and the server as isolated components, figuring out how much latency there is in each component, and then adding the two results together. Let's assume we have a client running with a main loop that is similar to that of the single-player example. This client is subject to the same input delay as the single-player game: half a client cycle (we'll call the client cycle *c*, so the influence lag is 0.5*c*.) After this half a cycle, the client is able to read the keystroke event and send it as a message over the network. As for incoming messages, these are subject to the same delays as input devices: they will arrive in the middle of a draw cycle. At the end of the draw cycle, the messages will be processed, which takes 0.5*c*. Then they must be drawn, which takes 1.0*c*. The player can see the results of the message after a total of 1.5*c* of observation lag.

Now we look at the server end: the server must receive messages from the client, which will typically arrive some-

FIGURE 3. *The stages of lag involved in cycle-inhibited client/server communication. Total lag is 2.0c + 1.5s.*

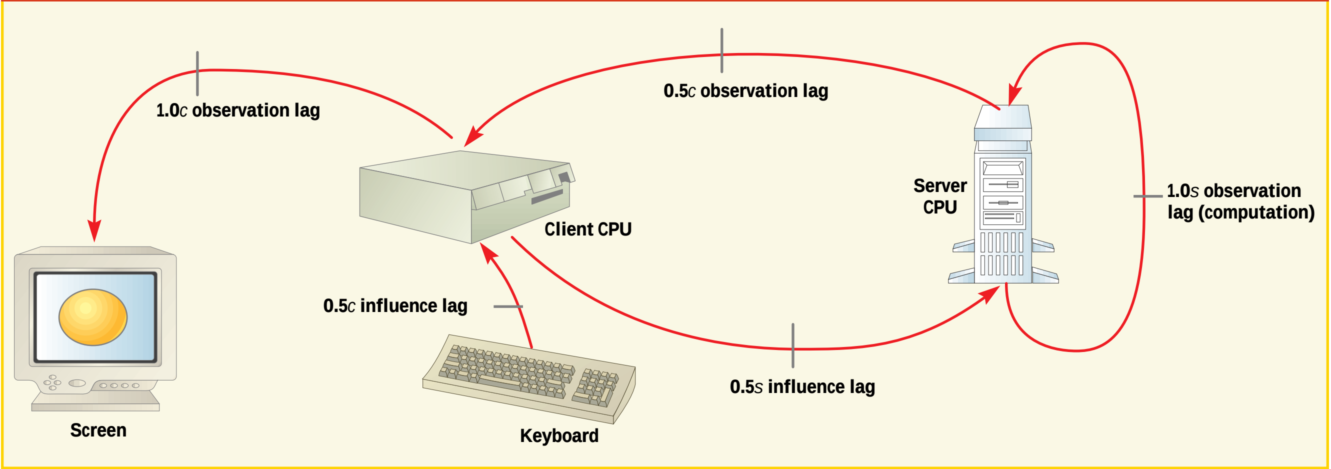
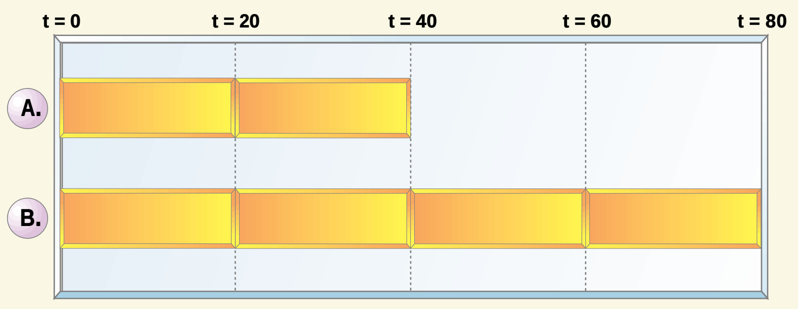


FIGURE 4. *When we attempt to transmit messages in rapid succession, the messages will queue up. This adds latency and induces variance. In A, we have tried to send two messages at once; it takes 40ms for both to complete. In B, we send four; it takes 80ms.*



where in the middle of the server cycle (which we will call *s*). After 0.5*s*, the events can be processed, having an effect on the world state; therefore the 0.5*s* is influence lag. It takes 1.0*s* to compute the results of the inputs on world state, after which time the results are sent to all clients. This 1.0*s* is probably observation lag (based on how the system is constructed).

Let's sum up this section and put the events in the proper sequence. On the client, we first have 0.5*c* of influence lag, then 0.5*s* for the message to get into the server. Then we have 1.0*s* of observation lag, then the response is sent to the client, which adds an additional 1.5*c* of observation lag. The total is 0.5*c* + 0.5*s* of influence lag and 1.0*s* + 1.5*c* of observation lag, for a total of 2.0*c* + 1.5*s* of general lag. The standard deviation is 0.29*c* + 0.14*s* (Figure 3).

Let's express this in real-world terms. Assuming both the client and the server are running at 20 FPS, that's 2.0*50 + 1.5*50 = 175ms of lag, deviating by 22ms, easily exceeding the 150ms ping time we mentioned earlier. Just for kicks, let's run these calculations for a low-end machine running at 12 FPS (where the server is still running at 20 FPS): 2.0(83) + 1.5(50) = 241ms, deviating by 31ms. Are we having fun yet? Next we'll attach a modem and see what happens.

Enter the Modem

Our modem will be an ideal modem that can transmit bits at a fixed rate, and aside from that is perfect in every way (no line noise, no overhead in setting up data for transmission, and so on). Our ideal modem will run at 28.8Kbps. (Yes, modems of higher speeds such as 33.6Kbps or an ostensible 56Kbps are common, but higher speeds are more susceptible to line noise, causing some serious problems for real-time games. Rounding down to 28.8Kbps will also help to compensate for other interference effects that this article is not taking into account.)

So if we're transmitting data serially at 28.8Kbps, each bit takes 1/28,800 sec, or $3.47*10^{-2}$ms to transmit. We'll call this unit of time *b*. Sending 32 bits over the modem will take 32**b* seconds, or 1.11ms.

A modem is an asynchronous communications device, which means that some signaling overhead is required to transmit messages. Typically, the modem must frame every 8 data bits transmitted with a start bit and a stop bit, so that it ends up transmitting 10 bits. So we need to multiply the number of bits we're sending from the application by 10/8 to get the number the modem is transmitting.

We'll want to measure our data in bytes, and a byte is 8 bits. So we'll multiply that 10/8 by 8 to convert from bits to bytes. So every byte we send takes 8*(10/8)**b* seconds = 10*b* seconds ≈ 0.35ms.

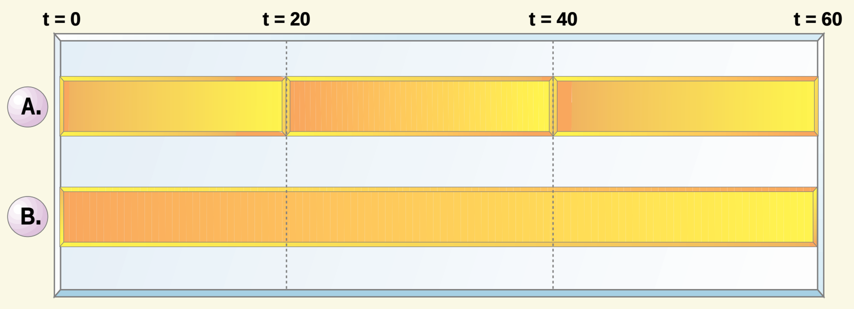
If we send a 64-byte message over our ideal modem, it will take 64* 10*b* = 22.21ms to get across. Assuming the message is not useful until the whole thing is received, this gives us about 22ms of extra lag to add to our previous computations.

Because it takes time to transmit bits, if we try to send messages too quickly, they'll pool up in their rush to get out. If our application sends two messages at the same time, the second message must wait for the first message to complete before it can begin its journey. Therefore the second message suffers further delay; the first went through in 20ms, but the second takes 40ms. Of course this phenomenon worsens with the number of simultaneous messages. We'll call this situation "message stall."

Message stall causes latency to change on a per-message basis; therefore it induces variance, even if our communication line is variance-free. Figure 4 illustrates the point.

In A, we send two messages. The first gets across the line after 20ms; the second gets across after 40ms. The average latency of our state messages is (20 + 40)/2 = 30ms, deviating by 10ms. In B, we send four messages. The average is (20 + 40 + 60 + 80)/4 = 50ms, deviating by 22ms.

FIGURE 5. *Three messages are sent simultaneously. In A, we send them as separate messages. The average latency is 40ms, deviating by 16ms. In B, we package the messages into a larger unit; the latency is now higher, at 60ms, but the deviation is 0.*



This is an important issue. Many optimized networking schemes would like to send a variable number of messages each frame, based on what is currently considered important to the game state. Nonetheless, varying the number of messages per frame isn't a good idea unless care is taken to compensate for the consequent extra variance.

When sending several messages to a client at once, we might choose to pack them all together into one message and send them as a unit. In the perfect case that we're studying here, this would be worse than sending them separately. Because the submessages are processed as a unit, none of them can be handled until the entire compound message has been received. This increases the average latency. The variance problem may still exist as well, because compound messages of different lengths will be lagged by different amounts (Figure 5).

Modem Guts

Of course, nobody who plays our game is going to have an angelic modem. Real modems and real communications lines introduce real problems. Telephone switches (the routers of telecommunications land), and all the other equipment involved in transporting and reproducing a telephone signal, will induce latency. Line noise can corrupt messages that we transmit. Sometimes, even on good phone lines, noise can come in bursts, causing blackouts of several hundred milliseconds, during which no messages successfully get through. Wacky changes in telephone line voltage can cause modem byte framing errors, requiring bytes to be retransmitted.

Error correction and compression schemes (such as those included in CCITT v.42bis) can be employed to combat line noise and increase bandwidth. Many of them introduce synchronous communication modes to eliminate the overhead of start and stop bits. However, such schemes introduce problems for real-time games. They usually packetize data into multibyte chunks, which increases latency because a message that ends in the middle of a modem-generated packet cannot be processed until the entire packet is received. Compressing and uncompressing data requires extra computation, and much of the other maneuvering that the modem must perform also increases latency. An error correction scheme can resend data in the case of line noise, but this is usually not what we want because we aren't transmitting stream data (see the discussion of TCP in the next section); this retransmission will delay the transmission of further data. We have found that for the sort of game we are discussing, in most cases it's best to turn off error correction and compression.

In this section, we haven't done much to quantify the influence of these effects on latency; this would be difficult because the effects vary so much from situation to situation. For now, we'll make ourselves content simply being aware of these issues and move on.

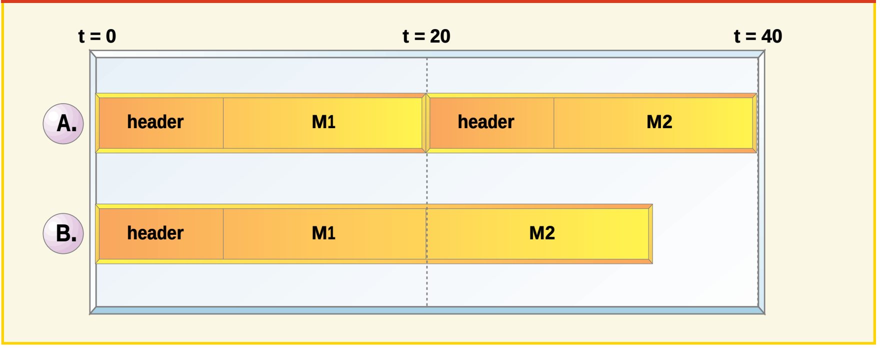


FIGURE 6. *Instead of transmitting two messages as two separate UDP packets, as in A, we can bundle them in one packet, as in B. In B, the message M1 arrives at its destination later than it did in A, but message M2 arrives earlier.*

Protocol Overhead

Now the excitement really begins! Any message that wants to travel on the Internet has to ride inside an Internet Protocol (IP) packet. In this section, we'll talk about the overhead involved in using IP over modem lines, as well as the higher-level protocols in the IP family, TCP and UDP.

The IP packet header contains information on the packet size, the source and destination addresses, and other transportation and maintenance information. The IP header is 20 bytes long — that's 20 extra bytes concatenated to any message we send.

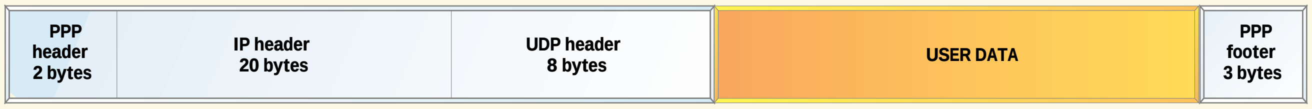
The IP header alone doesn't provide enough information, however; it is only sufficient to describe the source and destination hosts of a packet, but not what to do with the packet when it reaches the destination. For that you need to use a higher-level protocol such as TCP or UDP, and if you're smart, you won't use TCP. (Many reasons have been given for why TCP is not appropriate for real-time applications. Often, people cite issues such as the exponential backoff that can cause excessive retransmission delay. But there is a much more fundamental reason that is easy to understand. TCP is an order-preserving, guaranteed-delivery protocol, meaning all data is delivered to your application in sequence. If one small part of the stream gets lost on the network — say, one byte — all further incoming data is withheld from your application until the data loss is discovered and the missing data is successfully retransmitted. This is silly and harmful if the data is logically independent from other information in the stream. To improve TCP's real-time properties, its designers built in facilities such as urgent mode, but that doesn't really aid our case.)

So for Internet communication, unless you want to write your own IP-family protocol (I definitely don't — I have a game to write), UDP is the only reasonable choice. For the record, though, TCP packets induce 20 bytes of overhead in addition to the IP header.

The UDP header is smaller, only 8 additional bytes on top of the IP header, for a grand total of 28 bytes. So if you transmit 16-byte messages in your application (small messages to keep latency down), you're really transmitting 44-byte UDP packets, or 64 percent overhead.

Under oppressive conditions such as this, one might decide to pack multiple state updates into one UDP packet to

FIGURE 7. *How many licks does it take to get to the center of a datagram?*



mitigate the overhead of the UDP/IP header (Figure 6). But this causes different problems. As we've already determined, latency and variance will go up because none of the state messages can be processed until the entire packed-up message has been received (because the operating system won't give the message to our application). Of course, if our state messages are very small and we send them in separate packets, we end up sending mostly IP headers, and that does even worse things to our latency. Clearly, a balance must be struck between message size and the number of headers we wish to tolerate.

So now, when computing the latency of our messages, we'll add those 28 bytes of UDP and IP overhead. This overhead is starting to get nontrivial, so we'll phrase it as the lag function ModemLag(n), where n is a quantity of bytes. Previously, we had ModemLag(n) = n*10b; now we have ModemLag(n) = (n + 28)*10b. That's an annoying amount of overhead, but unfortunately, there's more.

IP is a device-independent protocol — it tells Internet routers what to do with a packet once the packet reaches them. But to transport a packet from one router to another, you also need a protocol that lets IP run on the physical communications layer. For modems, this is usually PPP (Point-to-Point Protocol), which maintains the Internet connection over a modem. As you'd guess, PPP message frames induce additional overhead; the overhead would be 8 bytes, except that it's usually negotiated down to 5 once a consistent connection is established (PPP is all about the machines on each end of the line negotiating connection parameters).

For starters, we'll need to add 5 more bytes to our previous 28, for a total of 33 (Figure 7). Besides that, PPP uses ASCII characters 0x7d and 0x7e as signals of its control protocol, so they must be escaped whenever they appear in application data; this is done by prefixing them with 0x7d, effectively doubling those bytes. Furthermore, to prevent problems with communications middlemen that might misinterpret ASCII codes 0x00-0x1f as nondata (for example, as flow-control signals), PPP can be configured to escape any or all of those 32 characters, the default being to escape them all. This configuration is decided during another one of PPP's frisky connect-time maneuvers, the ACCM negotiation.

The upshot is that if your message content is evenly distributed across ASCII (for example, random binary data), you will suffer anywhere from (2/256)*n to (34/256)*n in extra bytes transmitted, depending on the ACCM. (If your data contains a lot of 0s, and that byte is being escaped, performance could be a lot worse.)

Carrying on in our tradition of optimism, we will assume that the PPP ACCM negotiation has turned off escaping of all bytes but 0x7d and 0x7e. Then we have ModemLag(n) = (n + 33)*258/256*10b. (This isn't quite right because parts of the PPP header won't ever be quoted, but it's close enough for hand grenades.) That factor of 258/256 is pretty negligible, but we leave it in to remind ourselves that it could end up being much higher, especially if we're not in control of the circumstances surrounding the dial-up connection. If we were pessimistic about the ACCM negotiation results, our equation would be ModemLag(n) = (n + 33)*290/256*10b.

In other words, the PPP escaping will cause somewhere between 0.8 percent and 13.3 percent overhead, depending on configuration.

Let's try to get our bearings again by plugging some real numbers into these equations. For instance, how long does it take to send a 64-byte message with all these overheads?

ModemLag(64) = (97) * 258/256 *10b
= 33.9ms

And what is the fastest that we could possibly get a message from one end to another (a 0-byte message)?

ModemLag(0) = (33) * 258/256 * 10b
= 11.5ms

For what it's worth, PPP provides a header compression scheme that compresses the IP and TCP headers of TCP stream packets to become very small, eliminating much overhead. But that only works for TCP. There's no good reason why it doesn't work for UDP, except that nobody ever cared enough to do it. So we're stuck with this for now.

Ping

It's common to use the ping utility to measure round-trip time between two sites on the Internet. This utility sends an "echo request" ICMP packet; ICMP is the Internet Control Message Protocol, and its header size (in the case of an echo request or reply) is 8 bytes. Ping is usually set to transmit 56 bytes of random data by default; so, with all the headers included (5 bytes PPP, 20 bytes IP, 8 bytes ICMP), we are transmitting 89 bytes per ping. PingLag(89) = 89*258/256*10b = 31.1ms. Because the ping is making a round trip, we multiply this number by two, getting 62.2ms, which is the time a ping would take under ideal conditions over a 28.8Kbps modem. The veteran Internet user knows that measured numbers are usually much higher.

Second-Order Effects

Aside from the major lag-inducing effects that we've examined, there are billions and billions of weaker effects inhabiting the galaxy.

Sending and receiving data over networks involves the operating system handling the data, which may require context switches. We may be confronted with bus contention or network-device-instigated delays. If we've got an external modem, our serial port might have some issues. Maybe routers on the other end of the line feel a little bit congested, so they decide to hold onto our packets for an extra 25ms, on top of the time it takes them to process packets normally. Perhaps some rare interdimensional phenomenon slows down the speed of light in a zone near the middle of the telephone line (maybe they're filming a *Star Trek* episode there or something).

All these things and many more will increase our suffering. Analyzing them closely is beyond the scope of this article because they are so diverse and unpredictable. However, we may take comfort in the knowledge that, generally, their effects will be less drastic than the phenomena that we've already looked at.

Solutions?

We've looked at delays caused by the atomic nature of operations such as rendering, and we've looked at delays caused by serial communication over a modem. We've seen that both these types of delays are influenced by the behavior of the game software.

The amount of lag caused by atomic software operations is high. However, it's also dependent on frame time, so if we can get our clients and servers running at a very high frame rate, the problem will go away. However, there are economic pressures that drive frame rate down (the need to have graphics that are more impressive than those of other games and the need to pack as many people as possible onto each server machine). So it will be constructive to think of other ways of eliminating this software-induced latency. Here, we will present some ideas and shoot most of them down.

Q: Can we have the client read player inputs more than once per game cycle? That way, we could detect input earlier and reduce latency, right?

A: Yes, but no, but yes. Looking at the simple case of a single-player game, if we poll for events more often and update our motion simulator after each poll, the client can respond to inputs sooner, thus reducing influence lag. However, observation lag increases to pick up the slack, and the total amount of lag remains the same. There is a trick that can be exploited, however. In a client/server architecture, the server acts as a parallel processor that the client can farm events off onto while it's waiting on its own observation lag. In an ideal world, an input event such as a keystroke would cause an interrupt, immediately stopping our client long enough for it to put together a packet, which is sent to the server without delay. Then the client resumes its normal processing. This way, we'd eliminate an entire 0.5*c* of influence lag. On many platforms, we can't use an interrupt, so we'd settle for polling several times per update. Last we checked, though, reading the joystick on a PC was so painfully slow that it was a bad idea to do it even once per cycle.

Q: What if we had a multiprocessing machine for the client? Could we use two processors to render scenes in parallel, issuing them at alternating intervals and reducing observation lag?

A: Yes, we can reduce lag this way, but not as much as we might hope. If we set up two processors rendering frames that are phase shifted from each other by 50 percent of the frame time (Figure 8), we can reduce the amount of time for which an event has to stall before it can enter a rendering cycle. In fact, if the time that it takes for one processor to render one frame is *c* (and therefore, the client's display frame time is 0.5*c*), then we've reduced the expected stall time to handle an event from our original 0.5*c* down to 0.25*c*. However, once the event enters the rendering process, it still takes 1.0*c* to be drawn. Since a new frame is being issued every 0.5*c*, it now takes two frames for each processed event to become visible, so observation lag remains the same. That seems weird, but that's how this pipelined stuff goes.

Using this technique, we can eliminate 0.25*c* of the observation lag induced on incoming messages (I'm assuming that we already used the previous trick to reduce the influence lag on keyboard messages, so this technique has no effect on that). However, alternating the rendering job between

Lockstep: The Ultimate Networking Scheme?

Throughout this article, we have assumed a networking model in which clients and servers operate independently in an unsynchronized fashion, without any network entity waiting on another to proceed. As we have seen, this assumption leads to added latency, because messages are received by the clients and servers at inopportune times.

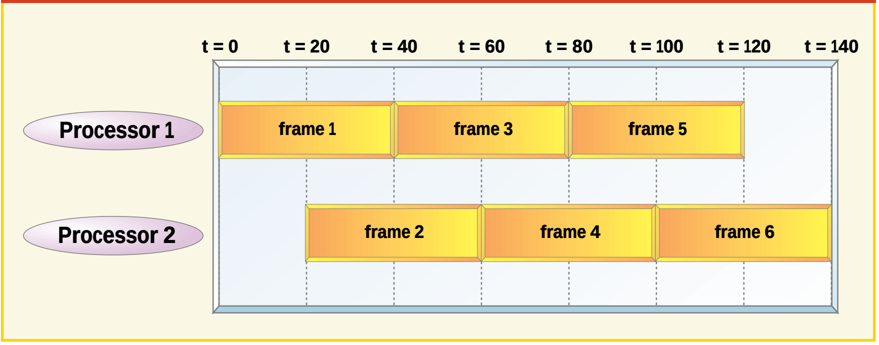
But there is a more primitive form of networking known as lockstep. This is the type of networking used by DOOM and other early commercial Internet games. These games typically use no server; instead each client communicates to all the others peer-to-peer, and each client simulates the state of the entire world privately.

Once per cycle, each client sends a message to all other clients describing events that have occurred that cycle. Each client will pause until it has received up-to-date messages from all other clients, at which point it will update its world state, draw the next frame, and repeat the cycle.

Under this scheme, each client's frame rate is locked to that of the slowest machine in the game (plus network lag). But the clients run in a synchronized manner, which means that much of the cycle-induced lag that we've discussed throughout the article just magically disappears.

In reality, this isn't appropriate for most games because first, the player with the slowest computer makes everyone else in the game suffer, and second, real-world communications problems make everyone in the game use words such as "suck." But it's amusing to think about.

FIGURE 8. *A two-processor machine runs the client program, with the job of rendering split between the two processors. This allows us to handle incoming messages a little bit sooner.*



multiple processors may be a tricky task because many graphics libraries and device drivers are not threadsafe. Even if they were, we might end up trying to render two different scenes on the same accelerator hardware at the same time, which isn't going to be a possibility anytime soon. So if our rendering is fill-limited, this whole idea is probably a wash.

If we had a rendering cycle that required some intense scene setup computation before any polygons were ever output, we could have one processor doing the scene setup stuff, then pass the result to the other processor, which would do the polygon outputs. Thus, each processor would always be doing the same job.

Q: Can we reduce lag on the server end by sending messages to clients about events as soon as they happen? For example, if a rock bounces off a wall in the middle of our simulation, might we interrupt the simulation to tell clients about the event, then continue? (This is analogous to sending out keystrokes immediately from the client.)

A: This is unlikely because of the limited bandwidth available for communicating to the client. With worlds of any complexity, there will be too much going on for us to inform each client of all events. A bandwidth-optimizing network scheme will look at events that have just occurred and decide which are most important for each client to know about. In order to compare events to see which are more important, we need a suitable selection of events to choose from, which means we need to wait for those events to occur. This implies the sort of per-cycle batch processing that we've already been assuming.

Q: Can't we reduce all that protocol overhead that slimes up our messages?

A: As individual game programmers, there's not much we can do. One big step would be an extension to PPP that allows compression of UDP/IP headers. There is no reason why PPP can't just see that we're throwing a bunch of UDP packets at the same destination, then negotiate away most of the headers as constant. It looks like we could cut the headers from 28 bytes to 5 bytes, if we're willing to play some checksum and length counter tricks. So if you're in the position to harangue someone working on PPP standards, bug them about this.

Recommendations

Now let's recap the basic ideas that we've looked at in the form of recommendations for future work:

1. We should design games with frame rates as high as possible. Even better, we should discover brilliant new computing paradigms that allow us to write software that isn't cycle-based. Polling is bad; event-drivenness is good!

2. We should take pains to handle events as soon as they occur, rather than waiting for convenient processing times. If we're locked into a system of cycles and polling, then we should poll many times per cycle for important inputs, handling them as soon as we see them, if possible.

3. We should make our network messages small, but not too small.

4. We should attempt to transmit messages evenly over time. As a random example, it may be better to send two updates to a client every server cycle, instead of four every two server cycles. Even though the former takes more bandwidth, latency and variance will probably be lower.

5. Let's exercise care when designing the form of messages that our game will send most often, thus ensuring that they won't be troublesome to lower-level protocols. Let's not design messages that contain bytes such as 0x7d all over the place, causing PPP to have a fit. ■

FOR FURTHER INFO

For more Internet protocol information than you can shake a stick at, see *TCP/IP Illustrated, Volume 1* by W. Richard Stevens (Addison-Wesley, 1994).

For a plate full of warm statistics goodness (and some clues about where that damn $\frac{\sqrt{3}}{6}$ comes from) see *Applied Statistics for Engineers and Physical Scientists*, Robert V. Hogg and Johannes Ledolter (Macmillan, 1992).

For a simple explanation of modem start and stop bits, partake thee of Dr. Joseph Williams' tasty slides at http://lamar.colostate.edu/~drj/Asynchronous_Communication.

Thorough documents describing telecommunications standards can be obtained, for proper amounts of filthy lucre, at www.eia.org and www.itu.ch.