Gamasutra - Designing Fast-Action for the Internet

As Internet-based video gaming services bring more and more games online, you may be wondering whether your super-speedy, eye-wearying, twitch-action game can cope with the Net's performance limitations. Wonder no longer. Given some careful design (or redesign) with the Internet's limitations in mind, your game might very well be playable over the Net.

This article focuses on network issues and techniques for making fast-action video games playable via modem on the Internet. Although there are some definite benefits to working with a specialized online game service (particularly in the administrative areas of player rendezvous, billing, tournament organizing, ranking, distribution, and maintenance, and in the technical areas of richer APIs and better network performance), the discussion here will apply to designing games for the Internet in general.

Network Performance

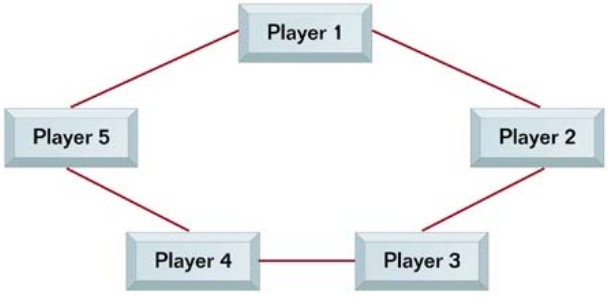
For fast-action games, network performance is the largest technical issue. We typically characterize network performance by two measures: bandwidth and latency.

Network bandwidth is the amount of data the network can transfer in a given time. In contrast, network latency is the time it takes a data packet (imagine an infinitesimally small packet) to travel across the network from sender to receiver. In short, latency represents a network link's built-in delay or lag, but bandwidth represents the network's throughput or data flow rate.

Network bandwidth and latency place critical constraints on fast-action online games. Bandwidth limits the scalability of a game, but latency limits a game's responsiveness. For example, bandwidth may limit the maximum supportable number of players in a game (since each additional player incurs additional data traffic), but high latencies may cause combatants in, say, a hand-to-hand fighting game to see significant delay between the initiation and visible result of an action.

As we'll see, bandwidth issues can often be resolved after a game is written (for example, by compressing data, prioritizing data, limiting the number of players, and so on). Latency issues, however, must be handled by design--preferably before the game is implemented.

Figure 1. Ring Topology



When we measure bandwidth and latency, we measure these along one link of a communications network. In a multiplayer game, there are typically many communications links, and any discussion about bandwidth and latency deserves some attention to the topology of the network. If you implement a ring-shaped network, for example, as shown in Figure 1, the latency from player 1 to player 4 is the sum of the latencies along several links in the network. The bandwidth is similarly limited by all the links between source and destination. The veteran Internet tank game, Bolo, uses this sort of a ring configuration.

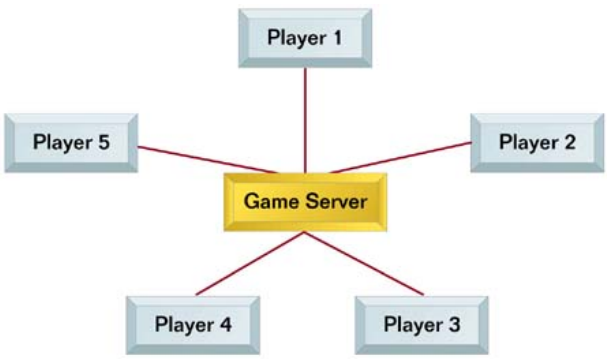
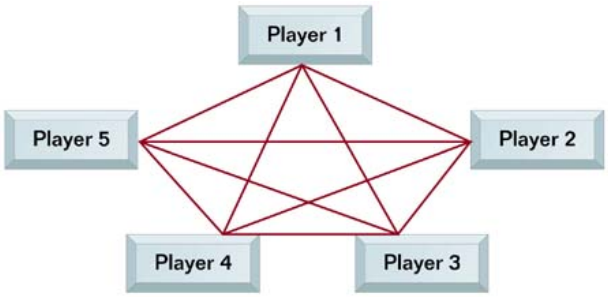


Figure 2. Star Topology

Generally speaking, you will be better served in terms of bandwidth and latency by a star configuration in which all game clients connect to a centralized, high-bandwidth server, like the one shown in Figure 2

In this configuration, the server maintains a connection to each game client and forwards data sent by one client to another. In a player-to-player message, bandwidth is limited by only two links of this network and is most likely constrained by the game players' modems. Latencies between any two players are similarly dependent on only two links of the network and are not, as in the ring configuration, equal to the sum of latencies along several serialized links. In the common case in which all players are dialed into the Internet via a modem dial-up connection, if the server can be positioned between (in latency space) all the players, the star network achieves near-optimal latency and bandwidth conditions.

Figure 3. All-to-All Topology



Another possible network configuration is an everyone-to-everyone topology, shown in Figure 3. This configuration might (but does not necessarily, as we'll see from the discussion of modem latencies later) achieve lower latencies between players, but comes at the cost of degraded bandwidth (since 28.8k modems don't perform as well in full-duplex) and added complexity in communications and player-management.

Reasonable Bandwidth Expectations

Assuming a star configuration (which is the most commonly used topology in today's online gaming services), network bandwidth is limited primarily by each player's modem speed and quality of telephone connectivity. When playing over the Internet using TCP or UDP, there is also some overhead for per-message header information. This overhead is typically 6 bytes per packet for TCP (with Jacoben PPP header compression enabled) and 34 bytes for UDP. Assuming a 14.4k modem, after allowing ample room for overhead, retransmits due to packet loss, and other in-band data (such as speech), and after planning to under-use the network to improve latency characteristics, I like games to target sending and receiving no more than 1,000 bytes per second.

A bit of very simple math will reveal that bandwidth is, in fact, a scarce resource. Consider this scenario. Suppose you plan to send 20 bytes of data at a rate of six times per second. Most of us would consider this already pretty lean. How many players using 14.4k modems can your game support? Well, each player will send 120 bytes per second. And, supposing he or she sends it to all the other players (a possibly over-conservative supposition), data can only be sent to a maximum of about eight other players. Thus, the game is limited to nine players. And this is if we only send a very careful 20 bytes of data!

Of course, sending data through an intelligent server might ease the bandwidth load. For example, a multicasting server could perform a broadcast on behalf of the player--freeing outbound bandwidth. Nevertheless, inbound bandwidth from other players will remain a limiting factor. Alternatively, to reduce the inbound bandwidth requirements of each client, a game server with game-specific intelligence could be used to filter which messages are sent to which clients. In any case, how much data you send and how the sent data affects bandwidth definitely must be carefully monitored and controlled.

Reasonable Latency Expectations

In contrast to bandwidth, there are many and various sources of network latency. If you've ever waited at the Netscape status line item that says "Host contacted. Waiting for reply", you've already had some experience with real-world latency. Round-trip latencies in the real world vary from a minimum of 120 to 160 msec to 5 seconds or more, depending a lot of variables. Due to the unreliable and inherently heterogeneous nature of the Internet, there are no upper bounds on Internet latencies. When I work with game companies, I like to design for a 200 to 250 msec round-trip latency and tolerate gracefully the rare instances when latencies might hit 1 second.

The ping tool is a good way to measure your latency to another machine. Try typing ping www.servername.com while dialed in to an ISP. Keep in mind that ping measures round-trip (that is, bidirectional) latency. Table 1 lists sources of latency at various layers of the communications medium.

You may be wondering why the best case round-trip latency for a dial-up player is so high: 120 to 160 msec. In most computer applications, the speed of the computer far surpasses the response times of humans, but here, we're talking about a tenth of a second, which borders on human sensibilities.

Most of this baseline latency occurs in the modem itself. Whether it's the buffering in the modem that occurs before sending data along, the V.42 protocol packetizing bits, the modem's Trellis coding algorithm, the 16550 UARTs, or TCP software buffering, I can't say exactly. Empirically, however, I've observed that each use of the modem simply incurs a 30 to 40 msec latency hit.

Considering that there are four modems involved in a round-trip ping (your modem outbound, ISP's modem

inbound, ISP's modem outbound, your modem inbound), that implies 120 to 160 msec as a best-case round-trip latency for dial-up users. Incidentally, we've started to talk about round-trip latencies partly because ping gives us results in round-trip numbers, but more importantly because, in a multiplayer dialup game, sending data from one player to another similarly involves four modems and thus yields latencies comparable to round-trip times.
Another significant cause for modem latency is the actual time spent in transmitting and receiving a message of any meaningful size. For example, if we are to measure modem latencies specifically as the time between the first byte of a message being sent and the last byte being received, there is in fact some nonzero time it takes to deliver the contents of the message. For a 14.4k modem, it's 1/(14400/8) = .56 msecs per byte of data.

However, again, four modems are involved in player-to-player communications, so you actually incur a latency of roughly 2 msecs per byte of data you deliver For example, a 26-byte message will incur an additional 56 msecs of round-trip latency on a 14.4k modem. This effect is slightly mitigated by modem compression, especially for larger packets of data, and is also mitigated on 28.8k modems. Still, time spent sending the data is significant.

If 120 to 160 msec is the baseline round-trip latency, why do ping results frequently show 300 msec to 1 second of round-trip latency? The biggest causes of latencies this large are routers. Without going into too much detail, suffice it to say that routers can easily cause hundreds of msecs of additional latency:

- Routers accumulate buffers of data before forwarding (thus incurring a latency penalty)
- A router's frequent routing table cache misses results in hard drive seek and read times
- Routers drop packets when overloaded (thus incurring, for TCP, at least a 3t+s latency penalty, where t is the one-way, end-to-end latency--a latency that includes two modems--and s is the inter-message delay. Note that 3t+s can easily be in the 450-msec range)
- Improperly managed networks commonly hit 10% dropped packet rates

One solution to this set of problems is to closely manage your network, deploy a geographically distributed set of game servers on an Internet backbone that is an ATM, switched, frame-relay network (and all but eliminates routers from the game traffic), and use permanent virtual circuits to assure prioritized data delivery and lower packet loss rates; this is the approach Mplayer takes. On the wider Internet, 25 to 700 msec latencies are unavoidable, and you will need to deal with them in your gameplay.

Modem buffers 16650 UARTs Interrupt handlers Modem compression algorithms	Consider turning off modem compression or error correct ion. No other good solutions.
Modem error correction algorithms	Consider turning off modem compression or error correct ion. No other good solutions.
Sizes of message (2 msec round-trip latency per byte of message, slightly mitigated by modem compressi on)	Send smaller messages.
PPP layer buffering	None.
TCP layer buffering	Carefully tune TCP settings, which are set by the setsockop t() call.
Thread priorities on both client and server	Use threads carefully and manage CPU use.
Speed of light (~60 msec for round-trip cross country, as the crow flies)	Sorry, we haven't beaten Einstein yet.
Store and forward routers	Use a well-managed network. Avoid routers.
Over-used routers causing packets to be dropped and retransmitted	Use a well-managed network.
Number of routers between source and destination	Use a strategically located, distributed set of servers.

Overcoming Latencies

It turns out that for round-trip latencies as high 250 msecs, appropriately designed fast-action games can play quite well over the Internet. Design is the key word here. Tolerance for 250 msec latencies really needs to be designed as part of the communications model of your game if you want to offer a rewarding fast-action experience. Optimizing for latencies as an afterthought will only lead to slipped schedules. One key insight and a variety of techniques can help your game design to compensate for and accommodate the Internet's latencies.

Network latency causes players' views of the world to be imperfectly synchronized, but, you know that's O.K.,

at least for short periods of time. Specifically, it's O.K. for different players to have different perceptions of the world, and it's even O.K. to not reconcile all the differences. The only thing you need to reconcile is the differences that are important to and relevant to interacting players (for example, the missile that one player fires at another). If you perform this reconciliation smoothly and gracefully, players will perceive no latency.

With 250 msecs of round-trip latency between players, you have just enough time to reconcile the outcome of an important event before human sensibilities perceive any lag. Perhaps an example will best clarify this point. In a car race, player A overtakes player B. Although you may instinctually try to synchronize the two computers' views of the world before allowing gameplay to continue, it's actually quite acceptable to let them remain out of sync for quite a long time, in fact, probably for as long as a second. Neither player will notice any difference in game-play! As long as the leadership position is reconciled before too much time passes and particularly before either player crosses the finish line, the network latency remains completely hidden to both players.

There are two forms of reconciliation: soft-reconciliation and dead-reckoning. During most of a car racing game, soft-reconciliation of the world view is acceptable. That is, during most of the game, players don't ever have to be exactly synchronized; they can afford to be loosely synchronized. However, after passing the finish line, dead-reckoning is necessary so that all players agree exactly upon who won and who lost. To minimize the effect of latency on a game, try to minimize the number of events in a game that require dead-reckoning. Also, use some of the following techniques to hide the minimum of 200 to 300 msec latency inherent in a dead-reckoning interaction.

Probably the most general and most important technique to enable graceful handling of latency in the 200 to 300 msec range is, whenever possible, to decouple the game communications from the game play. For example, decouple game communications from your graphics frame rate, keyboard, or joystick input rate, and from anything else that affects the pace of the game. Using a separate thread for communications and FIFO queues to communicate with that thread, implementing a communications model and user-interface that doesn't depend on messages from other players to continue, and taking advantage of other techniques described in Table 2 will decouple the network from your game and enhance the overall responsiveness of the game.

Also, regarding latencies, expect and plan for occasional network delays of up to a second. Use the rule of thumb that, even if you completely lose network connectivity for some period of time, your game UI should still respond at some level and, to the extent possible, gameplay should continue. Also, gameplay must be able to recover and continue naturally after such a network outage or abort gracefully if such an outage persists. As with latency, there are a variety of approaches to conserving bandwidth. Many are listed in Table 3.

Table 2. How to Overcome Latency

Decouple the communications from your game:

- Place blocking communications calls in a separate thread or use asynchron ous communications calls.
- Never await incoming data from another player before allowing game play to continue.
- Use predicting, interpolating, and reconciling later to improve game play.
- When using prediction, transmit not just position information, but also veloc ity and acceleration.
- Latch or queue user input (for example, keystrokes) until the next time com munications data is to be sent. If you use the more traditional method of ch ecking the keyboard at regular intervals during the game loop and perform this checking only at moments before sending data to the network, you will be prone to missing user input.

Hide latencies in other game elements:

- Schedule events in the future if you want them to happen simultaneously for all users.
- Require multiple shots to kill someone, and minimize the number of all-or-n othing, deterministic, latency-sensitive events.
- Make rockets take a long time in the air (and reconcile the outcome while t he rocket is flying).
- Require time to move from one location to another (don't allow any instanta neous teleporters).
- Make players, ships, and other objects move in ways that facilitate predictio n. For example, build inertia into the way entities in your game move.
- Use your imagination to incorporate latency within the context of your game concept.

TCP or UDP?

Now we must address whether to use TCP or UDP for your game traffic. Here are some of the considerations. UDP messages are sent unreliably (that is, no retransmits to correct errors, as in TCP). Also, UDP packets might be received out of order by the sender. UDP messages also have a higher packet overhead than TCP (34 bytes of overhead for UDP vs. 6 bytes for TCP when header compression is turned on). The greatest advantage for using UDP is that a dropped UDP packet doesn't kill latency. TCP, in contrast, offers guaranteed, in-order delivery, but you pay the price by experiencing very high latencies if a packet is dropped.

If you are considering UDP or even TCP in a multicasting environment (where packets might be lost in the multicast server), you must consider the possibility of a dropped packet. One tip is this: when sending data such as "current health" or "current amount of gold" in a message, always send the value itself and not a delta or change between the previously sent value. This approach offers a simple and fast (low-latency) way to recover from a lost message.

Table 3. Techniques to Conserve Bandwidth

- Compress your game and speech data.
- Send data asynchronously--that is, only when needed, rather than at regular intervals.
- Prioritize your data; then, think of your bandwidth resource as a fixed resou rce and budget according to your priorities. This way, if there is too much d emand for bandwidth, you degrade game play gracefully. Prioritizing data a nd allocating bandwidth accordingly also allows for better interoperation b etween 14.4k, 28.8k, ISDN, and other qualities of bandwidth.
- Use a multicasting or customized game server.
- Cache information about other players locally.
- Cluster repetitious game events (for example, a machine gun firing) into a s mall number of messages, rather than hundreds of messages.
- Have AI live on the local client machines, and send over only the random se eds and checksums to control the AI.
- Use teams and consider having players on the same team divide responsibi lities so only one player needs to send data on behalf of the entire team. Fo r example, multiple players can ride in a single tank that is controlled by just one player.
- Don't allow everyone to see everyone else. One simple way to do this is to subdivide the game world (for example, into sectors or rooms) and restrict communication among players based on the subdivisions.

In Sum

It's not easy to write an Internet-enabled, fast-action game. Given the latency and bandwidth limitations of the Net and modems, careful attention to the network is necessary to provide a rewarding action game experience. Nevertheless, it is possible. Moreover, the opportunity of millions of fans eagerly seeking to play your title over the Net is huge.

The Net adds significant, real value to the game experience: most would agree that the chance to compete against a live, intelligent human is far more rewarding and engaging than playing the computer AI. Is the reward great enough to make you go the extra mile and make it possible? That, given the technical challenges, is left entirely in your hands.

Yu-Shen Ng develops tools and services for game developers at Mpath Interactive, a company running Mplayer, an online service for Internet videogaming. He can be reached at yushen@mpath.com, http://www.mpath.com, or http://www.mplayer.com