CMSC 425: Lecture 22
Multiplayer Games and Networking

Reading: Today's lecture is from a number of sources, including lecture notes from University of Michigan by Sugih Jamin and John Laird, and the article "Network and Multiplayer," by Chuck Walters which appears as Chapter 5.6 in *Introduction to Game Development* by S. Rabin.

Multiplayer Games: Today we will discuss games that involve one or more players communicating through a network. There are many reasons why such games are popular, as opposed say to competing against an AI system.

- People are "better" (less predictable/more complex/more interesting) at strategy than AI systems
- Playing with people provides a social element to the game, allowing players to communicate verbally and engage in other social activities
- Provides larger environments to play in with more characters, resulting in a richer experience
- Some online games support an economy, where players can buy and sell game resources

Multiplayer games come in two broad types:

Transient Games: These games do not maintain a persistent state. Instead players engage in ad hoc, short lived sessions. Examples include games like *Doom*, which provided either head-to-head (one-on-one) or death-match (multiple player) formats. The are characterized as being fast-paced and providing intense interaction/combat. Because of their light-weight nature, any client can be a server.

Persistent Games: These games are run by a centralized authority that maintains a persistent world. Examples include *massively multiplayer online games* (MMOGs), such as "World of Warcraft" (more specifically an MMORPG), which are played over the Internet.

Performance Issues: The most challenging aspects of the design of multiplayer networked games involve achieving good performance given a shared resource (the network).

Bandwidth: This refers to the amount of data that can be sent through the network in steady-state.

Latency: In games where real-time response is important, a more important issue than bandwidth is the responsiveness of the network to sudden changes in the state. Latency refers to the time it takes for a change in state to be transmitted through the network.

Reliability: Network communication occurs over physical media that are subject to errors, either due to physical problems (interference in wireless signals) or exceeding the network's capacity (packet losses due to congestion).

Security: Network communications can be intercepted by unauthorized users (for the purpose of stealing passwords or credit-card numbers) or modified (for the sake of cheating). Since cheating can harm the experience of legitimate users, it is important to detect and minimize the negative effects of cheaters.

Of course, all of these considerations interact and trade-offs must be made. For example, enhancing security or reliability may require more complex communication protocols, which can have the effect of reducing the useable bandwidth or increasing latency.

Network Structure: Networks are complex entities to engineer. Let us describe the basics of network structure. (For more information, take a course such as CMSC 417.) In order to bring order to this topic, networks are often described in a series of layers, which is called the *Open System Interconnect* (OSI) model. Here are the layers of the model, from lowest (physical) to the highest (applications).

Physical: This is the physical medium that carries the data (e.g., copper wire, optical fiber, wireless, etc.)

Data Link: Deals with low-level transmission of data between machines on the network. Issues at this level include things like packet structure, basic error control, and machine (MAC) addresses.

Network: This controls end-to-end delivery of individual packets. It is responsible for routing (path determination and logical addressing) and balancing network flow. This is the layer where the Internet Protocol (IP) and IP addresses are defined.

Transport: This layer is responsible for transparent end-to-end transfer of data (not just individual packets) between two hosts. This layer defines two important protocols, TCP (transmission control protocol) and UDP (user datagram protocol). This layer defines the notion of a *net address*, which consists of an IP address and a port number. Different port numbers can be used to partition communication between different functions (http, https, smtp, ftp, etc.)

Session: This layer is responsible for establishing, managing, and terminating long-term connections between local and remote applications (e.g., logging in/out, creating and terminating communication sockets).

Presentation: Provides for conversion between incompatible data representations based on differences system or platform, such as character encoding (e.g., ASCII versus Unicode) and byte ordering (highest-order byte first or lowest-order byte first) and other issues such as encryption and compression.

Application: This is the layer where end-user applications reside (e.g., email (smtp), data transfer (ftp, sftp), web browsers (http, https)).

The OSI model is illustrated in Fig. 1. While the OSI model is an international standard, it is not the model used in the Internet. The Internet is based on a similar, but older model called *TCP/IP*.

TCP/IP was developed during the 1960s as part of the US Department of Defense's Advanced Research Projects Agency (ARPA) effort to build a nationwide packet-data network. It was

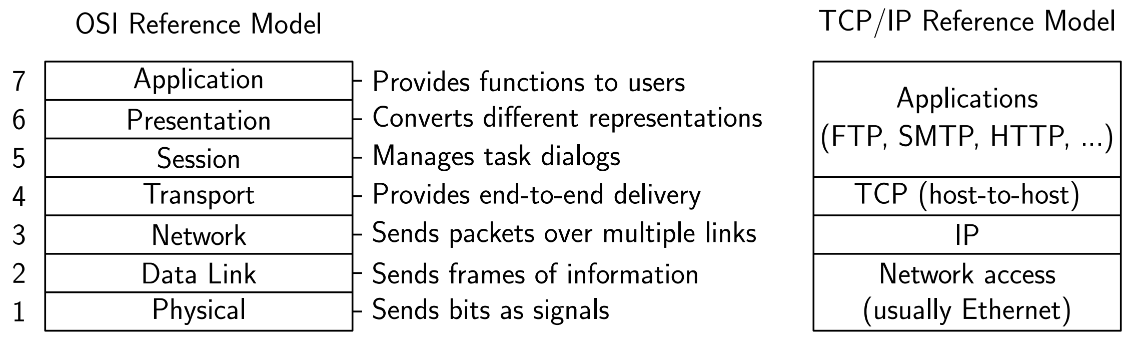
OSI Reference Model
7 Application - Provides functions to users
6 Presentation - Converts different representations
5 Session - Manages task dialogs
4 Transport - Provides end-to-end delivery
3 Network - Sends packets over multiple links
2 Data Link - Sends frames of information
1 Physical - Sends bits as signals
TCP/IP Reference Model
Applications (FTP, SMTP, HTTP, ...)
TCP (host-to-host)
IP
Network access (usually Ethernet)

Fig. 1: The Open System Interconnect (OSI) Model. (Courtesy of Ashok Agrawala's notes.)

first used in UNIX-based computers in universities and government installations. Today, it is the main protocol used in all Internet operations.

If you are programming a game that will run over the internet, you could well be involved in issues that go as low as the transport layer (as two which protocol, TCP or UDP, you will use), but most programming takes place at the application level.

Packets and Protocols: Online games communicate through a *packet-switched network*, like the Internet, where communications are broken up into small data units, called *packets*, which are then transmitted through the network from the sender and reassembled on the other side by the receiver. (This is in contrast to direct-link communication, such as through a USB cable or circuit-switched communication, which was used for traditional telephone communication.)

In order for communication to be possible, both sides must agree on a *protocol*, that is, the convention for decomposing data into packets, routing and transferring data through the network, and dealing with errors. Communication networks may be unreliable and may connect machines having widely varying manufacturers, operating systems, speed, data formats. Examples of issues in the design of a network protocol include the following:

Packet size/format: Are packets of fixed or variable size? How is data to be laid out within each packet.

Handshaking: This involves the communication exchange to ascertain how data will be transmitted (format, speed, etc.)

Acknowledgments: When data is received, should its reception be acknowledged and, if so, how?

Error checking/correction: If data packets have not been received or if their contents have been corrupted, some form of corrective action must be taken.

Compression: Because of limited bandwidth, it may be necessary to reduce the size of the data being transmitted (either with or without loss of fidelity).

Encryption: Sensitive data may need to be protected from eavesdroppers.

Later in this lecture we will discuss two commonly-used protocols that run at the Transport layer of the OSI model, TCP and UDP. Before doing this, let us discuss the main issue that arises in online games, latency.

The Problem of Latency: Recall that latency is the time between when the user acts and when the result is perceived (either by the user or by the other players). Because most computer games involve rapid and often unpredictable action and response, latency is arguably the most important challenge in the design of real-time online games. Too much latency makes the game-play harder to understand because the player cannot associate cause with effect. Latency also makes it harder to target objects, because they are not where you predict them to be. Also, as we will see in future lectures, latency can be exploited in some cheats in online games.

Note that latency is a very different issue from bandwidth. For example, your cable provider may be able to stream a high-definition movie to your television after a 5 second start-up delay. You would not be bothered if the movie starts after such a delay, but you would be very annoyed if your game were to impose this sort of delay on you every time you manipulated the knobs on your game controller.

The amount of latency that can be tolerated depends on the type of game. For example, in a Real-Time Strategy (RTS) game, below 250ms (that is, 1/4 of a second) would be ideal, 250–500ms would be playable, and over 500ms would be noticeable. (Recall that "ms" refers to 1/1000 of a second.) In a typical First-Person Shooter (FPS), the latency should be smaller, say 150ms would be acceptable. In car racing game or other game that involves fast (twitch) movements, latencies below 100ms would be required. Latencies in excess of 500ms would make it impossible to control the car. Note that the average latency for the simplest transmission (a "ping") on the internet to a geographically nearby server is typically much smaller than these numbers, say on the order of 10–100ms.

There are a number of sources of latency in online games:

Frame rate latency: Data is sent to/received from the network layer once per frame, and user interaction is only sampled once per frame.

Network protocol latency: It takes time for the operating system to put data onto the physical network, and time to get it off a physical network and to an application.

Transmission latency: It takes time for data to be transmitted to/from the server.

Processing latency: The time taken for the server (or client) to compute a response to the input.

There are various techniques that can be used to reduce each of these causes of latency. Unfortunately, some elements (such as network transmission times) are not within your control.

Coping with Latency: Latency can be reduced in various ways (more servers placed closer to players, faster machines), but it cannot be eliminated. What can the game programmer do to conceal latency from the player? Any approach that you take will introduce errors in some form. The trick is how to create the illusion to your user that he/she is experiencing no latency.

Sacrifice accuracy: Given that the locations and actions of other players may not be known to you, you can attempt to render them approximately. One approach is to ignore the time lag and show a given player information that is known to be out of date. The second is to attempt to estimate (based on recent behavior) where the other player is at the present time and what this player is doing. (See the material on dead-reckoning below.) Both approaches suffer from problems, since a player may make decisions based on either old or erroneous information.

Sacrifice game-play: Deliberately introduce lag into the local player's experience, so that you have enough time to deal with the network. For example, a sword thrust does not occur instantaneously, but after a short wind-up. Although the wind-up may only take a fraction of a second, it provides the network time to send the information through the network that the sword thrust is coming.

Dealing with Latency through Dead Reckoning: One trick for coping with latency from the client's side is to attempt to estimate another player's current position based on its recent history of motion. Each player knows that the information that it receives from the server is out of date, and so we (or actually our game) will attempt extrapolate the player's current position from its past motion. If our estimate is good, this can help compensate for the lag caused by latency. Of course, we must worry about how to patch things up when our predictions turn out to be erroneous.

- Each client maintains precise state for some objects (e.g. local player).
- Each client receives periodic updates of the positions of everyone else, along with their current velocity information, and possibly the acceleration.
- On each frame, the non-local objects are updated by *extrapolating* their most recent position using the available information.
- With a client-server model, each player runs their own version of the game, while the server maintains absolute authority.

Inevitably, inconsistencies will be detected between the extrapolated position of the other player and its actual position. Reconciling these inconsistencies is a challenging problem. There are two obvious options. First, you could just have the player's avatar jump *instantaneously* to its most recently reported position. Of course, this will not appear to be realistic. The alternative is to smoothly *interpolate* between the player's hypothesized (but incorrect) position and its newly extrapolated position.

Dealing with Latency through Lag Compensation: As mentioned above, dead reckoning relies on extrapolation, that is, producing estimates of future state based on past state. An alternative approach, called *lag compensation*, is based on *interpolation*. Lag compensation is a server-side technique, which attempts to determine a player's intention.

Here is the idea. Players are subject to latency, which delays in their perception of the world, and so their decisions are based on information that is slightly out of date with the current world state. However, since we can estimate the delay that they are experiencing, we can try to roll-back the world state to a point where we can see exactly what the user saw when they made their decision. We can then determine what the effect of the user's action would have been in the rolled-back world, and apply that to the present world.

Here is how lag compensation works.

(1) Before executing a player's current user command, the server:

   (a) Computes a fairly accurate estimate of the player's latency.
   (b) Searches the server history (for the current player) for the last world update that was sent to the player and received by the player (just before the player would have issued the movement command).
   (c) From that update (and the one following it based on the exact target time being used), for each player in the update, move the other players *backwards* in time to exactly where they would have been when the current player's user command was generated. (This moving backwards must account for both connection latency and the interpolation amount the client was using that frame.)

(2) Allow the user command to execute, including any weapon firing commands, etc., that will run ray casts against all of the other players in their interpolated, that is, old positions.

(3) Move all of the moved/time-warped players back to their correct/current positions

The idea is that, if a user was aiming accurately based on the information that he/she was seeing, then the system can determine this (assuming it has a good estimate of each player's latency), and credit the player appropriately.

Note that in the step where we move the player backwards in time, this might actually require forcing additional state information backwards, too (for example, whether the player was alive or dead or whether the player was ducking). The end result of lag compensation is that each local client is able to directly aim at other players without having to worry about leading his or her target in order to score a hit. Of course, this behavior is a game design tradeoff.

Reliability: Let us move on from latency to another important networking issue, reliability. As we mentioned before, in packet-switched networks, data are broken up into packets and then may be sent by various routes. Packets may arrive out of order, they may be corrupted, or they may fail to arrive at all (or after such a long delay that the receiver gives up on them). Some network protocols (TCP in particular) attempt to ensure that every packet is delivered and they arrive in order. (For example, if you are sending an email message, you would expect the entire message to arrive as sent.)

As we shall see, achieving such a high level of reliability comes with associated costs. For example, the user sends packets. The receiver acknowledges the receipt of packets to the sender. If a packet receipt is not acknowledged, the sender resends the packet. The additional communication required for sending, receiving, and processing acknowledgments can increase latency and use more of the available bandwidth.

In many online games, however, we may be less concerned that every packet arrives on time or in order. Consider for example a series of packets, each of which tells us where an enemy player is located. If one of these packets does not arrive (or arrives late) the information is now out of date anyway, and there is no point in having the sender resend the packet. Of course, some information is of a much more important nature. Information about payments or certain changes to the discrete state of the game (player X is no longer in the game), must be communicated reliably. In short, not all information in a game is of equal importance with respect to reliability.

Communication reliability is handled by protocols at the transport level of the OSI model. The two most common protocols are TCP (transmission control protocol) and UDP (user datagram protocol).

Transmission Control Protocol: We will not delve into the details of the TCP protocol, but let us highlight its major elements. First, data are transferred in a particular order. Each packet is assigned a unique *sequence number*. When packets are received, they are reordered according to these sequence numbers. Thus, packets may arrive out of order without affecting the overall flow of data. Also, through the use of sequence numbers, the receiver can determine whether any packets were lost. Second, the transmission contains *check-sums*, to ensure that any (random) corruption of the data will be discovered. The receiver sends acknowledgments of the receipt of packets. Thus, if a packet is not received, the sender will discover this and can resend it.

TCP also has a basic capability for *flow control*. If the sender observes that too many packets are failing to arrive, it decreases the rate at which it is sending packets. If almost all packets are arriving, it slowly increases this rate. In this way, the network will not become too congested.

Advantages:

- Guaranteed packet delivery
- Ordered packet delivery
- Packet check-sum checking (basic error detection)
- Transmission flow control

Disadvantages:

- Point-to-point transport (as opposed to more general forms, like multi-cast)
- Bandwidth and latency overhead
- Packets may be delayed to preserve order

TCP is used in applications where data must be reliably sent and/or maintained in order. Since it is a reliable protocol, it can be used in games where latency is not a major concern.

User Datagram Protocol: UDP is a very light-weight protocol, lacking the error control and flow control features of TCP. It is a *connectionless protocol*, which provides no guarantees of delivery. The sender merely sends packets, with no expectation of any acknowledgment. As a result, the overhead is much smaller than for TCP.

Advantages:

- Packet based—so it works with the internet
- Lower overhead than TCP in terms of both bandwidth and latency
- Immediate delivery—as soon as it arrives it goes to the client

Disadvantages:

- Point to point connectivity (as with TCP)
- No reliability guarantees

- No ordering guarantees
- Packets can be corrupted
- Can cause problems with some firewalls

UDP is popular in games, since much state information is nonessential and quickly goes out of date. Note that although the UDP protocol has no built-in mechanisms for error checking or packet acknowledgments, the application can add these to the protocol. For example, if some packets are non-critical, they can be sent by the standard UDP protocol. Certain *critical* packets can be flagged by your application, and as part of the packet payload, it can insert its own sequence numbers and/or check-sums. Thus, although UDP does not automatically support TCPs features, there is nothing preventing your application from adding these to a small subset of important packets.

Area-of-Interest Management: In large massively multiplayer games, it would be inefficient to inform every player on the state of every other player in the system. This raises the question of what information does a player need to be aware of, and how to transmit just that information. This is the subject of the topic of *area-of-interest management.* This subject is to networking what visibility is to collision detection. This is typically employed in large games, and so it is the server's job to determine what information each player receives.

There are two common approaches, grid methods and aura methods. *Grid methods* partition the world into a grid (which more generally may be something like a quadtree). Each cell is associated with the players and other entities that reside within this cell. Then, the information transmitted to a player is based on the entities residing within its own and perhaps neighboring grid cells.

One shortcoming of this method is that it neglects the fact that some entities may not correspond to individual points, but to entire regions of space. For example, a cloud of poisonous gas cannot be associated with a single point in space. The alternative is called an *aura method*, in which each entity is associated with a region of space, its *sphere of influence.* All players that lie within this region are provided information on this entity.