5.6 Network and Multiplayer

In This Chapter

Overview

This chapter introduces the concepts and terminology involved with network and multiplayer programming. It begins with an assessment of multiplayer game modes, followed by an exploration of network programming fundamentals, including network protocols, real-time data transfer, asynchronous environment guidelines, and game security.

Multiplayer Modes

Multiplayer games share a generic set of concepts, in addition to mode-specific details. This section surveys the common ground, describing three key differentiating factors of multiplayer design/implementation: *event timing*, *shared I/O*, and *connectivity*.

Event Timing

Games follow either a turn-based or a real-time *event-timing* model. Some games contain a mixture. In such cases, turn-based events take precedence due to their lock-step nature. Consequently, timing models influence design and implementation paths of various components.

Turn Based

Turn-based games restrict movement to a single player, making all other players wait for their turn; also referred to as *round robin*. Most board games and card games exhibit turn-based gameplay. These games tolerate high and/or variable latency and low-bandwidth conditions.

Real Time

Real-time games support simultaneous player interaction, often requiring arbitration to handle *race conditions*. Examples of such conditions include determining the first to cross the finish line, grab an object in the game world, or lose all health. A special category of real-time games, known as "twitch" games, relies on a constant flow of race conditions. All real-time games design around a rigid set of latency and bandwidth requirements, but twitch games tend to degrade with latencies above 150 ms, and become unplayable above 500 ms (0.5 seconds).

Shared I/O

Games run on a single computer often facilitate multiple players by sharing input and display systems. Players may share a single input device such as assigning different keys on a keyboard to each player, or simply passing the entire keyboard between turns in a turn-based game, or plug in additional input devices for exclusive use by each player. One could consider this a form of connected multiplayer. In fact, multiple input devices provide a good means to simulate players on a low-latency network. The next few sections describe models for sharing the displays.

Full Screen

A full-screen multiplayer game normally requires one of the following conditions:

Complete playfield visibility. Card games show the entire table. A checkers or chess game shows the entire board. A game with a virtual world such as a soccer playfield or a war game's battlefield must display the entire field. Without this constraint, one or more players may not see their game entities and subsequently fail in efforts to control them.

Player funneling. To facilitate multiple players in a snapshot of a larger game world, *player funneling* restricts players to stay within a virtual cage the size of the screen display. It's as if four people were each holding one corner of a blanket. If they all pull in opposite directions, the blanket stays put. If one person decides to move in the same direction of the person opposite him, the blanket moves in that direction. The popular role-playing game *Gauntlet* uses this technique to arbitrate player movement while scrolling through the game world.

Turn-based screen control. Since turn-based games only allow one person to move at a given time, the display only necessitates showing the current player's viewpoint. In this scenario, the display simply switches to the active player's camera.

Split Screen

With split-screen multiplayer, each player is allotted separate portions of the display to show personalized views of the game world. The following are common component separations required for each player:

- **Camera** (each player's point of view)
- **Cull data** (one for each screen since the point of view differs for each player)
- **Heads-Up Display, or HUD** (game stats relevant to each player)
- **Map data** (centered on current player)
- **Audio effects** (mixing required since normally only one audio system)

Splitting the screen incurs a performance hit due to the multiple render cycles. Updating each view during the render phase of the game loop keeps the display state consistent, the importance of which becomes relevant when two or more players exist in close proximity. One possible optimization, which breaks this consistency, involves round robin rendering. Only one view updates per render phase, and the updated view changes to the next player on the next render phase. In a four-player game, a hybrid of this could update two views each render phase. This helps keep the view consistent in the close proximity case; otherwise, each player will be three updates out of sync with one of the other views.

There are two standard methods for dividing the screen real estate: *viewports* and *render destinations*. Viewports render directly to the back buffer, and render destinations render to a texture placed on geometry, which then renders to the back buffer. Due to their performance benefits over render destinations, viewports experience greater acceptance. However, render destinations offer unique capabilities over viewports. The created textures map to geometry of any shape. This geometry may freely move in the virtual display, thus allowing varied size, rotation, translation, and overlap with transparency.

A hybrid option known as *windowed mode* may use either viewports or render destinations to display the contents within the window. This adds further performance degradation to the base modes, but on a PC, it allows players to drag their view

onto separate monitors, or otherwise customize their window layout. Usability studies suggest it is best to offer the player a few predesigned and playtested layouts to choose from; otherwise, the customizability tends to detract from gameplay because a new player doesn't necessarily know how to form a good layout to deal with particular game features.

Connectivity

The connection type determines latency and bandwidth. These two constraints then dictate game timing, number of controllable game entities (including players), and other game design elements. However, connected multiplayer games reuse many ideas from nonconnected multiplayer games. Some connected games use split-screen displays or player funneling. Others pass input data as if additional input devices were connected to the same computer.

Due to the lack of latency requirements, turn-based games work over a greater variety of connection mediums. In this case, data transfer need not be fast; the data just needs to get there intact eventually. For example, game moves saved to a file and then transferring that file by e-mail, FTP, or even saving it to a removable disk and walking it over to another computer are all acceptable means of turn-based game connectivity. The following categories offer real-time connectivity:

Direct link: Linking computers over a short connection normally guarantees low latency, while bandwidth depends on the medium. Popular cable links include a modified serial cable (a null modem cable) and a USB cable. Popular wireless links include infrared and Bluetooth. Each harbors specialized protocols to facilitate communications that tend to restrict to peering.

Circuit-switched network: The public phone networks provide an unshared direct connection or circuit. This maintains a consistent, low-latency medium, but is short on bandwidth and player distribution (only two player; call conference modem games never really took off). An Internet service provider (ISP) allows the circuit to attach to an Internet conduit (the modem at the ISP), which places the packet data traffic on the Internet. This solves the player distribution problem, but takes away the low-latency benefits of the direct circuit.

Packet-switched network. Data networks share virtual circuits that are created and released for each data packet. Network configurations vary in hardware, transmission medium, and protocols, and the Internet combines these smaller physical networks into a single large logical network, allowing people to play anybody from anywhere, at any time. However, the Internet suffers from a wide variance in bandwidth and latency. It is also less reliable than the public phone system.

Protocols

A *protocol* is an agreed-upon format for transferring data between devices. This format specifies some or all of the following methods:

- Packet length conveyance
- Acknowledgement methodology
- Error checking
- Error correcting
- Compression
- Encryption
- Control packet description

Packets

The logical transmission units of a protocol, otherwise known as *packets*, consist of two parts: a header section and a payload section. The header contains the format elements of the protocol. A protocol considers the payload as a Binary Large OBject (BLOB), which it does not modify; rather, it simply delivers it according to the terms of the protocol. The following demonstrates a simple packet structure:

```
struct packet
{
     // Header
     short PacketLength; // Length of this packet
     short PacketType;   // Control Information
     int   Checksum;     // Error Checking

     // Payload
     char [256] Blob;    // Higher layer protocol data
};
```

When creating packet structures, the structure may be formed such that it requires no special *serialization* (reformatting the data into a serial form). The following factors determine whether a packet structure requires serialization:

Pointers: Since pointers refer to local memory, the data pointed to needs to be serialized into a byte stream.

Abstract data types: ADTs commonly contain references, which require their extraction and placement in an array.

Byte alignment linkers: Default to word alignment for processor performance. To avoid this byte padding, use the following preprocessor directives:

```
#pragma pack (1) // Byte aligned, no padding
  // Add packet structures here
#pragma pack () // Set back to default alignment
```

Endian order: When building a game that connects across platforms, multibyte intrinsic types require *endian* synchronization. The Sockets API provides the following macros to place multibyte in the standardized *network order* (since routers need to inspect address variables): ntohs, ntohl, htons, and htonl. Following the same standard for endianess reduces confusion.

Specific intrinsic types: Use intrinsic types that have a specified width. For example, use "`__int32`" rather than "`int`," since the size of "`int`" differs on 32-bit and 64-bit CPUs.

Unicode strings: Start out using Unicode character strings at the beginning of a game development project to make localization easier at the end of the project. Otherwise, an additional conversion/serialization step needs to occur for cross-language packet exchange where one language uses 1-byte ASCII strings and the other uses 2-byte Unicode strings.

Request for Comments

Protocol specifications require distribution to get used. They also need to be constructively criticized or otherwise commented on to identify imperfections. From this need arose the Request for Comments [RFC] repository for new and existing protocol specifications. Most public Internet protocols have an associated RFC specification number associated with a detailed description on the RFC Web/FTP site.

Protocol Stack

The Open System Interconnect (OSI) specification formalizes interoperability between devices and software entities into logical layers. Figure 5.6.1 illustrates the flow of data between layers, pointing out common protocols that reside in each layer.

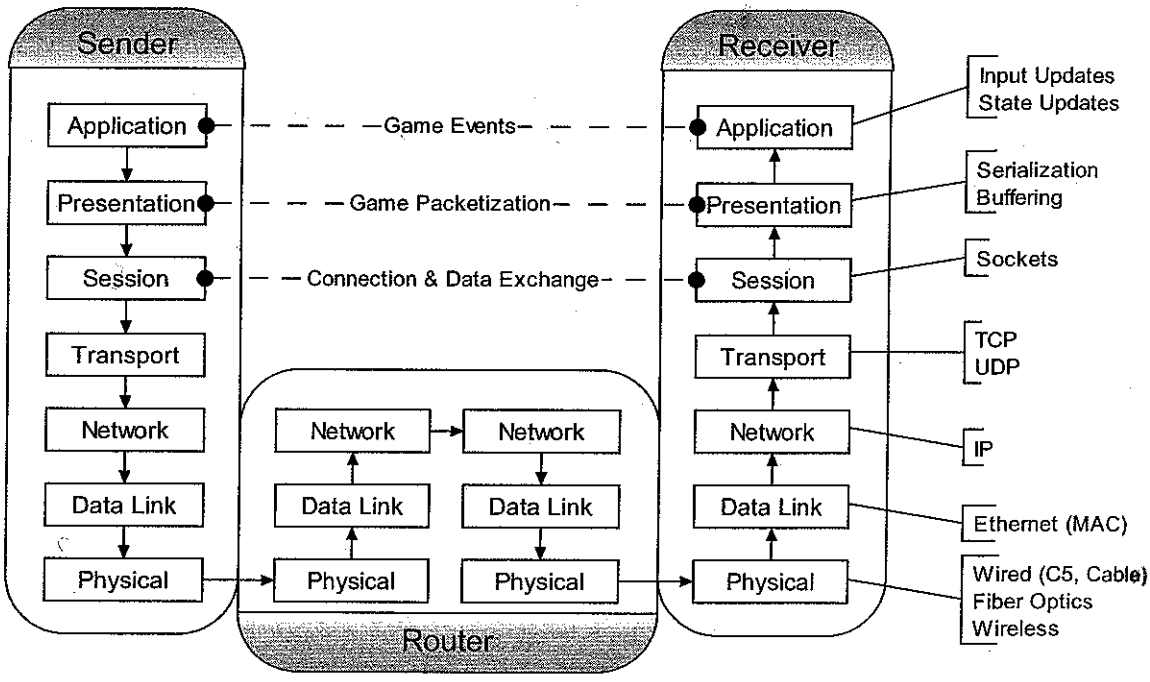


FIGURE 5.6.1 *The OSI model.*

The Internet model provides a variation of the OSI model that combines the Application, Presentation, and Session layers into one layer and calls it the Application layer. This simplifies the model for workers in the lower layers, but the core work in multiplayer game development takes place at the higher layers.

Physical Layer

The *physical layer* streams bits of data over a communication medium. Popular mediums include category-5 twisted-pair wire, coaxial cable, and various wireless frequencies. The game developer's primary concern at this layer regards latency, bandwidth, and reliability of the media.

Bandwidth and Latency

Bandwidth represents the data transfer rate from source to destination, commonly measured as bits per second and often confused with the more useful measurement of bytes per second due to its abbreviation "bps." *Latency* represents the delay a single bit of data experiences traversing from a source computer to a destination computer, commonly measured in millisecond time units. Bandwidth calculations must account for the initial latency cost; otherwise, the initial latency becomes amortized in the bandwidth calculation. Amortized bandwidth approaches actual bandwidth when line latency and total transfer time deviate, thus making line latency insignificant. The following demonstrates a not-so-deviant case:

Data (δ)	= 240 bits
Transfer Time (θ)	= 4 s
LATENCY (λ)	= 500 ms
BANDWIDTH (β)	= unknown
$\beta_{Amortized}$	$= \delta / \theta$
	= 240 b / 4 s
	= 60 bps
β_{Actual}	$= \delta / (\theta - \lambda)$
	= 240 b / (4s – 500 ms)
	= 240 b / 3.5s
	= 68.5 bps

With bandwidth increasing and latency decreasing, it may seem pointless to even consider these calculations. Realize that initial line latency is included only once, but packets normally transmit in intervals. This requires adding in the line latency time multiple times throughout the session. If 32 packets are delivered each second, the calculation must multiply the line latency by a factor of 32.

Media

Designing a connected game requires setting a minimum bandwidth. Determining the supported media dictates the bandwidth. Otherwise, designing the packet model and arriving at a minimum bandwidth will determine the media over which it may be played. Either way, one must know the bandwidth saturation of each medium. Table 5.6.1 lists some common bandwidth specifications.

Table 5.6.1 Max Bandwidth Specifications

Media Connection Type	Speed (bps)
Serial	20K
USB 1&2	12M, 480M
ISDN	128K
DSL	1.5M down, 896K up
Cable	3M down, 256K up
LAN 10/100/1G BaseT	10M, 100M, 1G
Wireless 802.11 a/b/g	b=11M, a=54M, g=54M
Power Line	14M
T1	1.5M

Note that both DSL and cable download 2 to 12 times faster than they upload data. The serial specification describes the uncompressed transfer rate a null modem experiences. Phone modems use the serial chip (UART), so the serial chip theoretically limits phone modems. Both 28.8K and 56K modems acquire the extra transfer rate through additional compression schemes. Direct serial transfers also contain smaller headers, which increases their bandwidth relative to TCP/IP, for example. The actual delivered bandwidth of any given medium tends to reliably hit about 70 percent of its advertised theoretical maximum.

Data Link Layer

The *Data Link layer* serializes the data for the Physical layer and manages the transmission to its neighboring node. The Ethernet adapter or network interface card (NIC) handles this serialization. Each NIC contains a MAC address to identify it as a unique node on the local network. Not all NICs contain unique MAC addresses; however, for a subnet to communicate, all NICs on that subnet must contain a unique MAC address.

Network Layer

The *Network layer* handles packet routing. Its most popular resident, Internet Protocol (IP), contains both the source and destination IP address for a packet. Richard

Stevens' book [Stevens94] provides clear, in-depth coverage of IP and its companion protocols TCP and UDP.

IP Addresses

Two common versions of IP exist on the Internet today: the popular old IP version 4 (IPv4), and IP version 6 (IPv6), also referred to as the next generation (IPng). The header formats differ in all but the first 4 bits of data that specify the IP version used. The major difference between these protocols centers on the size and format of the IP address entries. An IPv4 address contains 4 bytes, commonly displayed in 8-bit decimal sections:

255.000.255.000

An IPv6 address contains 16 bytes, commonly displayed in 16-bit hex sections:

FFFF:FFFF:0000:0000:FFFF:FFFF:0000:0000

IP entry GUIs should be designed to accept both forms of addresses. The Sockets API, responsible for passing the IP address to this layer, provides a generic means of dealing with either address. For further details, see the discussion on binding a socket to an IP address in the "Session Layer" section of this chapter.

Unicast

An individual's IP address, or unicast address, comes from one of the following sources:

Static (user assigned): The static assignment of an IP address is usually reserved for servers that require a well-known presence.

Dynamic Host Configuration Protocol (DHCP): Routers commonly use this protocol to assign IPs to a specific NIC. The DHCP server maintains an IP lease list containing the IP address assigned, the NIC MAC address assigned to, and the lease expiration time. When an IP address lease expires, it may automatically renew with the same IP or a different IP depending on the policy in place.

Automatic Private IP Addressing (APIPA): The fallback when a DHCP service is not available.

Special Addresses

The following commonly used IPv4 addresses have special meaning and may not be used as a unicast address.

Multicast

Range: *.*.*.{224-239}

Special "multicast" routers allow multiple IPs to enter a group. When a member of the group sends a packet, he sends one packet to the group address on the router, and the router redirects that packet to all members. Multicasting provides excellent bandwidth savings, but the hardware costs make this technology sparse.

Local Broadcast

Range: 255.255.255.255

Socket macro: INADDR_BROADCAST

Local broadcast packets deliver themselves to all adapters on the local subnet, reaching up to 222 IPv4 adapters.

Directed Broadcast

Range: *.*.*.{240-255}

Similar to local broadcasts, but instead of broadcasting on the local subnet, it broadcasts on the specified subnet. Although a nice feature, directed broadcasts are usually discarded by firewalls.

Loop Back

Range: 127.0.0.1

Socket macro: INADDR_LOOPBACK

Packets sent to this address never reach the physical layer. Instead, they are transferred from the send queue to the receive queue at the IP layer.

Address Any

Range: 0

Socket macro: INADDR_ANY

Computers with single adapters often use this address as the source address when setting up a listening socket, because it selects the IP associated with the only NIC on the computer. Computers with multiple NICs may also use this address to allow automatic selection of any available NIC for the socket to listen on.

Domain Name

A domain name provides a human-readable form of the IP address, and a layer of indirection through the Domain Name Service (DNS). For example, the Web server located at 16.15.32.1 provides less description than the domain name *www.gamedev.net*. The DNS indirection allows the gamedev.net site to move to another IP address at any time, but clients can use the same domain name. The DNS is a server infrastructure dedicated to fast domain name resolution. While DNS often meets its service goals, it has its downsides.

First, it adds a layer of complexity to the socket connection process if an address requires resolution, which takes some time since it must contact one or more DNS servers to do so. Working with an IP address directly would save connection time.

Second, the DNS server may be unreachable, leaving one unable to connect even with an active network. Again, using an IP address directly avoids such dependencies. Another problem with DNS centers on the changing of a domain name's associated IP. Moving a domain name to point to a new IP takes time to propagate through the DNS server infrastructure. Moving a domain to a new server may take hours or even days before all DNS nodes reflect the change. Caching the most recent IP resolution provides a good fallback solution if the DNS fails for some reason, although it's not 100 percent reliable.

The Sockets API contains methods to resolve a domain to an IP and look up all domain names associated with an IP, `getaddrinfo()`, and `getnameinfo()`, respectively.

Transport Layer

The *Transport layer* ensures data delivery between endpoints. It recovers from errors and controls the flow of data. It also provides the notion of "ports" as a logical extension to the IP address.

Ports

A *port* number works similar to an apartment number, where an IP address works similar to the address of the apartment complex. To deliver a piece of mail, the mail carrier needs both numbers. Network connections also require a complete "Net Address":

$$\text{Net Address} = \text{IP Address} + \text{Port}$$

Ports range between 0 and 64k. Common network services such as FTP, Telnet, and HTTP use "well-known ports" in the range of 0–1024. Historically, RFC 1700 maintained port specifications. Now, these numbers are maintained by the Internet Assigned Numbers Authority [IANA]. Additional, but less common, services map to the 1k–5k port space. While entirely valid, using port numbers below 5k may clash with other servers running on a LAN. For example, creating a server that listens on port 80 may never receive any traffic due to the router forwarding all port 80 traffic to the LAN's Web server.

TCP and UDP packet headers contain both a source and destination port entry. The listening port must be agreed upon by both endpoints before attempting a connection. If a connector lacks this information, it will require connection attempts on all 64k ports to find which port the host chooses to listen on. The connector must also specify the return port, but due to its inclusion in the packet, it may be selected at random.

Transmission Control Protocol

TCP works best for large data transmissions and data that must reach its destination. The following sections highlight the most-used features of TCP.

Guaranteed In-Order Delivery

TCP will not deliver data to the session layer out of order. If two bytes, byte "A" and byte "B," are sent in respective order, TCP guarantees that byte "A" will be delivered to the session layer before byte "B," even if byte "B" arrives before byte "A."

TCP supports a special flag in the header called "Out of Band," which allows sending/receiving of priority packets. However, the architecture required to use this facility is frowned upon. The recommended alternative entails the creation of a separate TCP connection to handle high-priority data.

Connected

TCP requires a connected state between endpoints that supports the following features:

Packet window: Although data flowing between the Transport and Session layers is considered a stream, TCP-to-TCP data transmission occurs through packets. This allows for a window of *N* outstanding packets, each with a window sequence number used for stream reconstruction, packet acknowledgment, and resending when necessary.

Packet coalescence: Also referred to as the Nagle algorithm or packet nagling, this combines smaller packets into a single larger packet to reduce network congestion caused by many small packets. If 1 byte of data were sent in a packet, 41 bytes would actually be sent: 40 for the header and 1 for the data. The downside to this is that data may sit in the TCP stack waiting for more data, causing unacceptable latencies. Nagle, on by default, may be turned off with the `TCP_NODELAY` socket option.

Connection Timeout: Off by default, this facility sends a simple "Timeout" packet after the line remains idle (no transmissions) for a specified period of time. The receiver of the heartbeat must reply with an acknowledgment in a given amount of time, or the TCP session closes. Use socket option `SO_KEEPALIVE` to configure TCP timeouts.

Streaming: Data transmitted over TCP comes as a stream to the Session layer rather than individual packets. Internally, TCP sends/receives data in packets, but these packets do not necessarily reflect the size of the data buffers made at the Session layer through a call to `send()`. TCP may split or combine individual send commands depending on TCP settings. This requires that the Presentation layer provide facilities to reconstruct data into packets if needed.

User Datagram Protocol

UDP communicates in a send-and-forget manner, not guaranteeing order of delivery or even delivery at all since no direct connection is made. Data transmits in packets instead of a stream, which assumes a connection. The lack of connection maintenance also reduces the UDP header size. The reduced header size in conjunction with the absence of resends and packet coalescence provides a better latency/bandwidth model than TCP.

Avoid writing a guaranteed layer on top of UDP, which bypasses all the time and effort built into TCP. Firewalls tend to render such solutions useless anyhow, because they commonly block all incoming UDP traffic. The policy to block all incoming UDP traffic stems from the security issue introduced by not verifying the return address. Stopping a network attack involves identifying its source—a difficult journey without a confirmed return address. Design the game to allow TCP as a fallback for denied UDP traffic.

Broadcasting

All broadcasting over IP networks takes place with UDP packets. This makes perfect sense with the connectionless nature of UDP. However, broadcasting floods network bandwidth, and firewalls normally drop incoming broadcast packets. This restricts its use to LANs where it provides an excellent method of player/game discovery. Lobby servers, such as GameSpy, replace actual UDP broadcasting on the Internet with a logical subscription-based broadcast in which one must connect to the server to receive TCP packets sent to all connections.

Session Layer

The *Session layer* manages connections between applications. Its responsibilities include establishing connections, terminating connections, and coordinating data exchange. The Sockets API provides a cross-platform Session layer model to handle these tasks.

Sockets

The *Sockets API* supports several basic implementation models, each simple at the outset to implement. The complexity emerges when maintaining multiple sockets, setting options for lower layers in the protocol stack, and ultimately using high-performance socket extensions. While sticking to the standard socket API methods promotes cross-platform portability, this model generally limits itself to client development. Servers often push connectivity limits and/or data throughput, which necessitate use of OS-specific socket extensions. About a dozen extensions exist that obscure the best approach to achieving high performance. The "Sockets Programming" references at the end of this chapter contain reams of information on both basic and high-performance sockets. Although not listed in the references, most modern programming languages support the Sockets API, including Java, C#, C/C++, and Visual Basic. The remainder of this section presents a general overview of sockets, and more importantly, the high-level differences in the ways to use sockets. The code snippets use WinSock defines, which occasionally differ ever so slightly from their Unix equivalents.

Origins

Sockets originated as an extension to the file I/O paradigm, which explains why the file descriptor "fd" abbreviation is scattered throughout sockets. To this day, serial

[Camp93], socket, and other connection types maintain file descriptor compatible handles. These handles pass to file I/O interfaces such as `read()` and `write()`, allowing data transfers to follow the standard reader/writer design pattern. Unix hosted the first Sockets API, which provided additional functionality to deal with the latent data transmissions and protocol control. Many years passed before third-party ports were made available on the Microsoft Windows platform, but soon after these first ports became available, Microsoft released its own implementation of the WinSock Sockets API.

WinSock

All the standard socket interfaces exist in WinSock, with specific extensions containing the "WSA" function name prefix. Microsoft provides nonstandard, briefly documented socket extensions external to the Winsock specification that allow for socket reuse and additional high-performance features.

All WinSock programs must use two such extension functions to allocate/free system resources; Unix sockets do not require such initialization:

- **`WSAStartup()`:** Call this before using any Winsock API methods.
- **`WSACleanup()`:** Call this to release all socket handles after closing them.

Socket Modes

Sockets are either *blocking* or *nonblocking*. By default, sockets use blocking mode. In a game that requires an active user interface, blocking sockets should only be used in separate threads, because blocking calls puts their thread to sleep until the action completes. To hack around the blocking problem in a single thread, one can "peek" for data available to read, since a socket `read()` call often blocks waiting for data. Setting nonblocking mode provides a better alternative to peeking because it actually completes the operation if possible or returns an error that it would have blocked if it were in blocking mode. In contrast, a successful peek polls for arrived data, and still requires an additional read operation to clear the data off the stack. This double read of kernel memory buffers degrades performance considerably in server applications.

To switch between blocking and nonblocking mode:

```
unsigned long arg = ?; //0=blocking 1=non-blocking
int status = ioctlsocket( fd, FIONBIO, &arg);
```

Once in either blocking or nonblocking mode, it remains in that mode for the duration of the process. If WinSock is shut down and reinitialized, it will be set back to blocking mode. To force a blocking call to return, either close the socket or make a call to `WSACancelBlockingCall()`.

Standard Socket Models

The sockets specification provides two socket models: standard and select. The select model provides a mechanism to handle sets of up to 64 sockets each. The following sections briefly cover the simplest form of the standard socket model and common usage patterns.

Socket Creation

This line of code creates a TCP socket descriptor:

```
SOCKET tcpSocket =
socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
```

This line of code creates a UDP socket descriptor:

```
SOCKET udpSocket =
socket(PF_INET, SOCK_DGRAM, IPPROTO_UDP);
```

TCP Connecting

To connect to a remote listening host socket, a destination net address is required. The following code creates an IPv4 compatible address structure and attempts a TCP connection:

```
SOCKADDR_IN addrV4;
addrV4.sin_family            = AF_INET;
addrV4.sin_port              = htons(4000);
addrV4.sin_addr.s_addr = inet_addr("10.2.15.89");

int error =
connect(tcpSocket,(SOCKADDR*)&addrV4,sizeof(addrV4));
```

The following code uses the preferred `getaddrinfo()` method to create an IP address compatible with both versions 4 and 6:

```
PADDRINFOT info = NULL;
ADDRINFOT           hints;
hints.ai_flags      = AI_NUMERICHOST;
hints.ai_family     = PF_INET;
hints.ai_socktype   = SOCK_STREAM;
hints.ai_protocol   = IPPROTO_TCP;

char strPort[10] = "4000";

int result =
getaddrinfo("10.2.15.89",strPort,&hints,&info);

int error =
connect(tcpSocket, info->ai_addr, info->ai_addrlen);

TCP LISTENER
```

The TCP host must bind the socket to a port and a local adapter with the `bind()` call. Next, the host must listen for incoming connections, and finally sit and wait to accept the connection.

```
int status =
getaddrinfo(htonl(INADDR_ANY),strPort,&hints,&info);

status =
bind(tcpSocket, info->ai_addr, info->ai_addrlen);

int connectionBacklog = 1;
status = listen(tcpSocket, connectionBacklog);

int addrLen=0;
PADDRINFOT remote = NULL;

SOCKET newSocket =
accept(tcpSocket, &remote.addr, &remote.ai_addrlen);
```

Stream Transmissions

After connection, the client and host may freely send and receive data. Among other error conditions, the send operation may error if the TCP buffer is too full to accept data, in which case the send should succeed once the buffers have time to transmit their contents.

```
#pragma pack (1)
struct Packet
{
        short   length;
        char            username[10];
};
#pragma pack ()

Packet pkt = {12,"testpkt"};
int flags = 0;  // No flags, rarely used

int err = send(tcpSocket,(char*)&pkt,sizeof(pkt),flags);
```

TCP receive operations may look very different between implementations due to its streaming nature. The following sample provides a common solution for packetizing the data stream. It assumes the first 2 bytes of all application layer packets contain the length of the packet. This allows the system to determine how big a buffer to create for the complete packet read operation.

```
// Handle Endian order if used cross platform
short pktLen;

// a short is used to represent packet length
short lenSize = sizeof(short);
```

```
int bytesToRead = lenSize;
int flags = 0; // ignore

// Read the packet length first
int bytesRead =
recv(tcpSocket, (char*)&pktLen, bytesToRead, flags);

// Allocate buffer for to read in entire packet
char* buffer = new char[pktLen];
memcpy( buffer, &pktLen, lenSize );

// Read remainder of packet
bytesToRead = pktLen - lenSize;
bytesRead =
recv( tcpSocket, &buffer[lenSize], bytesToRead,flags);
```

Datagram Transmissions

UDP's `sendto()` method acts similar to the combination of TCP's `connect()` and `send()` methods. UDP's `recvfrom()` method combines even more functionality in comparison to TCP's, with `recvfrom()` taking the place of `bind()`, `listen()`, `accept()`, and `recv()`. The following shows UDP's send method with previously declared parameters:

```
int status = sendto( udpSocket, (char*) &pkt, sizeof(pkt),
info->ai_addr, info->ai_addrlen );
```

In addition to combining so much functionality into one method, `recvfrom()` also avoids the two-step read of packet length followed by a read of the remaining packet. This is due to `recvfrom()` only reading one packet at a time. Simply create a buffer the size of the largest possible packet, and call `recvfrom()` with the buffer and its length:

```
int flags = 0; // ignore, not very useful
char buffer[MAX_PACKET_SIZE];

int bytesRead =
recvfrom(udpSocket, buffer, MAX_PACKET_SIZE, flags
&remote.addr, &remote.ai_addrlen);
```

The `recvfrom()` will only fill up to the size of a single packet and return. The method will never combine packets, as TCP would, because it does not stream data as TCP does.

High-Performance Socket Models

Standard socket models fall short with respect to performance in several areas, most notably in event notification and memory buffer copies. Additional shortcomings exist in the `accept()` architecture and in socket reuse. For implementation details on each of these subjects, search for the following keywords: WSAEventSelect, I/O Completion Ports, poll, and Kernel Queues. The following provides an overview of the issues involved with each problem.

Event notification entails use of multiple threads, and kernel level signaling of a blocking application thread waiting for an operation to complete, such as data arriving, data sent, or a socket closed. "Standard" sockets need a thread for each outstanding operation, and "select" sockets require a thread for every 64 sockets. Event notification requires only one main socket processing thread that receives the signals for all sockets. Once signaled, the processing thread can handle the operation or place the operation in a signaled queue to process in a worker thread from a thread pool.

Standard sockets copy data from kernel buffers to user-supplied buffers during a read call. Asynchronous I/O solves this performance issue by allowing the application or user thread to pass the kernel some number of buffers to use instead of its own. After a buffer fills, the kernel then signals the user thread about the readied data. This greatly reduces overhead with data transfers—critical for maximizing data throughput.

The Listen/Accept role of a TCP connection host using the standard socket model requires the host to accept a socket connection before receiving any data from the connector. Accept creates a socket that requires the lengthy process of the kernel allocating a descriptor. Two solutions exist for lessening the impact of this problem. One solution allows the passing of a created but unconnected socket descriptor to the accept method, thus bypassing the expensive socket creation hit at runtime. This still requires the connection to happen before any authentication. The other solution allows the connect method to send an initial data packet with the connect request. This allows the host to authenticate a connection request before the TCP connection request returns accepted.

Creating a socket descriptor consumes valuable resources. Reusing socket descriptors after they close allows servers to handle many transient connections with faster response times. This requires connections to confirm closure and that closed sockets not release their descriptor handle. The standard socket `close()` method frees socket descriptors in addition to closing the connection. Note that this only applies to TCP sockets, since UDP sockets inherit reuse since they never connect and thus never require closure.

Three methods exist in the Sockets API to control TCP, UDP, and IP protocol options and session layer I/O: `getsockopt()`, `setsockopt()`, and `ioctlsocket()`. These take a socket, a predefined operation code, and the arguments associated with the operation (plus other overhead parameters). Refer to the references at the end of this chapter for a thorough explanation of all these options and sockets programming at large.

Presentation Layer

The Presentation layer provides generic data conversion by preparing data for transmission and converting incoming data back into a format recognized by the Application layer.

Compression

Real-time data packets are relatively small compared to file transfers, normally on the order of 10–1000 bytes. They should not exceed the network's Maximum Transmission Unit (MTU), which dictates the largest size of a single packet. Packets larger than the MTU, normally around 2k bytes, must be split over multiple packets. Dealing with such small data sizes makes it counterproductive to use generic compressors such as Huffman encoding, due to their table overhead. A better generic alternative would be a custom encoder built into the packet serialization process. Such an encoder may implement the following data reduction methods:

Pascal strings: C/C++ commonly follows the NULL terminated string convention. This poses two problems. First, the containing buffer is normally created at a static maximum string length, which often wastes space, including the space required by the NULL terminator. Second, a packet stream works best if the length of the pending data comes first. With NULL terminated strings, the length remains a mystery until reaching the NULL terminator. Pascal strings reserve the first byte or two of the string to place the string length, and forego the NULL terminator.

String tables: Some strings are set once during a game and continuously used, such as a username in chat rooms. Keep a table of strings and associate an integer key with each string. Introduce a string to all players as a string/key pair, and then only send the key in all future references. New players need to receive a copy of the entire table upon entering the game.

Bit fields: Placing small enumerations and Boolean variables into bit fields conserves bandwidth. Implement this at the structure level to avoid having to serialize the data. Placing bit fields consecutively also reduces gaps of unused byte space.

Float to fixed: Often, floating-point accuracy is overkill for certain data representations, such as percentages. A fixed-point number commonly uses a single 2-byte integer and logically splits it into a two parts: the whole number and the fraction. This saves 2 bytes over the common 4-byte floating-point representation. Save more memory by splitting a short or char if precision requirements permit.

Matrix to quaternion: 3D orientation is commonly represented in matrix form for any number of reasons. A quaternion provides the same information and accuracy with fewer bytes.

Encryption

The most likely person to hack a game packet is the person running the game. Never pass sensitive data to a DLL, as they are easy to chain, allowing the user to replace the authentic DLL with his own to change the data, which it then passes on to the authentic DLL. This chaining process is also referred to as *shimming*. The WinSock

DLL may also be shimmed, rendering IP Security vulnerable to local data tampering. The best method to keep data from the gamers' prying eyes involves encrypting within the executable module.

Serialization

Structures may contain integer alignment padding, pointers, or other data not intended to leave the local computer. To solve this problem, serialize the data by using a secondary buffer and filling it with the exact byte stream to pass to the Session layer.

When using TCP, data is sent to and received from the Session layer as a stream. To work in packets over TCP, this layer must provide the logic to identify the size and type of packet as such:

```
struct packet
{
        ushort Length; // Size of this packet
        ushort ID;     // Predefined packet type
        ...                    // Additional header info
        char   Data;   // Data from Application Layer
};
```

The positioning of the `Length` variable as the first variable works well in the receiving architecture. First, post a 2-byte socket read operation. Create a buffer to hold the size of the packet. Then, post a second read operation to receive the remainder of the packet. Repeat for the next packet.

Buffering

The following sections describe different types of buffering found in the Presentation layer.

Packet Coalescence

Although supported by TCP, turn it off and create a customized coalescence system to avoid excessive latencies. Since UDP doesn't support this feature, such a system at the Presentation layer could be used with UDP. Game clients normally abandon all coalescence in lieu of absolute lowest latency. Game servers may actually lower overall latency through coalescence by freeing up processing time and bandwidth. The effects are most noticeable in servers with large numbers of clients.

Induced Latency

Ideally, all players act on the same input data from all users at the same time. The induced latency technique sends input as soon as it latches locally, and then stores it for some prescribed amount of time (the induced latency) before using it, as shown in Figure 5.6.2.

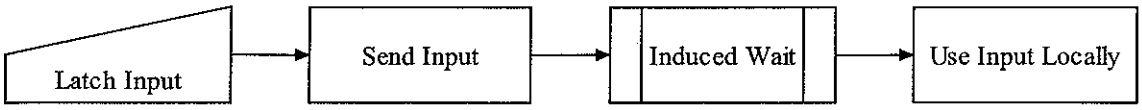


FIGURE 5.6.2 *Induced latency diagram.*

Adding a server that sends local input back at a steady rate allows the server to send previous input when current input does not arrive in time for the server sendoff. In this case, the local machine never stores their input and acts only on input received from the server. Of course, this runs the risk of stopping and waiting for input to arrive from the server, but the intent is to use this on a synchronous model, which must stop and wait for remote input as well.

Dead Data

Since UDP packets may be received out of order, a real-time game may not use old data, allowing the Presentation layer to discard such older packets. A more complex system uses old data to either confirm paths or help provide more accurate interpolation. Old data may also be entered into replay queues for smoother transitions and a more accurate original path. The game designer must decide whether the replay shows actual paths or viewed paths. Since actual paths may conflict with local paths, running the replay system should warp to discrete position/orientation nodes rather than rerun through a simulation with collision detection.

Large Packets

Data packets larger than UDP's MTU or larger than the TCP's send buffer must be split into multiple packets. The Presentation layer handles this subdivision and reconstruction. Such a system involves the addition of a new packet type ID for "large packets," a unique instance ID to differentiate it from other large packets, and a sequence number. A given large packet must either transmit the total number of packets in the first packet, or provide a special sentinel number for the final packet.

Application Layer

The *Application layer* deals directly with game data and game logic. While the Session and Presentation layers often contain generic implementations replaceable with middleware, the Application layer is always part of the game. Here, the update model and synchronization code form the core of a networked game.

Version Verification

When a quality assurance (QA) team starts testing, especially during development when many versions of game components exist, always run a sanity check on all component versions. This check includes comparing size, version, and checksums of all executable files and dynamic link libraries (even system DLLs used), data, and art

files. One risks a constant chasing down of nonbugs without using a robust validation system.

Update Models

The game's *update model* guides the design of the most intense packets in the game. The *input reflection model* presents packets sent over the network as if they were another input device attached to the computer. The *state reflection model* processes the input locally and sends the processed data, such as new positions, orientations, velocities, and accelerations.

Input Reflection

This model usually sends slightly processed input data, processed enough to make the input generic rather than deal with specific device nuances. The generic input for a single player using a mouse, joystick, and keyboard can usually fit in a packet less than 32 bytes in size, depending on the controls used. Such a small payload turns out smaller than the combined protocol headers of TCP/IP at 40 bytes. Expending more bandwidth on header data than on the actual packet payload data is generally frowned upon as an inefficient use of bandwidth. The low packet latency mandate overrides general network bandwidth efficiency, and in this respect, input reflection excels.

Human perception of delay plays a critical role in establishing the latency requirements of input reflection. The average person will notice anything more than approximately $1/10^{th}$ of a second delay in hand/eye coordination tasks. Frame rate also plays a role, since ideally each new frame changes with respect to player input. With human perception and average maximum frame rates capping at around 60 fps, an acceptable packet send rate ranges between 16 to 64 input packets per second.

Input reflection might use a synchronous or asynchronous play model. Synchronous play, where the game stops and waits until it receives input data from all users, looks horrible and plays jerky when latencies vary or rise, but it always remains 100 percent in sync. Asynchronous play, where the game predicts remote player input when not available, produces difficult game synchronization problems when wrong predictions result in dramatic in-game events such as a crash, death, or anything else that produces a "rising of the dead" on a resynchronization. Even without catastrophic events, the resynchronization of input reflection requires a complicated, comprehensive, and potentially memory-intensive state save for each input simulation past a prediction point until the actual input arrives. This allows recovery at any point past the prediction, so that the game goes back to a save point, discarding all future state (or in this case, current state), and resimulates the game from that point. Asynchronous play is better suited to the state reflection model or hybrids thereof, which may use input reflection as hints.

Pure input reflection works well with the synchronous "stop & wait" model. The prime directives of this model keep:

- Latency low
- Latency consistent
- All clients in sync

The last item requires rigid rules on randomness. The following tips help avoid unintentional randomness.

- The function `rand()` must use the same seed on all clients if used in simulation calculations. There should be no actual randomness between clients. Since only one `rand()` instance exists for a given process, consider creating a custom randomizer for the game if multiple systems need it.
- Use `rand()` in a reproducible manner; avoid sharing it between the physics system, which is consistent, and the graphics system, which introduces variance across clients.
- Fix all uninitialized variables. Clear all stack and heap variables to remove residual contents before setting. Partial initialization of dirty structures and arrays are very difficult bugs to locate.
- Fix all freed pointer reads. Rely on heap tools for this.
- Validate versions of all files between clients. Note that art files may affect physics calculations.
- Avoid using client system time for calculations, as they rarely match 100 percent.

State Reflection

The *state reflection model*, also known as the *positional update model* for passing object positions and other locality variables, provides a flexible environment geared toward prediction and synchronization at will. The data packets normally require much more information, but object prioritization allows distant objects to update less frequently. The simplified synchronization of this model eases the production of a "drop-in" game where players can join a game in progress; a common method of game entry for role-playing games (RPGs).

Synchronization

One of the most artistic tasks of a network game programmer revolves around keeping all clients in sync with minimal visual or game event related anomalies.

Dead Reckoning

Dead reckoning is a basic prediction method that uses the last known position, orientation, momentum, and acceleration to determine the most plausible current position. The object undergoes simulation over time assuming no changes to acceleration factors. The results work well for all but drastic changes in acceleration, which should be capped in the game's design.

AI Assist

Standard AI design involves setting waypoints for nonplayer characters (NPCs). Positional updates provide such waypoints but lack the smooth transition between

waypoints. Leveraging the AI to control the transition works well as long as the waypoints don't change too often. To achieve this, give waypoints a commitment time, and refrain from removing the waypoint for a tunable amount of time. This causes the game to run slightly out of sync, but avoids the distracting "bee wiggle" that results from wavering waypoints.

Arbitration

As in real life, some things just don't work out as planned and require unbiased third-party *arbitration* to rectify a situation with no clear outcome. Fuzzy logic helps build a weighted decision as to the correct outcome based on affected clients and the server's view of a situation. A simplified, dictator-style approach ignores the client views and forces clients to take the game server's view. Most games instantiate the dictatorship role in the server, but design the game with incremental states. The client may lessen the impact of dictation by delaying critical events, such as a dying sequence that starts with a badly wounded sequence while it waits for the final word from the server/arbitrator.

Real-Time Communication

Real-time networked games require a great deal of coding effort to efficiently handle waiting for data to arrive. Properly crafting multithreaded constructs to avoid spinning, dead lock, lock contention, and excessive context switches help to reduce the wait. These constructs grow in complexity with the number of interactions.

Reducing wait time often requires a judgment call on the following data-related issues:

Priority: Certain types of data impact the feel and fairness of gameplay, while other types of data merely support the game through added flair. Data that impacts gameplay needs to arbitrate outcome with other players.

Security: Encrypting all data traffic costs both CPU time and extra bandwidth, which in turn cause additional delivery delay. One encryption algorithm doesn't necessarily fit all. Consider the following optimizations:

- Lower bit strength encryption on less-sensitive data.
- Use *secret key* instead of *public key* for high-frequency transmissions.
- Use *message digest* instead of secret key if the intent is to tamperproof without hiding contents from prying eyes.
- Encrypt every other packet (or even less often) on high-frequency transmissions, and then use the encrypted packets to sanity check data in nonencrypted packets.

Compression: Converting, packing, or otherwise compressing data comes at a CPU cycle cost. While bandwidth reduction often takes priority over CPU cost, a very small bandwidth saving may incur a large CPU cost.

Reliability: Does the data need to arrive? If so, use a guaranteed protocol. If not, use a nonguaranteed protocol. Games often require data on time or not at all. Using UDP avoids resends, freeing bandwidth, socket buffers, and CPU cycles for required and on-time data.

Synchronicity: Will the game adopt a "stop & wait" policy on latent data? If so, the simulation may need to freeze, but the graphic subsystem should continue to update. Stopping graphical updates appears as if the system hung. The user needs fluent access to chat and menu systems to communicate with connected parities or take action on the latency issue by booting a player or simply exiting the game.

Connection Models

The connection models described in this section should not be confused with connection- and connectionless-oriented protocols such as TCP and UDP, respectively. Even though UDP is considered connectionless, a packet follows a path considered a pseudoconnection for this discussion.

Broadcast

Broadcasting simplifies player discovery, but should never be used for normal packet delivery. To receive a broadcast packet requires actively listening for all broadcasts. Broadcast packets must contain some special identifying mark to distinguish them from broadcasts sent by other applications (which you have no choice but to weed through). A Globally Unique IDentifier (GUID), built from a hash of unique numbers—IP address, MAC address, and current date/time—handles such circumstances. Creation usually occurs outside the application, followed by embedding it as a constant within the source code. Microsoft provides "guidgen.exe" to create these numbers, and similar tools exist on other platforms.

A GUID established for game broadcasts solves the problem of game discovery. To use broadcasting for all game packets during gameplay requires a second GUID for the game instance. While broadcasting provides a useful mechanism for game discovery on LANs, avoid its use for high-frequency game packets. If say 10 four-player games were played on the same LAN, each person would have to weed through 39 packets for each game event, while only three are of concern to any given player.

Peer to Peer

Peer-to-peer connectivity means every player connects directly to every other player, as shown in Figure 5.6.3. This model experiences the least amount of latency because packets avoid the additional scenic trip to a server. This latency benefit comes at the cost of increased bandwidth requirements and connection maintenance complexity as the number of players increases beyond two. Since two-player games are not affected by the adversities of peering, they normally use this model unless the game architect requires a server.

Client/Server

The *client/server* model suits most games over two players, better than peering, due to the easier connection maintenance and lower bandwidth.

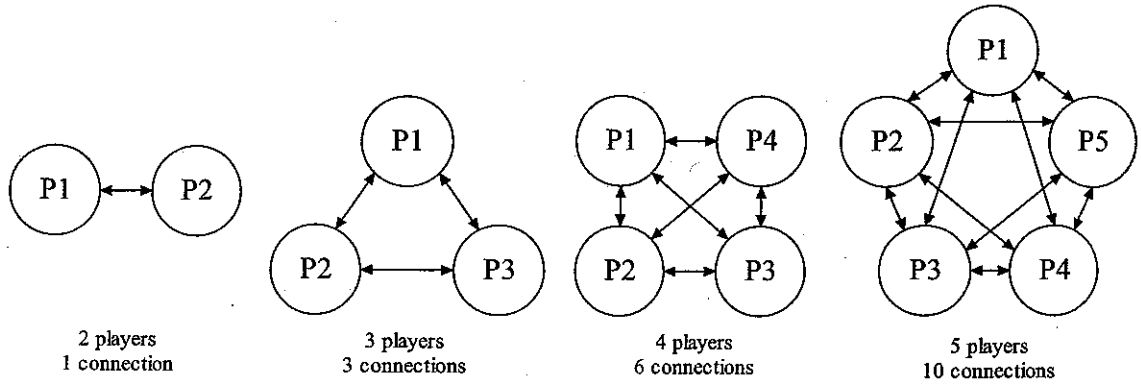


FIGURE 5.6.3 *Connections within the peer-to-peer model.*

Having a server in the connection model, as shown in Figure 5.6.4, offers many benefits over peering arrangements. Packet coalesce works best with on servers. "Lossy TCP," a construct that requires a server, works as follows. UDP packets are commonly refused by secure firewall configurations. If your game relies on high packet throughput in the absence of UDP, you must resort to TCP. TCP tends to back up, trying to redeliver packets that become outdated if not delivered immediately. A server can implement a "Lossy TCP" for clients that cannot accept UDP traffic by maintaining a buffer of size two; one slot for the item currently being sent, and the other slot for the latest packet. An incoming packet simply replaces the second slot item if one exists.

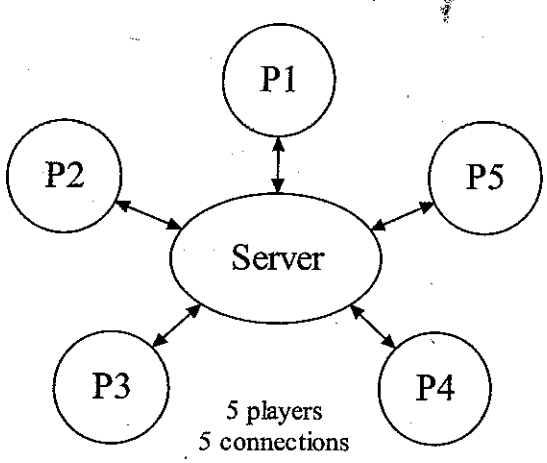


FIGURE 5.6.4 *Connections within a client/server model.*

Connection Complexity

Maintaining a connection requires monitoring the line for input, outputting data (duplicating effort for each additional player), keeping silent connections from clos-

ing with "keep alive" packets, and handling dropped connections either by ending the game or attempting to reconnect. Table 5.6.2 illustrates the number of connections required using different connection models.

Table 5.6.2 Number of Connections Required Using Different Connection Models (N = number of players)

	Broadcast	Peer to Peer	Client/Server
Connections	0	$\sum_{x=1}^{N-1} x$	Client = 1 Server = N

Bandwidth

Bandwidth costs increase linearly with the number of players. With bandwidth at nearly the same premium as latency, peering tends to fail beyond the simplest two-player game. In a fully connected game, going from two to three players more than doubles the bandwidth requirements. That is more than just application data, since each packet comes with the transport's header. Table 5.6.3 illustrates the number of times a single packet must be transmitted for each player using the different connection models.

Table 5.6.3 Number of Times a Single Packet Must Be Transmitted for Each Player (N 5 number of players)

	Broadcast	Peer to Peer	Client/Server
Send	1	$N-1$	Client = 1 Server = N
Receive	$N-1$	$N-1$	Client = 1 Server = N

Asynchronous Environments

An asynchronous environment exists where two or more branches of code run simultaneously. This section assumes a basic understanding of threads, signals, and critical sections. The following information provides tips for network game programming in a multithreaded environment.

Set network thread priority above or equal to the main render thread, but always below the audio thread. Make sure all network threads exit before exiting the main thread, or shared data will likely cause a null pointer exception.

Use signals and events to wake up or otherwise communicate the availability of new information across threads. The alternative, polling for state change, also goes by the name "spinning" because of the time wasted accomplishing nothing. Both events and signals are limited system resources, so plan accordingly. Remember that events

notify all subscribers, where signals usually notify one of *N* subscribers. Expect to run into signals with thread pools that contain multiple subscribers, but only one available thread should receive a wake signal to process the work item.

System timers signal a high-priority thread, so refrain from lengthy activities in the timer thread or risk skewing time. Choose a reasonable timer resolution; no need to go above 128 Hz for most cases. Make sure to use the proper high-resolution timer, as more than one timer may exist on the OS. Use the multimedia timer for Windows.

Use critical sections sparingly. Try to design with the least amount of critical sections, and always keep the amount of time in a critical section to a minimum. Always match a critical section entry with an exit. Avoid entering in one method and exiting in another, as this complicates the design. All shared data writes require critical sections if more than one thread writes to the data. Avoid calling another critical section within a critical section at all costs; otherwise, deadlocks may occur. For incremental data modifications, use `interlockedincrement()` and `interlockeddecrement()` in Windows or their OS equivalent rather than using a critical section. Precede all shared data definitions with the `volatile` keyword; using data in a critical section does not relieve the possibility of shared data getting stored in registers between context switches.

Security

As soon as someone starts boasting about his perfectly secure system, he gets hacked. One-hundred-percent security does not exist. Keys and passwords can always be compromised at some level. Security takes development time and affects game performance. Instead, make a great game and provide an environment *secure enough* against all but the most diligent delinquent. The best way to go about securing a system involves using multiple security mechanisms. Start with standard security measures, and then add some creative convolution to make breaking security a hands-on affair rather than leaving the system open to automated cracking. The remainder of this section describes security mechanisms, what they do, and where to use them.

Encryption

The three goals of encryption are:

Authentication: Verification of an entity's claimed identity
Privacy: Prevent unauthorized viewing of data
Integrity: Assurance that data was not tampered with after leaving its source, although it does not prevent tampering

Atomic security elements commonly address one or two of these goals, but not all three. Higher level security layers combine the atomics to meet additional goals.

Public Key (Asymmetric—Key Pairs)

The *public key encryption* algorithm creates two keys: one public and one private. The public key is made public so others may encrypt data with it. The encrypted data requires the private key to decrypt. This algorithm is stronger than secret key encryption, but requires much more computational time. Both symmetric and asymmetric key encryptions provide privacy only.

Secret Key (Symmetric—Same Key)

The *secret key encryption* algorithm creates one shared key used to both encrypt and decrypt data. Use this algorithm to encrypt real-time data, as it requires less computation time. To share a secret key, encrypt it with the remote end's public key and send it to the remote end.

Ciphers

Block and *stream ciphers* describe how the key mechanism interacts with the plain text to produce the cipher text. Block ciphers work with a fixed-size data block. If the plain text does not fill the data block, the block cipher adds padding. Plain text larger than the block is split across multiple blocks. Stream ciphers work with any size data. The simplest cipher, called Electronic CookBook (ECB), combines the key and the plain text to produce the cipher text. Other variations use the output from the previous cipher operation in addition to the key and plain text, providing stronger encryption [Sch96].

Message Digest

The relatively fast *message digest algorithm* produces a checksum to verify message integrity, but does not encrypt the data or provide authentication.

Certificates

Certificates, also known as *digital IDs*, provide authentication through a trusted third party [VeriSign]. The third party stores public keys associated with a verified entity and delivers them on request, encapsulated in certificates that confirm the key owner's identity.

Copy Protection

Stopping game software pirating may not work entirely, but several reasons make it a worthwhile endeavor to at least delay the inevitable. A chart-topping game makes a large percentage of its lifetime sales in the first month of its release. Consider purchasing a copy protection system as an insurance policy on these initial sales. Even though professional criminals will break the protection, many more would not think twice about burning a copy for their friends.

An old form of copy protection involved the use of a code sheet distributed with the game. The code sheet contained numerous entries that did not photocopy. The game would randomly ask for codes from the code sheet to be entered in order to continue play. A similar approach was to request words from specified sections of the

manual. The deterrence here was to ship a very large manual. Both code sheets and digital copies of the large manuals started popping up all over the Internet.

CD-ROM copy protection combines tricks using valid data in sectors marked as invalid on a CD-ROM, and encryption [Safedisc].

Watermarking

While not preventing illegal copying, digital watermarks make nonvisible alterations to art assets that can prove theft if such art appears in other works; for example, another game using artwork without permission from the creator.

Execution Cryptography

The following sections detail measures to take against attacks on the game execution modules.

Code Obfuscation

Stripping the code of its symbol names complicates reverse engineering. While not possible with interpreted languages, a code obfuscator changes all variable and method names to nondescriptive names. For example, a variable named "WorldPosition" would change to "v0001," leaving a hacker to figure out the intended purpose. Code obfuscators do not affect the actual byte code.

Heap Hopper

Tools are available that take snapshots of the heap before and after an event takes place, such as moving a player entity forward. After taking several snapshots between moves, certain variable locations consistently show change. A hacker can narrow in on the movement variable and change it manually, or make a program to automate changes. *Heap hopping* moves sensitive data around in the heap to make it more difficult to associate an action with a specific variable change. This may be done in numerous ways, but one strategy creates several heap buffers of the same size, copies data from one location to the next, and modifies the variables in the new location, thus preventing a hacker from narrowing in on the sensitive data.

Stack Overrun Execution

Stack overrun execution hacks take advantage of deficient packet data validation by the game. If a user sends malformed data that causes a function in the game to overrun its stack, the hack can modify the return instruction pointer to point another location, be it in the code or a data buffer the user sent in a packet. Such a hack usually requires a great deal of code analysis to craft, but there are determined hackers willing to spend the effort to make such an attack.

No Op Hacks

One of the easiest hacks involves changing the executable file by replacing method calls with "No Operation" byte codes. A hacker could use this to bypass certain validation checks, allowing further tampering of the code.

Timer Hacks

Many games use the computer's timers to regulate the game physics or otherwise control movement. Changing the system clock is very simple. To counter such hacks, verify that the timer never goes back in time or never takes unreasonable forward leaps.

DLL Shims

Game DLLs have method entry points. A shim DLL mimics these entry points and provides replacement code for each entry point so it silently replaces the original DLL. This allows monitoring of data passed to the methods of the DLL and often the changing of code execution. Several viable counters to this attack include using numeric entry points (ordinals) instead of method names, or providing only one entry point that returns class object pointers. Using a DLL with named method entry points makes the shim builder's job much easier, while a single entry point makes the hacker work harder.

User Privacy

Breaching a player's right to privacy creates headaches ranging from bad public relations to legal trouble. An online subscription game typically collects very personal data for billing purposes. Never divulge the following critical pieces of information:

- Real name
- User password
- Address
- Phone number(s)
- E-mail address
- Billing information
- Age (especially of minors)

Use strong cryptographic measures for both transmission and storage of such information. In addition, limit access to this secured data within the development team.

While not a listed element in the critically private information, a user's IP address justifies a fair degree of privacy. Peered connection architectures make sharing IP addresses unavoidable, but not their display. Displaying a user's IP allows even the nontechnical person to simply enter someone's IP into a program that could perform various network attacks.

Username and Password Interception

Using public key encryption for transmission of username and passwords, and not showing the users password as it is typed are sufficient standard practices to secure this information. Problems with username/password theft normally stem from impersonation over chat or e-mail channels, and fake utilities. Prevention by educating the user

is the best step to take on both accounts. Fake game-specific utilities often require the user to enter in his username and password, which it sends to the hacker through some untraceable means such as a hotmail e-mail account. To reduce the impact of such a breach, all changes to the account and access to billing should require confirmation through the user's e-mail.

Firewalls

A *firewall* either inspects packets to determine if they should pass through the firewall, or provides an encrypted session.

Packet Filter

Stateful protocol inspection, or *packet filtering*, looks at protocol headers to determine whether a packet should be allowed to pass. A port filtering device inspects the port entry of a TCP or UDP packet and either accepts or rejects the packet based on user settings for a given port. Users should keep ports blocked in case they inadvertently acquire a malicious program that attempts to communicate data from their computer. This applies twofold to computers running a game server, which should block every port not explicitly in use. Games that require certain ports be made available should allow configuration of exactly which port(s) the game uses.

Similar filters inspect IP address entries in IP packets and accept/reject packets based on the source. Such filters provide a way to ban specific IPs from access to the network. Entry of IP addresses into the banned table provides the most benefit when done by an automated process that detects numerous denied connection requests to elements on the network behind the firewall.

Proxies

Application *proxies* inspect the data within the packet. Such a proxy server could check MIME or FTP traffic for viruses.

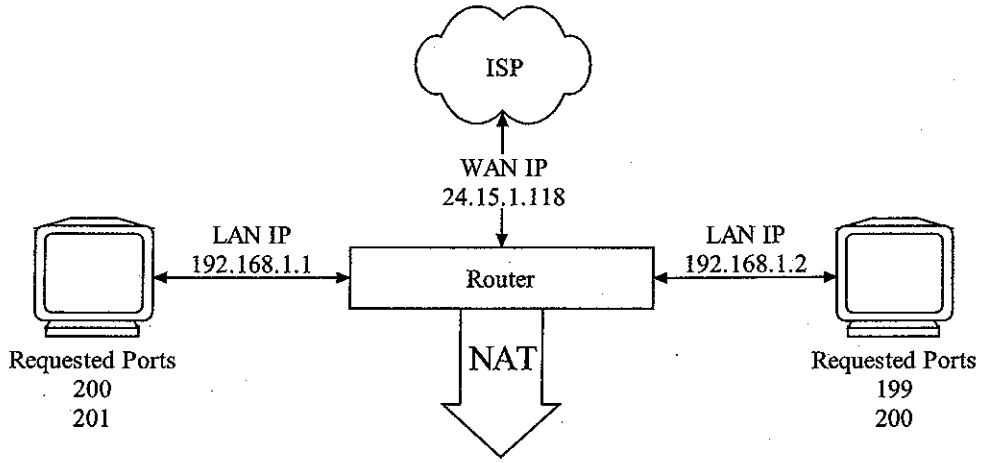
Circuit Gateways

Circuit gateways set up secure sessions, and ignore packet contents.

Network Address Translation (NAT)

The NAT protocol allows routers to share a single WAN IP address between all network adapters connected to the router. It accomplishes this by sharing the 64 K port space of the WAN IP between them. To share the port space, the router maintains a NAT table that maps LAN addresses, IP:Port, to WAN ports. This process hides LAN IPs from the Internet side of the router, which only sees the WAN IP. This indirection makes it more difficult to direct attacks on a specific address.

The NAT algorithm determines whether requested ports actually map directly to the WAN port. In the case of two requests for the same port, from separate adapters, one request will either be offered a different external port or return with an "in use" failure. Figure 5.6.5 illustrates this process.



LAN Address	WAN Address
192.168.1.1:200	24.15.1.118:200
192.168.1.1:201	24.15.1.118:201
192.168.1.2:199	24.15.1.118:199
192.168.1.2:200	24.15.1.118:4000*

FIGURE 5.6.5 *NAT at work.*

Port Forwarding

Hosting socket connections requires a preagreement between client and server as to the IP and port on which the server listens for client connections. When a computer behind a NAT listens on a port, there is no guarantee the WAN port is the same such port. To compensate for this problem, most routers allow the forwarding of specific ports or ranges of ports to a specific LAN IP. This in essence places a static entry into the NAT table.

Port Triggering

Port triggering enables transient port forwarding. Some routers enable port triggering with a table of process names and ports to forward, when requested, to the requesting computer. After the socket closes and the port is subsequently released, the port returns to the pool of ports available for translation by the NAT. This reduces the vulnerability caused by the static nature of port forwarding. It also allows the game, requiring the specific port(s), to work on different computers without the need to manually update port forwarding.

DMZ

Adding a LAN IP to the demilitarized zone (DMZ) setting on a router forwards all ports to that particular computer. This bypasses the NAT, and its security feature of hiding the computer's IP. A computer in the DMZ shares the WAN IP with the router.

Determining WAN IP

When a computer behind a router uses conventional means to determine its own IP address, it receives the LAN IP issued through the DHCP service. No clear-cut method exists to reliably retrieve the WAN IP programmatically. The most reliable, cross-platform-friendly and router-brand-agnostic method is by third party:

Third party: Send a packet to a third party requesting a response containing the IP address they see in the sender portion of the IP packet. Such a third-party tool may be written as a server dedicated to this purpose, or a simple server-side script built for access through HTTP.

UPnP: Universal Plug and Play contains methods to gain the WAN IP for newer routers that support it.

Parse router admin page: Routers have different ways to access this information through administration tools. The Web page admin interface is popular, and a developer could write code to parse the IP from the admin page. The problem is that each router admin page hierarchy and page format differ, thus requiring time-consuming vender-specific special-case code support.

Summary

This survey of multiplayer development began by looking at different multiplayer categories, from split-screen to real-time network connections. The core material focused on dissecting the OSI layers in an average network game. The OSI layers contained mediums, IP, TCP and UDP, sockets, packet presentation, and game logic related to controlling latency. Next, the real-time communication models of broadcasting, peer-to-peer, and client/server were analyzed for strengths and weaknesses, followed by tips for working in multithreaded environments. The chapter concluded with a glimpse into the necessary evils of game security.

No single book covers all the technical details of multiplayer development. Complete coverage would entail a discussion of all the dirty details on numerous platforms on the following subjects: serial communication, server design, network gear and infrastructure, socket programming, voice over IP (VOIP), tools of the trade, unit and beta testing, available middleware analysis, database development, Web development, asynchronous programming, and much greater depth in latency hiding/recovery for every game genre.

Exercises

Protocol Search

Use the RFC Web site (*www.rfc-editor.org/*) and the IANA Web site (*www.iana.org/assignments/port-numbers*) to answer the following questions:

1. What protocols do the following RFCs cover: RFC 791, RFC 792, RFC 793, RFC 768, RFC 2616, RFC 10, and RFC 9?
2. What protocols are associated with the following ports: 80, 3074, 20/21, 1433, and 3306?

Throughput Calculations

Assume the following conditions for problems 3 through 5:

- 256 kbps DSL connection on all endpoints
- Eight clients sending to a dedicated server
- Each client sends at 32 Hz
- Application packet data is 64 bytes per packet

3. Determine client sending saturation given a TCP protocol, packet, and send rate.
4. Determine bandwidth saturation using UDP.
5. How many clients could a server support with a bandwidth of 1 Mbps?

Packet Construction

6. Rewrite the following `Packet` so it does not need to be serialized and is as small as possible. Assume the following conditions:

- System supports a maximum application packet length of 300 bytes.
- Packets will be exchanged between varieties of platforms, including 64/32 bit systems and Windows/Unix.

```
typedef enum PktCode
{
     Pkt0=0,
     ...                          // Other IDs
     PktMax=65000
};

struct Packet
{
     PktCode   ID;
     BOOL      Lights;
     int       HourOfDay;         // 0-23
     short     DayOfWeek;
     int       Health;            // 0-100%
     int       PacketLength;
     char      UserName[64];
};
```

WinSock

Complete Exercise 7 using the ServerMon application server (on the companion CD-ROM) running on a remote system, preferably running Windows Server 2003 or greater with 2 GB of RAM. Client computers should also run with at least 2 GB of RAM for the "Critical Mass" tasks. Lower RAM will limit the system resources required for 30,000 connections due to the kernel page locked memory limitation of Windows.

7. Blackbox:
   a. (optional) If the instructor supplies a Web page containing the IP:Port of the server, acquire that information from the Web page using either the WinINet SDK or MFC HTTP reader class. A more advanced alternative entails reading the HTTP protocol RFC to format a page request and implement a simple HTTP protocol to acquire the Web page.
   b. Connect to that IP:Port given for the Blackbox server, and wait for a `Pkt_Message` (defined in the header file "PacketDefs.h" on the companion CD-ROM) packet containing further instructions. This program will test your ability to host and connect using TCP and UDP, and send and receive data using both protocols.

8. Critical mass:
   a. Make and maintain 30,000 TCP connections.
   b. Listen for, accept, and maintain 30,000 TCP connections (not supported in the provided ServerMon.exe).

Encryption

Use Microsoft's Crypto API to perform the following:

9. Implement public key encryption with a certificate:
   a. Generate a certificate using makecert.exe.
   b. Extract the public key from the certificate to encrypt some plain-text message.
   c. Get the private key associated with the certificate, which was placed in a key container you named during the creation of the certificate. Use this private key to decrypt the cipher text generated in Step b.
10. Implement symmetric key encryption:
    a. Generate a symmetric/secret key.
    b. Encrypt a plain-text message.
    c. Save the secret key to a file.
    d. Load the secret key from the file and use it to decrypt the cipher text generated in Step b.

References

Protocols

[Hind95] Hinden, Robert, "IP Next Generation Overview," available online at *http://playground.sun.com/pub/ipng/html/INET-IPng-Paper.html.*

[IANA] Internet Assigned Number Authority, "Well Known Ports," available online at *www.iana.org.*

[NSIP04] Network Sorcery, "IP, Internet Protocol," available online at *www.networksorcery.com/enp/protocol/ip.htm.*

[NSIP604] Network Sorcery, "IPv6, Internet Protocol version 6," available online at *www.networksorcery.com/enp/protocol/ipv6.htm#Version.*

[NSTCP04] Network Sorcery, "TCP, Transmission Control Protocol," available online at *www.networksorcery.com/enp/protocol/tcp.htm.*

[NSUDP04] Network Sorcery, "UDP, User Datagram Protocol," available online at *www.networksorcery.com/enp/protocol/udp.htm.*

[RFC] Internet Society, "The Request for Comments," available online at *www.rfc-editor.org/.*

[Stevens94] Stevens, Richard, *TCP/IP Illustrated, Volume 1, The Protocols*, Addison-Wesley, 1994.

Communication APIs

[Camp93] Campbell, Joe, *C Programmer's Guide to Serial Communications, Second Edition*, Sams Publishing, 1993.

[Darcy] Darcy, Jeff, "High-Performance Server Architecture," available online at *http://pl.atyp.us/content/tech/servers.html.*

[Jones02] Jones, Anthony, *Network Programming for Microsoft Windows, Second Edition*, Microsoft Press, 2002.

[Kegel00] Kegel, Dan, "Micro benchmark comparing poll, kqueue, and /dev/poll," available online at *www.kegel.com/dkftpbench/Poller_bench.html.*

[Lemon] Lemon, Jonathan, "Kqueue: A generic and scaleable event notification facility," available online at *http://people.freebsd.org/~jlemon/papers/kqueue.pdf.*

[Provos00] Provos, Niels, "Scalable network I/O in Linux," available online at *www.citi.umich.edu/techreports/reports/citi-tr-00-4.pdf.*

[Stevens04] Stevens, Richard, *UNIX Network Programming, Volume 1, Third Edition: The Sockets Networking API*, Addison-Wesley, 2004.

Middleware

[DirectPlay] Available online at *http://msdn.microsoft.com* keyword DirectPlay.

[Quazal] Available online at *www.quazal.com.*

Latency Compensation

[Aronson97] Aronson, Jesse, "Dead Reckoning: Latency Hiding for Networked Games," available online at *www.gamasutra.com/features/19970919/aronson_01.htm*.

[Caldwell00] Caldwell, Nick, "Defeating Lag with Cubic Splines," available online at *www.gamedev.net/reference/articles/article914.asp*.

[Haag01] Haag, Chris, "Targeting: A Variation of Dead Reckoning (v1.0)," available online at *www.gamedev.net/reference/articles/article1370.asp*.

Security

[Coleridge96] Coleridge, Robert, *The Cryptography API, or How to Keep a Secret*, available online at *http://msdn.microsoft.com/library/en-us/dncapi/html/msdn_cryptapi.asp*, August 19, 1996.

[Gibson02] Gibson, Steve, "Distributed Reflection Denial of Service," available online at *http://grc.com/dos/drdos.htm*.

[RSALabs00] RSA Laboratories, *RSA Laboratories' Frequently Asked Questions About Today's Cryptography*, Version 4.1, May 2000.

[Safedisc04] MacroVision, "Safedisc Copy Protection," available online at *www.macrovision.com/products/safedisc/index.shtml*.

[Sch96] Schneier, Bruce, *Applied Cryptography: Protocols, Algorithms, and Source Code in C, 2nd Edition*, John Wiley & Sons, 1996.

[TwoFish96] Schneier, Bruce, *The Twofish Encryption Algorithm*, John Wiley & Sons, 1996.

[Veri04] VeriSign, "Digital ID, A Brief Overview," available online at [VeriSign].

[VeriSign] VeriSign, "Protect Your Digital ID; Protect Your Private Key," available online at *www.verisign.com/repository/PrivateKey_FAQ/*.