ARCHITECTING NETWORKED GAMES

"For any aspiring game programmer, this book is a must read! Glazer and Madhav are some of the best at explaining these critical multiplayer concepts. I look forward to their next book!"

—**ZACH METCALF,** Game Programmer at Rockstar Games and USC Games Alum

MULTIPLAYER GAME Programming

Joshua **GLAZER**
Sanjay **MADHAV**

Multiplayer Game Programming

```
                    }
                }
            }
        }
    }
}
```

The routine begins by creating a listen socket and adding it into the list of sockets to check for readability. Then it loops until the application requests it do otherwise. The loop uses `Select` to block until a packet comes in on any socket in the `readBlockSockets` vector. When a packet does come in, `Select` ensures that `readableSockets` contains only sockets that have incoming data. The function then loops over each socket `Select` has identified as readable. If the socket is the listen socket, it means a remote host has called `Connect`. The function accepts the connection, adds the new socket to `readBlockSockets`, and notifies the application via `ProcessNewClient`. If the socket is not a listen socket, however, the function calls `Receive` to obtain a chunk of the newly arrived data and passes it to the application via `ProcessDataFromClient`.

> **note**
>
> There are other ways to handle incoming data on multiple sockets, but they are platform specific and less commonly used. On Windows, I/O completion ports are a viable choice when supporting many thousands of concurrent connections. More on I/O completion ports can be found in the "Additional Reading" section.

Additional Socket Options

Various configuration options control the sending and receiving behavior of the sockets. To set these values for these options, call `setsockopt`:

```
int setsockopt(SOCKET sock, int level, int optname, const char *optval, int
optlen);
```

`sock` is the socket to configure.

`level` and `optname` describe the option to be set. `level` is an integer identifying the level at which the option is defined and `optname` defines the option.

`optval` is a pointer to the value to set for the option.

`optlen` is the length of the data. For instance, if the particular option takes an integer, `optlen` should be 4.

`setsockopt` returns 0 if successful or –1 if an error occurs.

Table 3.4 lists some useful options available at the SOL_SOCKET level.

Table 3.4 SOL_SOCKET Options

Macro	Value Type (Windows/POSIX)	Description
SO_RCVBUF	int	Specifies the buffer space this socket allocates for received packets. Incoming data accumulates in the receive buffer until the owning process calls recv or recvfrom to receive it. Remember that TCP bandwidth is limited by the receive window's size, which can never be larger than the receive buffer of the receiving socket. Thus, controlling this value can have a significant impact on bandwidth.
SO_REUSEADDR	BOOL/int	Specifies that the network layer should allow this socket to bind an IP address and port already bound by another socket. This is useful for debugging or packet-sniffing applications. Some operating systems require the calling process to have elevated privileges.
SO_RECVTIMEO	DWORD/timeval	Specifies the time (in milliseconds on Windows) after which a blocking call to receive should time out and return.
SO_SNDBUF	int	Specifies the buffer space this socket allocates for outgoing packets. Outgoing bandwidth is limited based on the link layer. If the process sends data faster than the link layer can accommodate, the socket stores it in its send buffer. Sockets using reliable protocols, like TCP, also use the send buffer to store outgoing data until it is acknowledged by the receiver. When the send buffer is full, calls to send and sendto block until there is room.
SO_SNDTIMEO	DWORD/timeval	Specifies the time (in milliseconds on Windows) after which a blocking call to send should time out and return.
SO_KEEPALIVE	BOOL/int	Valid only for sockets using connection-oriented protocols, like TCP; this option specifies that the socket should automatically send periodic keep alive packets to the other end of the connection. If these packets are not acknowledged, the socket raises an error state, and the next time the process attempts to send data using the socket, it is notified that the connection has been lost. This is not only useful for detecting dropped connections, but also for maintaining connections through firewalls and NATs that might time out otherwise.

Table 3.5 describes the `TCP_NODELAY` option available at the `IPPROTO_TCP` level. This option is only settable on TCP sockets.

Table 3.5 `IPPROTO_TCP` Options

Macro	Value Type (Windows/POSIX)	Description
`TCP_NODELAY`	`BOOL/int`	Specifies whether the Nagle algorithm should be ignored for this socket. Setting this to true will decrease the delay between the process requesting data to be sent and the actual sending of that data. However, it may increase network congestion as a result. For more on the Nagle algorithm, see Chapter 2, "The Internet."

Summary

The Berkeley Socket is the most commonly used construct for transmitting data over the Internet. While the library interface differs across platforms, the core fundamentals are the same.

The core address data type is the `sockaddr`, and it can represent addresses for a variety of network layer protocols. Use it any time it is necessary to specify a destination or source address.

UDP sockets are connectionless and stateless. Create them with a call to `socket` and send datagrams on them with `sendto`. To receive UDP packets on a UDP socket, you must first use `bind` to reserve a port from the operating system, and then `recvfrom` to retrieve incoming data.

TCP sockets are stateful and must connect before they can transmit data. To initiate a connection, call `connect`. To listen for incoming connections, call `listen`. When a connection comes in on a listening socket, call `accept` to create a new socket as the local endpoint of the connection. Send data on connected sockets using `send` and receive it using `recv`.

Socket operations can block the calling thread, creating problems for real-time applications. To prevent this, either make potentially blocking calls on non–real-time threads, set sockets to non-blocking mode, or use the `select` function.

Configure socket options using `setsockopt` to customize socket behavior. Once created and configured, sockets provide the communication pathway that makes networked gaming possible. Chapter 4, "Object Serialization" will begin to deal with the challenge of making the best use of that pathway.

Review Questions

1. What are some differences between POSIX-compatible socket libraries and the Windows implementation?
2. To what two TCP/IP layers does the socket enable access?

3. Explain how and why a TCP server creates a unique socket for each connecting client.
4. Explain how to bind a socket to a port and what it signifies.
5. Update `SocketAddress` and `SocketAddressFactory` to support IPv6 addresses.
6. Update `SocketUtils` to support creation of a TCP socket.
7. Implement a chat server that uses TCP to allow a single host to connect and relays messages back and forth.
8. Add support for multiple clients to the chat server. Use non-blocking sockets on the client and `select` on the server.
9. Explain how to adjust the maximum size of the TCP receive window.

Additional Readings

Information Sciences Institute. (1981, September). *Transmission Control Protocol*. Retrieved from http://www.ietf.org/rfc/rfc793. Accessed September 12, 2015.

I/O Completion Ports. Retrieved from https://msdn.microsoft.com/en-us/library/windows/desktop/aa365198(v=vs.85).aspx. Accessed September 12, 2015.

Porting Socket Applications to WinSock. Retrieved from http://msdn.microsoft.com/en-us/library/ms740096.aspx. Accessed September 12, 2015.

Stevens, W. Richard, Bill Fennerl, and Andrew Rudoff. (2003, November 24) *Unix Network Programming Volume 1: The Sockets Networking API, 3rd ed*. Addison-Wesley.

WinSock2 Reference. Retrieved from http://msdn.microsoft.com/en-us/library/windows/desktop/ms740673%28v=vs.85%29.aspx. Accessed September 12, 2015.

This page intentionally left blank

CHAPTER 4

OBJECT SERIALIZATION

To transmit objects between networked instances of a multiplayer game, the game must format the data for those objects such that it can be sent by a transport layer protocol. This chapter discusses the need for and uses of a robust serialization system. It explores ways to handle the issues of self-referential data, compression, and easily maintainable code, while working within the runtime performance requirements of a real-time simulation.