# ﻿Multiplayer Game Programming: Architecting Networked Games (Game Design)

# ﻿Josh Glazer & Sanjay Madhav

# Connection Manager

The job of the connection manager is to abstract the connection between two computers over the network. It receives data from the layer above it, the stream manager, and transmits data to the layer below it, the platform packet module.

The connection manager level is still unreliable. It does not guarantee delivery of data sent to it.

However, the connection manager does guarantee a delivery status notification —that is to

say, the status of a request passed to the connection manager can be verified. In this way, it is possible for the level above the connection manager (the stream manager) to know whether or not particular data was successfully delivered.

The delivery status notification is implemented with a sliding window bit field of acknowledgments. Although the original Tribes Networking Model paper does not contain a detailed discussion regarding the implementation of the connection manager, an implementation of a similar system is discussed in Chapter 7 , “Latency, Jitter, and Reliability.”

# Stream Manager

The primary job of the stream manager is to send data to the connection manager. One

important aspect of this is determining the maximum rate of data transmission that is allowed.

This will vary depending on the quality of the Internet connection. An example given in the

original paper is where a user on a 28.8-kbps modem might have their packet rate set to

10 packets per second with a maximum size of 200 bytes per packet, for approximately 2 kB of

data per second. This rate and size is sent to the server upon connection of the client, in order

to ensure that the server does not overwhelm the client’s connection with too much data.

Since several other systems will ask the stream manager to send data, it is also the duty of the

stream manager to prioritize these requests. The move, event, and ghost managers are given

the highest priority when in a bandwidth-bound scenario. Once the stream manager decides

on what data to send, the packets are dispatched to the connection manager. In turn, the

higher-level managers will be informed by the stream manager regarding the status of delivery.

Because of the set interval and packet size enforced by the stream manager, it is very much

possible for a packet to be dispatched with multiple types of data in it. For example, a packet

may have some data from the move manager, some data from the event manager, and some

data from the ghost manager.

# Event Manager

The event manager maintains a queue of events that are generated by the game’s simulation.

These events can be thought of as a simple form of a remote procedure call or RPC , a

function that can be executed on a remote machine. RPCs are discussed in Chapter 5 , “Object

Replication.”

For example, when a player fires a weapon, this would likely cause a “player fired” event to be

sent to the event manager. This event can then be sent to the server, which will actually validate

and execute the weapon firing. It is also the purview of the event manager to prioritize the

events—it will try to write as many of the highest priority events as possible until any of the

following conditions are true: the packet is full, the event queue is empty, or there are currently

too many active events.

The event manager also tracks the transmission records for each event marked as reliable.

In this way, it is very simple for the event manager to enforce reliability. If a reliable event is

unacknowledged, then the event manager can simply prepend the event to the event queue

and try again. Of course, there will be some events that are marked as unreliable. For these

unreliable events, there is no need to even track their transmission records.

# Ghost Manager

The ghost manager is perhaps the most important system in terms of supporting up to

128 players. At a high level, the job of the ghost manager is to replicate or “ghost” dynamic

objects that are deemed relevant to a particular client. In other words, the server sends

information about dynamic objects to the clients, but only the objects that the server thinks

the client needs to know about. The game’s simulation layer is responsible for determining

what a client absolutely needs to know and what a client ideally should know. This adds an

inherent prioritization to game objects in the world: “need to know” objects are the highest

priority, while “should know” objects are lower priority. In order to determine whether or not

an object is relevant to a particular client, there are several different approaches that can be

employed. Chapter 9 , “Scalability,” covers some of these approaches. In general, determining

object relevancy is very game-specific.

Regardless of how the set of relevant objects is computed, the job of the ghost manager is to

transmit object state from server to client for as many relevant objects as possible. It’s very

important that the ghost manager guarantees that the most recent data is always successfully

transmitted to all of the clients. The reason for this is that the game object information that is

ghosted will often contain information such as health, weapons, ammo count, and so on—all

cases where the most recent data is the only information that matters.

When an object becomes relevant (or “in scope”), the ghost manager will assign some

information to the object, which is appropriately called a ghost record . This record will include

items such as a unique ID, a state mask, the priority, and status change (whether or not the

object has been marked as in or out of scope).

For transmission of the ghost records, the objects are prioritized first by status change and

then by the priority level. Once the ghost manager determines the objects that should be sent,

their data can be added to the outgoing packet using an approach similar to what is covered in

Chapter 5 , “Object Replication.”

# Turn Timers

Since every game instance is performing an independent simulation, it makes sense to utilize

a peer-to-peer topology. As discussed in Chapter 6 , “Network Topologies and Sample Games,”

one advantage of a peer-to-peer model is that data can reach every computer more quickly.

This is because the server is not acting as a middleman. However, one disadvantage is that each

player needs to send their information to every other player, as opposed to just a single server.

So for example, if player A issues an attack command, then every game instance needs to be

aware of this attack command, or their simulations would diverge from each other.

However, there is another key factor to consider. Different players are going to run the game

at different frame rates, and different players are going to have different quality connections.

Going back to the example where player A issues an attack command, it’s just as important

that player A does not immediately apply the attack command. Instead, player A should only

apply the attack command once players B, C, and D are all ready to simultaneously apply the

command. But this introduces a conundrum: If player A’s game waits too long to execute the

attack command, the game will seem very unresponsive.

The solution to this problem is to introduce a turn timer to queue up commands. With the

turn timer approach, first a turn length is selected—in the case of Age of Empires , the default

duration was 200 ms. All commands during these 200 ms are saved into a buffer. When the

200 ms are over, all the commands for that player’s turn are transmitted over the network to all

other players. Another key aspect of this system is a turn execution delay of two turns. What

this means is that, for example, commands that are issued by the player on turn 50 will not be

executed by any game until turn 52. In the case of a 200-ms turn timer, this means that the

input lag , the amount of time it takes for a player’s command to be displayed on screen, could

be as high as 600 ms. However, the two turns of slack allows for every other player to receive

and acknowledge the commands for a particular turn. It may seem slightly counterintuitive for

an RTS game to actually have turns, but you can see the hallmarks of the turn timer approach in

many different RTS games, including StarCraft II . Of course, modern games can have the luxury

of shorter turn timers since bandwidth and latency are much better for most users today in

comparison to the late 1990s.

There is one important edge case to consider with the turn timer approach. What happens

if one of the players experiences a lag spike and they can no longer keep up with the

200-ms timer? Some games might temporarily pause the simulation to see if the lag spike can

be overcome—eventually, the game may decide to drop the player if they continue to slow

down the game for everyone else. Age of Empires also tries to compensate for this scenario by

dynamically adjusting the rendering frame rate based on network conditions—thus a computer

with a particularly slow Internet connection might allocate more time to receive data over the

network, with less time being allotted for rendering graphics. For more detail on the dynamic

turn adjustment, consult the original Bettner and Terrano article listed in the references.

There’s also an extra benefit of transmitting the commands issued by the clients. With such an

approach, it does not take much extra memory or work to save the commands issued over the

course of an entire match. This directly leads to the possibility of implementing savable match

replays, as in Age of Empires II . Replays are very popular in RTS games because it allows players

to evaluate matches to gain a deeper understanding of strategies. It would require significantly

more memory and overhead to create replays in an approach that transmitted unit information

instead of commands.

# Changes in World State

Because each host maintains its own copy of the world state, it is not necessary to replicate

the entire world state in a single packet. Instead, the sender can create packets that represent

changes in world state, and the receiver can then apply these changes to its own world state.

This way, a sender can use multiple packets to synchronize a very large world with a remote host.

When replicating world state in this manner, each packet can be said to contain a world state

delta . Because the world state is composed of object states, a world state delta contains one

object state delta for each object that needs to change. Each object state delta represents one

of three replication actions:

1. Create game object

2. Update game object

3. Destroy game object

Replicating an object state delta is similar to replicating an entire object state, except the sender

needs to write the object action into the packet. At this point, the prefix to serialized data is getting

# Network Topologies

By and large, Chapters 1 to 5 have focused specifically on the issue of two computers

communicating over the Internet and sharing information in a manner that is conducive to

networked games. Although there absolutely are networked two-player games, many of

the more popular games feature higher player counts. But even with only two players, some

important questions arise. How will the players send game updates to each other? Will there

be object replication as in Chapter 5 , or will only the input state be replicated? What happens

if the computers disagree on the game state? These are all important questions that must be

answered for any networked multiplayer game.

A network topology determines how the computers in a network are connected to each other.

In the context of a game, the topology determines how the computers participating in the

game will be organized in order to ensure all players can see an up-to-date version of the game

state. As with the decision of network protocol, there are tradeoffs regardless of the selected

topology. This section explores the two main types of topologies used by games, client-server

and peer-to-peer, and the small variations that can also exist within these types.

Client-Server

In a client-server topology, one game instance is designated the server, and all of the other

game instances are designated as clients. Each client only ever communicates with the server,

while the server is responsible for communicating with all of the clients. Figure 6.1 illustrates

this topology.

A screenshot of a computer

Description automatically generated

In a client-server topology, given n clients there are a total of O(2n) connections. However,

it is asymmetric in that the server will have O(n) connections (one to each client), while each

client will only have one connection to the server. In terms of bandwidth, if there are n clients

and each client sends b bytes per second of data, the server must have enough bandwidth to handle b ̇ n incoming bytes per second. Similarly, if the server needs to send c bytes per second

of data to each client, the server must support c ̇ n outgoing bytes per second. However, each

client need only support c bytes per second downstream and b bytes per second upstream.

This means that as the number of clients increase, the bandwidth required for the server will

increase linearly. In theory, the bandwidth requirements for the client will not change based on

the number of clients. However, in practice, supporting more clients leads to more objects in

the world to replicate, which may lead to a slight increase in bandwidth for each client.

Although by no means the only approach to client-server, most games that implement clientserver

utilize an authoritative server. This means that the game server’s simulation of the

game is considered to be correct. If the client ever finds itself in disagreement with the server,

it should update its game state based on what the server says is the game state. For instance,

in the sample Robo Cat Action game discussed later in this chapter, each player cat can throw

a ball of yarn. But with an authoritative server model, the client is forbidden from making a

determination of whether or not the yarn hits another player. Instead, the client must inform

the server that it wants to throw a ball of yarn. The server then decides both if the client is even

allowed to throw a ball of yarn and, if so, whether or not the other player is hit by the ball of yarn.

By placing the server as an authority, this means there is some amount of “lag” or delay in

actions performed by the client. The topic of latency is discussed in great detail in Chapter 7 ,

“Latency, Jitter, and Reliability,” but a brief discussion is in order. In the case of the ball throw,

the server is the only game instance allowed to make a decision on what happens. But it will

take some time to send a ball throw request to the server, which in turn will process it before

sending the result to all of the clients. One contributing factor of this delay will be the round

trip time , or RTT , which is the amount of time (typically expressed in milliseconds) that it

takes for packets to travel to and back from a particular computer on the network. In an ideal

scenario, this RTT is 100 ms or less, though even on modern Internet connections there are

many factors that may not allow for such a low RTT.

Suppose there is a game with a server and two clients, Clients A and B. Because the server

sends all game data to each client, this means that if Client A throws a ball of yarn, the packet

containing the yarn throw request must first travel to the server. Then the server will process

the throw before sending the result back to Clients A and B. In this scenario, the worst case

network latency experienced by Client B would be equal to ½ Client A’s RTT, plus the server

processing time, plus ½ Client B’s RTT. In fast network conditions, this may not be an issue, but

realistically, most games must use a variety of techniques to hide this latency. This is covered in

detail in Chapter 8 , “Improved Latency Handling.”

There is also a subclassification of types of servers. Some servers are dedicated , meaning

they only run the game state and communicate with all of the clients. The dedicated server

process is completely separate from any client processes running the game. This means that the

dedicated server typically is headless and does not actually display any graphics. This type of

server is often used by big-budget games such as Battlefield , which allows the developer to run

multiple dedicated server processes on a single powerful machine.

# Server Game Instance

Before going on, it is worthwhile to disambiguate some of the overloaded meanings of the

word “server” when used in various contexts. Sometimes “server” refers to an instance of the

class in code that simulates the one true version of the game world and replicates it to clients.

Other times, it refers to the process listening for incoming connections, hosting that class

instance. Still other times, it refers to the physical piece of hardware running that process, as in

“check out all the servers I can fit on this rack.”

To avoid confusion, this chapter uses the term server game instance or just game instance to

represent the entity that simulates the game world and replicates information to clients. The

concept is an abstraction that represents a single reality shared by a group of players playing

together. If your game supports 16-player battles, then a server game instance is a running

16-player battle. In League of Legends it is typically a 5 versus 5 game in the “Summoner’s Rift”

level. In matchmaking terms, it is a single match.

# Game Server Process

A game instance does not exist in a void. It lives inside a game server process , which updates

it, manages its clients, interacts with the operating system, and does everything else a process

typically does. It is the embodiment of your game, as far as the operating system is concerned.

In all previous chapters, the concepts of game server process and game instance were not

separated because there was a one-to-one mapping between them. Each game server process

was responsible for maintaining only one game instance. However, in the world of dedicated

server hosting, that can change.

In properly abstracted code, a single process can manage multiple game instances. As long as

the process updates each instance, binds a unique port for each instance, and does not share

mutable data between the instances, multiple game worlds can coexist peacefully in the same

process.

Multiple instances per process can be an efficient way to host multiple games, because it allows

sharing of large immutable resources like collision geometry, navigation meshes, and animation

data. When multiple game instances run in their own processes, they each need copies of this

data, which can cause unnecessary memory pressure. Games employing multiple instances

per process also benefit from finer control of scheduling: By iterating through each instance

each update, they can assure a roughly regular update pattern across instances. With multiple

processes on the same host, this is not necessarily the case, as the operating system scheduler

decides which process is updated when. This is not always a problem, but finer-grained control

can be useful at times.

The significant advantages of the multi-instance approach may seem compelling, but the

disadvantages of the tactic are just as significant. If a single instance crashes it can bring down

the entire process, with all of its contained game instances. This can be particularly nasty if an