Contents

CHAPTER 1

OVERVIEW OF NETWORKED GAMES

Although there are notable exceptions, the concept of networked multiplayer games didn't really catch on with mainstream gamers until the 1990s. This chapter first gives a brief history of how multiplayer games evolved from the early networked games of the 1970s to the massive industry today. Next, the chapter provides an overview of the architecture of two popular network games from the 1990s—*Starsiege: Tribes* and *Age of Empires*. Many of the techniques used in these games are still in use today, so this discussion gives insight into the overall challenges of engineering a networked multiplayer game.

A Brief History of Multiplayer Games

The progenitor of the modern networked multiplayer game began on university mainframe systems in the 1970s. However, this type of game didn't explode until Internet access became common in the mid-to-late 1990s. This section gives a brief overview of how networked games first started out, and the many ways these types of games have evolved in the nearly half century since the first such games.

Local Multiplayer Games

Some of the earliest video games featured **local multiplayer**, meaning they were designed for two or more players to play the game on a single computer. This included some very early games such as including *Tennis for Two* (1958) and *Spacewar!* (1962). For the most part, local multiplayer games can be programmed in the same manner as single-player games. The only differences typically are multiple viewpoints and/or supporting multiple input devices. Since programming local multiplayer games is so similar to single-player games, this book does not spend any time on them.

Early Networked Multiplayer Games

The first **networked multiplayer games** were run on small networks composed of mainframe computers. What distinguishes a networked multiplayer game from a local multiplayer game is that networked games have two or more computers connected to each other during an active game session. One such early mainframe network was the PLATO system, which was developed at the University of Illinois. It was on the PLATO system that one of the first networked games, the turn-based strategy game *Empire* (1973), was created. Around the same time as *Empire*, the first-person networked game *Maze War* was created, and there is not a clear consensus as to which of these two games was created first.

As personal computers started to gain some adoption in the latter part of the 1970s, developers figured out ways to have two computers communicate with each other over serial ports. A **serial port** allows for data to be transmitted one bit at a time, and its typical purpose was to communicate with external devices such as printers or modems. However, it was also possible to connect two computers to each other and have them communicate via this connection. This made it possible to create a game session that persisted over multiple personal computers, and led to some of the earliest networked PC games. The December 1980 issue of *BYTE Magazine* featured an article on how to program so-called Multimachine Games in BASIC (Wasserman and Stryker 1980).

One big drawback of using serial ports was that computers typically did not have more than two serial ports (unless an expansion card was used). This meant that in order to connect more than two computers via serial port, a **daisy chain** scheme where multiple computers are connected to each other in a ring had to be used. This could be considered a type of network topology, a topic that is covered in far more detail in Chapter 6, "Network Topologies and Sample Games."

So in spite of the technology being available in the early 1980s, most games released during the decade did not really take advantage of local networking in this manner. It wasn't until the 1990s that the idea of locally connecting several computers to play a game really gained traction, as discussed later in this chapter.

Multi-User Dungeons

A **multi-user dungeon** or MUD is a (usually text-based) style of multiplayer game where several players are connected to the same virtual world at once. This type of game first gained popularity on mainframes at major universities, and the term originates from the game *MUD* (1978), which was created by Rob Trushaw at Essex University. In some ways, MUDs can be thought of as an early computer version of the role-playing game *Dungeons and Dragons*, though not all MUDs are necessarily role-playing games.

Once personal computers became more powerful, hardware manufacturers began to offer modems that allowed two computers to communicate with each other over standard phone lines. Although the transmission rates were extraordinarily slow by modern standards, this allowed for MUDs to be played outside the university setting. Some ran MUD games on a **bulletin board system** (BBS), which allowed for multiple users to connect via modem to a system that could run many things including games.

Local Area Network Games

A **local area network** or LAN is a term used to describe several computers connected to each other within a relatively small area. The mechanism used for the local connection can vary—for example, the serial port connections discussed earlier in this chapter would be one example of a local area network. However, local area networks really took off with the proliferation of Ethernet (a protocol which is discussed in more detail in Chapter 2, "The Internet").

While by no means the first game to support LAN multiplayer, *Doom* (1993) was in many ways the progenitor of the modern networked game. The initial version of the id Software first-person shooter supported up to four players in a single game session, with the option to play cooperatively or in a competitive "deathmatch." Since *Doom* was a fast-paced action game, it required implementation of several of the key concepts covered in this book. Of course, these techniques have evolved a great deal since 1993, but the influence of *Doom* is widely accepted. For much greater detail on the history and creation of *Doom*, read *Masters of Doom* (2003), listed in the references at the conclusion of this chapter.

Many games that support networked multiplayer over a LAN also supported networked multiplayer in other ways—whether by modem connection or an online network. For many years, the vast majority of networked games also supported gaming on a LAN. This led to the rise of LAN parties where people would meet at a location and connect their computers to play networked games. Although some networked multiplayer games are still released with LAN play, the trend in recent years seems to have developers forgoing LAN play for exclusively online multiplayer.

Online Games

In an **online game**, players connect to each other over some large network with geographically distant computers. Today, online gaming is synonymous with Internet gaming, but the term "online" is a bit broader and can include some of the earlier networks such as CompuServe that, originally, did not connect to the Internet.

As the Internet started to explode in the late 1990s, online games took off alongside it. Some of the popular games in the earlier years included id Software's *Quake* (1996) and Epic Game's *Unreal* (1998).

Although it may seem like an online game could be implemented in much the same way as a LAN game, a major consideration is **latency**, or the amount of time it takes data to travel over the network. In fact, the initial version of *Quake* wasn't really designed to work over an Internet connection, and it wasn't until the *QuakeWorld* patch that the game was reliably playable over the Internet. Methods to compensate for latency are covered in much greater detail in Chapter 7, "Latency, Jitter, and Reliability" and Chapter 8, "Improved Latency Handling."

Online games took off on consoles with the creation of services such as Xbox Live and PlayStation Network in the 2000s, services that were direct descendants of PC-based services such as GameSpy and DWANGO. These console services now regularly have several million active users during peak hours (though with expansion of video streaming and other services to consoles, not all of these active users may be playing a game). Chapter 12, "Gamer Services," discusses how to integrate one such gamer service—Steam—into a PC game.

Massively Multiplayer Online Games

Even today, most online multiplayer games are limited to a small number of players per game session—somewhere from 4 to 32 is commonly the number of supported players. In a **Massively Multiplayer Online Game** (MMO), however, hundreds if not thousands of players can participate in a single game session. Most MMO games are role-playing games and thus called **MMORPGs**. However, there are certainly other styles of MMO games such as first-person shooters (MMOFPS).

In many ways, MMORPGs can be thought of as the graphical evolution of multi-user dungeons. Some of the earliest MMORPGs actually predated the widespread adoption of the Internet, and instead functioned over dial-in networks such as Quantum Link (later America Online) and CompuServe. One of the first such games was *Habitat* (1986) which implemented several pieces of novel technology (Morningstar and Farmer 1991). However, it wasn't until the Internet became more widely adopted that the genre gained more traction. One of the first big hits was *Ultima Online* (1997).

Other MMORPGs such as *EverQuest* (1999) were also successful, but the genre took the world by storm with the release of *World of Warcraft* (2004). At one point, Blizzard's MMORPG had over

12 million active subscribers worldwide, and the game became such a large part of popular culture that it was featured in a 2006 episode of the animated series *South Park*.

Architecting an MMO is a complex technical challenge, and some of these challenges are discussed in Chapter 9, "Scalability." However, most of the techniques necessary to create an MMO are well beyond the scope of this book. That being said, the foundations of creating a smaller-scale networked game are important to understand before it's possible to even consider creating an MMO.

Mobile Networked Games

As gaming has expanded to the mobile landscape, multiplayer games have followed right along. Many multiplayer games on these platforms are **asynchronous**—typically turn-based games that do not require real-time transmission of data. In this model, players are notified when it is their turn, and have a large amount of time to make their move. The asynchronous model has existed from the very beginning of networked multiplayer games. Some BBS only had one incoming phone line connection, which meant that only one user could be connected at any one time. Thus, a player would connect, take their turn, and disconnect. Then at some point in the future, another player would connect and be able to respond and take their own turn.

An example of a mobile game that uses asynchronous multiplayer is *Words with Friends* (2009). From a technical standpoint, an asynchronous networked game is simpler to implement than a real-time one. This is especially true on mobile platforms, because the platform APIs (application program interfaces) have built-in functionality for asynchronous communication. Originally, using an asynchronous model for mobile games was somewhat out of necessity because the reliability of mobile networks is comparatively poor to wired connections. However, with the proliferation of Wi-Fi–capable devices and improvements to mobile networks, more and more real-time networked games are appearing on these devices. An example of a mobile game that takes advantage of real-time network communication is *Hearthstone: Heroes of Warcraft* (2014).

Starsiege: Tribes

Starsiege: Tribes is a sci-fi first-person shooter that was released at the end of 1998. At the time of release, it was well regarded as a game featuring both fast-paced combat and a comparatively massive number of players. Some game modes supported 128 players over either a LAN or the Internet. To gain some perspective on the magnitude of the challenge in implementing such a game, keep in mind that during this time period, the vast majority of players with an Internet connection used a dial-up service. At best, these dial-up users had a modem capable of speeds up to 56.6 kbps. In the case of *Tribes*, it actually supported users with modem speeds of only 28.8 kbps. By modern standards, these are extremely slow connection speeds. Another factor was that dial-up connections also had relatively high latency—a latency of several hundred milliseconds was rather common.

It may seem that a networking model designed for a game with low bandwidth constraints would be irrelevant in the modern day. However, it turns out that the model used in *Tribes* still has a great deal of validity even today. This section summarizes the original *Tribes* networking model—for a more in-depth discussion, refer to the article by Frohnmayer and Gift referenced at the end of this chapter.

Do not be concerned if some of the concepts covered in this section don't entirely make sense right now. The intent is that by looking at a networked multiplayer game's architecture at a high level, you will gain an appreciation for the numerous technical challenges faced and decisions to be made. All the topics touched on in this section are covered in much greater detail throughout the remainder of this book. Furthermore, one of the sample games built throughout this book, *RoboCat Action*, ultimately uses a model similar to the *Tribes* networking model.

One of the first choices made when engineering a networked game is to choose a **communications protocol**, or an established convention by which data is exchanged between two computers. Chapter 2, "The Internet," covers how the Internet works and the commonly used protocols. Chapter 3, "Berkeley Sockets," covers a ubiquitous library used to facilitate communication via these protocols. For the sake of the current discussion, the only thing you need to know is that, for efficiency reasons, *Tribes* uses an *unreliable* protocol. This means that data sent over the network is *not* guaranteed to be received by the destination.

However, using an unreliable protocol can be problematic when a game needs to send information that is important to all the players in the game. Thus, the engineers needed to consider the different types data they wanted to send out. The developers of *Tribes* ultimately separated their data requirements into the following four categories:

1. **Non-guaranteed data.** As one might expect, this is data that the game designates as nonessential to the game. When bandwidth-starved, the game can choose to drop this data first.
2. **Guaranteed data.** This data guarantees both arrival and ordering of the data in question. This is used for data deemed critical by the game, such as an event signifying when a player has fired a weapon.
3. **"Most recent state" data.** This type of data is for cases where only the most recent version of the data is of importance. One example is the hit points of a particular player. A player's hit points 5 seconds ago are not terribly relevant if the game knows what their hit points are right now.
4. **Guaranteed quickest data.** This data is given the highest priority in order to transmit as quickly as possible *with* guaranteed delivery. An example of this type of data is player movement information, which is typically relevant for a very short period of time, and thus should be transmitted quickly.

Many of the implementation decisions made in the *Tribes* Networking Model center on providing these four types of data transmission.

Another important design decision was to utilize a client-server model instead of a peer-to-peer model. In a **client-server model**, players all connect to a central server, whereas in a **peer-to-peer model**, every player connects to every other player. As discussed in Chapter 6, "Network Topologies and Sample Games," a peer-to-peer model requires $O(n^2)$ bandwidth. This means that the bandwidth grows at a quadratic rate based on the number of users. In this case, with n being as high as 128, using peer-to-peer would lead to very little bandwidth per player. To avoid this issue, *Tribes* instead implemented a client-server model. In this configuration, the bandwidth requirements of each player remain constant, while the server must handle only $O(n)$ bandwidth. However, this meant that the server needed to be on a network that would allow for several incoming connections—the type of connection that only a company or university might have owned at the time.

Next, *Tribes* split up their networking implementation into several different layers—one can think of this as a "layer cake" of the *Tribes* Networking Model. This is illustrated in Figure 1.1. The remainder of this section briefly describes the composition of each of these layers.

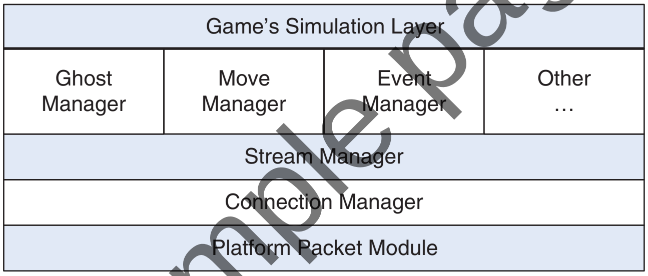

Figure 1.1 The main components of the *Tribes* Networking Model

Platform Packet Module

A **packet** is a formatted set of data sent over a network. In the *Tribes* model, the **platform packet module** is the lowest layer. It is the only layer in the model that is platform-specific. In essence, this layer is a wrapper for the standard socket APIs that can construct and send various packet formats. The implementation of this layer might look rather similar to the systems implemented in Chapter 3, "Berkeley Sockets."

Since *Tribes* utilized an unreliable protocol, the developers needed to add some mechanism to handle the data they decided needed to be guaranteed. Similar to the approach discussed in Chapter 7, "Latency, Jitter, and Reliability," *Tribes* implemented a custom reliability layer. However, this reliability layer is not handled by the platform packet module; instead the higher level managers such as the ghost manager, move manager, or event manager are responsible for adding any reliability.

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.”

Move Manager

The responsibility of the **move manager** is to transmit player movement data as quickly as possible. If you've ever played a fast-paced multiplayer game, you are likely cognizant of the fact that accurate movement information is extremely important. If the information regarding a player's position is slow to arrive, it could result in players shooting at where a player used to be instead of where a player is, which can result in frustrating gameplay. Quick movement updates can be an important way to reduce the perception of latency on the part of player.

The other reason the move manager is assigned a high priority is because input data is captured at 30 FPS. This means there is new input information available 30 times per second, so the latest data is sent as quickly as possible. This higher priority also means that, when move data is available, the stream manager will always first add any pending move manager data to an outgoing packet. Each client is responsible for transmitting their move information to the server. The server then applies this move information in its simulation of the game, and acknowledges the receipt of the move information to the client who sent it.

Other Systems

There are a few other systems in the *Tribes* model, though these are less critical to the overarching design. For example, there is a datablock manager, which handles transmission of game objects that are relatively static in nature. This differs from the relatively dynamic objects that are handled by the ghost manager. An example for this might be a static vehicle such as a turret—the object doesn't really move, but it exists to serve a purpose when a player interacts with it.

Age of Empires

As with *Tribes*, the real-time strategy (RTS) game *Age of Empires* was released in the late 1990s. This means that *Age of Empires* faced many of the same bandwidth and latency constraints of dial-up Internet access. *Age of Empires* uses a **deterministic lockstep** networking model. In this model, all the computers are connected to each other, meaning it is peer-to-peer. A guaranteed *deterministic* simulation of the game is concurrently performed by each of the peers. It is *lockstep* because peers use communication to ensure that they remain synchronized throughout the game. As with *Tribes*, even though the deterministic lockstep model has existed for many years, it is still commonly used in modern RTS games. The other sample game built during the course of this book, *RoboCat RTS*, implements a deterministic lockstep model.

One of the largest differences between implementing networked multiplayer for an RTS instead of an FPS is the number of relevant units. In *Tribes*, even though there are up to 128 players, at any particular point in time only a fraction of these players is going to be relevant to a particular client. This means that the ghost manager in *Tribes* rarely has to send information about more than 20 to 30 ghosts at a time.

Contrast this with an RTS such as *Age of Empires*. Although the player cap is much smaller (limited to eight simultaneous players in the original game), each player can control a large number of units. The original *Age of Empires* capped the number of units for each player at 50, whereas in later games the cap was as high as 200. Using the cap of 50, this means that in a massive eight-player battle, there could be up to 400 units active at a time. Although it is natural to wonder if some sort of relevancy system could reduce the number of units that need to be synchronized, it's important to consider the worst-case scenario. What if a battle toward the end of a game featured the armies of all eight players? In this case, there are going to be several hundred units that are relevant at the same time. It would be hard for the synchronization to keep up even if a minimal amount of information is sent per unit.

To alleviate this issue, the engineers for *Age of Empires* decided to synchronize the *commands* each player issued, rather than synchronizing the units. There's a subtle but important distinction in this implementation—even a professional RTS player may be able to issue no more than 300 commands per minute. This means that even in an extreme case, the game need only transmit a few commands per second per each player. This requires a much more manageable amount of bandwidth than transmitting information about several hundred units. However, given that the game is no longer transmitting unit information over the network, each instance of the game needs to independently apply the commands transmitted by each player. Since each game instance is performing an independent simulation, it is of the utmost importance that each game instance remains synchronized with the other game instances. This ends up being the largest challenge of implementing the deterministic lockstep model.

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.

Synchronization

Turn timers alone are not enough to guarantee synchronization between each peer. Since each machine is receiving and processing commands independently, it is of the utmost importance that each machine arrives at an identical result. In their paper, Bettner and Terrano write that "the difficulty with finding out-of-sync errors is that very subtle differences would multiply over time. A deer slightly out of alignment when the random map was created would forage slightly differently—and minutes later a villager would path a tiny bit off, or miss with his spear and take home no meat."

One concrete example arises from the fact that most games have some amount of randomness in actions. For instance, what if the game performs a random check in order to determine whether or not an archer hits an infantry? It would be conceivable that player A's instance decides the archer does hit the infantry, whereas player B's instance decides the archer doesn't hit the infantry. The solution to this problem is to exploit the "pseudo" prefix of the **pseudo-random number generator** (PRNG). Since all PRNGs use some sort of seeding, the way you can guarantee both players A and B arrive at the same random results is to synchronize the seed value across all game instances. One should keep in mind, however, that a seed only guarantees a particular sequence of numbers. So not only is it important that each game instance uses the same seed, it's equally important that each game instance makes the same number of calls to the random generation number—otherwise the PRNG numbers will become out of sync. PRNG synchronization in a peer-to-peer configuration is further elaborated in Chapter 6, "Network Topologies and Sample Games."

There is also an implicit advantage to checking for synchronization—it reduces the opportunity for players to cheat. For example, if one player gives themselves 500 extra resources, the other game instances could immediately detect the desynchronization in the game state. It would then be trivial to kick the offending player out of the game. However, as with any system, there are tradeoffs—the fact that each game state simulates each unit in the game means that it is possible to create cheats that reveal information that should not be visible. This means that the so-called "map hacks" that reveal the entire map are still a common issue in most RTS games. This and other security concerns are covered in Chapter 10, "Security."

Summary

Networked multiplayer games have a lengthy history. They began as games playable on networks of mainframe computers, such as *Empire* (1973), which was playable on the PLATO network. Networked games later expanded to text-based multi-user dungeon games. These MUDs later expanded to bulletin board systems which allowed for users to dial in over phone lines.

In the early 1990s, local area network games, led by *Doom* (1993), took the computer gaming world by storm. These games allowed for players to locally connect multiple computers and play with or against each other. As adoption of the Internet expanded in the late 1990s, online games such as *Unreal* (1998) became very popular. Online games also started to see adoption on consoles in the early 2000s. One type of online game is the massively multiplayer online game, which supports hundreds if not thousands of players in the same game session at once.

Starsiege: Tribes (1998) implemented a network architecture still relevant to a modern-day action game. It uses a client-server model, so each player in the game is connected to a server that coordinates the game. At the lowest level, the platform packet module abstracts sending packets over the network. Next, the connection manager maintains connections between the players and the server, and provides delivery status notifications. The stream manager takes data from the higher-level managers (including the event, ghost, and move managers), and based on priority, adds this data to outgoing packets. The event manager takes important events, such as "player fired" and ensures that this data is received by the relevant parties. The ghost manager handles sending object updates for the set of objects deemed relevant for a particular player. The move manager sends the most recent movement information for each player.

Age of Empires (1997) implemented a deterministic lockstep model. All computers in the game connect to each other in a peer-to-peer manner. Rather than sending information about each unit over the network, the game instead sends commands to each peer. These commands are then independently evaluated by each peer. In order to ensure the machines stay synchronized, a turn timer is used to save up commands over a period of time before sending them over the network. These commands are not executed until two turns later, which gives enough time for each peer to send and receive turn commands. Additionally, it is important that each peer runs a deterministic simulation, which means, for example, pseudo-random number generators need to be synchronized.

Review Questions

1. What is the difference between a local multiplayer game and a networked multiplayer game?
2. What are three different types of local network connections?
3. What is a major consideration when converting a networked game that works over a LAN to work over the Internet?
4. What is an MUD, and what type of game did it evolve into?
5. How does an MMO differ from a standard online game?
6. In the *Tribes* model, which system(s) provide reliability?
7. Describe how the ghost manager in the *Tribes* model reconstructs the minimal necessary transmission in the event that a packet is dropped.
8. In the *Age of Empires* peer-to-peer model, what is the purpose of the turn timer? What information is transmitted over the network to the other peers?

Additional Readings

Bettner, Paul and Mark Terrano. "1500 Archers on a 28.8: Network Programming in Age of Empires and Beyond." Presented at the Game Developer's Conference, San Francisco, CA, 2001.

Frohnmayer, Mark and Tim Gift. "The Tribes Engine Networking Model." Presented at the Game Developer's Conference, San Francisco, CA, 2001.

Koster, Raph. "Online World Timeline." *Raph Koster's Website*. Last modified February 20, 2002. http://www.raphkoster.com/gaming/mudtimeline.shtml.

Kushner, David. *Masters of Doom: How Two Guys Created an Empire and Transformed Pop Culture*. New York: Random House, 2003.

Morningstar, Chip and F. Randall Farmer. "The Lessons of Lucasfilm's Habitat." In *Cyberspace: First Steps*, edited by Michael Benedikt, 273-301. Cambridge: MIT Press, 1991.

Wasserman, Ken and Tim Stryker. "Multimachine Games." *Byte Magazine*, December 1980, 24-40.