Distributed Computing and Computer Networking in Multiplayer Games

David Le

University of Manitoba

August 14, 2020

One of the biggest growing genres of video games currently are online multiplayer games. Online multiplayer games are games that can be played with other players over the internet. There are several different types of multiplayer games, with varying amounts of concurrent players and population. A first-person shooter can be played with a lobby of two to sixteen players, while a massively multiplayer online role-playing game (MMORPG) can host a world containing hundreds to thousands of players at a time. What will be discussed are some distributed computing and networking techniques used to create multiplayer games. These include different types of distributed architectures, transport protocols for different types of games, and common synchronization techniques and methods for both client-server and peer-to-peer architectures.

When developing a game with a multiplayer component, consider what kind of distributed structure and architecture to use. Two common types of distributed architectures are the client-server architecture, and the P2P or peer-to-peer architecture. Both the usage of a client-server architecture and peer-to-peer architecture will be discussed.

A client-server architecture is a system that works by having a server host and manage services and messages that are delivered to clients. When developing a game, a developer can approach the client-server model in different ways with the concept of load balancing, that is the process of distributing tasks over a set of servers. For example, in a massively multiplayer online game (MMO), a single server can be responsible for the entire world, have multiple servers for each distinct world, or have each instance/region of the MMO in different servers (Yahyavi and Kemme, 2013, p. 10). Other servers can be used for different services as well, such as a server to maintain a chat service. Depending on the load balancing approach taken, it can help prevent a single point of failure, reduce server downtime, and create a high availability.

When using a client-server architecture to help build a game server, one thing to consider is having a stateful server versus a stateless server. In a stateful server, the server holds the state, while in a stateless server, the client generally holds the state. Many games use both a stateful and stateless server. Many games also have varying levels of relationships between client and server which scales from being a fully authoritative server to a more non-authoritative server.

The majority of games using a client-server architecture have authoritative stateful servers. This means that the server holds the state, and clients simply display and replicate the

state of the game. Thus, the server is responsible for the game logic and decisions and is the final authority, while the client only needs to render results. A fully authoritative game has the advantage of being resilient to cheating and malicious behaviour, as the server is fully responsible for logic and can verify the data. They are also easy to synchronize states between multiple clients, as the server maintains the state of the game, and the client simply displays what it is sent. A drawback to this client-server relationship is that the server will need to do more calculations, as it is responsible for all of the logic, which increases server hardware costs. Clients may also have input delay, in which client-side prediction will be needed. Examples of games that use a fully authoritative server are League of Legends, a competitive MOBA (Massive Online Battle Arena), and Overwatch, a competitive first-person shooter. These types of competitive game demand real-time synchronization and anti-cheat, in which an authoritative server can easily implement.

A non-authoritative server is similar to a stateless server, in which the client has more control and information about the state of the game. A non or partial-authoritative server requires less bandwidth between client and server as the client does most of the calculations and sends data to the server. This results in a cheaper operating cost as the client will do most of the work and will relay data to the server. Despite this, there are several disadvantages to having this client-server relationship. Limitations of the relationship include desynchronization issues happening more often, as the server is not the final authority game state, and that cheating and malicious behaviour is much more common and easier to do (IT Hare on Soft.ware, 2015). Examples of games that are non-authoritative or mostly non-authoritative are Rust and The Forest.

Generally, multiplayer game servers are stateful, but many components of games can be stateless. Some multiplayer games use RESTful APIs, an application program interface that is based on the REST architectural style. REST uses HTTP, a stateless protocol, to send, receive, update, and delete data from a web server. Examples of where RESTful APIs can be used would be ranking leaderboards or match history in a first-person shooter, or a friend list and inventory screen in an MMORPG. RESTful APIs are used here as there is not a need for a constant connection, and rather can be called when needed. Examples of RESTful APIs used in

multiplayer games would include the Riot Games API, used to retrieve stats and data for Riot games such as League of Legends and Valorant.

The advantages of using RESTful APIs are that it allows for easy horizontal scaling, that is scaling by adding more machines, is platform-independent, and allows for easy integration for services required in the game (Cooper and Walt Scacchi, 2015, p. 162). Drawbacks to a server using RESTful APIs are that there is a higher latency to process client requests due to using HTTP, more overhead between the server and the database, and that the database can become a bottleneck in the system from constant queries. (Cooper and Walt Scacchi, 2015, pp. 162-163).

A peer-to-peer architecture is a system that works by having each peer or client communicate with one another. Each peer is a node, and share messages with one another. In multiplayer games, there are generally two approaches to peer-to-peer architectures. One approach mimics a client-server architecture by having one client become a server host, and all the other clients connect to the server host. The latter approach is to have each peer maintain their own copy of the game state and share their state and events to other peers on the network.

Following the latter approach, the peer-to-peer architecture is based on fully connected overlay networks, a network that is built on top of another network, which in this case is a network over the internet (Ferretti, 2005, p. 15). Each peer maintains its own copy of the game state and are connected with one another. Each event that is produced by a player will be sent to all of the other players. Thus, each peer depends on one another to send state updates to progress the game.

Types of games that commonly take advantage of a peer-to-peer architecture are most console games, fighting games, strategy and turn-based strategy games, and many cooperative games. Two games that use a peer-to-peer architecture would be the turn-based strategy game series Civilization, or the real-time strategy (RTS) game StarCraft. Games in general can be characterized by a set of states that are modified by events and logic as the game plays through (Neumann et al, 2007, p. 2). These two games have very large game states; players often control hundreds to thousands of units at a time. Because of this, it is more efficient for each peer to calculate game logic and events, and then send updates to each other to synchronize game state, rather than rely on a server to do all of the calculations. This helps reduce the amount of latency in games with very large game states.

There are several advantages when using a peer-to-peer architecture. The advantages of a peer-to-peer architecture in multiplayer games are their robustness (Ferretti, 2005, p. 16). As there is no centralized server, there is not a single point of failure. It is inexpensive to implement, as each peer will be sharing data with each other, and there will be no server to host and manage. It has a reduced message latency, as there is no intermediary to relay messages, and instead have peers directly communicate with each other. Players do not have to worry about defunct servers, so long as there are other peers that are willing to play the game.

There are several problems with using a peer-to-peer architecture for multiplayer games. One issue is that in a peer-to-peer architecture, there is no authority, as each peer maintains its own state of the game. The events and data that are sent by each peer are not verifiable by a server. Thus, unless anti-cheats are implemented, malicious and cheating behaviour can easily occur. Peer-to-peer architectures do not scale well, because as more nodes connect with one another, the stability of the network decreases, and latency increases (Li, n.d, pp. 2-3). Thus, multiplayer games using a peer-to-peer architecture for gameplay are more suited to having a small number of players in each game session, compared to a client-server architecture that can serve a large number of concurrent players.

Comparing the two different architectures, a client-server architecture generally holds the game state, and distribute it around clients, while a peer-to-peer architecture contains multiple states held by the multiple clients. Peer-to-peer architectures are easier and cheaper to implement, while client-server architectures are far more expensive and harder to implement. Client-server architectures are more stable and scalable, while peer-to-peer architectures have bad scalability as more players are added per game session. From this, a client-server architecture is preferred for scalability and is suitable for large-scale multiplayer games or for games that have large concurrent player counts. A peer-to-peer architecture would be used for a cheaper implementation of a multiplayer component, and for games with a small lobby of players playing (often between two – sixteen players). Despite the differences in architecture, some games use both architectures. For example, a small-scale first-person shooter may have a dedicated server whose sole purpose is to be a matchmaking service, pairing players together for them to play the game through a peer-to-peer network, while having RESTful APIs and web services to update the scoreboard and create a match history.

When developing a multiplayer game, an important decision to make is choosing what kind of transport protocol to use. Two of the most common protocols that are used are the Transmission Control Protocol (TCP), and the User Datagram Protocol (UDP). TCP is a connection-oriented transport protocol, while UDP is a connectionless transport protocol. TCP creates a two-way connection between a client and a server, and a connection must be established before data can be sent. TCP is generally reliable, as data sent using TCP is guaranteed to be delivered to the client, and lost data will be resent. TCP sends data in order, so a client will receive data in the same order as they were sent. UDP is an unreliable protocol that sends data constantly to a client whether or not the client receives them. UDP does not need an established connection, does no error checking, and can send data in any order. This leads to UDP being much quicker than TCP, at the cost of order and reliability.

When developing a real-time game such as a first-person shooter, there is a real-time requirement on packet delivery (Fiedler, 2008). Real-time games generally do not care about what happened in the past, but rather only care about the most recent event. Thus, many real-time games employ the use of UDP, as retransmission from TCP will create a lot of overhead and lag, and small amounts of packet loss will not affect gameplay much.

Though many games use UDP as their protocol, TCP can be used as well. Multiplayer HTML5 games such as Agar.io, Slither.io, and MMORPGs such as World of Warcraft are entirely built with TCP (Wu et al, 2009, p. 8). Although HTML5 games are forced to use TCP due to being built on WebSockets, TCP can be useful as it guarantees that every data will be received. Due to its high reliability, it can be used for player sensitive content and services that require ordered and reliable data to be sent such as player authentication and chat services, or for turn-based games where speed and time are not an issue.

The biggest disadvantage of games using TCP is that it can significantly increase the amount of latency and input lag a client may have. As TCP guarantees that a client receives the data that was sent, dropped data will be resent which can cause hang-ups and freezes as later important data will be sent and received after retransmission.

Despite the different qualities of both UDP and TCP, many multiplayer games integrate both the use of UDP and TCP as their transport protocol. TCP can be used for when reliable, ordered data is needed, and UDP will be used for input and state. For example, TCP can be used

for lobbies, chat services, and non time-sensitive gameplay while UDP can be used for the time-sensitive gameplay.

A complex issue that comes with creating a multiplayer game is synchronization and state. Depending on the type of game, synchronization can play a huge role in the quality of the game. While a turn-based strategy game may only need to change state at each turn, time-sensitive games will require players to be synchronized in real-time. When changing and synchronizing the state of the game, it is generally not feasible to transmit the full state of the game, as this would take far more bandwidth for one update compared to updating a few states in the game, which can result in higher latency. Thus, there are several techniques and algorithms useful for synchronization depending on the architecture, complexity, and size of the game. There are several algorithms and protocols used to synchronize state across multiple players, such as lockstep and its derivatives for peer-to-peer architectures, and RPC and pub/sub message broker models for client-server architecture games.

A commonly used algorithm to synchronize peer-to-peer architecture games is the Lockstep algorithm. Basic lockstep works by collecting the data of each peer and only continue with game logic once all data is received by every peer (Cronin, Filstrup and Kurc, 2001, p. 23). This data would include player states such as the position of a character, or an event caused by the player. Basic lockstep guarantees that each peer will see the same game state. The major downside of a lockstep algorithm is that the game speed is dictated by the peer with the highest latency. Thus, a peer with high latency will slow down the game for everyone, creating an unenjoyable experience.

However, if the game is deterministic, that is every input sent will always produce the same state on all machines and systems, then a deterministic lockstep algorithm can be used instead. Similar to the basic lockstep, deterministic lockstep sends input, rather than data to every peer, and continues with game logic once all inputs are received (Sun, 2019). Thus, a player will update their own game state by inputting the commands from other peers. Since the game is deterministic, then every player will have the same state once game logic continues. This significantly reduces the amount of bandwidth overhead used, as inputs take fewer resources than data and events, and helps lessen the strain that a high latency player will have.

A simple way a multiplayer game using a client-server architecture can change states is by using remote procedure calls (RPC). Remote procedure calls are functions that are called locally but execute on a remote host. In multiplayer games, a client may invoke a function to a server, and a server may invoke a function to one or more clients. RPCs are primarily used for in-game events that are small, short, cosmetic, or infrequent (docs.unrealengine.com, n.d.). Examples of such events would be a server invoking a function to a client to play sounds and effects, or a client invoking a function to a server to change the state of an object in the game.

A publish-and-subscribe message broker is a common and viable solution to synchronizing states across several clients. A message broker is an intermediary or middleman between the clients, and several servers/services. They help the client and server communicate and exchange info through messages. The publish-and-subscribe pattern has a publisher/sender send a message as a topic, and any subscribers/recipients that are subscribed to the topic will receive the message. The message broker does things such as decoupling messages through message queues and routes messages.

A pub/sub model can be used for different parts of a game. Two different types of pub/sub models would be a topic-based, where messages are published to topics for subscribers to receive, and content-based, where messages are only sent to the subscriber if the content of the message matches the constraints set by the subscriber (Cañas et al, 2014, p. 2). A simple example would be using a topic-based pub/sub model to create an in-game service such as a chat system. Each player can connect and subscribe to different channels that serve as a topic. Then, a user sends a message, it will be published to the topic, in which topic subscribers will receive the message in-game.

A larger example would be using a content-based pub/sub model to specify a player's location in an open-world game and share with other players (Cañas et al, 2014, p. 6). A player's location can be attributed with two to three variables, an x-coordinate, y-coordinate, and in a 3D game, a z-coordinate. A player can set a constraint where other players will only be visible if they are within a certain x, y, z-coordinate. Then, a player can subscribe to the constraint and listen to players that exist within a certain x, y, z-coordinates. From this, a player only needs to publish their coordinates, and every player whose constraint is within the coordinates will be able to see that player.

While time-sensitive games may not have perfect synchronization, latency issues can be alleviated through several methods and techniques. Not going too in-depth, some methods can include client prediction through extrapolation and interpolation (Bernier, 2001). These techniques can help smooth out gameplay and make the game look like it has less lag than it should have.

In conclusion, several different distributed architecture structures can play a major role in developing a robust multiplayer component of a game. Client-server architectures are more scalable, resilient to cheating, at the cost of being more expensive to their peer-to-peer counterparts that are cheaper and easier to implement but is more prone to cheating does not scale as well. Some games use both architectural styles depending on the services they need. The use of UDP protocol is preferred when dealing with real-time gameplay, compared to TCP, which is used when reliability and ordered delivery is needed. A combination of both transport protocols can be effectively used for different services and aspects of the game. Several different types of synchronization techniques can be used for different types of distributive structures.

Reference list

Bernier, Y. (2001). *Latency Compensating Methods in Client/Server In-game Protocol Design and Optimization*. [online] Available at: https://web.cs.wpi.edu/~claypool/courses/4513-B03/papers/games/bernier.pdf.

Cañas, C., Zhang, K., Kemme, B., Kienzle, J. and Jacobsen, H.-A. (2014). Publish/subscribe Network Designs for Multiplayer Games. *Proceedings of the 15th International Middleware Conference on - Middleware '14*.

Cooper, K.M.L. and Walt Scacchi (2015). *Computer Games and Software Engineering*. Boca Raton: Chapman & Hall/Crc.

Cronin, E., Filstrup, B. and Kurc, A. (2001). *A Distributed Multiplayer Game Server System*. [online] Available at: https://www.cs.ubc.ca/~krasic/cpsc538a/papers/quakefinal.pdf.

docs.unrealengine.com. (n.d.). *RPCs*. [online] Available at: https://docs.unrealengine.com/en-US/Gameplay/Networking/Actors/RPCs/index.html.

Ferretti, S. (2005). *Interactivity Maintenance for Event Synchronization in Massive Multiplayer Online Games*. [online] Available at: https://pdfs.semanticscholar.org/0869/4530132ad182a62f8016b360627ff8d45d31.pdf.

Fiedler, G. (2008). *UDP vs. TCP*. [online] Gafferongames.com. Available at: https://gafferongames.com/post/udp_vs_tcp/.

IT Hare on Soft.ware. (2015). *On Cheating, P2P, and [non-]Authoritative Servers from "D&D of MMOG" Book*. [online] Available at: http://ithare.com/chapter-iii-on-cheating-p2p-and-non-authoritative-servers-from-dd-of-mmog-book/.

Li, W. (n.d.). *Scalability and Consistency in Peer to Peer Based Network*. [online] Available at: https://liwei8257.files.wordpress.com/2014/05/p2p-network-game.pdf.

Neumann, C., Prigent, N., Varvello, M. and Suh, K. (2007). Challenges in peer-to-peer gaming. *ACM SIGCOMM Computer Communication Review*, [online] 37(1), p.79. Available at: http://ccr.sigcomm.org/online/files/p2p_gaming.pdf.

Sun, R. (2019). *Game Networking Demystified, Part III: Lockstep*. [online] ruoyusun.com. Available at: https://ruoyusun.com/2019/04/06/game-networking-3.html.

Wu, C.-C., Chen, K.-T., Chen, C.-M., Huang, P. and Lei, C.-L. (2009). On the Challenge and Design of Transport Protocols for MMORPGs. *Multimedia Tools and Applications*, [online] 45(1–3), pp.7–32. Available at: https://www.iis.sinica.edu.tw/~swc/pub/tcp_mmorpg.html.

Yahyavi, A. and Kemme, B. (2013). Peer-to-peer architectures for massively multiplayer online games. *ACM Computing Surveys*, [online] 46(1), pp.1–51. Available at: http://www.contrib.andrew.cmu.edu/~ayahyavi/files/Yahyavi-CSUR13-P2PMMOG.pdf.