Viability of Peer-to-Peer Architectures for Large Fast-Paced Videogames

Jonah Henriksson, Computer Science, Dr. Vincent Robles, Department of Computer Science

For information on network architectures, please read Appendix A, which covers what Client-Server and Peer-to-Peer architectures are and their differences.

Objective

Multiplayer games vary in game speed (how fast the game progresses) but usually this rate can be classified either as slow-paced or fast-paced. This classification is more useful than the classification of turn-based or real-time, which only describes the style of game speed. Fast-paced games are more constrained than slow-paced: latency can't be too high, or the immersion is broken. Slow-paced games don't have the same constraints with latency, which means that they can support a larger number of players. For the sake of this project, let's define fast-paced as requiring a latency of lower than 100ms and large as having at least 25 players in a match. Most games in both categories have been implemented to run on C/S architectures. However, there has been research showing the viability of slow-paced games on P2P architectures. Meanwhile, there is little research on the viability of fast-paced games on P2P architectures, and even less (if any) on large fast-paced games. This project will implement and test a large, real-time demo on a P2P architecture to determine the viability of P2P architectures for such multiplayer video games.

Background and Significance

Previous research into P2P architectures for multiplayer games has focused on slow-paced games, like Massively Multiplayer Online (MMO) games, which have an extremely large capacity for players, but sacrifice latency to do so. Many authors have made a focus of using *locality* (separating players into manageable groups, also called *areas of interest*) to support the large capacity of these games (Peer-to-Peer Support for Massively Multiplayer Games; Rooney, S., et al.; Triebel, Tonio, et al.). Locality isn't a technique limited to P2P architectures but can be used to organize P2P networks. However, locality

makes sense for slow-paced games like MMOs because there isn't a need for all players to be interacting at once. However, in a fast-paced game, like a first-person shooter (FPS) that has a team deathmatch mode, all players are in one area, so locality can't be used. The Donnybrook architecture aims to be P2P architecture for fast-paced FPS games (Bharambe, Ashwin, et al.). It addresses locality and proposes using *interest sets* which determine which updates a player receives, which is more suited for fast-paced games. Donnybrook focuses on disseminating updates through multicast trees, which raises the question of how this architecture works on the public internet (multicast can only be done on local networks or through tunnels). Donnybrook also introduces the use of *forwarding groups* (machines with extra bandwidth that can route updates) for lowering peer bandwidth. The most promising architecture is the Distributed Referee Anti-Cheat Scheme (DRACS), which introduces the use of roles (Webb, Steven Daniel.). Peers are given roles of *player* and *referee*, where referees work to ensure that players aren't cheating. Updates are sent via Peer-Peer (PP) mode in normal network conditions, but can be elevated to Peer-Referee-Peer (PRP) mode if issues, like cheating, arise. The use of players and referees addresses the main problem with many P2P architectures: a lack of security. For some games, security isn't a concern, but for competitive games, there needs to be ways to defend against cheaters. However, while DRACS looks promising, it has only been tested on slow-paced games or small fast-paced multiplayer games.

None of these papers experimented with large, fast-paced multiplayer games. To that end, this project will implement DRACS, along with any techniques necessary to make it suitable for large, fast-paced games. Additional techniques will be explored from the aforementioned, as well as techniques used in other types of applications. For instance, there are P2P architectures for streaming video, which have valuable methods of compressing and sending data (Baccichet, Pierpaolo, et al.).

Research Methods

To experiment with a P2P architecture, I need an implementation of one. Rather than try to modify an existing implementation, I plan to write my own. Writing my own allows me to interface closely with the underlying game engine. I have chose to write my architecture as a plugin for the young

Bevy game engine in the Rust programming language. Bevy is a modular game engine with an Entity-Component-System (ECS) architecture. This modularity makes it easier for me to write my plugin without being too tied into the internals of the game engine. I chose Bevy over larger game engines like Unity and Unreal because it shows real promise and I enjoy writing in Rust. The network architecture of the plugin will be based on DRACS. The actual data update method will be *state synchronization* (input and state will be sent by peers). I also plan to support *remote procedure calls* (RPCs), including reliable and unreliable calls, however that may be after this project.

While writing this plugin, I'll also have to write a library for an implementation of the Interactive Connectivity Establishment (ICE) protocol. ICE is used to create P2P connections across the internet. It's necessary because most computers are behind *network address translation* (NAT), which hides their real address from incoming connections. Rust has a few libraries for ICE, but they aren't very documented and many are meant to work with certain libraries. I am also interested in making my implementation for Trickle ICE, a variation that is faster than the original "Vanilla" ICE protocol. Besides ICE, I'll also have to write a matchmaking library that can start the game session.

Once I have an implementation of a P2P architecture, I'll be able to use it to create a very simple multiplayer game. To accurately simulate a fast-paced game, I'll create a simple FPS, which would require sending the positions, rotations, and inputs of the players, as well as the positions and rotations of any game objects. The tick rate of the demo will be at least 30hz.

To test whether my implementation is viable for real-world use, I'll test the demo with 25, 50, and 100 players (representing large deathmatch games at 25, large team games and small battle royales at 50, and extra-large team games and large battle royales at 100) and record the average latency and bandwidth for each peer. The goal is to have latency under 100ms and ideally under 50ms and have bandwidth under 2mb and ideally under 500kb. I won't be testing on real-world hardware, but a network simulation in VMware Workstation 17 Pro via GNS3, which can replicate real-world network topologies, with features like routers and NATs. I'll be running the demo in a headless mode (without graphics) to

keep the simulation performant and to make sure that my hardware is not a constraint. Once I confirmed the viability on a simulated network, I'll use AWS "t3a.medium" instances (2 CPUs, 4 GiB RAM, 5 Gigabit internet) to test the three test cases again, but on a real network.

This is a very large and complex project, and I expect it to take longer than expected. I believe I can get the ICE crate complete within 2-3 months, but the rest could take up to 8 months. I also have other plugins I maintain for the Bevy community, which I update when Bevy updates (4 times a year), which can two weeks to complete as I add new features. So, to be safe, I believe I can complete this project within 2 years.

Expected Outcome

This project will result in a Rust crate (a library), published on Crates.io (the Rust community's crate registry), that contains the Bevy plugin. The crate will be dual-licensed under MIT/Apache-2.0 (the Rust community's favorite licensing, due to MIT being GPL compliant and Apache-2.0 being easier to use; dual-licensing allows for users to choose the license they wish to use). There will also be another crate for matchmaking, and likely another crate for storing common types used by both of the aforementioned. There will also be the ICE protocol crate, designed to be as generic as possible to work with multiple asynchronous runtimes (there are some limitations due to Rust not having all the capabilities needed to make a truly generic asynchronous library, notably associated traits), which would mean that the library could be used in many other projects as well.

I will post on many social media platforms about the release of the crate, to share it with the rest of the Rust community, especially the Bevy community. I will also write a blog post on my website (https://jonahplusplus.dev), documenting the experience of writing the crate, as well as the technicalities behind it. If it is well-received, it will open the doors for other developers to write their own P2P multiplayer games, which will make the genre more accessible to small, indie developers. And if my

research proves that P2P architectures are viable for large, fast-paced games, I'll be able to make my own game using this crate.

Literature Review

Baccichet, Pierpaolo, et al. "Low-Delay Peer-to-Peer Streaming Using Scalable Video Coding." Packet Video 2007, 2007, pp. 173–81. IEEE Xplore, https://doi.org/10.1109/PACKET.2007.4397039. The paper proposes a Scalable Video Coding (SVC) standard that can use network more efficiently than H.264/AVC on a P2P network. SVC allows the encoding of a video at different qualities within a layered bit stream, which allows for quickly adapting to network conditions. SVC was tested through an implementation of the Standford Peer-to-Peer Multicast (SPPM) protocol, which was designed for low-latency video content delivery.

Bharambe, Ashwin, et al. "Donnybrook: Enabling Large-Scale, High-Speed, Peer-to-Peer Games." ACM SIGCOMM Computer Communication Review, vol. 38, no. 4, Aug. 2008, pp. 389–400. October 2008, https://doi.org/10.1145/1402946.1403002. The paper proposes Donnybrook, a P2P architecture for large-scale, high-speed games. It works by reducing bandwidth for lower priority game objects and scales by creating multicast trees for updates to propagate. The authors conclude that the techniques described would not only be useful for P2P, but also C/S applications.

Peer-to-Peer Support for Massively Multiplayer Games | IEEE Conference Publication | IEEE Xplore. https://ieeexplore.ieee.org/abstract/document/1354485. Accessed 15 Nov. 2022. This paper presents an architecture designed for massively multiplayer games (MMGs). The primary focus of the architecture is locality: updating game objects only when needed by organizing players into groups based on areas of interest. The authors conclude that P2P is viable for massively multiplayer games, and that the shared state distribution and replication mechanism can be applied to other P2P computing applications.

Rooney, S., et al. “A Federated Peer-to-Peer Network Game Architecture.” IEEE Communications Magazine, vol. 42, no. 5, May 2004, pp. 114–22. IEEE Xplore, https://doi.org/10.1109/MCOM.2004.1299353. The paper proposes a federated peer-to-peer architecture, where an application is broken into smaller islands for different areas of interest. Each area is supported by a P2P network to which players can subscribe to by entering the area. Stability is ensured using multicast reflectors, which maintain lists of current subscribers and disseminate packets among them, and control servers, which manage players and can adapt the P2P network to changes in network conditions.

Triebel, Tonio, et al. “Peer-to-Peer Infrastructures for Games.” Proceedings of the 18th International Workshop on Network and Operating Systems Support for Digital Audio and Video, Association for Computing Machinery, 2008, pp. 123–24. ACM Digital Library, https://doi.org/10.1145/1496046.1496079. This paper presents Planet ΠΙ4, a Massively Multiplayer Online Game (MMOG), that demonstrates scalability for MMOGs. It uses a modular architecture that allows for the game to be adapted to different systems. At the time of writing, the authors adapted the game to IP-Broadcast, Skype and SpoVNet, to demonstrate the flexibility of their game architecture.

Webb, Steven Daniel. Referee-Based Architectures for Massively Multiplayer Online Games. Curtin University, 2010. espace.curtin.edu.au, https://espace.curtin.edu.au/handle/20.500.11937/498. The paper proposes a Referee Anti-Cheat Scheme (RACS), a peer-to-peer architecture that operates by having referees, which manage games and players, that play the game. Players can communicate directly in Peer-Peer (PP) mode or indirectly in Peer-Referee-Peer (PRP) mode, should issues, like suspected cheating, arise. The paper also provides extensions to RACS, including Mirrored RACS (MRACS), which uses mirrors to provide extra scalability, and Distributed RACS (DRACS), which allows for complete P2P by

having multiple referees per game, whereas RACS and MRACS only have one, which effectively makes them hybrid architectures.

Preliminary Work and Experience

I am an experience Rust developer and have been programming for around 8 years (since 7th grade). I am the creator and maintainer of "bevy_atmosphere" (>4.3k downloads, initially released October 2021), a procedural sky plugin for Bevy and "bevy_spectator" (>50 downloads, initially released November 2022), a spectator camera plugin for Bevy. I've been researching network programming for a few years now, particularly for video game networking.

I have already begun progress on this project. The barebones of the crate has been prepared, including code for defining custom protocols, marking data to be replicated, a command queue for managing sessions, and a system for encoding/decoding messages sent between peers (for now it is a bit packer, but I may investigate adding statistical compression). I have also started work on the ICE protocol crate. It currently provides an interface for defining bindings for asynchronous runtimes (unlike most languages, Rust doesn't come with an async runtime, but allows users to define their own suited to their needs).

IRB/IACUC statement

This project does not need IRB or IACUC approval.

Budget

The project will need funding for the following:

"t3a.medium" hourly on-demand rate: $0.0376

025 "t3a.medium" instances for 10 hours	$009.40
050 "t3a.medium" instances for 10 hours	$018.80
100 "t3a.medium" instances for 10 hours	$037.60
VMware Workstation 17 Pro	$199.00
Total	$264.80

Appendix A: C/S vs P2P

Network architectures fall into three categories: Client-Server (C/S), Peer-to-Peer (P2P) and hybrid (mixed between C/S and P2P). There are a lot of misconceptions surrounding these terms, so let's take the time to define them. With C/S, there is one dedicated server and many clients: it's asymmetrical. With P2P, there are many peers. One might be tempted to call P2P symmetrical, but this is only true for certain types of P2P architectures; it's possible to have an asymmetrical P2P architecture where peers fall into roles. The misconception of P2P being symmetrical is rooted in the association of a specific P2P architecture as synonymous to the term P2P itself (this specific architecture is called "lockstep"). Rather than define an architecture by the capabilities of the parties, I opt to define an architecture by the trustworthiness of the parties. With C/S, there is communication between one trusted server and many untrustworthy clients. With P2P, there is communication between many untrustworthy peers. A hybrid architecture involves both: there is communication between a trusted server and many untrustworthy clients, but the clients can also communicate between themselves.

C/S is the most common type of architecture; most of the internet runs on C/S today and C/S can be thought of as the "traditional" architecture. But for good reason: C/S is a relatively simple type of architecture to implement and get correct. It's easy to manage connections for C/S: for *n* clients, only *n* connections are needed for C/S. In a P2P architecture with *n* peers, there will be *at least n* connections, up to $\sum_{i=1}^{n}(i-1)$ (all peers interconnected). It's also easy to manage security on C/S: you can trust the code executed on your server to not be tampered with, while with P2P, you can't trust peers to not have been modified by malicious users. C/S is also more bandwidth efficient for clients: in P2P, peers may have to send data to multiple other peers, while in C/S, the client only must communicate to the server.

So, what is P2P good for? There is a common misconception that C/S is more scalable than P2P, but it's the opposite: P2P architectures are capable of better scalability than C/S. The reason for this

misconception goes back to associating P2P with lockstep. However, other P2P architectures don't suffer from the same limitations of lockstep. The reason P2P architectures can scale better than C/S is that the bandwidth of P2P networks grows with the number of peers, while the bandwidth of C/S networks only grows with the number of servers. That means that as more clients join a C/S network, more servers must be provisioned to handle them; P2P doesn't have this problem. This means that P2P networks are cheaper to maintain, since there are no server costs, though there is more of an initial investment to develop, due to P2P's complexity.