Towards Cheap Scalable Browser Multiplayer

Yousef Amar, Gareth Tyson, Gianni Antichi
Queen Mary University of London
{y.amar, gareth.tyson, g.antichi}@qmul.ac.uk

Lucio Marcenaro
University of Genova
lucio.marcenaro@unige.it

***Abstract*—The barrier to entry for the development of independent, browser-based multiplayer games is high for two reasons: complexity and cost. In this work, we introduce and evaluate a method and library that aims to make this barrier as small as possible, by utilising appropriate development abstractions and peer-to-peer communication between players. Our preliminary evaluation shows that we can lower both the technical development overhead, as well as minimise server costs, at no loss to performance.**

***Index Terms*—peer-to-peer, p2p, browser, multiplayer, game, networking, development**

I. Introduction

The web browser as a target platform for modern games has been unpopular due to its limitations, for example lack of multithreading support. In recent times, developers have been deploying less and less to the browser. [1], [2] However, as modern HTML5 APIs such as WebGL and WebAssembly are beginning to see widespread support in browsers and devices, there is an opportunity for independent game developers to rediscover the browser as a serious target for deployment. As independent developers are more limited when it comes to labour and resources, many shy away from large-scale, real-time multiplayer game development.

The most common approach to supporting multiplayer is a client-server model, where one or more authoritative servers maintain communication between clients. This holds true for both browser and non-browser games. This is usually (at least at first) the least complex solution and fits well within existing internet architecture and paradigms.

The potential of P2P architectures for game networking has always been recognised however. These have the capacity to lower costs for game developers and provide a more reliable service to the players.

In this work, we introduce a library for easily and scalably networking players in the browser that was designed with a games-first approach. Our primary contributions are a method for scaling efficiently to many players and a library that allows this to be done in modern web browsers. Solving the challenges to make a system like ours scalable has historically not been worthwhile when compared to simply shifting the development cost to deploying additional servers, and the browser has only recently become a platform where it is possible to implement this for with some difficulty. We focus primarily on Multiplayer Online Battle Arena (MOBA)-style games (where players communicate with a limited number of other players at a time) although we aim to support Massively Multiplayer Online (MMO)-style games (where players could need to communicate with an unlimited number of players at at time) in the future.

Further, we present preliminary evaluations of this library and highlight reductions in complexity and server utilisation/costs at no cost to game performance.

978-1-7281-1884-0/19/$31.00 ©2019 IEEE

II. Related Work

Systems such as Kiwano [3], VoroGame [4], and [5] seek to solve issues of scalability for MMOs. These systems utilise many of the methods we explore, however they are either cloud-based or a hybrid of cloud and P2P. We seek to remove dependency on the cloud as much as possible in order to hosting costs.

Solipsis [6], [7] was one of the first approaches at pure P2P architectures for virtual environments. This was taken further through VON [8] and subsequent work [9]. These approaches use virtual avatar position data to compute the peer network topology based on area of interests. Most methods use Gnutella-like initialisation through distributed hash tables (DHTs) or similar. Newer methods account for variable spatial concentration of players by using Voronoi-based or derivative partitioning schemes. There have been several extensive survey papers covering these methods [10], [11].

While our proposed method builds on a lot of this past work, we take it further by allowing arbitrary metrics for computing ideal topologies. In addition to this, we guarantee that our peer networks are fully connected and resilient to a certain extent of link failures. Further, we consider the challenges of implementation within the browser; past methods have been platform-agnostic.

III. Design

This section details the design of our library based on our requirements, as well as detailing and justifying our design decisions. It is split into and overview and subsections for our two main solutions for these two goals respectively, and a third to discuss server-side design.

A. Overview

At the highest level, our goals are to minimise costs that manifest themselves in server and codebase development and maintenance. As we minimise maintenance by encapsulating general and game-specific functionality in our library, the main requirement in the context of this paper is that our

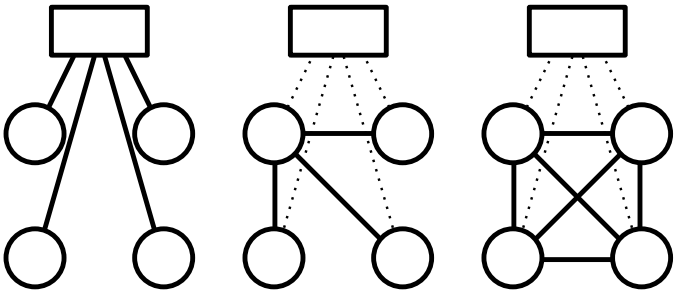

Fig. 1. Three networking topologies of interest between servers (rectangles) and peers/clients (circles) – client-server model (left), hosted P2P (middle), and full P2P (right)

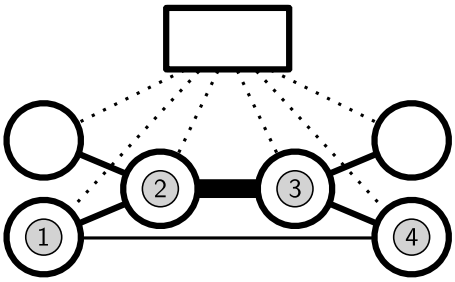

Fig. 2. An example of a computed MST topology where peers with better connections (② and ③) act as supernodes, and with redundancy (① and ④)

system significantly reduces server costs and these costs scale minimally as we increase the number of players.

The majority of traffic that passes through servers in most games is simply broadcasting information from one client to one or more other clients. We remove the middleman by creating a peer network that allows us to broadcast directly between peers.

What makes our system different from a standard P2P mesh, is that we also connect the peers in a peer network in such a way that peers do not have to maintain too many connections, while at the same time not having to rely on single peers for the fidelity of the communicated data. Currently, this peer network topology is coordinated by the signalling server based on the quality of a connecting between pair-wise peers. We discuss this in more detail in this section.

B. Reducing server costs

The obvious problem with a completely connected P2P approach is that it does not scale for N players as well as a client-server model does, as clients can be resource/bandwidth-weak in comparison to servers that have the resources to maintain N connections. We mitigate this by allowing for alternative peer network topologies (which define how the peers are connected), such as the middle one in figure 1 where a peer is designated "host" and acts as a de facto authoritative server, as opposed to a completely connected topology such as the right one in figure 1. This limits the server costs of the game provider to simply acting as a lobby/directory for finding these rooms/groups, but at the same time, the number of players that a host can support is more limited than a standalone server.

A server is still required for signalling purposes to exchange the information required to establish a connection between peers. This low-traffic server connection only needs to be maintained if the peers need to be notified when a new peer joins their network, which translates to joining their game room/area/instance/arena/match/etc. Otherwise it can be severed after a peer network is set up.

C. Reducing code complexity

Our second goal is maximising ease of development and developer barrier to entry. To do this, we encapsulate all networking functionality behind (*i*) event emitters and (*ii*) shared objects that sync between peers automatically. This is important because we want to avoid code duplication and maintaining strongly related code in more than one place. This is common in client-server applications where changes to client communication requires changes to server communication in parallel, and vice versa. With our system, developers need only maintain peer code.

While figure 1 is simplistic, it alludes to a spectrum between a topology with the least number of connections possible — e.g. with a minimum spanning tree (MST) over all peers — and a completely connected topology where each peer is connected to every other peer.

As our peer networks should be able to have arbitrary topologies, it is important that we do not introduce heterogeneity between peers in code as this can quickly become unmanageable. Ultimately, the same peers should be able to switch between an MST topology and a completely connected topology without touching the game logic whatsoever.

D. Signalling server

Our P2P approach introduces other non-negligible challenges. In a client-server context, there is a stark trade-off between how much game logic the server carries or attests (which increases server costs) and players' ability to cheat by modifying the client. An example of this are games where all physics simulation is done client-side to minimise lag and a player can modify a client to manipulate their position and clip through solids.

In a peer-to-peer context, this problem persists between peers, however the cost of validating game states falls to the clients. As no authoritative servers exist, peers have to decide either manually or autonomously to disconnect and/or blacklist cheating peers by maintaining an array of IPs of past offenders seen first-hand.

On the spectrum between MST and completely-connected, we cannot therefore simply go for MST as it is more efficient. We must introduce redundant connections for two reasons: (*i*) resilience and (*ii*) accountability. Figure 2 shows a visual example of this where peer ④ relies on just peer ③ for updates, as peer ③ has a good connection to the left side of the network. We add redundancy by connecting ④ to ① also.

For the first, it is imaginable that a node goes offline for whatever reason, and a new MST must be computed. To avoid the overhead and potential lag in repairing the peer network, redundant connections are an advantage. For the second, if a peer relies on only one other peer to update their global game state, that peer can spoof the game state. While every peer can perform their own validation for impossible game states, or states that imply cheating, there are edge cases where this becomes less obvious when a peer receives realistic but divergent states from two or more other peers.

It is therefore important for the topology to be set up in such a way that each peer can choose to validate data by comparing it between more than one source. A peer that is caught cheating (by e.g. spoofing their in-game position) can then be blacklisted and the network can sever the connection to them. We plan to extend our library to allow this by comparing data across source peers, and flagging peers that diverge from the majority. Additional validation (such as checking if a player's position is changing too fast) is left to the developer and we provide the API for a peer to blacklist them in that case.

Finally, we can take advantage of other information for setting up a peer network topology (the weights for our MST) to reflect that peers with more resources can take on more connections. We do this by periodically checking the latencies between each pair of peers. From this information, we can optimise peer network topology so that a weak peer does not inadvertently become a supernode. To minimise code complexity, we encapsulate this behaviour in the signalling server.

IV. Implementation

Our peer library and signalling server implementations are open source under an MIT license and available on GitHub under *yousefamar/p2p-peer* and *yousefamar/p2p-sig-serv* respectively, and the peer library can be installed through NPM as *p2p-peer* and used as a Node.js module with common browser bundlers such as Browserify and Webpack, or as an ES6 module. Our peer library is 231 SLOC / 7.61 KB of uncompressed, unminified code, and our signalling server is a mere 64 SLOC / 1.53 KB, both pure JavaScript.

P2P communication has begun to see widespread support in modern browsers through WebRTC. While WebRTC was mainly intended to enable audiovisual communication between browsers (such as for video chat applications), it can also be used to communicate arbitrary data. While modern browsers have had this capability for a while, browser-based games have not yet seen widespread adoption of WebRTC for P2P communication between players.

Even outside of games, most applications of WebRTC are limited to chat applications and specialised use cases such as for exchanging metadata in the WebTorrent protocol. WebRTC is not very developer-friendly, and while libraries that simplify P2P communication exist, these are bare-metal and not aimed at game development.

We use WebRTC to enable P2P communication between browsers. Our library exposes a `PeerNetwork` object which handles peer connections and emits events based on different events that happen in the network (such as peers connecting or disconnecting). We allow arbitrary namespacing through *rooms* of which a peer can join multiple.

It also exposes a set of methods that can be called directly if required. These are:

- `signal(event, ...args)` – Sends an event and data to the signalling server
- `join(roomID)` – Joins a room with a particular ID
- `leave(roomID)` – Leaves a room with a particular ID
- `broadcast(event, ...args)` – Sends an event and data to all connected peers
- `async connect(sigServURL)` – Connects to a signalling server

`PeerNetwork` also has three properties; `ownUID` which is the peer's own UID in the network, an array called `peers` which contains instances of `Peer` for all connected peers, and an array called `rooms` which will be explained shortly.

One can `.disconnect()` from or `.send(event, data)` directly to each `Peer`, and on the other side, peers will emit events based on what is sent that can be listened to. These events do not need to be defined anywhere beforehand.

As an added layer of abstraction over network events, each `room` contains an `eventEmitter syncedData` object which implements an Observer design pattern such that any changes made to this object (or any nested objects at any depth) emit an event distinct to a property's path in this object. The value, along with the path of the added/modified property's path (e.g. `.foo.bar`) is propagated through the peer network and all peers in a room can expect to see the property at that path updated. For added efficiency, we only broadcast data that has been modified. We also keep track of which data has been set by which peer for data ownership and to prevent broadcast storms in the peer network.

Namespacing within the object and how rooms are organised is left to the developer to define. As mentioned before, rooms can map directly to an arena instance in a MOBA context for example. Another possibility is rooms mapping to a 2D or 3D spatial grid where players join and leave rooms seamlessly as they navigate through the game environment, prioritising connections to the players who are closest to them in the game.

Similarly, rooms can be "nested" into a tree structure for optimisation purposes, such as for limiting update frequency for peers that are further away from each other in-game (e.g. position updates) creating network Levels of Detail (LODs). This is especially powerful in an MMO context.

While the organisation of rooms is left to developers, and the current abstractions are not just game-centric, we plan to provide example code for the most common use cases such as these. We also plan to survey the most common kinds of data transmitted in networked browser games (e.g. position updates vs events like jumping) and build functionality for doing this seamlessly.

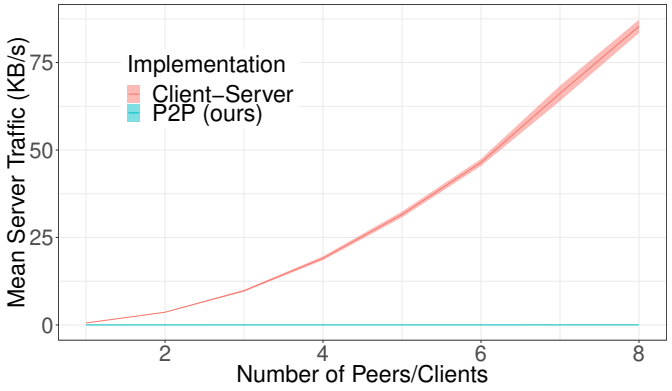

Fig. 3. Mean server network traffic (0.95 confidence interval) against number of clients/peers for a traditional client-server model versus our optimised P2P version

V. Evaluation

This section describes a simple evaluation of our method and library. Our aim is to demonstrate how we achieve significant reductions in server traffic, thereby lowering required bandwidth and sever costs.

A. Setup and Method

We deploy up to eight client/peers and two servers. The first server acts as a signalling server for the P2P implementation, and the second exposes a WebSocket endpoint that allows clients to broadcast messages to other clients in the same room through the server.

The clients/peers then make use of our library to set a random floating point number on a set room's `syncedData` object every 100 milliseconds. Our library propagates these values through the network.

We use `nethogs` to then measure the traffic in KB/s for the server processes. We collect traffic data for several minutes before incrementing the number of clients/peers and repeating the process. The measurements for the P2P implementation and the client-server implementation are done separately.

B. Results

Figure 3 shows how the mean server network I/O changes as we increase the number of clients/peers. While the client-server implementation scales exponentially as we add more clients, the server traffic for our P2P implementation increases linearly and very slowly.

This is because the P2P signalling server only sends periodic heartbeats to each peer to check if a connection is still alive. The data does not need to be sent through the client and we shift the burden of reaching other peers to the peer. For few peers (<30), modern devices can cope well with a completely connected network. Beyond that, our optimised topologies make a significant difference, which we aim to demonstrate in future work.

VI. Conclusion and Future Work

We have identified server costs and development overhead as two deficiencies in multiplayer browser game development that create a barrier to entry for independent game developers. We have addressed these deficiencies by introducing a library that takes advantage of modern HTML5 APIs to simplify P2P communication for games and make it scalable. Finally, we have empirically demonstrated the advantages that our library provides by evaluating it against the standard approach.

While our library can be sufficient for low-throughput networked browser games, there are some cases where games might need unreliable, but higher-throughput, streaming communication (cf. UDP). For this, our event-based system may be unsuitable. A natural extension of our library would be more bare-metal APIs for that type of data.

Currently, the signalling server orchestrates the topology of the peer network, introducing some centralisation. We can decentralise our system even further by instead employing a more peer-centric method, where e.g. peers dynamically reconnect to better peers over time, thus eventually converging on optimal networks. This minimises points of failure.

Finally, we aim to extend the scalability of our system such that it would be suitable for use in an MMO where a single peer may need to communicate with a very large number of peers can cannot be expected to broadcast data to all of them. To do this requires further investigation and evaluation of peer network topologies, however can yield significant cost benefits.

References

[1] Game Developers Conference (GDC), "State of the game industry 2017," Tech. Rep., 2017.

[2] ——, "State of the game industry 2018," Tech. Rep., 2018.

[3] R. Diaconu and J. Keller, "Kiwano: A scalable distributed infrastructure for virtual worlds," in *2013 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE, 2013, pp. 664–667.

[4] E. Buyukkaya, M. Abdallah, and R. Cavagna, "Vorogame: a hybrid p2p architecture for massively multiplayer games," in *2009 6th IEEE Consumer Communications and Networking Conference*. Ieee, 2009, pp. 1–5.

[5] S. Kulkarni, "Badumna network suite: A decentralized network engine for massively multiplayer online applications," in *2009 IEEE Ninth International Conference on Peer-to-Peer Computing*. IEEE, 2009, pp. 178–183.

[6] J. Keller and G. Simon, "Toward a peer-to-peer shared virtual reality," in *Proceedings 22nd International Conference on Distributed Computing Systems Workshops*. IEEE, 2002, pp. 695–700.

[7] D. Frey, J. Royan, R. Piegay, A.-M. Kermarrec, E. Anceaume, and F. Le Fessant, "Solipsis: A decentralized architecture for virtual environments," in *1st International Workshop on Massively Multiuser Virtual Environments*, 2008.

[8] S.-Y. Hu, J.-F. Chen, and T.-H. Chen, "Von: a scalable peer-to-peer network for virtual environments," *IEEE Network*, vol. 20, no. 4, pp. 22–31, 2006.

[9] S.-Y. Hu, C. Wu, E. Buyukkaya, C.-H. Chien, T.-H. Lin, M. Abdallah, J.-R. Jiang, and K.-T. Chen, "A spatial publish subscribe overlay for massively multiuser virtual environments," in *2010 International Conference on Electronics and Information Engineering*, vol. 2. IEEE, 2010, pp. V2–314.

[10] A. Yahyavi and B. Kemme, "Peer-to-peer architectures for massively multiplayer online games: A survey," *ACM Computing Surveys (CSUR)*, vol. 46, no. 1, p. 9, 2013.

[11] J. S. Gilmore and H. A. Engelbrecht, "A survey of state persistency in peer-to-peer massively multiplayer online games," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 5, pp. 818–834, 2011.