A CRYPTOGRAPHICALLY SECURED REAL-TIME PEER-TO-PEER MULTIPLAYER FRAMEWORK FOR BROWSER WEBRTC

Haochen Han[1] and Yu Sun[2]

[1]Troy High School, 2200 Dorothy Ln, Fullerton, CA 92831
[2]California State Polytechnic University,
Pomona, CA, 91768, Irvine, CA 92620

ABSTRACT

P2P(peer-to-peer) multiplayer protocols, such as lockstep and rollback net-code, have historically been the cheaper, direct alternative to the Client-Server model. Recent advances in WebRTC technology raise interesting prospects for independent developers to build serverless, P2P multiplayer games on the browser. P2P has several advantages over the Client-Server model in multiplayer games, such as reduced latency, significantly cheaper servers that only handle handshakes, etc. However, as the browser environment does not allow for third-party anti-cheat software, having a secure protocol that catches potential cheaters is crucial. Furthermore, traditional P2P protocols, such as deterministic lockstep, are unusable in the browser environment because different players could be running the game on different browser engines. This paper introduces a framework called Peercraft for P2P WebRTC games with both security and synchronization. We propose two P2P cheat-proofing protocols, Random Authority Shuffle and Speculation-Based State Verification. Both are built on known secure cryptographic primitives. We also propose a time-based synchronization protocol that does not require determinism, Resynchronizing-at-Root, which tolerates desynchronizations due to browser instability while fixing the entire desynchronization chain with only one re-simulation call, greatly improving the browser game's performance.

KEYWORDS

Cyber Security, Anti-Cheat, Peer-to-Peer multiplayer, WebRTC.

MODULE 1, INTRODUCTION

Our framework is built on top of WebRTC, a browser peer-to-peer [1] technology that allows for both video/audio calls and the sending of arbitrary data[2], through UDP or TCP. Because each peer is hiding behind a NAT, a handshake server is required to perform hole punching[3]. This framework uses Peer.js, a high level wrapper for WebRTC API [4], so the hole-punching is handled automatically [5]. Users can also choose to implement their own Peer.js hole punching server, but this is out of the scope of this paper. Once a direct connection is established, the handshake server is no longer needed.

I, Contributions

The main focus of this paper will be about preventing cheating and maintaining synchronization while optimizing for a browser environment. Security protocol is much more important in browsers, as browser games rely on third-party anti-cheats [6]. It is also significantly easier to manipulate browser games state as the codes are open source. For this purpose, we introduced two new protocols, Random Authority Shuffle and Speculation Based State Verification , to let the network catch cheaters. Random Authority Shuffle uses established cryptographic primitives(hash commitment scheme) to perform synchronized switching of the authority of clients. It ensures at a given tick, a player is only authoritative of the data of a random, remote player. This is combined with speculative state verification, where if a cheater attempts to manipulate the data based on its assigned authority over a random, remote player, non-cheating clients will compare their non-authoritative data and deny the manipulated data from the cheater, even if the cheater has authority. We also introduced a novel state synchronization technique that does not require determinism while minimizing the amount of re-simulation required due to desync, called Resynchronizing at Root.

II, Existing Methods

As of now, there are 3 major types of netcode implementation for browser multiplayer game:

1, Client-Server with authority server
2, Peer-to-Peer with one single authority "client host"
3, Peer-to-Peer with deterministic lockstep, input delay, and/or rollback

Peer-to-Peer style multiplayer, either with client host or lockstep, offers advantage in terms of lower cost and latency due to elimination of third party servers [7]. Deterministic lockstep, where clients broadcast its input rather than state, is commonly used in RTS(real-time-strategy) games as the sheer amount units makes sharing and synchronizing state impractical. Rollback, in which clients predict next remote input from previous remote input, is commonly used in fighting games, where the fluidity and low latency of peer-to-peer structure help promote better gameplay experience. Most multiplayer IO games, such as agar.io, uses Client/server method. As the number of connections needed for each client increases with the number of players, peer-to-peer networking tends to be less practical than client/server methods. Therefore, the focus of this paper and framework will be lobby-based multiplayer match games with less than 30 players rather than massively multiplayer IO games. The P2P single-authority client host methods are commonly used in games with private lobbies. Although convenient, it grants the host an unfair advantage as it has authority over all other clients. Both lockstep P2P and rollback P2P distribute authorities to each client. In other words, they are fully decentralized. Strictly deterministic lockstep aims to reduce the amount of data transmitted between clients by sending inputs from each player rather than state, and allow each client to calculate the future game state based on input. If the game is deterministic, then clients should always yield the same state. Network latency is solved through input delay, where clients issue a command with an execution time guarantee to be later than the package receive time on other clients. For example, client A issues a "move" command at tick 1, but specifies execution at tick 3. At tick 2, client B receives the command. At tick 3, both clients execute the move command.

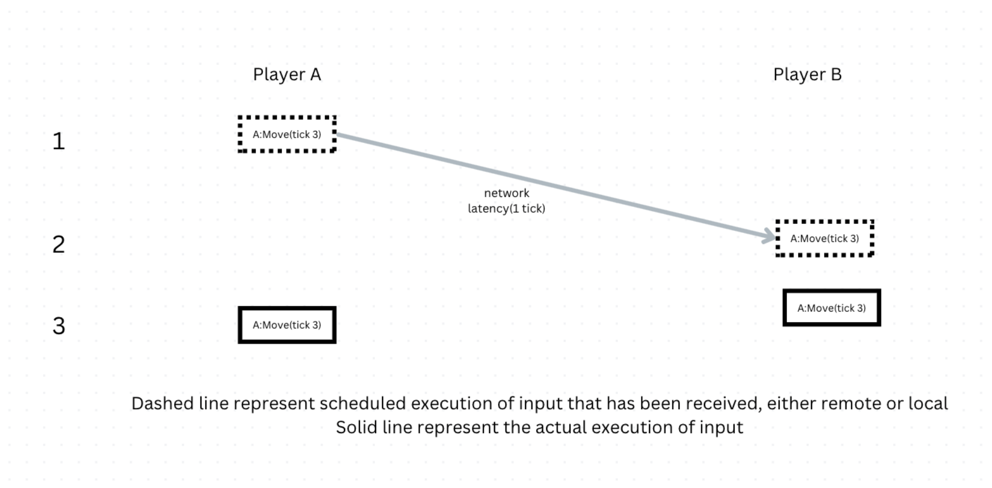
Player A
Player B
1
2
3
A:Move(tick 3)
network latency(1 tick)
A:Move(tick 3)
A:Move(tick 3)
A:Move(tick 3)
Dashed line represent scheduled execution of input that has been received, either remote or local
Solid line represent the actual execution of input

Figure 1. Figure of example

The unresponsiveness of input delay is usually solved with graphical feedback, in which local players' character immediately execute a command but run cause game-state altering at the synchronized execution time.

Another solution is using rollback to reduce the input delay interval. In rollback, clients predict movement of other players, often based on previous input. Inaccuracy in prediction is corrected when a synchronization packet from the other player arrives. There are also newer netcodes which completely eliminate the element of input delay, such as in Skullgirls, where clients are allowed to "steal frames" in order to catch up to inputs. Because human reaction time for visual stimuli is 250 millisecond when untrained, 2-15 frames of input delay is usually acceptable outside of fighting games(Reaction Time Study: https://www.scientificamerican.com/article/bring-science-home-reaction-time/) .

Existing lockstep ensures synchronization through waiting for the slowest client to send end-of-tick packets(often containing state). This is slow and impractical for many fast paced browser IO games,where it is not necessary to wait for a lagging player. Existing industry solution is deterministic lockstep + input delay. Some frameworks implement rollback to reduce the input delay time. The GGPO rollback SDK is currently the industry standard for implementing rollback in Peer to Peer games. This paper proposes a new framework with an alternative solution to rollback specifically targeting WebRTC and the browser environment [11]. There are several problems that we will solve in this paper. As the browser environment is nondeterministic, it is previously unrealistic to eliminate the lockstep wait through determinism. This paper proposes a new protocol called Timing Synchronization, in which it takes advantage of date/time on each device to heuristically ensure lockstep(without having to wait for the laggiest player), with regular state correction packets to eliminate simulation differences on each machines due to non-determinism. Secondly, most existing P2P multiplayer netcode does not provide an effective scheme for anti-cheat. This paper introduces a preliminary input/state verification system to allow players to catch cheaters. There are caveats to this system—- although universally applicable in all games in preventing unwarranted gameplay input or state manipulation on remote clients, it does not allow for private data. As all data are shared between all clients, this framework is more applicable to fast-paced shooter games or any games that do not include a de facto fog of war.

III, Paper structure

In Module 2, we explain the detail of Random Authority Shuffle and the cryptographic primitives behind it. In Module 3, we introduce the synchronization suite for this framework- time based lockstep, input delay, and Resynchronizing at Root. In module 4, we discuss the specific implementation detail in a browser environment, and network-related solutions such using UDP packets while maintaining reliability. In module 6, we reiterate the main ideas presented in this paper. In module 7, we suggest Zero-Knowledge-Proof as a future research direction for allowing cheat-proof private data in generalized use cases beyond TCG games.

One key concept to understand is that in this protocol, the term authority is always relative. To prevent state manipulation, if the client with authority deviates significantly from other players, it will be detected as a cheater. In our proposed Desynchronization Tracing method, a non-authority client can detect the data from an authority client is faulty due to authority client desynchronization, and thus ignore the data. The authority client's resimulation will then bring it to the same true state as the non-authority client.

Module 2, Cryptographical Cheat-Proof

A. Threat Model

In P2P, there are no impartial third parties. There are several potential ways in which a player could cheat.

1, Host Cheating - if a player is granted all authority as the de facto "server" in a pseudo client-server simulation, it could alter state to its will. As our protocol uses peer-to-peer state authority, this is not an issue.
2, Peer to Peer state manipulation - if state authorities are distributed among all peers, then one peer could report manipulated state to other peers
3, Data Peeking - in a game where "fog of war" exists, a player should not be able to see the secret data of the other player.
4, Probability Manipulation

#1 is not a problem as it only applies to pseudo client-server p2p. This paper introduces techniques, namely, Random Authority Shuffle and Speculation Based State Verification to address #2. This paper also uses a cryptographic commitment scheme to generate unbiased, synchronized random numbers.

I)Prevent State Manipulation

In a distributed environment, where each player has authority over their own data, it is very easy for a player to falsely report his/her own data to give himself/herself an unfair advantage.

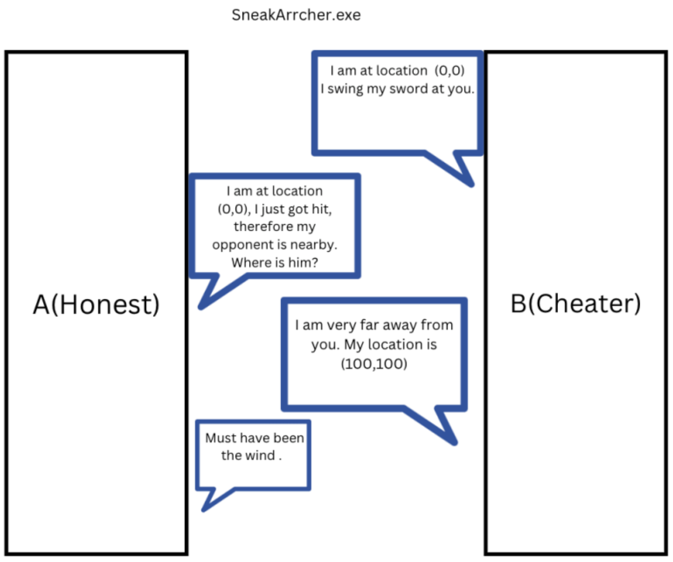
SneakArrcher.exe
A(Honest)
B(Cheater)
I am at location (0,0) I swing my sword at you.
I am at location (0,0), I just got hit, therefore my opponent is nearby. Where is him?
I am very far away from you. My location is (100,100)
Must have been the wind .

Figure 2. Sneak Arrcher

To catch state manipulation, the common approach is to implement a verification function for each state field and input to catch improbable reporting. For example, the game programmer could hardcode state rules like “positional change between tick cannot be greater than speed of a player, unless the previous 10 input stack has blink skill used”. When these rules are violated, it is likely that the opponent is a cheater. For input, rules can be like “player cannot send attack input twice in one single tick”. These rules, although successful in catching clear violations, often fall short in covering all potential cases. This is because cheaters can perform subtle manipulation on the state based on specific situations to get around rules. For example, when a cheater is about to get hit, being 0.1 meter away would result in avoiding the damage completely, thus giving the cheater a significant advantage while not being detected. To mitigate this, we propose two techniques.

1, Randomized Authority Shuffle
2, Speculation Based State Verification

Random Authority Shuffle

1)Logic

If a player has absolute authority over its data, the only way an honest player can catch a cheater is through careful inspection. To add in an extra layer of security, we propose Random Authority Shuffle protocol, which allows for three constraints

1, players can only be in “authority” of the state of a random remote player
2, before each tick, player cannot predict who’s state it will be in charge of
3, the entire network, however, will reach a consensus on authority responsibility at that time.

Consider a hypothetical cheater. To give itself an advantage, the cheating client will report that its health is full 100/100, despite being hit. If the cheater has authority over its own data in every tick, and reports the manipulated data, then other remote clients will falsely assume that they have desynchronized and thus change their state of B to 100/100. B will thus never take damage. However, in the Random Authority Shuffle, B will never be in authority of its own state. This limits the possibility of cheating into sabotaging the opponent's data. But since on every tick, the cheater is reassigned a random player's data to be in authority of, it is very hard for it to target down its opponent. Of course, the cheater could report manipulated state like health = 0 every tick for player A until the cheater is in charge of the state of player A. However, consider the previous example where the cheating player moves 0.1 meter to avoid getting hit. It is highly specific to the data of the tick. If the cheating player wants to perform this cheat by moving A away, it has to be lucky enough to be in charge of A's state, at the exact tick player A hits the cheater. If player A attacks once per second(15 ticks), In a lobby with n players, the probability of catching the attack is 1/(n-1)*1/15. This makes subtle, carefully tuned specific state manipulation unviable in a lobby with a large number of players. The cheater, forced to resort to brute force method such as reporting " player X(the player cheater happen to be in charge of) has health 0" on every tick, can be easily catched by the conventional method of Rule Based input verification, and our method of Speculation Based State Verification.

2)Cryptographic Commitment Implementation

For Random Authority Shuffle to work, the entire network needs to agree on an unbiased random value at a given tick. We will use a PRNG(pseudo random number generator) algorithm, because it is deterministic. For every 10 ticks, the decentralized network will generate an agreed-upon unbiased random seed, and use it for 10 ticks. We used a cryptographic commitment scheme to ensure the true randomness of the scheme [8]:

1.all players determine a number - this number can be generated through a true random number generation algorithm, or can be manipulated by a cheater. Once the number is determined, each player hashes the number and sends the hash string to every other player. This hash string will be "commitment"

a.SHA-256 is preferred as it avoids collision

2.Once all hashes have been received on all clients, players send the actual number to each other. Players can then hash these numbers to confirm that other players did not deviate from commitment.

3.All players then add up the numbers together. The sum will be used as the random seed for the next 10 ticks.

This commitment prevents a situation where a cheater intentionally delays sending out its number to receive all numbers from other clients first. Without commitment, the cheater can simply sum up other clients' numbers and send out a reverse engineer number that will result in a random seed that can give a favorable result for the cheater.

Once the random seed is synchronized, each player would then independently use that seed to determine the authority arrangement of the network. Since the random algorithm is deterministic, this is guaranteed to result in the same authority arrangement. If a cheater intends to cheat by claiming that it has authority over player A's state, player A and player B will simply ignore the cheater's authority state patch for player A.

1.Assume there are 3 players, A, B, C

2.each player calculate three(number of players in the network) random value, assign them to A, B, C, respectively in the same order

a.because the random algorithm is deterministic, and the seeds are the same, the three random value for A,B,C will be the same across all peers

3.Each player then compares the value for A, B, C and arranges them in Descending order. Let array P be the ordered arrangement of player, arrangement is determined as following
a.P[i] has authority over P[i+1]
b.if P[i] is the last player in the array(i+1 > OP.length), P[i] has authority over P[0]

As there are still chances for the cheating player to be lucky enough to manipulate the state of a specific player while also getting pass rule-based state verification, we use another technique, called Speculation Based State Verification.

Speculation Based State Verification

Since each player runs a near-deterministic simulation of all other players in our framework, it is possible to use the simulation of the remote player's state, also known as speculation, to detect cheating. If multiple players disagree with the potential cheater, then they reject the state patch from that potential cheater even if it has authority. See example

1.The cheater has authority over the state of client A after random assignment.
2.The cheater sends out a state patch to all clients, claiming that client A's health equals to 0.
3.Client A receives the authority state patch from the cheater for client A. It detects discrepancy.

a.client A's speculation for its own state is {health_of_A = 100}
b.the state patch from cheater is {health_of_A = 0}
c.client A request a Convention of Speculators to all clients
d.all clients will send to all other client their own speculation of client A's data
e.all clients then tally the result
f.each clients will discover that client A, B, C all speculated client A's health to be 100, while the cheater claim client A's s health is 0
g.the cheater is detected

II)Cryptographic Random Number Generation

The same algorithm for Random Authority Shuffle is also used for network random number generation. It is modified to satisfy the following constraints

1.cannot be manipulated
a.Since each client needs to send a commitment hash, they cannot modify their partial seed number after receiving other clients' partial seed number to make the seed sum favorable to them. The existing solution for Random Authority Shuffle already satisfies this constraint.
2.cannot be predicted
a.in Random Authority Shuffle, players reconvene every 10 ticks to repeat the seed generation process. This means players can only predict future states up to ten ticks. However, this is not good enough for turn-based games. Take the example of Monopoly. One player can gain a significant advantage by knowing what the next two dice rolls will be and prepare accordingly. Therefore, modification is needed

To make the game truly unpredictable, where a client cannot cheat by simply look ahead by running a copy of the PRNG model with the shared random seed, the seed should change, ideally every tick [9]. It is impractical to re-execute the same cryptographic commitment seed generation process on every tick. Therefore, we used a different technique basing off player input. Input data, unlike state, is not subject to indeterminism or desynchronization as we designed the UDP packet to guarantee arrival of input through redundancy(more on implementation detail section).

Therefore inputs for a tick in the past is guaranteed to be the same across all players. We utilize this to modify the seed in an unpredictable yet synchronized manner

1.At given tick X, find inputs that are executed at tick X-2 for all clients.
   a.Because we use input delay, inputs for tick X -2 are likely sent at tick X-4. This margin heuristically ensures that the inputs have arrived across all machines. The margin can be increased to compensate for network latency.
2.Convert the input names into ASCII numbers and add them together, resulting in number N.
3.The new seed will be the previous seed (S%N)*(N%S) * S.
   a.modulus is used to cause greater fluctuation while maintaining integer operations
   b.A Sin function could also be used, however, it must be implemented in a way that does not break determinism through floating point as small error in random seed can result in big difference in NRNG generated number.

Because player inputs are most likely unpredictable, no player can guess the new random seed to gain an advantage.

Module 3, Protocol

A.Designing for a Browser Environment

This section introduces the technique we used to achieve near-lossless synchronized multiplayer simulation on Peer to Peer WebRTC browser environment [12]. Browser environment are fundamentally different from PC game environment because are several constraints:

a.No guarantee on continuous simulation:

Browsers can pause a page's event loop and execution when deemed necessary. This means any framework need to account for such pause in order to be stable

b.inherently non-deterministic

Unlike PC games, developers cannot enforce determinism as players could play on different browser engines AND different computers.

Constraint (a) means that tick rate and packet arrival time is not-guaranteed to be consistent across different machines. In a traditional lockstep, players wait for tick packet confirmation to continue simulation. This delay is not feasible as browsers could freeze webpage when deemed necessary, stopping the simulation for the entire lobby. Traditionally, this requirement on waiting for tick updates is lifted by input delayed determinism. But there are two problems with input delay determinism on browsers: 1st, determinism is infeasible. 2nd, the tick rate is not guaranteed to be synchronized. We designed a novel solution that does not require determinism for synchronization nor waiting for player's tick advancement confirmation, while ensuring all clients run on the same tick. This framework can be divided into two components. First, time bound lockstep simulation with input delay. Second, synchronization protocol through state patch packets.

B.Time-Based lockstep

Ensuring that all connected clients run the same tick at any given time is crucial for synchronizing because the protocol performs catch-up/correction based on global tick. To achieve this, a frequent, consistent update loop is needed,preferably equivalent to 60fps. The update loop is done through web-worker multi-threading, with detail explained in the framework implementation module.

We used a time-bound technique to ensure tick synchronization. This is under the assumption that browser javascript function Date.Time() returns an accurate relative representation of how much the time has passed. Note that Date.Time() does not have to be synchronized — it could differ based on different implementations. Our protocol have the following unique advantages

1.Synchronized Tick
2.Does not need to wait for lagging player to catch up- lagging player would automatically catch up without delaying the whole network

Below is the specific detail of the tick network synchronization protocol

1.The "host" client is responsible for distributing an accurate start time. Before the game starts, the host client sends ping packets, calculating the RoundTripValue.
a.Network latency for each connected client is calculated as NetworkLatency[i] = RoundTripValue[i]/2
b.to ensure accuracy, this process is repeated several time to find Median Value
i. MedianNetworkLatency[i] - the median network latency of client i
2. The host sends out a start packet with specified starting delay based on median network latency.
a. host A find the client M with biggest MedianNetworkLatency[M],
i. calculate global delay value as D = MedianNetworkLatency[M]+ padding(30ms)
b. send out start packet with specified starting delay(S)
i. for each client i, S(i) = (MedianNetworkLatency[M] - MedianNetworkLatency[i])+ D
c. when each client i receive the packet, start the game after delaying for S(i)
3. This ensures all clients start the game exactly D millisecond after host A sends out the start packet

After D millisecond, all clients would start at roughly the same same time, assuming the MedianNetworkLatency[i] is accurate. All clients will capture this time and use Date.Time() on their local machine to store the moment as InitialTime. Specific implementation of the delay will be discussed in the Framework section. Afterward, all clients jumpstart the tick loop. Some slight difference in framerate is likely unavoidable. For example, despite our protocol, there could still be a 10ms difference between the InitialTime of each client. To solve this, we designate a 15 tick per second tickrate. At any given moment T(fetched through Date.Time()), tick is equal to floor((T-InitialTime)/66). All inputs and state adjustment packets will be sent at the end of each tick and executed at the start of specified execution tick. This allows toleration for 66ms error, while not breaking the tick-to-tick synchronization.

C.Lag Catch-up

In certain scenarios, browser execution of the game may freeze temporarily. Although usage of worker thread prevents most of the issue, there could be situations in which players miss a few ticks. This is designed to ensure the simulation is lag-proof

1.At the start of each tick loop function, players fetch time T from date.time() and calculate its own tick based on floor((T-InitialTime)/66).
    a.assume there is a 2 tick delay and player missed tick 3 and tick 4 , and is currently on tick 5

2.for each missed tick(3, 4), run the Tick Execution Sequence, fetching and running state patch and input for tick 3 and 4, instantly

a.input delay, documented later, ensures all input runs on a specified frame across all clients.

b.State Patch is explained in the Input/State Synchronization section

3.run tick 5

D.Input delay

Because ticks are ensured to be synchronized, we use input delay to make sure input executes at the same tick on all clients. The delay amount is heuristically calculated and would be discussed in the implementation section.

E.Concurrent Simulation/State Patch/Authorities

All examples here are simplified to not include Random Authority Shuffle.

First, some terminology/symbols should be specified.

- Xa = position of a
- Xb = position of b
- Ha = health of a
- Hb = health of b
- State Patch - a packet containing state of a remote player at a certain tick, and usually arrive at a tick later than the specified tick
- Speculation - players perform inputs received from remote clients and simulate their state for them. However, since no determinism is guaranteed, the state for remote player may not match the authority state on remote player and thus is mere speculation
- Local Authority state - the state of a player, on its machine.
- Resynchronize - when a state patch for tick X for player B arrives, player A check for its speculation at tick X for player B. if they are different, player A revert all state back to tick X with correct state for player B (based on state patch), and instantly rerun all of the input after tick X in order
- Tick Packet - each tick’s input, state patch, and desync label is packaged and sent out in one UDP packet. This packet could be dropped. See Framework/implementation section for detailed solution to dropping frames.

Each client runs a simulation for players of self and other clients. The specific implementation of this structure is discussed in the implementation section. Each client only has authority on its own, local player data. For player data of remote clients, the local client maintains a speculative prediction called Speculation based on synchronized input simulation. In an ideal situation, where all clients maintain perfect determinism, this prediction is enough. Consider the following case:

- Player have 2 skills, move and attack(which only work if two players are adjacent)

In a simulation lacking determinism, the following could happen:

1.Player A and Player B start at Xa = 0, Xb = 2, respectively. For simplicity sake, there are only one dimension
2.At tick 1, Player A issues a move command(which would change Xa to 1), scheduled to execute at tick 3. (optionally, user of the framework could choose to extrapolate player to location without delay to improve responsiveness)
3.At tick 2, Player A issue attack command ,scheduled for tick 4
4.At tick 3, both player A and player B execute the move command. On Player A’s machine, Player A ends up at Xa = 1. However, due to non-determinism, on Player B’s machine, Player A ends up at Xa = 0.9.
5.At tick 4, both player A and player B execute the attack command. On A’s A's attack will register, causing B to lose health. However, on B’s machine, A’s attack would not register, causing a desynchronization.

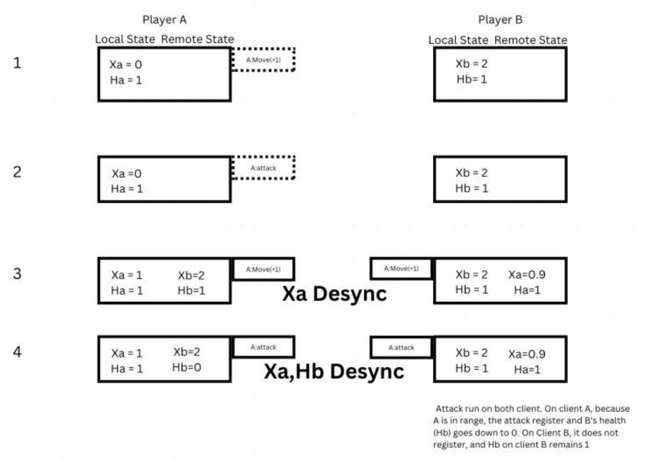

Figure 4. Player A vs Player 2

Note that in A’s speculation of B is that B lost health, while on B’s machine’s authority data, A’s attack missed, and B maintained full health. However, B’s machine’s calculation of full health is based on faulty simulation. Naive approach would simply rubber band A’s speculation of B to B’s true state. There are two problem, however:

1.A’s speculation of B, now accurate, will be 2 ticks behind because B’s state patch takes 2 tick to arrive
2.The root cause of this desynchronization is based on B’s faulty speculation - addressing the most recent desynchronization by simply rubberbanding makes the game appear illogical to both parties.

Our approach is a variation of rollback. Every tick, the player sends a State Patch packet, which specifies the current true state of the local player at the current tick rate. This state patch packet will arrive to the other client around 2-3 ticks later. The following protocol ensures proper state adjustment if there are speculation errors on remote clients. This is based on the previous scenario

1.At tick 5, remote client B receives state correction packet of client A for tick 3
 a.Packet content: true position of A is Xa = 1 at tick 3
2.Player B compare the packet content with stored speculation for Player A at tick 3
 a.if they are the same, do nothing
 b.if they are not the same(as in this scenario), proceed with the algorithm

3.Player B waits for all player (player C, player D) state synchronization packets for tick 3 to arrive. While player B is waiting, it is also actively speculating.

4.Player B reverts game state for player A on its local machine to tick 3 based on authority value given by player A. Player B then instantaneously Resynchronize all the input from the point on.

a.Resynchronize tick 3 with all input from tick 3 to 5. this means now player 3 recognize the hit, the hit gets registered, and two players are in sync

Because this technique traces the root of desynchronization and Resynchronize, the first State Patch after desynchronization should correct all of the following desynchronization on the client receiving the state patch, thus matching it with the sender [10]. In the example above, player B receives a state patch for player A's position. After resimulation, player B's local authority state of health now matches player A's speculation of player B's health because now with the correct position for tick 3 for player A, player B can detect and register the attack from player A. In this technique, the local authority state is not necessarily always the true state of a player across the network. In the example, player B falsely calculates local authority state of health = 1, because it did not register a hit due to faulty speculation of A's location being 0.9 instead of 1. In its essence, local authority state may be subject to change if it is calculated based on faulty speculation of remote clients' authority state from previous ticks.

However, this presents a problem if a state patch is being sent on every tick. It could be that before B has corrected its state, B sends a faulty State Patch to A.

See a possible scenario

1.at tick 4, player B's local authority state for health is 1.
  a.this is because player B had a wrong speculation of A's position at tick 3 and thus did not register the hit
  b.player B sends state patch, stating that its local authority state is {hB = 1}
2.at tick 5, player B Resynchronize, changing its health to 0
  a.player A's state patch packet arrives at tick 5, stating that player A's authority state at tick 3 is Xa = 1. Player B detects desync as its speculation of A's state at tick 3 is Xa = 0.9
  b.player B sends state patch again, now local authority state is {hB = 0}(which is the same as player A's speculation)
3.at tick 6, player B's faulty state patch{hB= 1}() for tick 4 arrives on player A
  a.player A Resynchronize
    i.now at tick 6, player A change player B's speculative state to {hB=1} from {hB=0}
4. at tick 7, player B's packet for tick 5 arrives
  a. player A Resynchronize
    i. speculation for B changes back to {hB = 0}

There are several unnecessary resimulations. Beside costing resources, it could lead to further smaller desync. Player A could have non-deterministic results on each simulation, for example. Or player A could calculate a different local state because of a faulty desync, and send out that state, which will induce a chain reaction across all clients, letting them Resynchronize over and over for every few tick intervals. Although this is not an infinite loop(as resimulation only happens at most once per tick), the true state for each client may be lost in this constant resimulation. Theoretically, since all players only broadcast their own state, when the tick loop stop, and the resimulation for each client's state packet of last tick finish, all players would ultimately be resynchronized again, with the chain of events sorted back all the way to the original desync. But this is highly improbable as the chain of resimulation expands exponentially with more players. Evidently, having all clients constantly sending state packets is not an efficient method and may result in highly complex resimulation based on delayed faulty state.

To ensure no client will receive faulty state packets when a remote client is desynced, we introduced a technique called Resynchronizing At Root. The technique guarantees that if a state patch causes a resimulation, it is at the very root of the desynchronization, and ignores all desynchronization that happened after the root because they will be automatically corrected by resimulation. In the example above, at tick 5, when player A's state patch for tick 3 arrives, player B would have a resimulation of tick 3 to correct the faulty speculation of Xa(from 0.9 to 1), which results in player B registering the attack input (scheduled for tick 4). On the other hand, at tick 6, when player B's faulty state synchronization packet for tick 4 arrive(falsely reporting that B's health Hb = 1 because of previous desync of Xa = 0.9), player A would know it is not the root of the desynchronization, and thus not trigger a resimulation. This is achieved by delaying the execution of a resimulation(from detecting difference between speculative state of remote player and state patch of remote player), until it is confirmed that all the previous tick up till the current tick is synchronized.

The protocol is as following

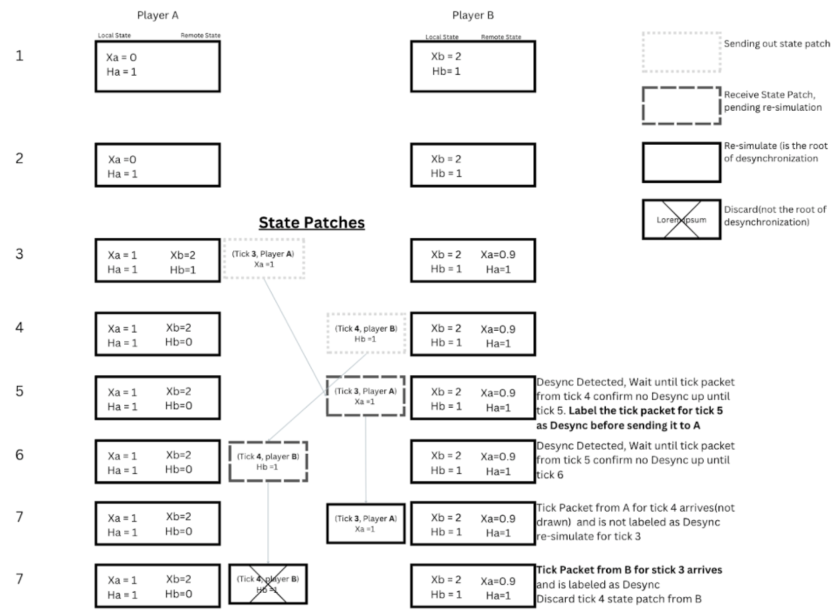
Player A
Player B
Local State
Remote State
Local State
Remote State
1
Xa = 0
Ha = 1
Xb = 2
Hb= 1
Sending out state patch
Receive State Patch, pending re-simulation
2
Xa =0
Ha = 1
Xb = 2
Hb = 1
Re-simulate (is the root of desynchronization
Discard(not the root of desynchronization)
Lorem Ipsum
State Patches
3
Xa = 1
Ha = 1
Xb=2
Hb=1
(Tick 3, Player A) Xa =1
Xb = 2
Hb = 1
Xa=0.9
Ha=1
4
Xa = 1
Ha = 1
Xb=2
Hb=0
(Tick 4, player B) Hb =1
Xb = 2
Hb = 1
Xa=0.9
Ha=1
5
Xa = 1
Ha = 1
Xb=2
Hb=0
(Tick 3, Player A) Xa =1
Xb = 2
Hb = 1
Xa=0.9
Ha=1
Desync Detected, Wait until tick packet from tick 4 confirm no Desync up until tick 5. Label the tick packet for tick 5 as Desync before sending it to A
6
Xa = 1
Ha = 1
Xb=2
Hb=0
(Tick 4, player B) Hb =1
Xb = 2
Hb = 1
Xa=0.9
Ha=1
Desync Detected, Wait until tick packet from tick 5 confirm no Desync up until tick 6
7
Xa = 1
Ha = 1
Xb=2
Hb=0
(Tick 3, Player A) Xa =1
Xb = 2
Hb = 1
Xa=0.9
Ha=1
Tick Packet from A for tick 4 arrives(not drawn) and is not labeled as Desync re-simulate for tick 3
7
Xa = 1
Ha = 1
Xb=2
Hb=0
(Tick 4, player B) Hb =1
Xb = 2
Hb = 1
Xa=0.9
Ha=1
Tick Packet from B for stick 3 arrives and is labeled as Desync Discard tick 4 state patch from B

Figure 5. Protocol

Note: tick packet is sent for every frame and thus not drawn.

Note: desync label from player A's tick packet represents player A has a faulty speculation. Therefore, if player B receives a state patch from player A that shows player B has a faulty speculation, but then receives another tick packet indicating player A is already desynchronized, player B does not Resynchronize.

1.At each tick X, the local player A may receive a state patch(as part of tick packet) for tick Y from a remote player a.tick Y is usually one or two tick behind tick X, such as X-2, depending how many tick it takes for a packet to arrive

2.local player A detect state patch for tick Y is different from its speculation for the past tick Y

a.does nothing. It is not determined whether or not tick Y is the root of desynchronization

b.mark tick packet for tick X with data {desync: tick Y}, this will arrive at tick Z.

c.if there are no difference between state patch and speculation, mark tick packet for tick X with data {synced:tick Y}

3.As the remote player B performs the exact same algorithm, player A is expected to eventually receive a tick packet containing the mark {synced: tick (Y-1)} or {desynced: tick (Y-1)}.

a.if {synced: tick (Y-1)}, Resynchronize as it means there is no desynchronization up till tick 3 thus it is the root cause of the desynchronization.

b.{desynced: tick (Y-1)}, does not Resynchronize – because player A's speculation may as well be the true state of the B, once B corrects its faulty speculation of A at tick Y-1 and Resynchronize up till tick Y.

If the local player receives a state patch that is different from speculation, we include in the current tick's tick packet an indicator that a previous tick is desynchronized. For example, if player B has a faulty speculation for player A's state at tick 3, and discover it at tick 5 after receiving tick 3 state patch from player A, player B would add the data along the line of "desync at tick 3" in its tick packet to indicate that it might Resynchronize because of a previous desynchronization. The flag Desync is for tick 3. It marks the tick in which the desynchronization happens, not the tick in which desynchronization is detected. In the previous example, the flag would specify there is a desync at tick 3 {Desync: 3}, while being packaged in the tick packet for tick 5. The flag allows player A to know that later state patches, like patch for tick 4 from player B, are not the root desynchronization and thus should not cause a resimulation. On the other hand, if player A received a state patch from player B that does not have desynchronized data, it will send out a tick packet with data {synchronized:3}, broadcasting that up till tick 3, there is no desynchronization. When player B receives this tick packet, it could safely assume that any desynchronization immediately after tick 3 is the root desynchronization. In the fringe case in which both players have speculation error at tick 3 for each other, it would also be root traced and corrected respectively.

Module 5, Implementation Details

The implementation will target browsers. Here we list down some specific key implementation details that make the framework efficient.

I, Web-Worker Multithreading

Main thread of a webpage in the browser is usually tied to the tab. This means when a tab goes idle, the main thread is subjected to being paused. Because of this, we implemented our framework in Web-Worker. Web-Worker provide the several advantages

1.separate from UI and game, thus will not be blocked by them

2.Can run on background, thus allowing update tick to run continuously

This median article https://medium.com/teads-engineering/the-most-accurate-way-to-schedule-a-function-in-a-web-browser-eadcd164da12 tested several different update loop methods in javascript [13].

- setInterval in main thread
- setInterval in web worker
- requestAnimation frame

It concludes that setInterval in web workers provides the most accurate delay based on the specified interval.

II, Managing UDP Packets

WebRTC's data channel API allows for both TCP and UDP packets to be sent across. However, we opted for UDP packet because there are several problems with TCP packets

- Ordered- causing jitter
  - ◇ packets arrive in irregular intervals due to network jitter. By forcing packets to arrive on the recipient's machine in order, TCP exacerbates the jittery effect.
- Reliable-causing lag
  - ◇ Because TCP waits for dropped packet to be resending before sending any new packet, it exacerbate the network jitter and also causing unnecessary lag

In situations where a continuous stream of data is favored over reliability, such as movie streaming, real-time fast paced games, UDP is the better choice because it aims to make the data arrive as fast as possible [15]. But raw UDP packets are unsuitable for our needs. Because our protocol does not require determinism, it is crucial for input packet and state patch for each tick to arrive on the remote player. Even one missed input could break both the simulation and the Random Authority Shuffle protocol. Therefore we packet previous ticks data into the current tick, until it is confirmed that they are received on a specific clients machine

1. At tick X, player A sends a tick packet to player B
   a. The packet contains input and state patches for tick X, tick X-1, and tick X-2.
   b. It also indicate that player A has received player B's packet up to tick X-1
2. Player B receives the tick packet (sent from step 1) at tick X +2.
   a. because the packet indicate player A has received player B's input and state up to tick X-1, player B only send a tick packet containing input and state of tick X, tick X+1, and tick X+2
   b. player B also indicate in its packet that it has received player A's packet up to tick X

This method allows for packets to arrive as fast as possible, while eliminating the unreliability of UDP. Because input and state are labeled for each tick, the packets can arrive in various order and still result in the same effect.

III, Input Delay

Our Protocol implements a variation of input delay. This paper will not dive into the detail of input delay as it has been thoroughly discussed in other literature. In our implementation of input delay, because our tick rate contains a 66ms interval, we allow for multiple input in order. This is because our tick runs at 15 ticks/second, while most browser updates at 60 frames/second. Therefore there could be multiple keyboard registers on the game engine in one tick(for example, a player pressing down W key for 60ms, resulting in 3 movement input).

Module 6, Discussion

In this paper, we introduced a framework for browser specific peer to peer multiplayer. Theoretically, the idea behind the protocol can work for desktop applications as well. But desktop applications usually would not need it as it has more security and anti-cheat options, such as anti-cheat software that detects RAM modification [14].

For security, we introduced two novel protocols to ensure no player can manipulate data to gain an unfair advantage: Random Authority Shuffle and Speculation Based State Verification. Random Authority Shuffle is a novel concept based on the core principle of using probability to reduce the success of targeted cheating by assigning each player to manage the authority of an unpredictable, random remote player. This way, no player can continuously target a specific player and any fine tuned cheating will likely fail due to the luck that it requires for cheater to be assigned with its target at the right time whereas any drastic cheating that significantly alters the state of remote player will be caught with Speculation Based State Verification. Speculation Based State Verification is a modified version of rule-based state verification, where if there is a significant discrepancy from data authority holder, players can convene and determine democratically if the data authority holder manipulated the data by simply checking if it differs from all other clients' simulation.

For Synchronization, we introduced a novel technique called Resynchronizing At Root , where it prevents unnecessarily re-simulation by ensuring that a re-simulation only happens if the tick to be Resynchronized is the root of desynchronization. Because re-simulation simulates the tick with all the input after it in order, it will remove the desynchronization chain that is caused by the root. We also used time based lockstep to ensure ticks are synchronized across all machines.

Module 7, Future Research

Although this paper addresses the state manipulation cheat from players, it is unsuitable for games that require secret data. We did not address how a client could hide private, strategic data while proving it is not cheating to other clients.For example, in a situation where a player wishes to hide its data(such as poker), how could a remote client make sure that player does not cheat by manipulating the private data? This could be solved with a Zero-Knowledge-Proof protocol.

ACKNOWLEDGEMENTS

I would like to express my gratitude toward Dr.Sun (Cal Poly Pomona) for his generous support and encouragement for this paper and my research into Cryptography.

REFERENCES

[1] Y. Dodis and A. Yampolskiy, "A verifiable random function with shortproofs and keys," in Public Key Cryptography (PKC'05). Springer,2005, pp. 416–431.
[2] D. Chaum, "Blind signatures for untraceable payments," in Advances incryptology. Springer, 1983, pp. 199–203
[3] El Saddik, Abdulmotaleb & Dufour, A.. (2003). Peer-to-peer suitability for collaborative multiplayer games. 101- 107. 10.1109/DISRTA.2003.1243003.
[4] Denning, T., Lerner, A., Shostack, A., & Kohno, T. (2013). Control-Alt-Hack: the design and evaluation of a card game for computer security awareness and education. Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security.
[5] Hincapié-Ramos, Juan David & Henao, Andres. (2007). P2P Game Network Framework - Communications and Context Framework for Building P2P Multiplayer Games (Intro).

[6] Blanchet, Bruno. "Security protocol verification: Symbolic and computational models." International Conference on Principles of Security and Trust. Springer, Berlin, Heidelberg, 2012.

[7] El Rhalibi, Abdennour, Madjid Merabti, and Yuanyuan Shen. "Aoim in peer-to-peer multiplayer online games." Proceedings of the 2006 ACM SIGCHI international conference on Advances in computer entertainment technology. 2006.

[8] Kamara, Seny, and Kristin Lauter. "Cryptographic cloud storage." International Conference on Financial Cryptography and Data Security. Springer, Berlin, Heidelberg, 2010. G., and Richard P. Mislan. "Mobile device analysis." Small scale digital device forensics journal 2.1 (2008): 1-16.

[9] Wortman, Paul, et al. "P2M‐based security model: security enhancement using combined PUF and PRNG models for authenticating consumer electronic devices." IET Computers & Digital Techniques 12.6 (2018): 289-296.

[10] Pfurtscheller, Gert, and FH Lopes Da Silva. "Event-related EEG/MEG synchronization and desynchronization: basic principles." Clinical neurophysiology 110.11 (1999): 1842-1857.

[11] Sredojev, Branislav, Dragan Samardzija, and Dragan Posarac. "WebRTC technology overview and signaling solution design and implementation." 2015 38th international convention on information and communication technology, electronics and microelectronics (MIPRO). IEEE, 2015.

[12] Johnston, Alan, John Yoakum, and Kundan Singh. "Taking on webRTC in an enterprise." IEEE Communications Magazine 51.4 (2013): 48-54.

[13] Jensen, Simon Holm, Anders Møller, and Peter Thiemann. "Type analysis for JavaScript." International Static Analysis Symposium. Springer, Berlin, Heidelberg, 2009.

[14] Powers, Scott, et al. "RAM, a gene of yeast required for a functional modification of RAS proteins and for production of mating pheromone a-factor." Cell 47.3 (1986): 413-422..

[15] Gu, Yunhong, and Robert L. Grossman. "UDT: UDP-based data transfer for high-speed wide area networks." Computer Networks 51.7 (2007): 1777-1799.

AUTHOR

Haochen Han

Haochen Han is a student living in Orange County with a passion for Computer Science, Networking Programming, and Cryptography. Author of Treasure of Forrealm, a multiplayer 2D PVP party game.