Master Thesis
Algebraic Server Reconciliation for Online Games

Philipp Hinz
Reviewed by Dr.-Ing. Guido Rößling

25. April 2025

1 Modelling

We define a game world using a game state S, an initial game state $s_0 \in S$ and a progression function $f : (S, I) \to S$ as $G = (S, s_0, f)$ where I is some external input. A game state at time t can be progressed using some input $i_t \in I$ to time $t + 1$ using the progression function: $s_{t+1} = f(s_t, i_t)$. We can combine the input and state to a frame $r_t = (s_t, i_t)$. We say that a machine $m_t \in M$ is running the game G with state s_t at time t. Clients are machines $c_t \in S \subseteq M$ which contribute input $i_t^c \in I^c$ to form the whole input $i_t^c \in i_t$. We assume that a machine is designated as the host and all clients can only communicate with the host. The state in the host is the universal truth.

1.1 Naive model

We have one host h and n clients c_i. Each client c_i has a rount trip time of t_i^r. We define the rount trip time as the sum of the send and receive latency $t_i^r = t_i^s + t_i^e$ Every tick the server sends its state s_t^h to each client, while each client sends their input $i_t^{c_i}$. The server applies the most recently received input, therefore each client will only be able to react with input $i_t^{c_i}$ to state $s_{t-t_i^r}^h$. We call this the naive model.

1.2 Fair naive model

This is not fair, as each player can only respond to a state s_t^h with an input t_i^r time later and t_i^r is different for every player. We can make it fair by only processing input on the server, once we received input from all players. Every player is able to react to state s_t^h with the same delay of $\max_i t_i^r$. Intuitively we artificially delay delay what each player can see to the slowest player. We call this the fair naive model.

Every game is based on reaction, doing an action in response to an event. A major problem with the current method is that we can only always respond to past events, even if fair, wont feel natural to players. (Improve writing)

1.3 Lockstep

One common solution in slow strategy games where the times interval is not fixed, is for each player only to react to an input, when it is received. This means, that for every s_t^h the player will respond with i_t^c and the server will use all i_t^c to generate s_{t+1}^h. We now have a fair game while giving players the most recent information to react. This model is known as and called here the lockstep model. It is currently in use in strategy games like starcraft.

Unfortunately most real-time games require an fluent time interval, often around 30 updates per second or more. That would force the rount trip time to an unrealistic $\max_i t_i^r < 33.3\text{ms}$, ignoring rendering and reacting to a frame by the user using an corresponding input. From here on out we assume that the latencies are larger in magnitude than the update interval of the state.

1.4 Other player limitation

We are also bound in the time a player is possible to react to other players inputs. One player has to first send their inputs and another player can only then receive them. Any changes to the state from player c' will never be visible to an other player c faster than $t_c^e + t_{c'}^s$.

1.5 Client side prediction

While there is an limitation in seeing other players inputs, there is no limitation on seeing state changes to your own inputs sooner. We define a player progression function $f^c : I^c \to S \to S$ which only progresses the state based on one single client input i_t^c. Each client now receives state s_t^h and directly applies their input to get predicted state $p_{t+1}^c = f^c(i_t^c, s_t^h)$. This method is called client-side prediction and used in most modern games as it provides the user with instantous feedback - making gameplay fluent.

The difficulty with this method is to apply new incoming host state s_{t+1}^h correctly. Given a client predicted p_{t+1}^c, it will receive an response to input i_t^c in at least t_c^r time as the state $s_{t+t_c^r}^h$. An previous state change arriving as s_{t+1}^h would override our predicted state p_{t+1}^c making the game snap. The original solution to this problem is to not replace our current state but define a partial application function $f^{\text{partial}} : S \to S \to$

S which compares the predicted state with the actual state and only replaces diverging parts. During normal operation, the prediction will be close enough causing only rare desynchronizations which will be visually visible as snapping. We should note, that some parts of the state will always diverge, if state changes are caused by other players or randomness.

Client side prediction is not fair, as each player will produce an input while seeing a different predicted state. Intuitively this can be explained, as beeing able to visually see parts of game state after prediction, which other players can not see. That can for example be another around the corner, who will only see you around a rount trip time later.

One very popular game that is using slightly modified variation of this method is minecraft. In minecraft the game does not send its direct inputs, but its predicted state. Most important parts of the state to be predicted are breaking blocks and all character movement. The partial application does not happen at client side, but on server side. Therefore the server validates if the incoming input looks good enough and doesnt resimulate it if not necessary. When detecting diverging state, the host sends state correction. Unfortunately this limits anti-cheat capabilities to how sophisticated the validation in the partial application function is.

One major problem with client side reconciliation is that finding a good partial application function can be difficult. The reason is that we are looking at divergences between s^h_{t+1} and $p^c_{t+t^r_c}$, therefore a difference in time of $t^r_c - 1 \approx t^r_c$. Therefore even a perfect prediction will divergence in scale of the round trip time.

Before introducing the modern solution, one alternative approach would be to not compare the current state for divergence, but memorize past states. Given a interval time of t^L, the client needs to memorize $\frac{t^r_c}{t^L}$ gamestates, which can be partial. This solution unfortunately doesnt solve the snapping when divergences are found, since it forces the client from a predicted state in time $t + t^r_c$ to $t + 1$.

1.6 Server reconciliation