Algebraic Server Reconciliation for Online Games

Algebraische serverseitige Zustandskorrektur für Online-Spiele
Master thesis by Philipp Hinz
Date of submission: October 04, 2023

1. Review: Dr.-Ing. Guido Rößling

Darmstadt

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Erklärung zur Abschlussarbeit gemäß § 22 Abs. 7 APB TU Darmstadt

Hiermit erkläre ich, Philipp Hinz, dass ich die vorliegende Arbeit gemäß § 22 Abs. 7 APB TU Darmstadt selbstständig, ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt habe. Ich habe mit Ausnahme der zitierten Literatur und anderer in der Arbeit genannter Quellen keine fremden Hilfsmittel benutzt. Die von mir bei der Anfertigung dieser wissenschaftlichen Arbeit wörtlich oder inhaltlich benutzte Literatur und alle anderen Quellen habe ich im Text deutlich gekennzeichnet und gesondert aufgeführt. Dies gilt auch für Quellen oder Hilfsmittel aus dem Internet.

Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass im Falle eines Plagiats (§38 Abs.2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Bei einer Thesis des Fachbereichs Architektur entspricht die eingereichte elektronische Fassung dem vorgestellten Modell und den vorgelegten Plänen.

Darmstadt, 04.10.2023

Philipp Hinz

Abstract

This is a template to write your thesis with the corporate design of TU Darmstadt.

Contents

List of Symbols

- t - time

List of Figures

1 Introduction

2 Related work

3 Network model

To rigorously discuss and analyze networking techniques for online games, particularly those involving state synchronization and reconciliation, it is essential to first establish a clear and formal understanding of the underlying concepts. This chapter lays the theoretical groundwork by defining fundamental components of a game from a networking perspective, such as game state, state progression, and the roles of clients and the server. Building upon these definitions, we will then explore several established network models, from naive approaches to more sophisticated techniques like client-side prediction and server reconciliation. This exploration will highlight the challenges inherent in achieving responsive and consistent gameplay in a networked environment. Finally, this chapter will introduce the core theoretical concepts behind delta states and culminate in the formal definition of algebraic server reconciliation, the novel method investigated in this thesis.

3.1 Overview

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 $m \in M$ is running the game G with state s_t at some time t. Clients are machines $c \in C \subseteq M$ which contribute input $i_t^c \in I^c$ to form the whole input $i_t^c \in i_t$. We assume that one machine is designated as the host and all clients can only communicate with the host. The state on the host is considered the truth if states between machines differ.

3.1.1 Naive model

We have one host h and n clients c_i. Each client c has a round trip time of t_c^r. We define the round trip time as the sum of the send t_c^s and receive t_c^e latency $t_c^r = t_c^s + t_c^e$. At each tick, the host sends its state s_t to each client and each client sends input i_t^c.

Both the client and the host run with a fixed time interval t^i which is usually smaller than t_c^r. Therefore the host cannot wait with processing all input i_t^c and will use the most recent input instead. This means that each client will only be able to react with input i_t^c to state $s_{t-t_c^r}$. We call this the naive model.

3.1.2 Fair naive model

Each client c can only respond to a state s_t with an input arriving at the host t_c^r time later. Since t_c^r is different for every player, this is not fair. We define a fair model as: given a fixed delay δ, every client can respond to the same state s_t with input $i_{t+\delta}^c$. We can make the previous model fair by only processing input on the host, once we received input from all players. Every player is able to react to state s_t with the same delay of $\delta = \max_i t_i^r$. Intuitively, we artificially delay what each player can see to the slowest player. We call this the fair naive model.

Many games are based on reaction, doing an action in response to an event. A major problem with the (fair) naive model is that player inputs in reaction to a present state will only be applied to a future state on the host.

3.1.3 Lockstep

In the fair naive model, the host is applying old inputs because the interval time is fixed and usually faster than t_c^r. We can loosen this requirement and assume a dynamic interval time. Players only send input i_t^c, after receiving s_t. The host only progresses to state s_{t+1} when all i_t^c have arrived. We now have a fair game while giving players the most recent information to react to.

This model is known as the lockstep model and popular in strategy games, where latency or a longer interval are not a big problem.

3.1.4 Player observation limitation

We are bound in the time a player is able 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 another player c faster than $t_c^e + t_{c'}^s$.

When the interval rate is fixed to a time $\delta < t_c^e + t_{c'}^s$, which is usually the case, a client c will already have made an input i_{t+1}^c without there being a possibility of receiving input $i_t^{c'}$ from client c'. Therefore according to our definition, games with a fixed interval cannot be fair.

3.1.5 Client side prediction

In the (fair) naive model, a client will only be able to apply their input i_t^c to a state $s_{t-t_c^r}$. Therefore a client sees responses to their own input with a latency of t_c^r.

While we previously showed an absolute limitation in seeing other players' inputs, there is no such limitation on seeing state changes to your own inputs. We define a prediction function $f^p : I^c \times S \to S$ which only progresses the state based on one single client input i_t^c. We write $p_{t,d}^c$ for a client c, predicting from timestamp t, d ticks to the future. Each client now receives state s_t and directly applies their input to get predicted state $p_{t,1}^c = f^p(i_t^c, s_t)$. Since responses to i_t^c will be received in $s_{t+t_c^r}$, we predict the state up to $p_{t,t_c^r}^c$. This method is called client-side prediction and used in most modern games as it provides the client with instantaneous feedback - making gameplay fluent.

The difficulty with this method is to apply new incoming host state s_{t+1} correctly. Given a client predicted $p_{t,t_c^r}^c$, it will receive a response to input i_t^c in at least t_c^r time as the state $s_{t+t_c^r}$. A previous state change arriving as s_{t+1} would override our predicted state $p_{t,t_c^r}^c$, making the game snap, caused by the time difference. The original solution to this problem is to not replace our current state but define a partial application function $f^{\text{partial}} : S \times S \to S$ 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 observe, that some parts of the state will always diverge, if state changes are caused by other players or randomness.

One very popular game that is using a 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 host-side. Therefore the host validates if the incoming input looks good enough and doesn't resimulate it if not necessary. When detecting diverging state, the host sends state corrections. 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_{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 diverge on the scale of the round trip time.

One alternative approach would be to not compare the current state for divergence, but memorize past states. Given an interval time of t^i, the client needs to memorize $n_{\text{predict}} = t^r_c / t^i$ game states, which can be partial states. This solution unfortunately doesn't solve the snapping when divergences are found, since it forces the client from a predicted state in time $t + t^r_c$ to $t + 1$.

3.1.6 Server reconciliation

When merging an update s_{t+1} with a diverged predicted state p^c_{t,t^r_c}, we previously overrode the state with state s_{t+1}. This skip from $t + t^r_c$ to $t + 1$ causes snapping. What we can do alternatively is to memorize inputs $i^c_k : t + 1 \leq k \leq t + t^r_c$. We replace our state with s_{t+1} and then apply all memorized inputs to get an alternative predicted state $p^{c\prime}_{t,t^r_c}$ which would have been reached if the divergence didn't happen. Since we now have a state in time $t + t^r_c$, we do not have snapping behavior. This method is called rollback server reconciliation and is one of the most fundamental techniques in modern games.

Server reconciliation is the process of combining a past received state s_t with a present predicted state p^c_{t,t^r_c}. Our partial application function is a form of server reconciliation. The goal is to fix divergences in state without the client noticing.

Unfortunately, since we have to rollback to the last received state, we must memorize all predictions we made. In the worst case, we memorize the whole s_t. When receiving s_t, we apply the prediction function n_{predict} times. Therefore an input i^c_t is predicted for the first time at $t - n_{\text{predict}}$ and the last time at t. So each input is processed by the client n_{predict} times. This means that especially clients with longer ping have to do more processing, increasing with a smaller tick interval.

3.1.7 Delta State

The host is sending the whole state s_t each tick. This means we will send redundant data as the state rarely changes completely. We define delta states $\Delta s_t \in \Delta S$ as changes in state which can be applied using a delta application function $f^A : S \to \Delta S \to S$. We define an alternative progression function $f^\Delta : I \to S \to \Delta S$ returning a delta instead of the whole state and write $f^\Delta(i_t, s_t) = \Delta s_t$. We can derive the original progression function as $f(i_t, s_t) = f^A\big(s_t, f^\Delta(i_t, s_t)\big)$, therefore all our previous results hold true when using delta states. The other way around we can convert any normal state to delta state given progression function f and state S by defining $f^A = f$, $\Delta S = S$ and $f^A(\cdot, s_t) = s_t$. This means, that in practice we can always implement delta state and use normal state on top.

The host will each tick send Δs_{t+1} to the clients. In the (fair) naive or lockstep model, a client has currently loaded a state s_t and can simply use the delta application function to progress the state. Using rollback reconciliation, the client can use the delta application function after the rollback to s_t and predict the actual state afterwards. If delta state can be used with the partial application reconciliation depends on the state and cannot be generalized because it is not possible to get back to a state s_t.

3.2 Algebraic server reconciliation

3.2.1 Definition

An Abelian group is a set A, together with an operation $+ : A \times A \to A$ which is associative and commutative. Additionally, it has an identity element $a + e = a$ and for every element a an inverse $a + a' = e$.

We assume a delta state that defines an abelian group with $S = \Delta S$, $e = s_e$ and $+ = f^A$, where f^A satisfies the required properties. We can observe, that we still have advantages over non-delta state, since we do not need to transfer state, when $\Delta s = s_e$.

We define a delta prediction function $f^{\Delta p} : I^c \times S \to \Delta S$ producing a delta $\Delta p^c_{t,d} = f^{\Delta p}\left(i^c_{t+d}, p^c_{t,d}\right)$ instead of the whole state when predicting the next state. Therefore $p^c_{t,d+1} = p^c_{t,d} + \Delta p^c_{t,d}$ and $p^c_{t,d} = s_t + \sum_{k=0}^{d} \Delta p^c_{t,k}$.

The client memorizes $n_{\text{predicted}}$ predicted deltas $\Delta p^c_k : t \leq k \leq t + t^r_c$. We define a new server reconciliation method as follows: For each incoming Δs_{t+1}, we add a correction update $\epsilon = \Delta s_{t+1} - \Delta p^c_t$ to our current state $p^c_{t+1,t^r_c} = p^c_{t,t^r_c} + \epsilon$. Assuming that predictions are mostly uncorrelated $\Delta p^c_{t,d} \approx \Delta p^c_{t+1,d-1}$, we can show convergence:

$$
\begin{aligned}
& p^c_{t,t^r_c} + \epsilon \\
&= p^c_{t,t^r_c} + \left(\Delta s_{t+1} - \Delta p^c_{t,0}\right) \\
&= s_t + \sum_{k=0}^{t^r_c} \Delta p^c_{t,k} + \left(\Delta s_{t+1} - \Delta p^c_{t,0}\right) \\
&= \left(s_t + \Delta s_{t+1}\right) + \sum_{k=1}^{t^r_c} \Delta p^c_{t,k} + \left(\Delta p^c_{t,0} - \Delta p^c_{t,0}\right) \\
&= s_{t+1} + \sum_{k=1}^{t^r_c} \Delta p^c_{t,k} \\
&\approx s_{t+1} + \sum_{k=0}^{t^r_c - 1} \Delta p^c_{t+1,k} \\
&= p^c_{t+1,t^r_c - 1}
\end{aligned}
$$

Therefore we can reach any prediction p^c_{t+d,t^r_c} over time. We call this reconciliation method algebraic server reconciliation.

3.2.2 Limitations

4 Networking Architecture

The goal of a networking system is to provide a seamless interaction with other players inside a game. How to achieve a seamless interaction with other players was investigated in Section 3. The focus of systems architecture is on how to implement these models within the game's underlying system. Consequently, this chapter is of less concern to players and more pertinent to developers. Just as we aim for seamless interaction for players, we also strive for seamless game development for developers. The architecture here is only one possible approach to networking architecture, which will be different depending on needs and game architecture.

The here introduced architecture will give an overivew how previously introduced concepts in game networking can be implemented practice and which considerations are made when implementing. It will enable us how our newly introduced method integrates in a networking stack and interacts with other components. It will also give a better understanding in what is happening, when we present our experiments and data afterwards.

4.1 Overview

Our networking system is designed as an independent layer (the “networking layer”) positioned beneath the game system (the “game layer”). The goal is to expose minimal complexity, making game development akin to creating a single-player game. The game system comprises a “world” (representing the game state), game logic that modifies this world, and events. These events facilitate communication between different logic elements and trigger user-facing indicators, such as animations.

This separation of concerns within the game system itself facilitates a cleaner interface with the underlying networking layer, as illustrated below.

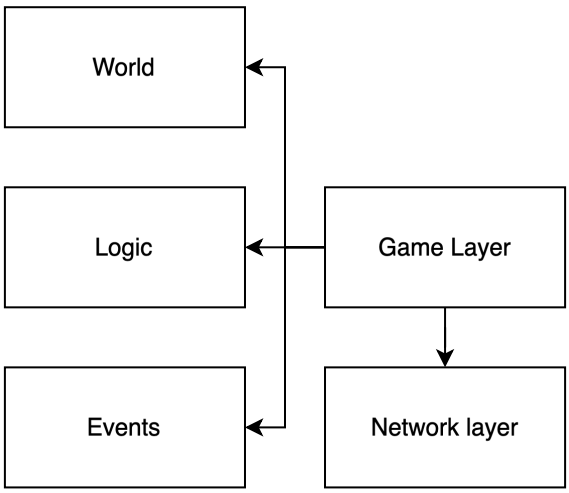

Figure 4.1: The fundamental division between the game layer, responsible for gameplay mechanics and presentation, and the network layer, responsible for communication.

4.1.1 Four quadrants model

Visually, the game and networking layers can be divided into four quadrants. The horizontal axis differentiates between the server and the client. As in Section 3, we use the classic server/client model, where the server has authority over the game world. Clients can only interact with the game world by sending their inputs, where the server will send updates to the game world (called replication) and events (for example, to trigger animations). The version of the world maintained on the client, which is partially updated through replication, is referred to as the "world view."

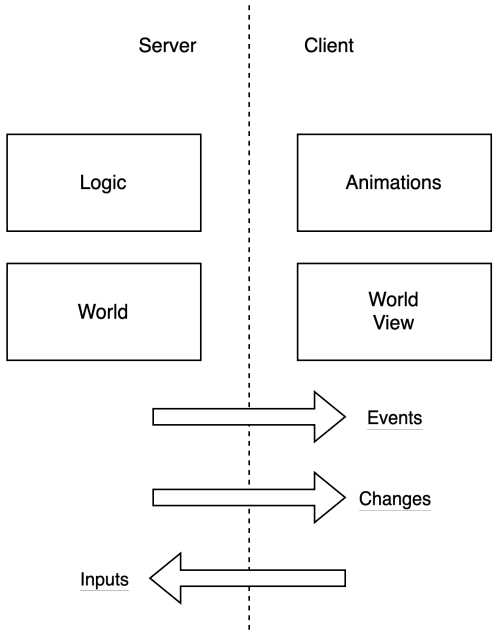

Figure 4.2: Building on this client-server distinction, we now incorporate the differentiation between the game and network layers.

4.1.2 Server and Client

The vertical axis, as previously introduced, differentiates between the game and network layers. Both the game and networking layers must serve two distinct scenarios: operating as a server or as a client. First, looking at the game layer, the server usually does not want to render the world. This independence not only boosts performance by avoiding unnecessary rendering overhead on the server but also simplifies porting the server logic to environments without graphical capabilities.

Most game logic is also only needed on the server, as changes are replicated authoritatively to the client. An exception could be user-controlled elements of the game world, such as the player character. However, similar to the approach in Section 3, we distinctly define this as "prediction logic" to improve cohesion. Additional event-based visuals, like animations, are typically only required on the client, as they do not impact the core game logic. This ensures that client-side presentation details do not impose constraints or unnecessary complexity on the authoritative server-side game simulation.

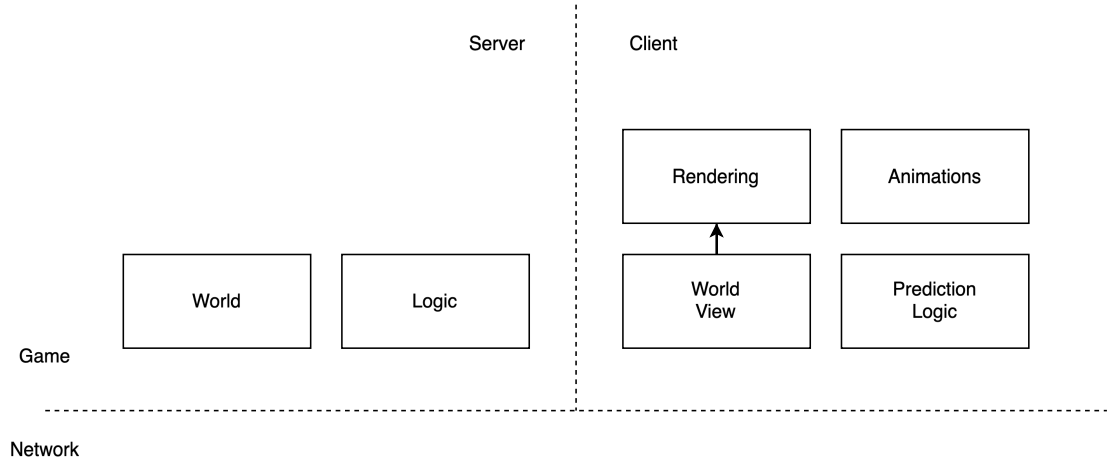

Figure 4.3: The server manages the authoritative World and Logic, while the client maintains a World View, handles Prediction Logic, Rendering, and event-based Animations.

4.1.3 Game and Network

The networking layer must cater to the server logic by providing a list of player inputs each game tick. Before replication, the networking layer has to look for changes in the server world. As introduced with “delta state” in Section 3, the system aims to send only the changes in the game world (deltas) rather than the entire world state each time. The efficiency of this delta state transmission is crucial for minimizing bandwidth usage and server load.

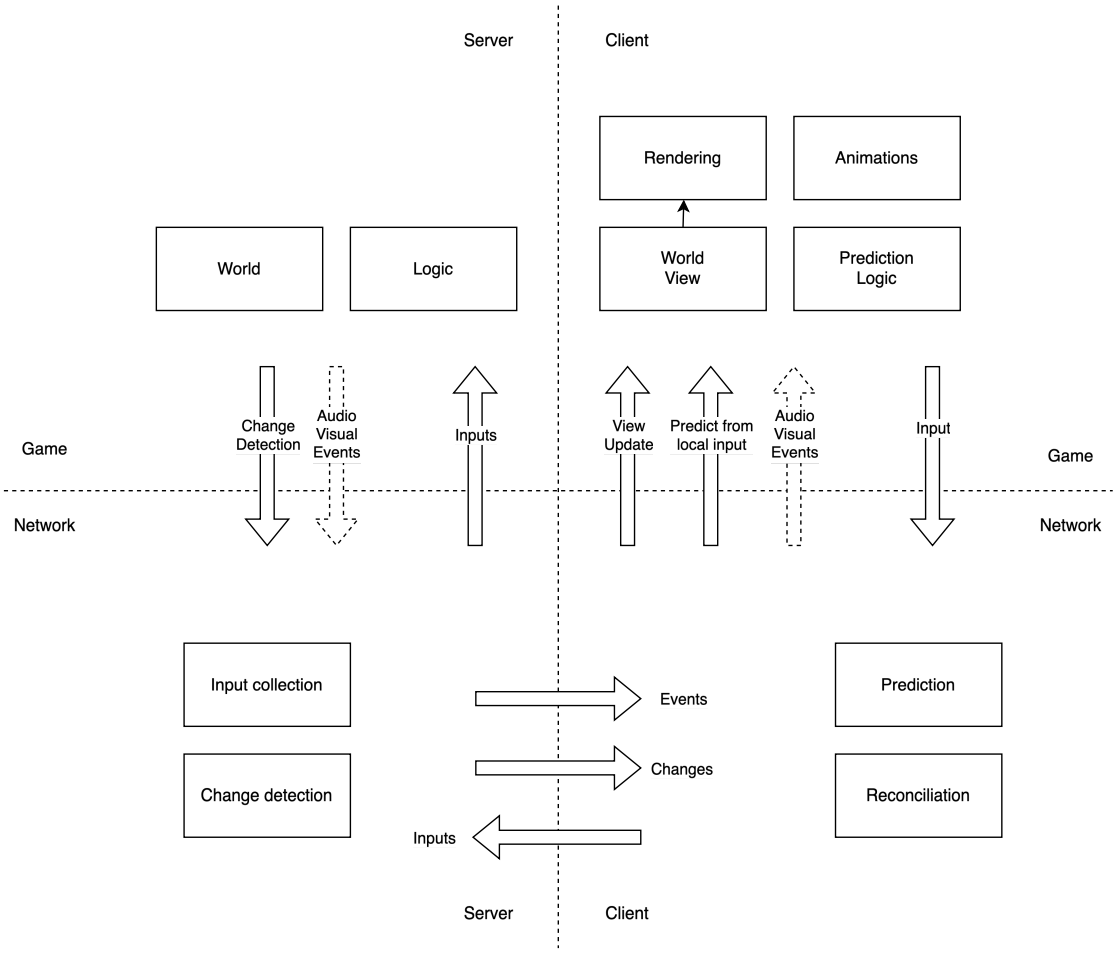

Figure 4.4: The complete four-quadrant architecture, illustrating the interplay between Game and Network layers on both Server and Client, including data flows.

Depending on how the world in the server is structured, it might provide additional helpers to detect changes. When the client receives changes, it applies them to its local game world, provided these changes are not already predicted. Otherwise, we need to use reconciliation to update our world view. This continuous loop of input collection by the network layer, server processing, state replication, and client-side reconciliation (if necessary) forms the core of the real-time networked experience.

4.2 Server networking layer

Previously, we explained an overview of our architecture and how components abstractly interact. Here, we want to dive deeper into the server-side dataflow. The following diagram displays all relevant components of the server-side networking layer, including their interaction with the game state.

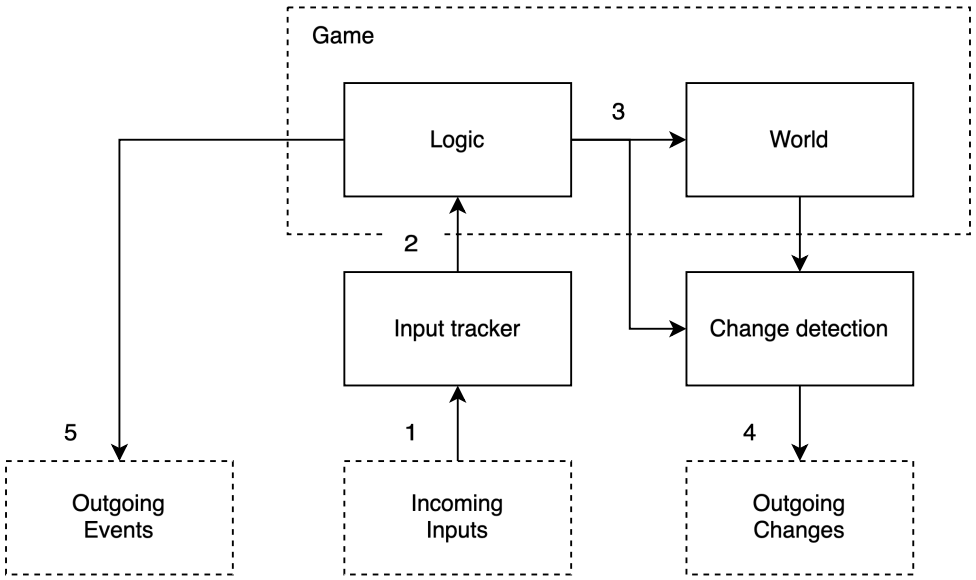

Figure 4.5: Server-side networking components and dataflow, illustrating the processing of incoming inputs and the generation of outgoing changes and events.

We have three primary communication channels with clients: incoming inputs from players, outgoing changes to replicate the game world, and outgoing events. Events are not strictly part of the game world but are needed for richer visual experiences on the client. The input tracker collects incoming inputs from players, providing a consolidated input to the game logic each tick. It is also responsible for providing stale or empty inputs if a new input has not arrived in time. The change detection mechanism is responsible for replicating the game world to all clients, ensuring each client has the same (or a consistent partial) world view by exchanging deltas.

The dataflow can be summarized by the following steps, which correspond to the numbered arrows in the diagram:

1. Users send their inputs to the server. The server receives these inputs, and the input tracker keeps track of them. Since inputs can be sent via unreliable communication methods, this component might also include logic for filtering inputs (e.g., processing only the most recent ones) or handling out-of-order packets.
2. Each tick, a set of all relevant inputs (one per player, potentially synthesized by the input tracker if an actual input is missing) is provided to the game logic. The game logic itself is fully defined in the game layer but is driven by inputs supplied by the networking layer.
3. Using these inputs, the game logic in the game layer modifies the world state. Optionally, the game logic may directly inform the change detection component about specific changes that have occurred.
4. Modifications to the world state need to be communicated to all clients. This is managed by the change detection component, which generates deltas representing these modifications. Since world state changes happen frequently and quickly, unreliable communication methods might be preferred for sending these deltas to clients. However, using unreliable transport for state updates adds complexity, such as the need for the server (or client) to track applied changes and potentially re-send missed updates to ensure eventual consistency.
5. Processing the game logic can also trigger occurrences that are not direct modifications of the persistent game world but are important for the player experience (e.g., sound effects, temporary visual effects). These are transmitted directly to the corresponding clients as events. Events are often sent reliably if they are critical, or unreliably if they are transient and missing one isn't detrimental.

This entire process is repeated each game tick while the game is running.

4.3 Client networking layer

The overall structure and communication pathways of the client-side networking layer were introduced in the architectural overview. Having inspected the dataflow within the server-side networking layer, we now turn our attention to the client-side components. This part is particularly important for our discussion, as it is where our new algebraic reconciliation method integrates. The following diagram presents the client-side dataflow.

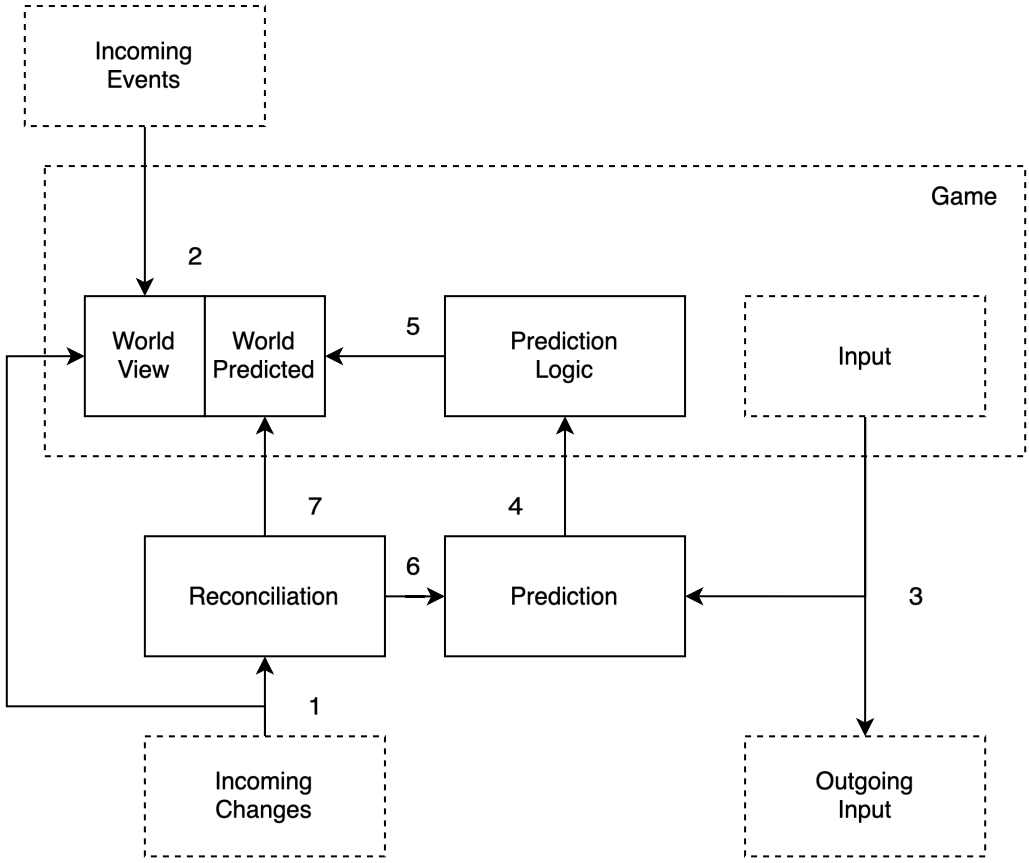

Figure 4.6: Client-side networking components and dataflow, detailing the processing of local player inputs, incoming server changes and events, and the interaction between prediction and reconciliation modules.

Similar to the server-side, the client has three primary communication channels: incoming events from the server, incoming state changes from the server, and outgoing player input. We will not focus extensively on event handling in this section, assuming that the world view component processes them (e.g., by triggering local visual or audio effects). The reconciliation, prediction, and prediction logic components collectively form the most interesting part of this architecture for our purposes. The interactions between these three elements differ depending on the specific reconciliation strategy employed, as will be detailed in the following subsections. We assume the client's representation of the game world is conceptually split into a "normal" part (the world view, reflecting acknowledged server state) and a predicted part (reflecting speculative local changes).

The dataflow can be summarized by the following steps, which correspond to the numbered arrows in the diagram:

1. Each tick, the server replicates state changes (deltas) to the client. These changes can correspond to either the predicted or non-predicted parts of the client's world representation. If a change affects a non-predicted part, it is directly applied to the world view. Otherwise, if it affects a predicted part, it is passed to the reconciliation module.
2. In parallel with receiving world state changes, the client also receives events from the server. As mentioned, we assume the world view handles these, for instance, by initiating animations or sound effects that do not alter the core game state logic.
3. Each tick, the game layer captures inputs from the player. These inputs are sent to the server as quickly as possible, often using an unreliable communication channel to minimize latency. Critically for client-side responsiveness, these inputs are also passed to the local prediction module.
4. After the prediction module receives inputs from the game layer, these inputs are fed into the game-specific prediction logic. If rollback reconciliation is being used (as detailed in Section 4.3.1), this prediction module is also responsible for memorizing these inputs for potential future re-simulation.
5. The game-layer defined prediction logic uses the local inputs to modify the predicted part of the world, most commonly affecting entities directly controlled by the player, such as their character. This immediate local modification provides the player with responsive feedback to their actions.
6. When incoming state changes from the server (received in step 1) affect parts of the world that have been locally predicted, these discrepancies must be handled by

the reconciliation module. This module receives the authoritative server change and information about the current predicted state from the prediction module.

7. If rollback reconciliation is employed, the reconciliation module will use the memorized inputs (from step 4) and the authoritative server state to re-simulate the player's actions and compute a corrected predicted part of the world. For other reconciliation methods, like algebraic reconciliation (detailed in Section 4.3.2), this step will differ. After processing, the reconciliation module updates the predicted part of the world to align it more closely with the authoritative server state, aiming to correct any mispredictions smoothly.

Similarly to the server-side dataflow, this entire process is repeated each game tick while the game is running.

4.3.1 Rollback reconciliation

When an incoming authoritative state change (delta) from the server affects the predicted part of the client's game world, reconciliation is necessary to correct any mispredictions. The most common reconciliation method currently used in games is rollback reconciliation, which was introduced conceptually in Section 3.1.6. The core idea behind rollback reconciliation is to revert the client's game state to the last known authoritative state before the misprediction occurred, apply the incoming server correction, and then re-simulate all local player inputs that have occurred since that authoritative state.

The interactions for this process within our client-side architecture are illustrated below.

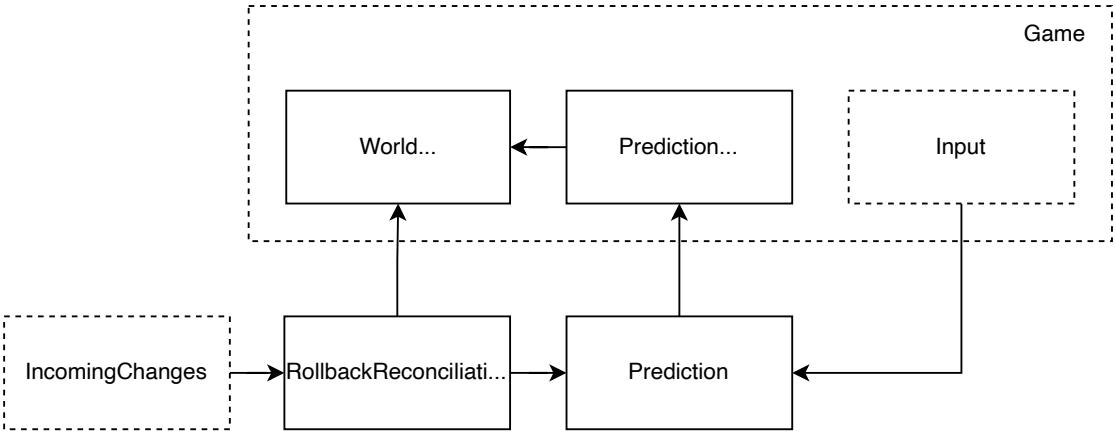

Figure 4.7: Dataflow for rollback reconciliation on the client. Incoming changes trigger a rollback, replay of memorized inputs via the prediction module, and update of the predicted world.

The primary challenge in implementing rollback reconciliation lies in efficiently reverting the game state. When working with delta states (as discussed in Section 3.1.7), two common approaches are:

1. **State Cloning**: Maintain at least two distinct copies (or snapshots) of the predicted part of the world. One copy represents the state before the latest batch of local

predictions is applied. When an incoming server change necessitates a rollback, the current predicted world (with the mispredictions) is discarded, and the client reverts to the previous clean snapshot. The server change is applied to this snapshot, which then becomes the new baseline for re-applying subsequent local inputs.

2. **Storing Reverse Deltas**: For each local prediction applied to the predicted part of the world, store a corresponding "reverse delta" or "undo operation." To roll back, these reverse deltas are applied in reverse chronological order to the current predicted state, effectively reverting it step-by-step to the desired past authoritative state. Once rolled back, the server's delta is applied, and then local inputs are re-simulated.

After successfully rolling back the state to the point of divergence, the reconciliation module needs to re-apply the local inputs that the player has generated since that point. To facilitate this, the prediction module, as mentioned in the client-side dataflow (see step 4), memorizes these applied inputs in a buffer. The rollback reconciliation process accesses this buffer to retrieve the sequence of inputs that must be re-played by the prediction logic against the corrected state.

4.3.2 Algebraic reconciliation

A significant drawback of rollback reconciliation, as highlighted in Section 3.1.6, is the computational cost of re-simulating predictions. Each time an authoritative server update corrects the client's predicted state, the client might need to re-process multiple ticks of input. This re-simulation effort can increase with the client's network latency, potentially affecting performance, especially in games with many frequently predicted game objects or for players with higher ping times.

The dataflow for applying algebraic reconciliation within our client architecture is depicted in the figure below.

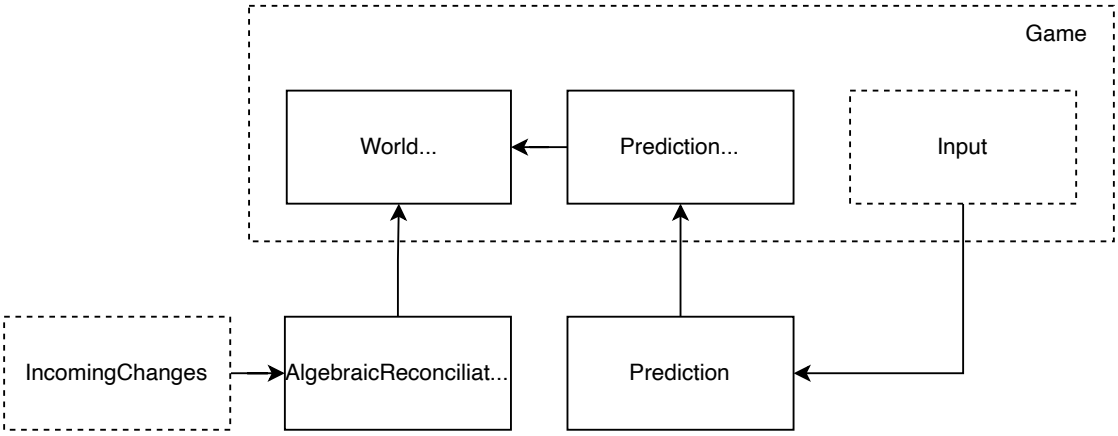

Figure 4.8: Dataflow for algebraic reconciliation on the client. Incoming server deltas are used by the reconciliation module to calculate a direct correction delta.

Our proposed algebraic server reconciliation method, whose theoretical basis was introduced in Section 3.2, offers an alternative designed to avoid this repetitive prediction overhead. This method relies on representing game state changes as "delta states." For algebraic reconciliation to work effectively, these delta states need to possess certain well-defined mathematical properties, specifically those of an Abelian group, as detailed in our theory section. In essence, this means the deltas can be reliably combined, effectively undone, and the sequence of their combination does not alter the final result, much like arithmetic operations on numbers. When these conditions are met, corrections to the

client's predicted state can often be calculated and applied as a single, direct adjustment, rather than requiring a full re-simulation of past inputs.

The implementation of algebraic reconciliation within the client-side networking layer primarily modifies how the reconciliation module processes incoming server changes and updates the client's predicted part of the world. The client's prediction logic, when handling local player inputs, conceptualizes its effects as a series of "predicted deltas." The prediction module tracks the cumulative outcome of these locally generated deltas, effectively maintaining the client's speculative view of the game state built upon the last acknowledged server state.

Concurrently, the client receives authoritative state updates from the server, also in the form of deltas. When such an authoritative server delta arrives, the reconciliation module's task is to compute a "correction delta." This is achieved by comparing the server's authoritative delta for a given time period with the net predicted delta that the client had generated for that same period. The discrepancy between these two deltas isolates the misprediction. This "difference," which is itself a delta, forms the required correction. The ability to reliably calculate this correction delta hinges on the algebraic (Abelian group) properties of the delta states, which allow for a meaningful "subtraction" or "inversion" of the predicted deltas relative to the server's authoritative ones, as outlined in Section 3.2.

Once this single correction delta is determined, it is applied directly to the client's current predicted part of the world. This direct application adjusts the client's state to more accurately reflect the server's view, effectively integrating the server's information without the need to rewind and individually re-process past local inputs. This avoidance of extensive input re-simulation is a key advantage over the rollback method. While the re-simulation of old inputs is bypassed, the client naturally continues to apply new local inputs via the prediction logic to its newly corrected state to maintain immediate responsiveness. It's important to note that for this process to function, the prediction module must maintain a record of the predicted deltas it has generated, as this information is essential for calculating the correction delta. This is distinct from rollback reconciliation's requirement to store the raw inputs themselves for potential re-simulation.

By directly adjusting the predicted state through these well-behaved delta manipulations, the algebraic approach can offer a more computationally efficient reconciliation process. This is particularly beneficial when the game state can be effectively represented by deltas that adhere to the required algebraic properties, as this allows for the direct calculation and application of corrections.

5 Experiments

6 Discussion

7 Future work

Bibliography

[1] “Das Bild der TU Darmstadt.” Accessed: May 1, 2020. [Online]. Available at: https://www.intern.tu-darmstadt.de/media/medien_stabsstelle_km/services/medien_cd/das_bild_der_tu_darmstadt.pdf