Things I Know About Client-Side Prediction

Phoenix Kahlo

1 BACKGROUND

1.1 What is video games? What is time?

We'll establish some basic components of a singleplayer game, focusing on a Minecraft-like game as an example:

- **State**: The game has state in memory which can change due to operations. This state can be sub-divided into smaller pieces of state, such as blocks at particular coordinates, entities with particular UUIDs, what non-instantaneous actions players are currently performing, etc.
- **Rendering**: A graphical representation of the current game state will be repeatedly rendered onto the user's screen. This ideally occurs once for every time the user's monitor refreshes, but may happen slower if the user's computer cannot render that fast.
- **Inputs**: Input actions performed by the user, such as pressing keyboard keys or using the mouse, may translate to operations which affect the game state. For example, right-clicking may place a block; tapping the space bar may trigger the avatar to jump; and depressing the W key may place the avatar into a state of trying to walk forward.
- **Updates**: Some pieces of game state will be subjected to operations that occur not as the result of player input but as a result of the passage of time, to give the impression of changing continuously over time. For example, an object subject to physics may be updated every 60th of a second by simulating one 60th of a second of physics for it.

From this, we can model our singleplayer game as an event loop in which state is repeatedly updated, in response to both user inputs and the passage of time, and then rendered.

There are actually many subtle design decisions in how one programs this outer loop. However, we can describe some ideal properties:

(1) Ideally, a render operation occurs every time the user's monitor refreshes.
(2) Ideally, an update operation occurs in between each render operation, and simulates the passage of the amount of time that elapsed in the real world since the last update operation.
(3) Ideally, processing some input happens shortly after the user performed the input.

From this, we can start to build up a notion of these input, output (rendering), and update operations in terms of their relationship with real world time.

In the logical world of the computer, we can envision the game as a sequential chain of game states, in which there is a state, followed by an operation that alters that state, followed by the subsequent state, and then the next operation, etc.

In the real world, wall-clock time passes continuously. The user performs input actions such as pressing keys at particular instants in real-world time, and a rendering operation causes its version of the game state to be displayed to the user at a particular instant in real-world time.

There is an implicit notion that a game state at some point in the chain *reflects the consequences* of all inputs that were processed earlier in the chain. By extension, there is a notion that the physical image resulting from a render operation transitively reflects the consequences of all input operations earlier in the chain.

Let us declare a new ideal property in terms of these notions of things reflecting the consequences of other things:

- The consequences of some input should be rendered shortly after the input.

Clearly, ideal properties 1 and 3 are natural consequences of this.

There is a similar concept at play here in terms of updates. We say that an update operation simulates the passage of a certain amount of time. If an update operation simulates $\frac{1}{60}s$ of physics, there is a notion that the subsequent game state *reflects* the passage of $\frac{1}{60}s$ more time than the previous state. This is a cumulative property; if two states in the chain have 5 different $\frac{1}{60}s$ update operations between them, then the second state reflects the passage of $\frac{5}{60}s$ more time than the first.

As such, we can declare this final intuition:

- A rendered image, in contrast to a previously rendered image, should reflect the passage of an additional amount of time equal to the amount of real-world time elapsed between those renders.

Clearly, ideal property 2 is a natural consequence of this.

1.2 Wouldn't it be nice...

Welp, now to take this beautiful architecture and streeeetch it out into a distributed system. First, some basic definitional things:

- There are multiple clients.
- Each client must perform rendering operations locally.
- Input events may originate from any client.

We can generalize our previous principle here and say: Ideally, the consequences of some input should be rendered *on all clients* shortly after the input. Of course, we now have some additional complications.

Consider a peer-to-peer configuration where there are 2 or more clients all connected to each other directly, with no server. We can fantasize about something that would be nice: Wouldn't it be nice if all operations were commutative?

If you could take any bag of input processing operations and update operations, and apply them all to the same starting state, and always get the same ending state no matter what order you applied them in, that would be very convenient. That would allow for the following algorithm:

- When an input occurs physically, the local client immediately applies it locally, and also transmits it to all peers.
- When a client receives an input from a peer, it applies it locally.
- Clients do update and rendering operations as they would normally.

This is, of course, a description of how CRDTs work, and does achieve the optimal input-to-render latency possible for all pairs of inputs and clients. However, the defining property of CRDTs is that they are painstakingly designed to be data types for which all operations commute–this is not true in the general case of video game logic.

There are many cases where the order of operations matters:

- The order between player 1 attempting to walk through a door and player 2 closing the door may determine whether player 1 successfully walks through the door or is blocked.
- The order between a player attempting to jump and that player experiencing an update may determine whether the player successfully jumps or whether it slides off of solid ground before being able to do so.

Later, we'll discuss how to exploit cases where operations *do* commute–but for now, we'll assume the worst case scenario that nothing commutes, and explore how to cope with that.

1.3 Rollback

So, since operations don't commute, it's not enough for the different clients to converge on a chain with the same set of operations in it; they must converge on a chain with the same *sequence* of operations in it. Each client locally applies its local inputs immediately after they occur, which is optimal and thus correct as per the ideal of minimizing input-to-render latency. The problem, then, must be that each client is applying its peers' inputs too late in the chain.

So: have the system track the wall-clock timestamp of each operation, and make it so when a client receives an input from a peer, if the timestamp of the input is earlier than the timestamp of the last operation in its local chain, it "buries" it earlier in the chain, somehow applying it retroactively "underneath" the operations which it did afterwards. This is non-trivial.

Probably, this requires the client to:

(1) Revert the game state to the last version before the operation should have occurred.
(2) Apply the newly introduced operation.
(3) Re-apply the sequence of operations afterwards.

As for how the client is able to revert its game state to a previous version, the main ways to do this are:

- **Copy-on-Write**: The client can simply keep a complete copy of each version of the game state. This is simple, but has obvious CPU and RAM costs. However, it can work for kinds of games where the game state is only a few kilobytes.
- **Undo Log**: In this approach, whenever the client does an operation on its state, it also generates some "undo" record, which contains whatever information the client needs to undo that exact change, and saves it for later. Thus, to revert to a previous state in the chain, the client goes through all the undo records corresponding to operations after that state, and un-does each one in reverse order.

For most operations, the undo records can simply contain a copy of the previous version of all sub-pieces of state the operation affects–thus becoming effectively a more granular version of copy-on-write. The undo log approach is more compatible with large amounts of game state, however, it becomes a cross-cutting piece of complexity that infects all game logic.

1.4 Serialization (Server/Host)

There's two completely unrelated definitions of "serialization" in computer science: the commonly used, Comp Sci 200, bad definition, which is "encoding an object into bytes," and the less commonly used, Comp Sci 700 definition, which is "making things happen one after another."

Let's say you don't want to do rollback. There are many valid reasons for that. To avoid ever having to roll back an operation, each client must ensure that any operation it performs really is the next operation in what will ultimately be the agreed upon correct order of operations. In other words, each client must *avoid* performing an operation until it is somehow sure there *won't* be any not-yet-known operations before it.

Achieving this involves picking some node in the system to be a central choke-point through which all events must flow. For example, one of the clients, or perhaps an entirely new node known as the "server," might be designated as the "host," and the system could have the following algorithm:

- When an input occurs physically, the local client transmits it to the host.
- When the host receives an input from a client, it transmits it back to all clients in a message which conveys that it's now "committed."
- When a client receives a message from the host conveying that an input is now committed, it applies it locally.

Unfortunately, this is not compatible with achieving optimal input-to-render latency in all cases. Each client will experience a delay equal to its network round trip latency with the host. If two clients are physically 30 light milliseconds apart, there is no physically possible way to avoid at least one client having a delay of at least 30 light milliseconds. There is no way to cheat this.

1.5 More realistic client/server architecture

Nevertheless, there are a large number of practical engineering advantages to this client/server architecture, which is why the vast majority of multiplayer games in the wild are far closer to this than to a peer-to-peer architecture. Games in the real world are also likely to diverge from this abstract model in some additional ways:

- The server and client likely have slightly different versions of game state–even beyond the sense in which operations on the client may lag behind the server. The server likely has some pieces of server-only state, such as for NPC AIs. The client likely has some pieces of client-only state, such as handles to resources for rendering graphically.
- Rather than inputs being transmitted from clients to the server and then back to clients without actually modifying them, the system likely defines two different types of message: "**actions**", which flow from clients to the server, and "**edits**," which flow from the server to clients.

As such, the more realistic story of an input travelling through a client/server architecture sounds more like this:

(1) *Input*: The user right-clicks. The client reads the client-state and determines that the user is trying to place a block at some coordinates. It sends the server an action: "place ${BLOCK} at ${XYZ}."
(2) *Action*: The server receives this. It reads the server-state and confirms that 1. no block currently exists at ${XYZ} 2. the user has a ${BLOCK} to place. It sets server-state block ${XYZ} to ${BLOCK}, and it sends all clients an edit: "set block ${XYZ} to ${BLOCK}."
(3) *Edit*: The client receives this. It sets client-state block ${XYZ} to ${BLOCK}.

One large advantage to splitting the game logic up like this is that it relieves most of it from having to be deterministic. When the client processes an input, or the server processes an action, they can: read local-only state, use RNGs, do input, do non-deterministic parallelism, have non-determinism bugs, run client-only or server-only mods, tolerate floating point anomalies, etc. Only the edit operation, and corresponding server-state mutations, are expected to be fully deterministic and reproducible, and the logic for them is usually very simple, akin to operations on a key/value store.

Another advantage to a client/server architecture is avoiding cheating. Anti-cheating in video games is less like the mathematical guarantees of cryptosystems and more like fairness at the Olympics, or perhaps two species of fish evolving to eat each other. Still, though, locking the client down to just sending actions it conceivably could do, and letting the server determine the consequences of them, is less radically abusable than a protocol where any client can broadcast edits throughout the system.

Finally, there are the advantages of a server being a reliable and fixed point. The server is expected to stay running at a stable WAN-accessible IP address. When a client joins the game, it initializes its client-state by downloading the current state from the server–thus allowing clients to come and go from a continuously running game. The entity administrating the game can unilaterally manage the server. More decentralized architectures must come up with more complex contrivances for all of these things.

2 CLIENT-SIDE PREDICTION

2.1 What is client-side prediction?

For these reasons, most video games are stuck to the model that the server-state is the "canonical" state, and that a thing happening is defined as it happening on the server. Thus, when a user tries to perform an action, there is an up-stream network delay before it actually happens, followed by a down-stream network delay before the user observes the effects of it.

For some actions, this delay is acceptable. However, for things like walking around, or placing and breaking blocks, this may seriously degrade the experience. As such, we need techniques for the client to *predict and immediately display the expected consequences of at least some actions, while still staying synchronized with the server if those predictions are incorrect.* Moreover, this must be done in a way that:

- Is subjectively aesthetically unobtrusive.
- Performs efficiently.
- Avoids infecting all game logic with a need to be perfectly reproducible.

2.2 A basic prediction algorithm

Placing and breaking blocks is a simpler case for client-side prediction than walking around, because the player's body is continuously being subjected to physics updates, whereas blocks usually are not (more on that later).

First off, we'll assume our network protocol includes acks, such that when the server receives and processes an action from a client, in addition to transmitting the consequent edits to all clients, it also acknowledges to the original client that it processed that message. Also, we'll hand-wave away the possibility of the client being in a partially received state; the client receives all the edits resulting from some action plus acking of that action atomically.

From these building blocks we can implement a basic client-side prediction algorithm. Actions subject to prediction will need 2 additional procedures implemented:

- Actions subject to prediction will need a client-side "predict consequences" operation. This operation reads and mutates the client-state to emulate the consequences of the action being processed on the server.
- When doing so, it must generate and store some "undo" record(s) which contain whatever information is needed to undo those exact changes, so that an "undo" operation can later be performed to restore the original client-state.

This is very similar to the rollback technique discussed earlier. However, it is done in a way that limits the scope of what logic needs to be made reproducible. Only actions that the developer wishes to make client-side-predictable need to have a "predict consequences" procedure implemented, and only mutations made by "predict consequences" procedures need to have rollback functionality implemented.

Given those operations, the algorithm looks like this:

- When the client sends the server a predictable action, the client also pushes it to the back of a queue of actions.
- When the server acks one of those predictable actions, the client pops it from the front of the queue of actions.
- Before rendering, the client ensures that all actions in the queue have their predictions applied, by running the "predict consequences" operation on each one front-to-back.
- Before applying edits received from the server, the client ensures that any applied predictions are reverted, by running the "undo" operation on all undo records in reverse order.

To illustrate how this handles a successful prediction, let's tell a story where block ${XYZ} starts as AIR, then Alice places WATER at ${XYZ}, then 3ms later Bob places STONE at that same ${XYZ} (note: placing STONE where there is WATER overwrites the WATER with STONE), and the latency is 10ms:

(1) **0ms**: Alice sends "place WATER at ${XYZ}" to the server, and stores it in her prediction queue. Alice locally sets block ${XYZ} to WATER, and locally stores the undo record "set block ${XYZ} to AIR."
(2) **3ms**: Bob sends "place STONE at ${XYZ}" to the server, and stores it in his prediction queue. Bob locally sets block ${XYZ} to STONE, and locally stores the undo record "set block ${XYZ} to AIR."

(3) **10ms**: Server receives Alice's action, sets block ${XYZ} to WATER, transmits "set block ${XYZ} to WATER" to both clients, and transmits an ack to Alice.
(4) **13ms**: Server receives Bob's action, sets block ${XYZ} to STONE, transmits "set block ${XYZ} to STONE" to both clients, and transmits an ack to Bob.
(5) **20ms (Alice)**: Alice receives server's first edit, and ack.
  (a) **Revert predictions**: Alice applies her undo record, setting block ${XYZ} to AIR, and discards it.
  (b) **Prediction acked**: Alice discards her prediction.
  (c) **Apply edit**: Alice sets block ${XYZ} to WATER.
(6) **20ms (Bob)**: Bob receives server's first edit.
  (a) **Revert predictions**: Bob applies his undo record, setting block ${XYZ} to AIR, and discards it.
  (b) **Apply edit**: Bob sets block ${XYZ} to WATER.
  (c) **Predict consequences**: Since Bob's action is still un-acked, he locally sets block ${XYZ} to STONE, and locally stores the undo record "set block ${XYZ} to WATER."
(7) **23ms (Alice)**: Alice receives server's second edit. With no predictions active, she simply sets block ${XYZ} to STONE.
(8) **23ms (Bob)**: Bob receives server's second edit, and ack.
  (a) **Revert predictions**: Bob applies his undo record, setting block ${XYZ} to WATER, and discards it.
  (b) **Prediction acked**: Bob discards his prediction.
  (c) **Apply edit**: Bob sets block ${XYZ} to STONE.

Now, consider how this would have gone differently if instead of placing WATER at ${XYZ}, Alice had placed DIRT. This would have caused Bob's attempt to place STONE at ${XYZ} to fail, as, unlike placing a block where there is WATER, which simply overwrites the WATER, trying to place a block where there is some other block such as DIRT cannot be done. In terms of how the algorithm would handle that:

- At 13ms, the server would not set block ${XYZ} to STONE, because there would already be DIRT there. However, it still would send Bob an ack.
- At 20ms, Bob's action would still be un-acked, so he would still predict its consequences, but instead of predicting that block ${XYZ} would be set to STONE, he would correctly predict that it would have no consequences, because in his client-state, there would already be DIRT there.

Moreover, even if the predicted consequences of an action are completely wrong, the client-state will converge back to the correct state after the action is acked. The main risk of desynchronization here is if the client's logic for undoing predictive edits doesn't work right. However, the complexity here is manageable since these undo records can usually simply contain a copy of the previous version of edited sub-pieces of state, rather than more complex logic.

2.3 Predicting updates

We can extend this algorithm to predict updates as well. The server must perform updates frequently. When it does so, we'll have it transmit to all clients the current wall-clock timestamp. This plays a role similar to acks–and similarly to acks, we'll hand-wave to say that the client receives all edits resulting from some update plus that update's corresponding timestamp atomically.

We'll give the client a "predict update" procedure. It takes a delta time as input, and reads and mutates the client-state to emulate the consequences of that long of an update being performed on the server. When doing so, it generates undo records to undo these mutations, just like the predict consequences operation.

Before rendering, in addition to the client ensuring it has all predicted actions applied, it also ensures it has predicted updates applied equal to the duration from the wall-clock timestamp of its received state to the wall-clock timestamp of the current moment at which it is rendering.

Similarly to how not every action needs a "predict consequences" procedure implemented, the client's "predict update" procedure does not need to predict the entire set of likely consequences of a server-side update. In particular, it may predict only the physics of some sub-pieces of state for which prediction is particularly important, such as players' bodies.

2.4 The up-stream latency problem

What would happen if we tried to apply this prediction algorithm to the players walking around? Unfortunately, there will be some additional complications.

Consider a simplified version of physics and walking, which is less similar to walking and more similar to operating a jetpack in outer space.

- Each player has a *pos* vector and a *vel* vector.
- The update operation is: $pos \leftarrow pos + dt \cdot vel$.
- The permitted action is "accelerate by ${DVEL}", which is performed as: $vel \leftarrow vel + \${DVEL}$.

Doing an accelerate action doesn't cause any visible effects on its own, as it only changes the velocity, not the position. It isn't until updates occur after the accelerate action that the effect is apparent. Here, the lack of commutativity between the accelerate action and the update operation is a fundamental part of what they are–the effect of an accelerate action plus 100ms of updating wholly depends on whether the accelerate action happens before the update, after the update, or in between two 50ms updates.

The client-side prediction algorithm we established amounts to always rendering the state last received from the server with active predictions applied on top of it. It's worth noting that *this isn't actually a very "accurate" prediction.* When the client sends the server a "place ${BLOCK} at ${XYZ}" action, it won't be processed until after an up-stream network delay. If the client were trying to predict the current server-state as accurately as possible, it would measure the network RTT and predict that the action would occur after 0.5 RTT. That would be wrong, though, because accuracy is the wrong goal.

We treat the server-state as the canonical version of state for practical reasons, but the ideal behavior is still to render the consequences of some input shortly after the input. When the user right-clicks an AIR block at ${XYZ} with a STONE block, the consequence is probably that block ${XYZ} is set to STONE. By predicting that consequence immediately, the client achieves the ideal of low input-to-render latency. It is the server-state which doesn't reflect the consequences until after a delay.

Let us ask, then, what are the consequences of an "accelerate by ${DVEL}" action? The consequences are that, for state 3ms of

wall-clock time after the action occurred *pos* is 3ms · ${*DVEL*} beyond what it would otherwise be, and for state 6ms of wall-clock time after the action occurred *pos* is 6ms · ${*DVEL*} beyond what it would otherwise be, etc. Herein lies the problem: due to the upstream network delay, the client and the server will disagree on what instant of wall-clock time the action occurred–and, unlike in the case of placing a block, that means they will disagree on its consequences.

2.5 The "one step ahead" technique

Let's say the network latency is a constant 10ms in each direction, the player starts with a position and velocity of 0, and at 0ms the client does an "accelerate by ${DVEL}" action. This means from 0ms to 20ms the client will be rendering server-state from before the server processed the action, with two predictions applied over it: 10ms of predicted updates, and the predicted "accelerate by ${DVEL}" action. We must ask: how does the client decide whether to apply the predicted action before the 10ms predicted update, after the 10ms predicted update, or in between two predicted updates totalling to 10ms?

(1) To always predict the action before the update would be patently absurd. That would mean that at [1ms, 2ms, 3ms], the client's received state would have been sent by the server at [-9ms, -8ms, -7ms]–before the client even sent the action–yet the client would be predicting that the server would process the action as the very next operation on the server before any further updates.

(2) Conversely, to always predict the action after the update would be completely ineffacious. At [17ms, 18ms, 19ms], the client's received state would have been sent by the server at [7ms, 8ms, 9ms], and the client would be predicting that the server would process the action at [17ms, 18ms, 19ms]. In addition to this being unrealistically *late*, it would produce no visible effects–as an accelerate action only produces visible consequences when updates are applied *after*.

(3) One might be tempted to make the client always predict that the server would process the action at 0ms. So, at 1ms, the client's received state would have been sent by the server at -9ms, and the client would predict 9ms of update, then the action, then 1ms of update. At 9ms, the client's received state would have been sent by the server at -1ms, and the client would predict 1ms of update, then the action, then 1ms of update. However, at, say, 13ms, the client's received state would have been sent by the server at 3ms, but the action would still be un-acked. At that point, the client would like to predict -3ms of update, then the action, then 13ms of update–but predicting negative duration of update isn't an operation we're assuming it can do. So it's unclear what would happen there. Perhaps from 10ms to 20ms it would simply predict the action followed by 10ms of update.

(4) A modified version of that approach would be to estimate the network RTT and always the predict that the server would process the action 0.5-RTT after the client sent it–in this case, 10ms. This means that, from 0ms to 10ms, the client would predict 10ms of update, but forego predicting the action at all. At 11ms, the client would predict 9ms of update, then the action, then 1ms of update. At 19ms, the client would predict 1ms of update, then the action, then 9ms of update. This approach is the most accurate so far at predicting the current server-state, but, as established, accuracy is the wrong goal, and this approach fails to fully minimize input-to-render latency.

(5) That "accurate" approach *is* looking like the closest to what we want so far, at least in shape. The main problem is that everything is just happening 10ms too late. So... what if... instead of predicting what state the server has now... we predicted what state the server would have *10ms in the future*?

Consider this algorithm: the client estimates the network RTT and always predicts that the server would process the action 0.5-RTT after the client sent it. When the client renders, in addition to predicting all un-acked actions, it predicts updates from the wall-clock timestamp of the received state to 0.5-RTT *after* the wall-clock timestamp of the current moment at which it is rendering.

At 1ms, the client's received state was sent at -9ms, and the client predicts 19ms of updates, then the action, then 1ms of updates. At 10ms, the client's received state was sent at 0ms, and the client predicts 10ms of updates, then the action, then 10ms of updates. At 19ms, the client's received state was sent at 9ms, and the client predicts 1ms of updates, then the action, then 19ms of updates.

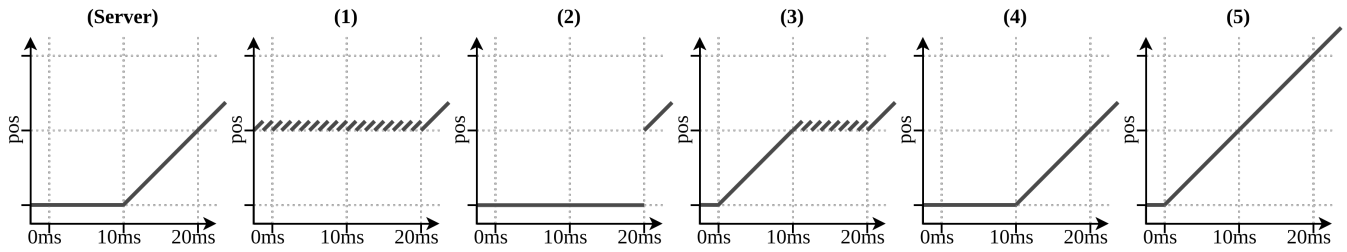
Although this approach might seem deranged, it actually checks off a lot of our boxes:

- **Input-to-render latency**: Renders begin to reflect the consequences of some input immediately after the input.
- **Passage of time**: Successive renders reflect the consequences of the passage of the amount of time between those renders.
- **Engineering constraints**: The client only has to roll back its predictions, and the server doesn't have to roll back anything.

However, this approach will still need a bit more refinement before it's quite ready to implement.

2.6 Philosophizing why that works

You might not be content to simply accept that something like this works just because it seemed to work in this example. This section



will try to conceptualize why doing this isn't a problem–skip it if you don't care.

When articulating our ideals of how the progression of game states corresponds to the progression of real-world time, it's worth noting that we never rely on absolute concepts of time, but only concepts of a relative amount of time passing when comparing two things. A game-state doesn't inherently have any signature absolute instant in time it represents, and a wall-clock timestamp has no meaning when considered in isolation. Rather, one game-state relative to a previous game-state can reflect the consequences of a certain duration of time being simulated since the former game-state, and one wall-clock timestamp can be said to be a certain duration after a previous wall-clock timestamp. The important relationship to uphold between game-states and real-world time is not to ensure that a render at a particular wall-clock instant renders a specific game-state, but to ensure that a render at a particular wall-clock instant reflects the consequences of the correct amount of time passing relative to previous renders and previous physical inputs.

We see the progression of game-states on the server as the canonical progression of game-state, and because of this it is tempting to see the game-state in the server's memory at a particular moment in time as canonically corresponding to that moment in time. However, this is a confusion of concepts. While it's true that physical computations occur over certain real world intervals of time, all that ultimately matters is when these computations manifest physically by physically displaying a rendered image.

As such, it may be more helpful to think of this algorithm not as the server holding the canonical present version of state and the client as rendering a predicted future version of state, but rather, as the server holding a canonical *past* version of state and the client as rendering the predicted *present* version of state. That is:

- The wall-clock moment represented by some point in the game-state chain is by definition the moment it is rendered by the client.
- The server adds updates to the game-state chain to keep it updated to 0.5-RTT in the past.
- Since it takes 0.5-RTT for that state to reach the client, the client's received state is always 1-RTT behind the present, and so the client must predict 1-RTT of updates.
- When the client performs an action, it wishes for the action to occur at the present moment. It sends it to the server, and it arrives 0.5-RTT later, at which point the game-state in the server's memory represents the moment the action is supposed to occur–thus, the server can simply apply it upon receiving it.

What, then, *is* the sense in which the progression of game-states on the server is the "canonical progression of game-state?" Events that occur on the server are canonical in the sense that they are *committed*. When an event occurs on the server, it is forever "locked in" as the very next event that occurs after all previous events; no event will ever be buried "underneath it." This allows the server to perform these operations in a way that renders it inherently or practically unable to undo them–it will never have to roll back to a previous state.

Recall something that was said in a previous section: "To avoid ever having to roll back an operation, each client must ensure that any operation it performs really is the next operation in what will ultimately be the agreed upon correct order of operations. In other words, each client must *avoid* performing an operation until it is somehow sure there *won't* be any not-yet-known operations before it." Given that the server wishes to avoid rollbacks, this explains why the server-state must lag in the past by the up-stream network delay: The ultimately correct order of operations must entail actions occurring at the point representing the moment the action was sent. Thus, at any given moment, the server can only be sure of the correct order of operations up to one up-stream delay in the past–since the correct order of operations after that could be many possible permutations of updates and actions depending on messages-in-transit the server does not yet have access to. This is why the server must avoid performing update operations until it is sure it has received all messages sent by a certain point.

2.7 Variable network delay

The algorithm described above is extremely close to what we'll go with. The main problem left to address is the assumption that network delay is constant. In truth, network delay may fluctuate noisily due to both external conditions and internal queueing delays. As such, we'll devise an algorithm that makes no reference to estimated RTT whatsoever.

We'll create an abstraction called a "time stream." The shared game world will contain multiple time streams: a single time stream for the server, plus one time stream for each client currently playing the game. For each time stream, both the server and each client will store a wall-clock timestamp, the moment that time stream has progressed to.

Rather than having a single update operation that simulates time for all game-state uniformly, we'll say that each sub-piece of state subject to updates belongs to a single time stream. Most sub-pieces of state will belong to the server time stream, but each client's avatar will belong to that client's time stream. We'll have to split the update procedure up into one that simulates a certain duration of server-time, and one that simulates a certain duration of a certain client's client-time. We'll also have to split up the client-side "predict update" procedure in the same way.

For any time stream, the server has the ability to advance that time stream forward to a given target wall-clock timestamp. When the server does so it:

(1) Performs that time stream's update operation for the duration from the time stream's timestamp to the target.
(2) Sets the time stream's timestamp to the target timestamp.
(3) Broadcasts to all clients a "server-time advanced to ${TS}" or "client-time of ${CLIENT} advanced to ${TS}" message.

From those building blocks, this will be our algorithm:

- Each client must frequently (eg. as often as it renders) send the server a "declare time ${TS}" message, where ${TS} is the current wall-clock timestamp. When the client does so, it also stores ${TS} locally as its "declared" timestamp.
- When the server receives a "declare time ${TS}" message from a client, it advances that client's time stream to ${TS}.

- Frequently, the server advances the server time stream to the current wall-clock timestamp.
- When a client receives a "(time stream) advanced to ${TS}" message, it stores ${TS} locally as the timestamp of that time stream's received state. As usual, we're hand-waving to assume the client receives that message plus all corresponding edits atomically.
- Before rendering, the client ensures that it has the following predictions applied:
  - For each time stream, predicted updates to that time stream for the duration from that time stream's received timestamp to the client's declared timestamp.
  - For each un-acked action, the predicted consequences of that action, inserted between predicted updates to the client's own time stream at the point representing what the client's declared timestamp was upon the client sending that action.

This algorithm ensures that the server-state of a client's avatar always lags behind the present by that client's current up-stream network delay, and all rendered state maintains a sum of received and predicted updates that matches the progression of real-world time, even if up-stream and down-stream latency are fluctuating wildly.

Unfortunately, this algorithm loses the ability to have clients immediately predict actions on sub-pieces of state other than their own avatar. For example, a "shove" action that lets a client apply acceleration to an NPC or another player could not be applied immediately. However, actions such as placing and breaking blocks can still be applied immediately, as blocks, not usually being subject to updates that cause the consequences of actions to amplify over time, are effectively not really subject to any time stream.

2.8 Hole-punching

A super cool bonus of this architecture is that it is actually extremely conducive to TCP/UDP hole-punching, aka. NAT traversal, where the server is able to convince / trick two clients' routers into letting the clients talk to each other directly, with lower latency than having information be relayed through the server.

To exploit this, we can have pairs of hole-punched clients send their actions to each other in addition to the server. We'll also need generalize our acking system so the server informs a client when it has processed one of its hole-punched peer's actions, in addition to its own.

This means that, at a given moment in time, in addition to a client having a prediction queue of its own un-acked actions, it may also have a queue of some of its peer's un-acked actions. As such, when the client predicts updates to that peer's time stream, it can interject the predicted consequences of that peer's actions, just as it would its own.

A ping of 40ms is considered basically fine, but this hole punching technique could allow a client to observe the consequential actions of a peer after only 20ms, which is considerably better. Furthermore, this would allow two people playing in the same physical building to observe the consequences of each others' actions almost immediately, even if the actual server is far away.

3 MISCELLANEA

3.1 Discrete and continuous updates

We have been assuming that update operations can be split into many smaller updates and still have the same cumulative effect so long as they sum up to the same delta time. However, it is likely overly burdensome to uphold this invariant for all update logic.

For example, acceleration on objects due to gravity is usually not actually applied continuously, but rather applied as an instantaneous change in velocity once each time step. This lack of smoothness is not noticeable on its own, however, if the server and client are applying acceleration at different frequencies than each other, it actually causes noticeable desynchronization.

To solve this, we implement two different update procedures:

- A "continuous update" procedure, which accepts a delta time and *is* required to have the same effect regardless of how it's broken up into smaller time steps.
- A "discrete update" procedure, which is guaranteed to run once every $\frac{1}{20}s$.

The continuous update moves objects via swept collision, drives animations, and does whatever else actually needs to change slightly between each rendered frame for aesthetic reasons. The discrete update does acceleration, as well as anything else that doesn't have to go in the continuous update.

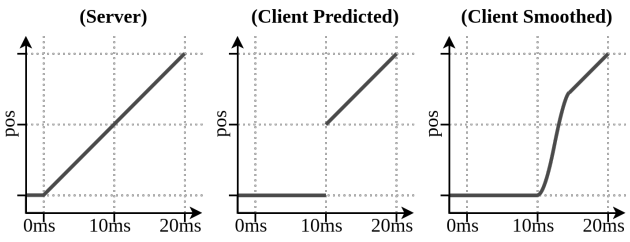
The client and server agree on a fixed schedule for discrete updates by designating some t_0, such that a discrete update occurs at t_0 and $t_0 + 50ms$ and $t_0 + 100ms$ and $t_0 + 150ms$, etc. Continuous updating happening between them.

Thus, when an update operation is invoked to advance some time stream from its current wall-clock timestamp to a target timestamp, it does so by performing the appropriate sequence of continuous and discrete updates in accordance with the schedule defined by t_0

3.2 Smoothing

If a client is receiving state about an entity X seconds in the past, and then predicting updates up to the present, and it receives an unexpected change to that entity's velocity in the past, the predicted present position will "snap" to a new position.

Although this achieves the ideal of the client reflecting the expected consequences of all events as soon as it becomes aware of those events, the appearance of teleporting is aesthetically jarring enough that it's worth smoothing over. To achieve this, we must add a new layer of abstraction *on top of* the prediction layer of abstraction that visually smooths over these unexpected shifts.



4 CONCLUSION

Go forth; all the knowledge oyu need is within you.