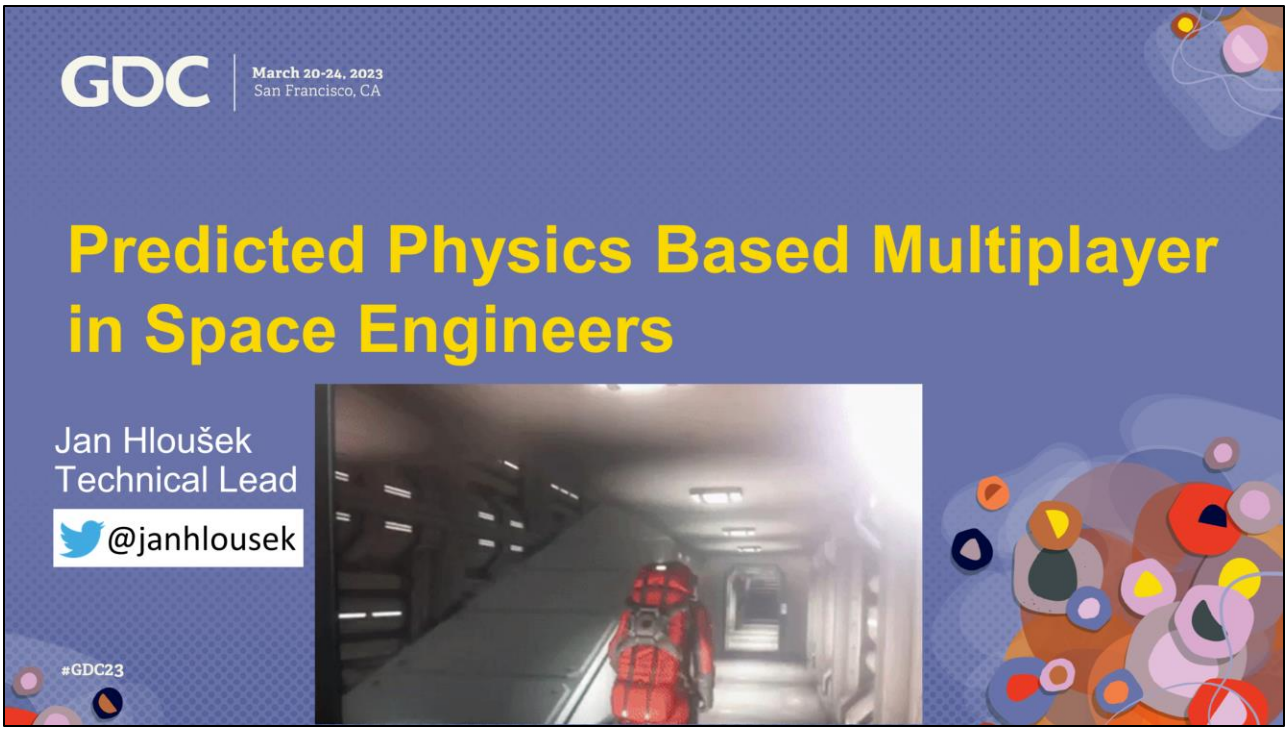
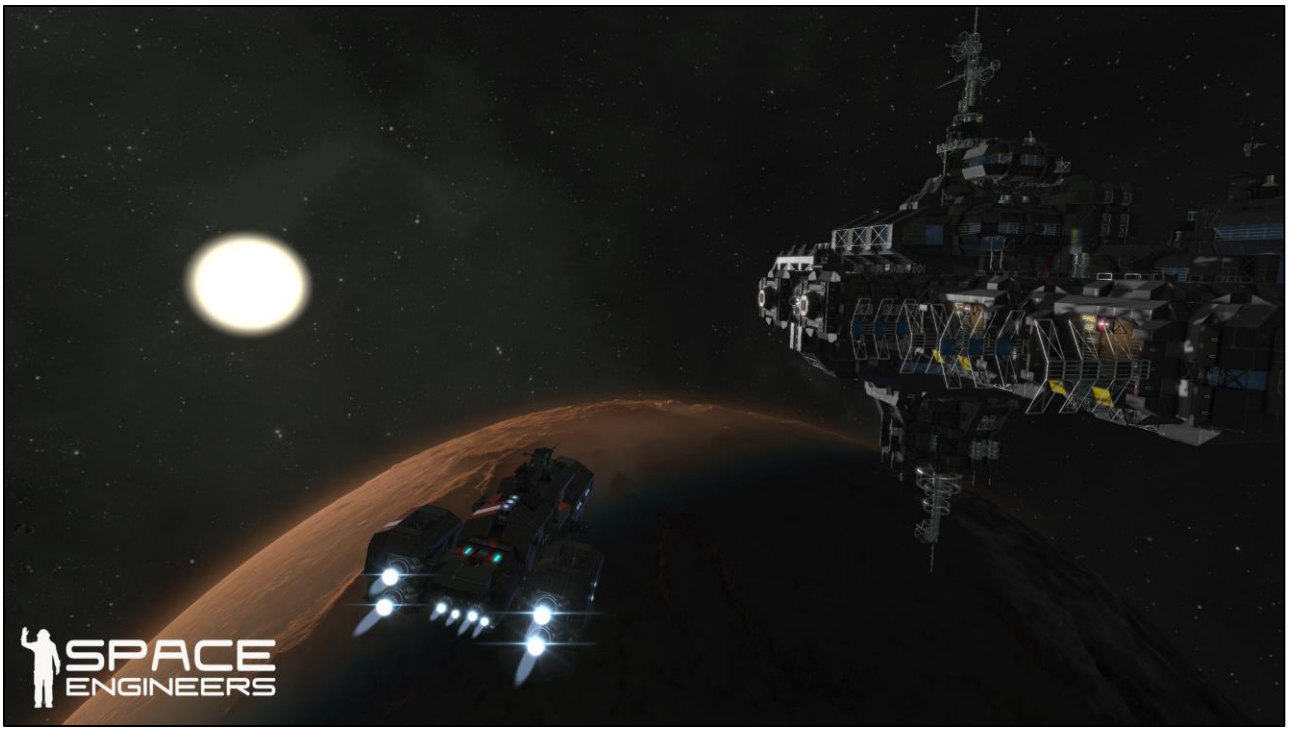


I am Jan Hlousek, technical lead at Keen Software House.

Currently, my team is developing the next generation of our inhouse engine.

Today, I will talk about improvements of entity prediction techniques in physics-based games like our Space Engineers.

Space Engineers is a creative sandbox game with solar system scale simulation.



Players can build static bases or flying and riding vehicles.

Players can do so in the multiplayer environment as they either cooperate to achieve the shared goals…

Or compete each other alone or in factions.

Environments are fully dynamic.

Planets and asteroids are made out of voxels and players can alter them via drilling or voxel hand tool.

Our design principle is that anything in the world can be constructed, deconstructed or destroyed.

Space Engineers in numbers

- ~10 years on Steam
- Xbox & Playstation
- 4M+ copies sold
- 500k mods on Steam & mod.io
- Up to 100 players per server
- 10k active entities per server

The game is 10 years on steam, 3 years on Xbox and coming soon to Playstation.

It has sold over 4 million copies and still doing well.

It is heavily modded by design. There are over 500 thousand mods and other user creations on Steam Workshop and mod.io

There are servers with 16 or 32 players, but biggest servers have up to hundred players.

There are tens of thousand individual entities called grids per server, which are composed out of thousands of blocks.

Space Engineers custom engine

- C# .net 4.7
- DX11
- Deferred PBR renderer
- Voxels
- Havok

Game is built on top of the inhouse developed engine called VRAGE.

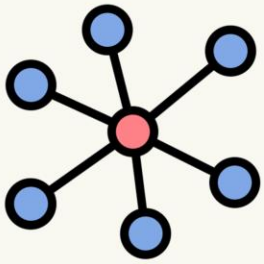
We are using C# for 99% of the code, including engine and renderer.

PC and Xbox platforms are using DirectX 11 with deferred PBR renderer and dynamic global illumination.

We have developed our custom Voxel technology and we are using Havok 2012 for physics.

Space Engineers multiplayer

- Authoritative server
- Star network topology
- Steam networking for PC only
- EOS networking for crossplay
- UDP (with acks)
- Locally hosted MP or DS
- Replication
- State updates



Multiplayer networking uses an authoritative server, where clients are connected via P2P in a star network topology, so clients never communicate directly with each other.

We are using steam for PC only networking and Epic Online Services for crossplay with consoles.

Our protocol is based on UDP with custom acknowledgements for some types of packets.

The servers are either locally hosted - which means players can set game discovery to enabled and others can connect to them
or players can spawn a dedicated server.

Server holds the complete world state and portions of it are replicated to clients.

Delta state updates are regularly sent from server to clients. Clients send only controls updates like button presses to the server.

Space Engineers multiplayer

Players can build any physical contraption, be it a ship, rover or hybrid, connecting it with rotors and pistons and then control it in multiplayer scenarios.

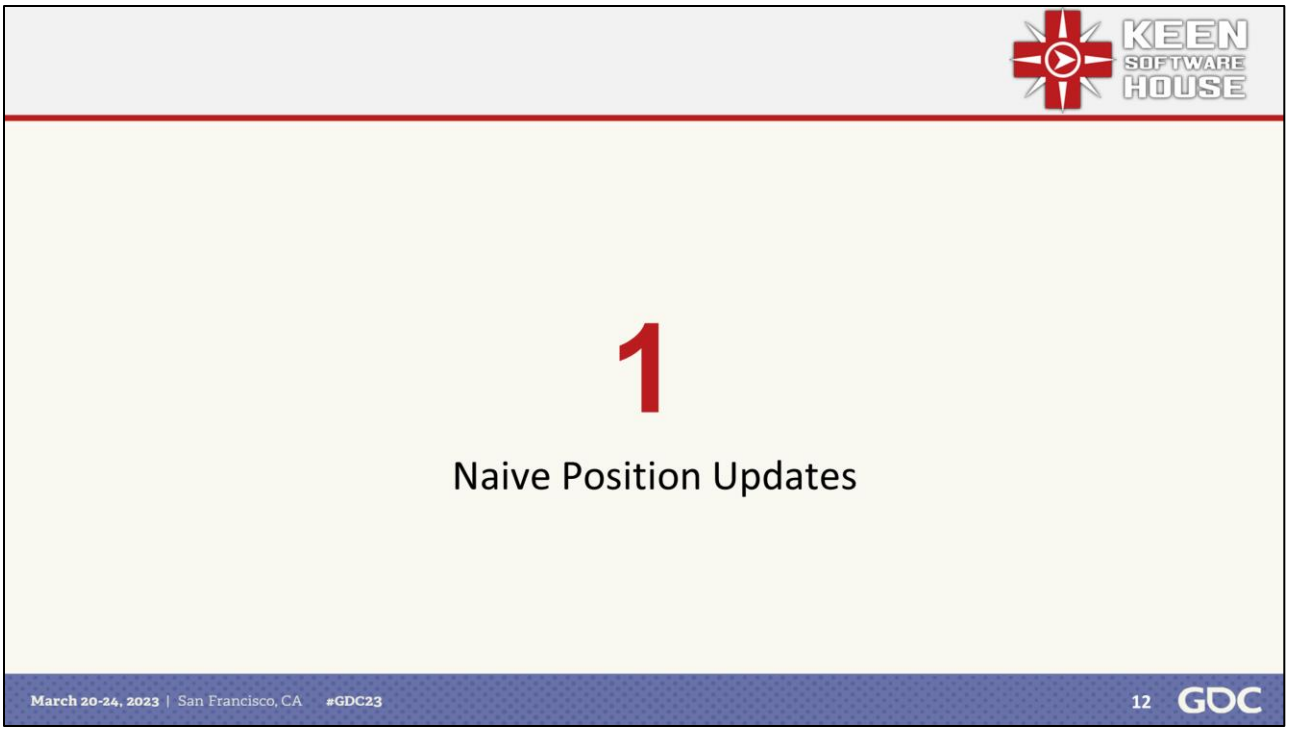
Agenda

1. Naive Position Updates
2. Relative Position Updates
3. Lag
4. Basic Prediction
5. Prediction Correction
6. Time Paradox
7. Relative Prediction
8. Client Physics Setup
9. Problems
10. Summary

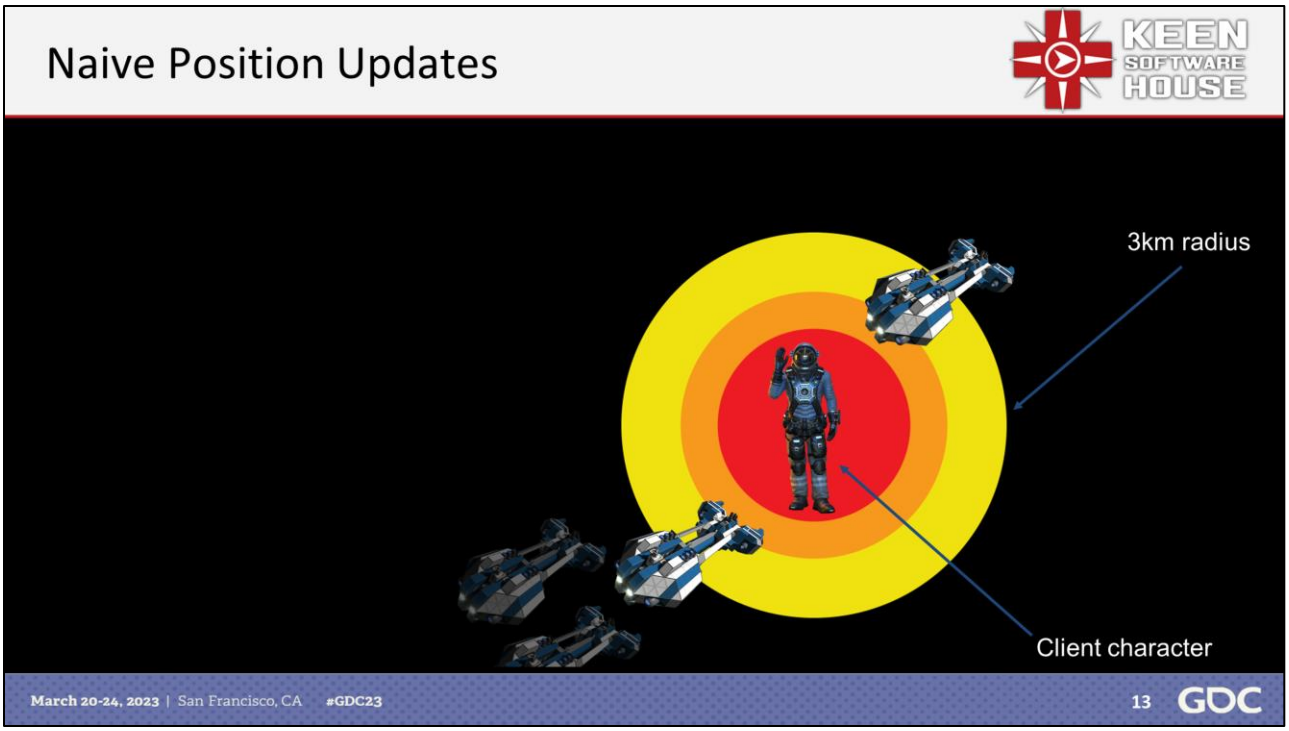
That's all for the intro.
Now, let me explain how we predict and synchronise entity transforms in our game:

- First I will establish how the positions are being updated to clients for all entities
- Then, I will show some issues of that approach along with improvements by relative positions in chapter 2
- Such solution have a big deficiency and that's a lag from user's input to screen, explained in chapter 3
- I will introduce basic prediction to fix that along with example correction in chapters 4 & 5
- Such approach works for static worlds like Counter Strike or Fortnite, I will explain why that does not work well for Space Engineers in chapter 6
- And continue with new proposal of relative prediction in chapter 7
- In chapter 8, I will touch on some specifics of physics setup needed
- And follow up with problems approach can bring and the solutions in chapter 9

So, we have quite long agenda, let's get to it.



Let’s start having our entity positions updated based on players actions.



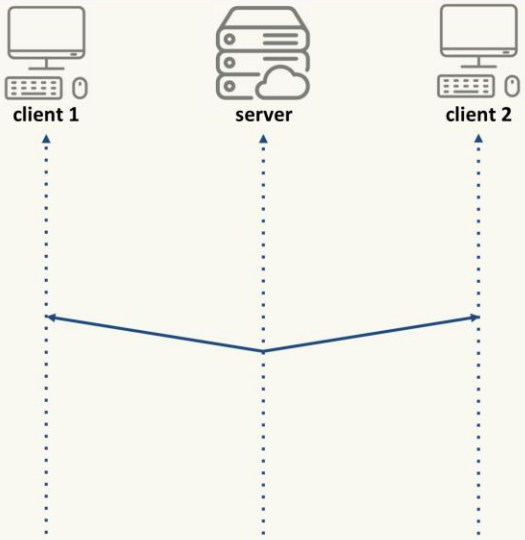
First, server world subset is replicated to the client based on the position of the client controlled entity, in this case his character.

Only client replicated entities get updates for their states and transforms. Server keeps track of replicated entities for each client.

Updates are sent sparsely the further the entity is from the client's controlled entity up to 3km radius.

So for example entity in red circle which has 20m radius gets updates every 4 frames, entity close to the edge of 3km replication radius gets updates every 60th frame, which means roughly every 1 second.

Position updates

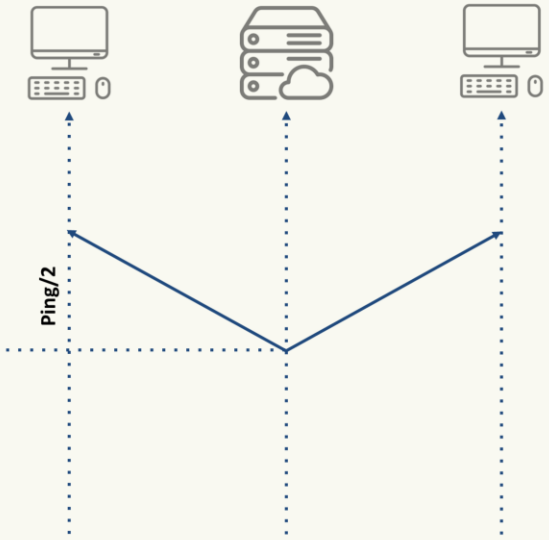


Let’s see the packet flow from server to clients.

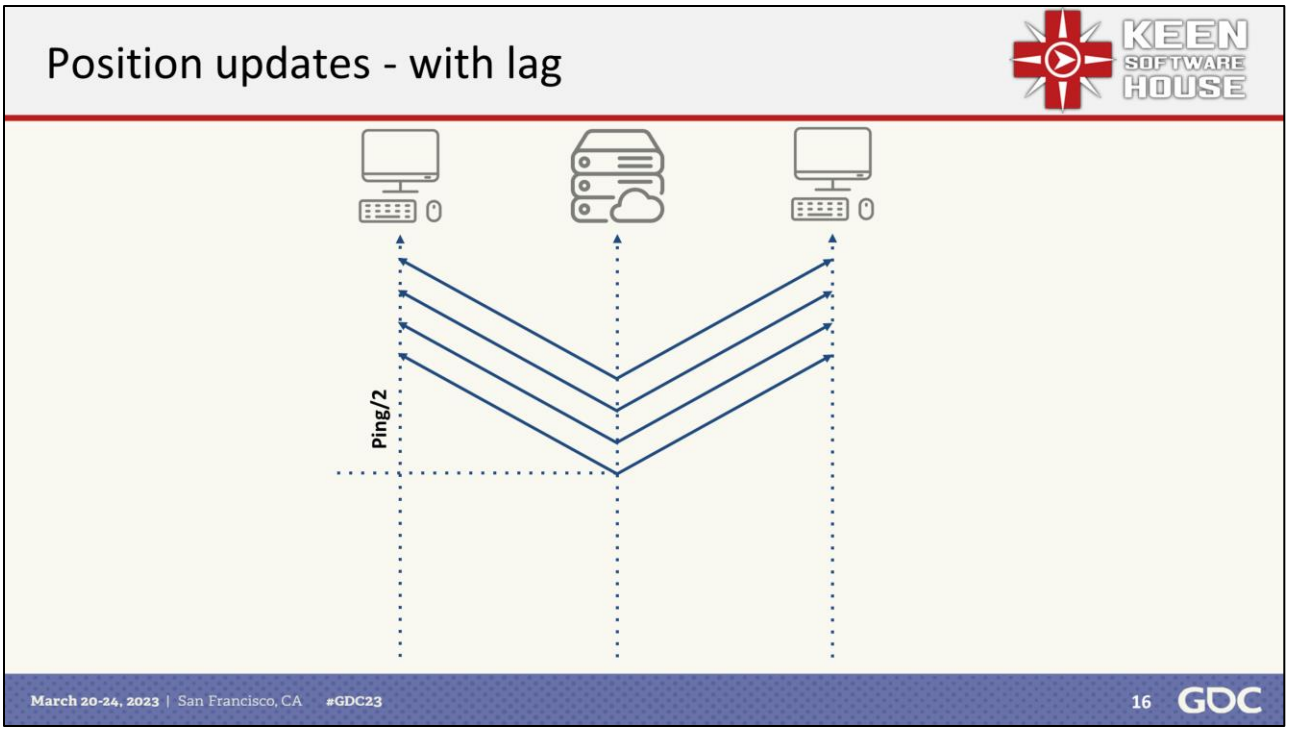
In the middle, there is a server and clients are on both edges of the diagram.

This is an ideal case of local network conditions without any lag.

Position updates - with lag

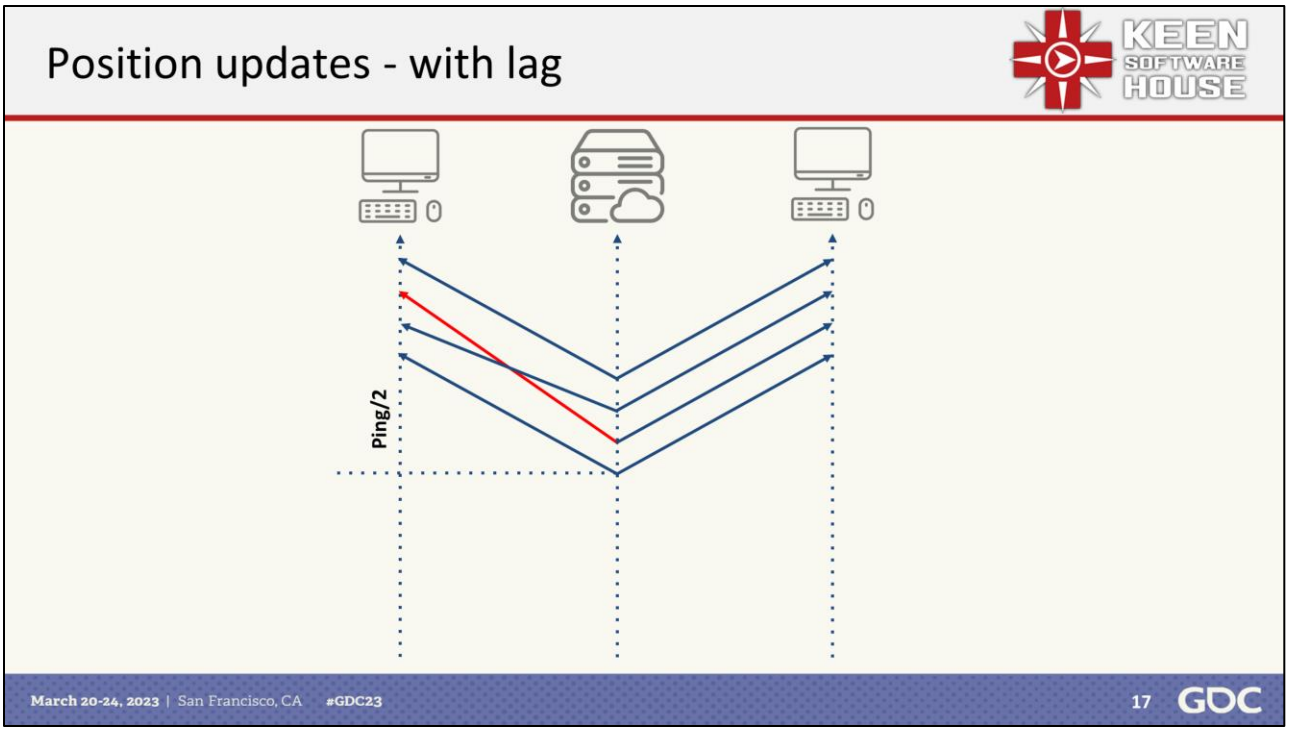


In real network conditions, packet travels from server to client for certain amount of time, which is half of the ping time.



Position updates are streamed using UDP protocol, which means server does not wait for confirmation.

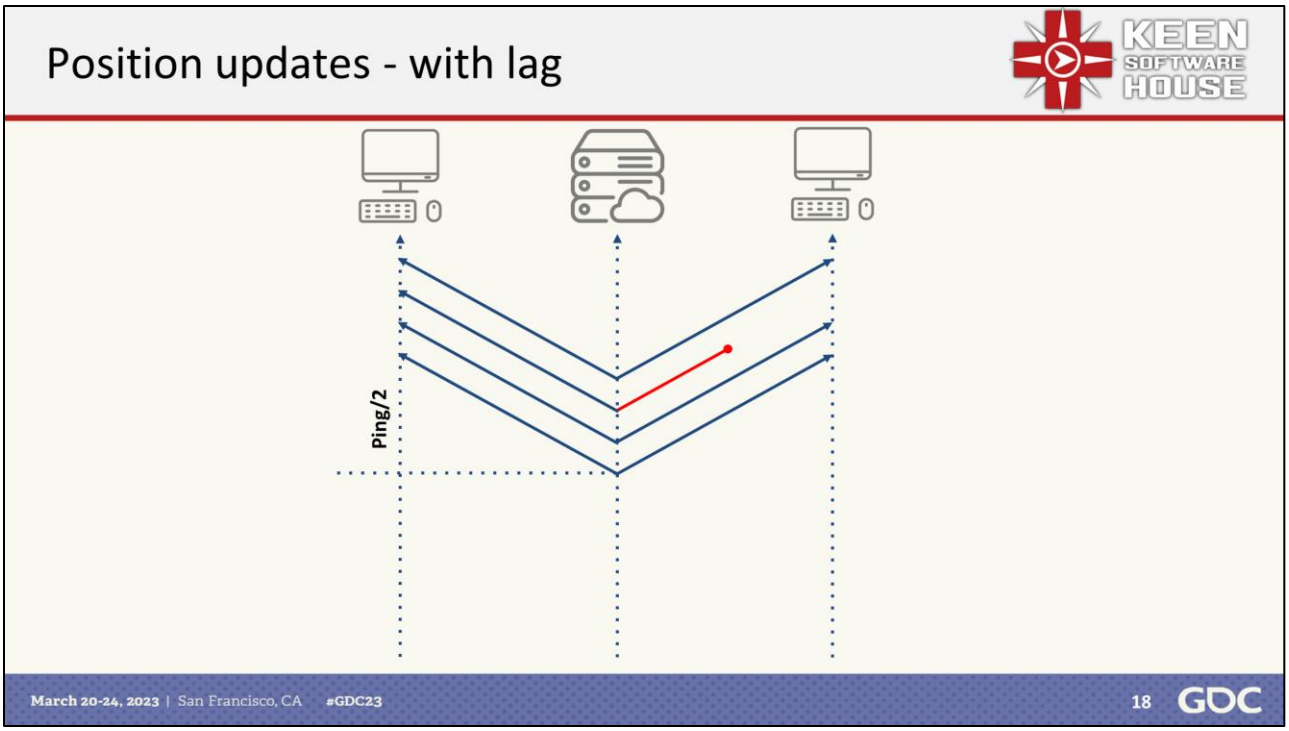
New data always supersede the old one.



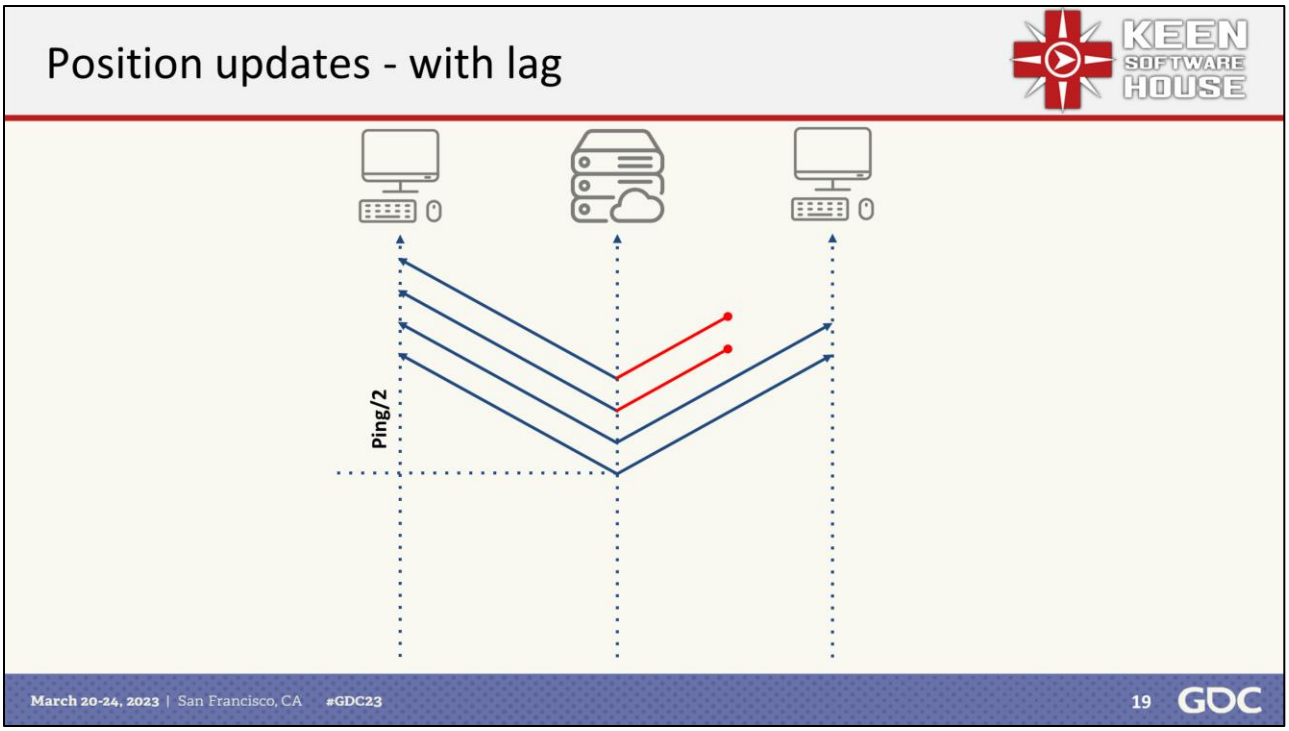
Real UDP networks exhibit behaviors like out of order delivery of the packets, see red arrow on the left side.

Client interpolates on the last 60ms of data, effectively operating 64ms plus half ping time late from the server.

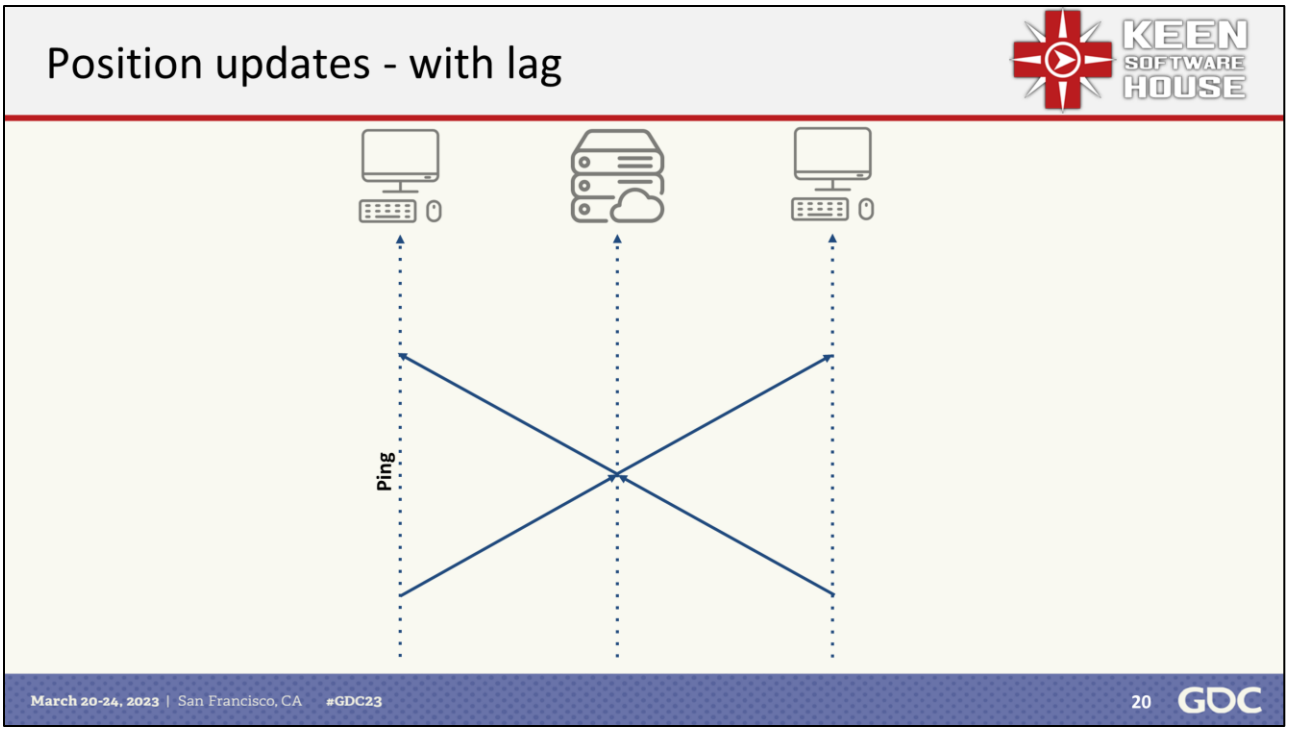
If the out of order packet is delivered within this time period, it is still entered into the position history and used. Otherwise it is discarded.



Another behavior of real networks is dropped packet. Since new data will come very soon, we just ignore it and interpolate on data we have.



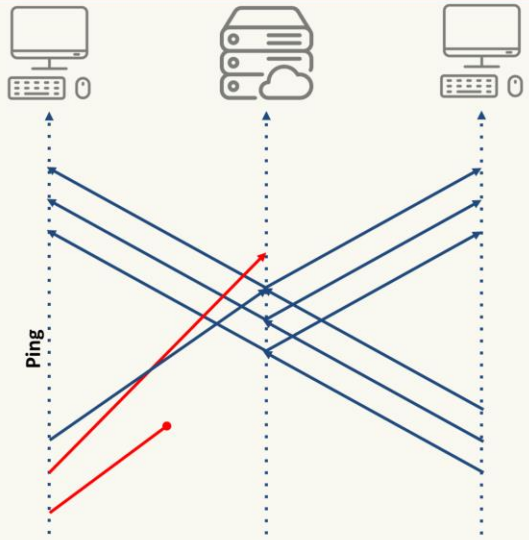
In case there is not enough transforms to interpolate, client **extrapolates** from older transforms.



From the other end, client sends controls updates (like button presses),

server applies them to simulation and sends results.

Position updates - with lag

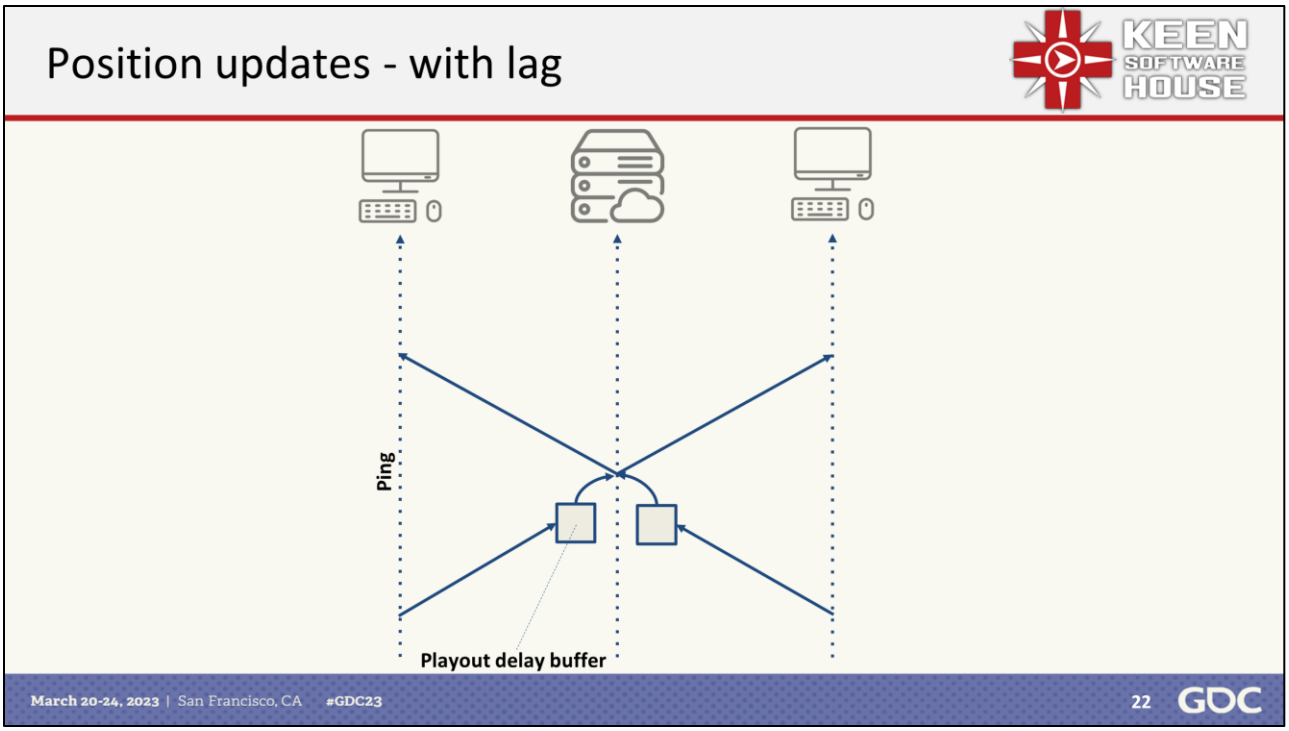


Real network conditions are a big issue in control updates, much bigger then in position updates:

Position updates seen on previous slides bare the complete information and every new packet makes the previous one obsolete.

On the other hand, with control updates, every packet have meaningful information and failing to apply it in correct simulation step changes the simulation outcome significantly.

It is very important to deal with such situations.

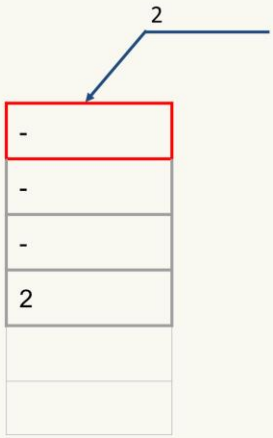


Hence we introduce a small buffer with 4 packets on server, which helps to cope with network conditions.

It is called playout delay buffer.

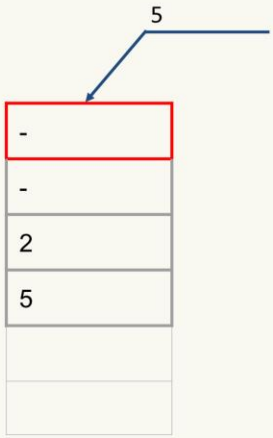
Let me show you how that works.

PDB - out of order



Let’s say, at the beginning we get packet with timestamp #2 and since it is first delivered packet, we enter it at the end of the buffer.

PDB - out of order

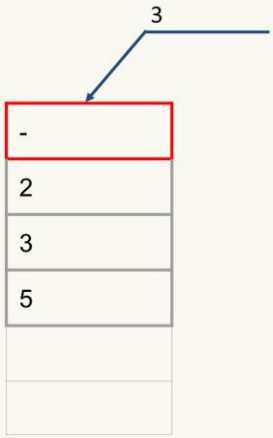


Next frame, we push buffer by one place.

We get packet #5 and order it behind the packet #2 since we miss packet 3 and 4.

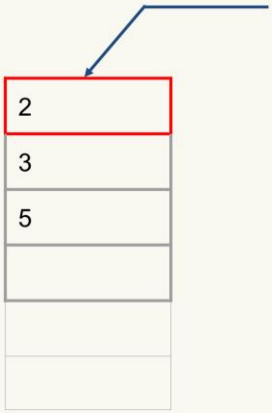
We are still not applying the packets to the simulation, so there is time to deliver them.

PDB - out of order



Here we go, out of order packet #3 is delivered and ordered into the buffer.

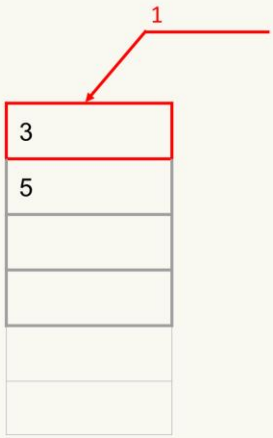
PDB - out of order



Next frame, no packet is delivered, but we anyway push the buffer by one place as in every frame

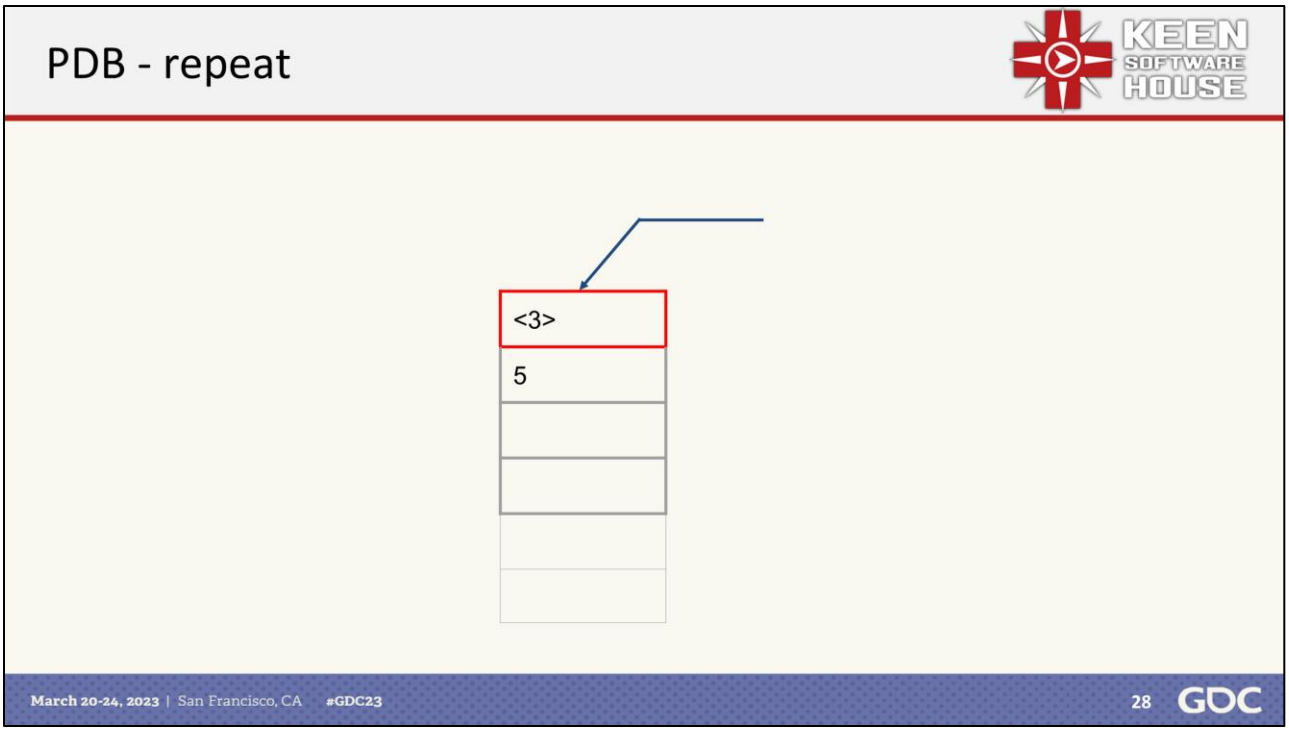
now the packet #2 reaches the buffer's head and we apply it to the simulation

PDB - too old



Next, packet #1 is delivered, but it is too old.

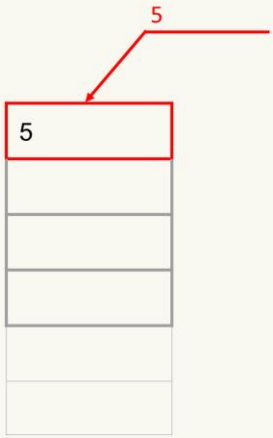
Applying out of order may change the simulation in unwanted way, so we rather discard it and instead apply the buffer's current head, packet #3.



Since we yet did not get the packet number 4, we apply again packet number 3 instead.

Since the packets hold the controls like arrow keys pressed in the frame, it is desirable controlled entities won't stop moving for every packet missing. It would result in jerky movement.

PDB - duplicated



Next frame, the duplicated packet #5 gets delivered. It is discarded, but packet #5 already present in the buffer is applied.

PDB - congestion

Next frame, we are out of packets, but we keep applying last delivered packet #5.

PDB - congestion too long

We stop repeating last packet after few frames, otherwise character will feel it is having its own agenda.

This was artificial case for our example purposes.

In real networks, these issues are much more rare and rather come in waves.

Our game contains various means of loosely attaching entities together using rotors and pistons.

Each attachment has some degrees of freedom, but is affected by motion from the attached entity.

In physics it is called constraints.

Relative Position Updates

When we interpolate each entity's transform separately, it creates disconnected movement.

It gets more pronounced when we get further from the entity, since we send position updates sparsely.

Relative Position Updates

- Based on physical connections
  - rotors and pistons
- Stream transformations local to parent
- Interpolate / Extrapolate in local space
- Convert to world space
  - similar to bone animation systems

The solution is to create hierarchy of entities, much like with character skeletal animation. Children stream transforms relative to its parent.

Each transform gets interpolated independently in local space.

Whole hierarchy is then resolved at once.

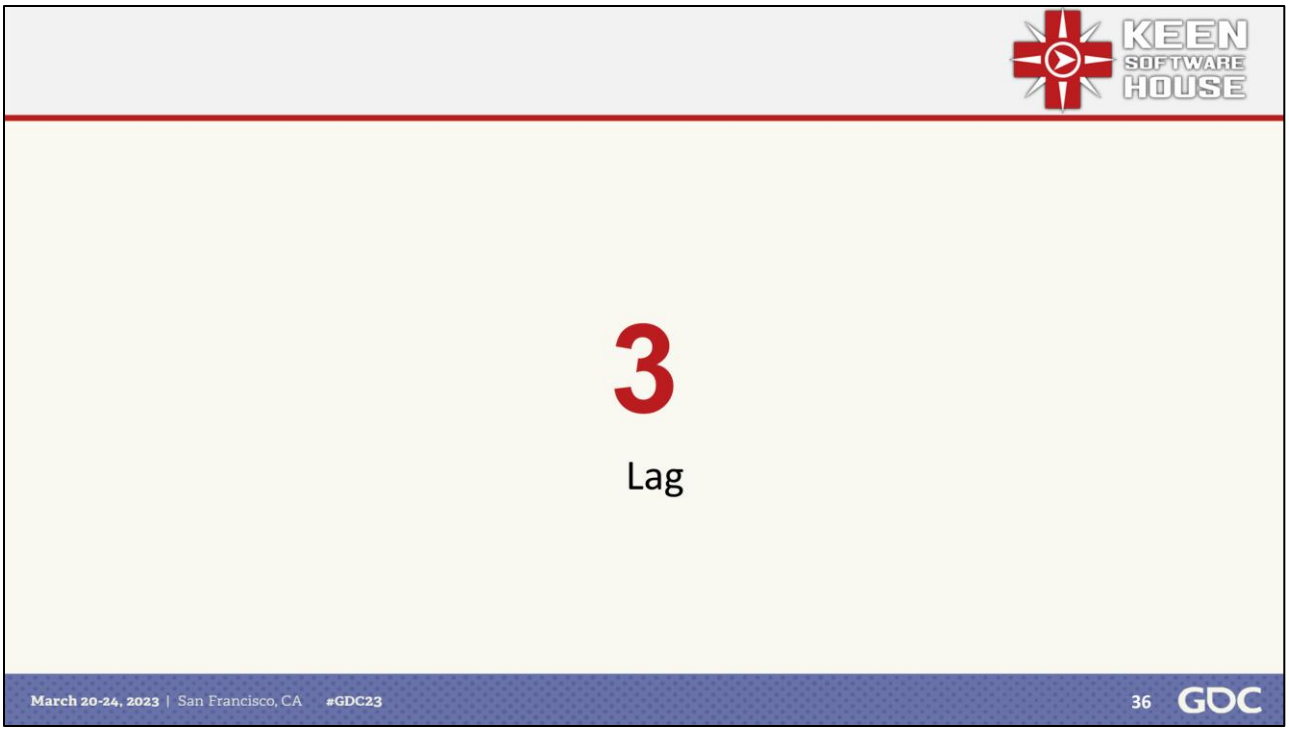
The challenge is to figure out which entity is root and which is child. With physical constraints, there is no such notion.

We have to account for cyclic dependencies as well.

There is a heuristic in place, prioritizing either controlled entity as root or at least largest entity.

Relative Position Updates

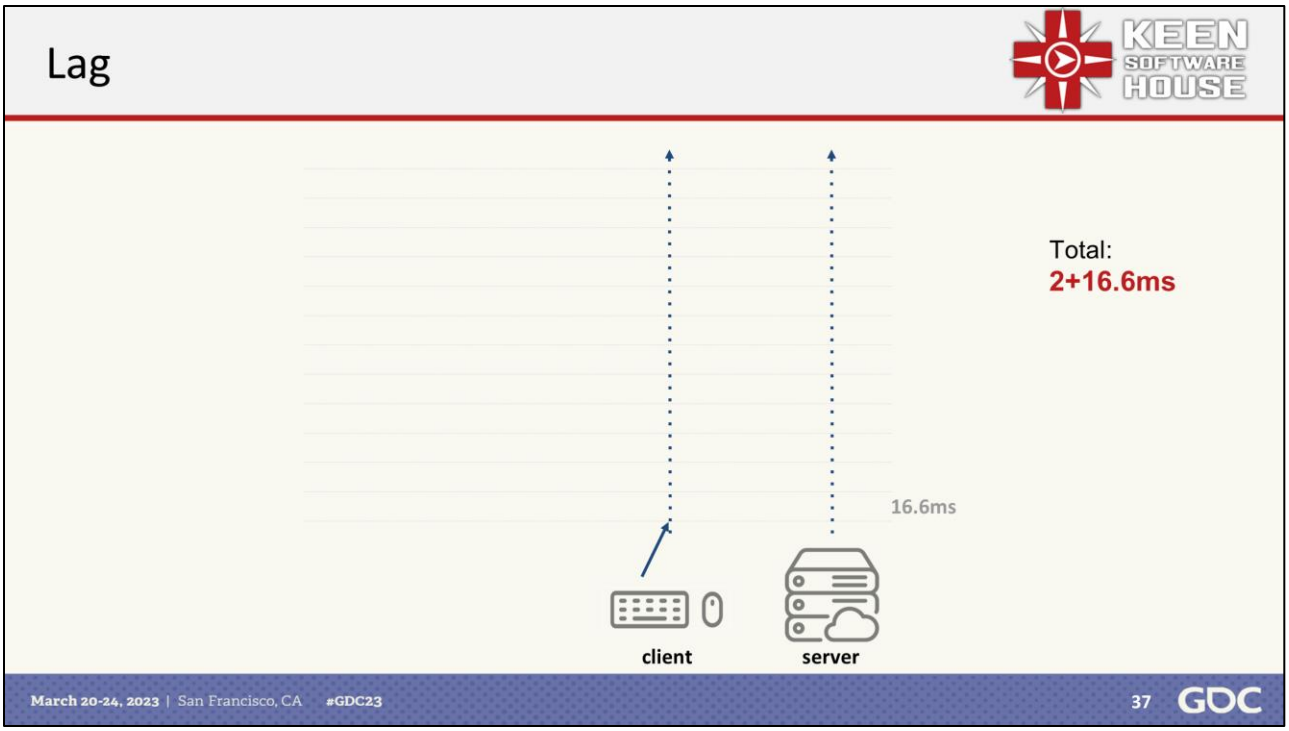
With this approach, the animation feels coherent even with sparsely sent position updates.



So far, we have a working prototype of entity synchronization.

But there is one big issue - time it takes for the action from a button press to appear on the screen.

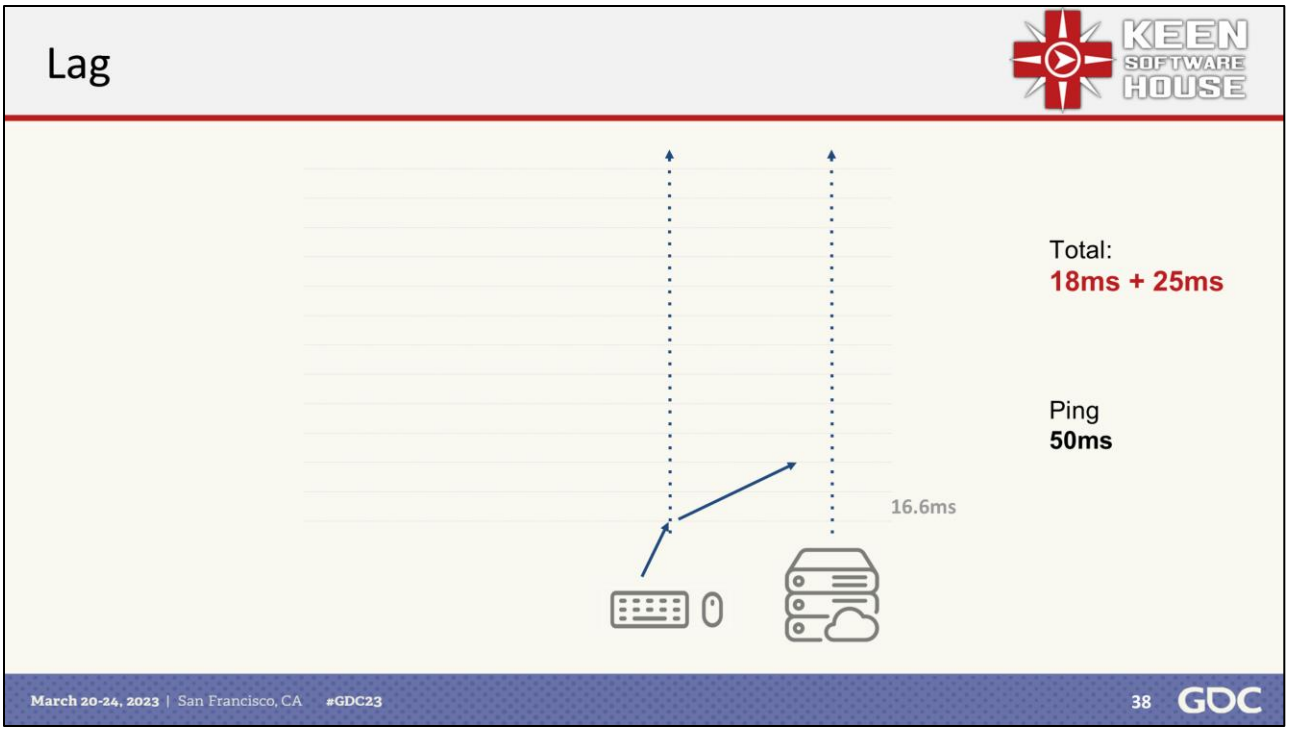
This time is referred to as a lag and the game is perceived laggy by now.



Let’s break down the order of operations, that it takes from action as a button press on the client through server simulation to screen.

First we press a button, which is registered by operating system in about 2ms.

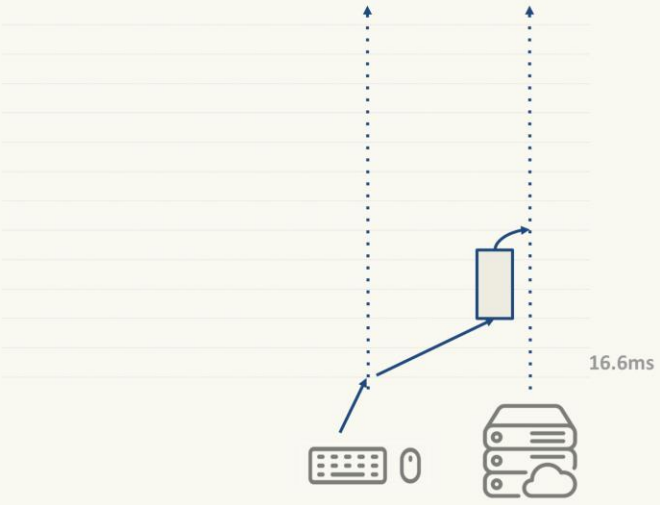
But with discrete frame steps at 60Hz it can actually take up to 18ms to get registered if we are **unlucky**.



We sample the input at the beginning of our frame step and send control updates to server right away.

With 50ms ping time, it takes 25ms to reach the server.

Lag

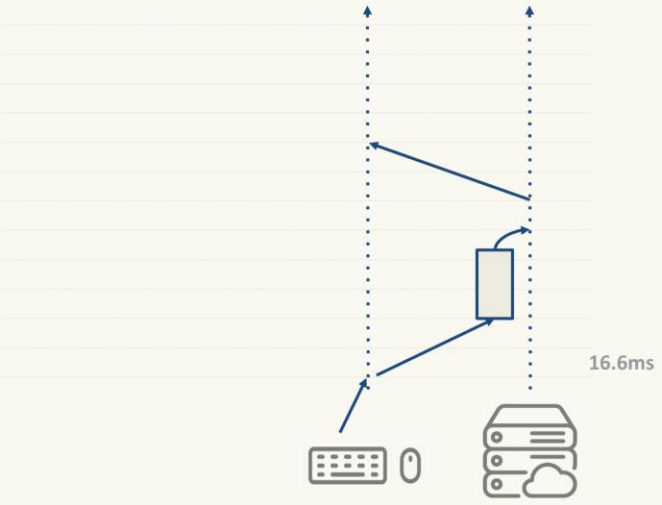


Total:
43ms +
4x16.6ms

Ping
50ms

<point>
On the server side, it sits in playout delay buffer for 4 frames, adding another 66ms.

Lag



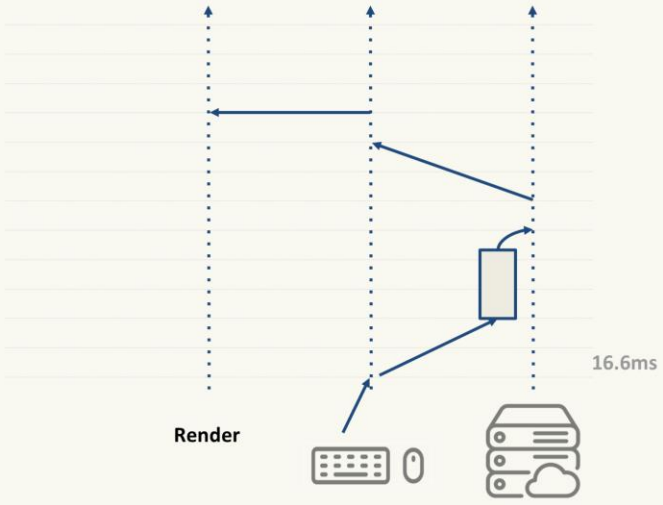
Total:
109ms +
16.6ms +
25ms

Ping
50ms

Server simulates the input and sends the result back to client at the end of the next frame.

It adds one frame latency plus half ping time to overall lag.

Lag

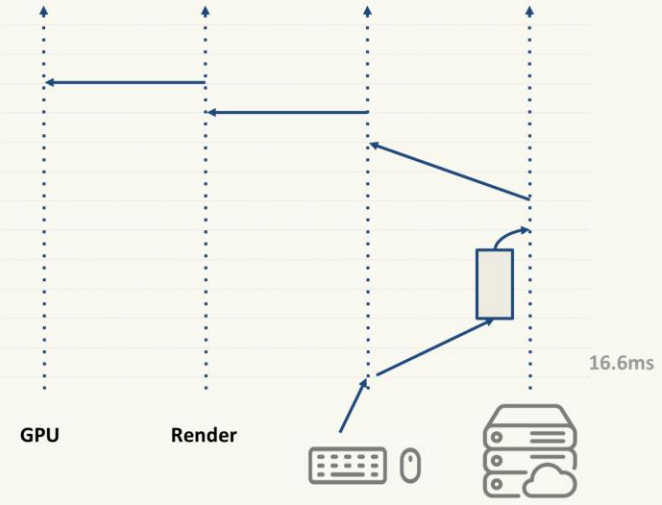


Total:
151ms + 16.6ms

Ping
50ms

Client gets the data delivered and applies them in its simulation tick and sends results to renderer. This adds another frame latency.

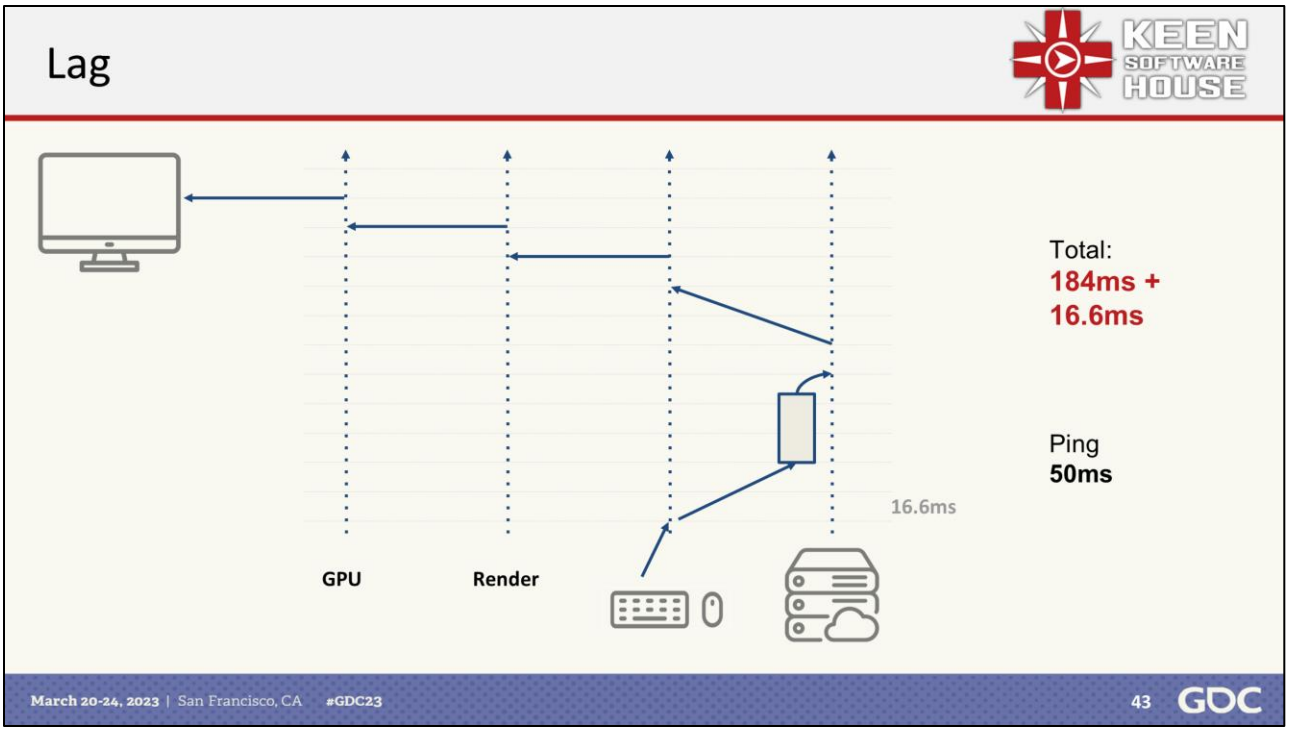
Lag



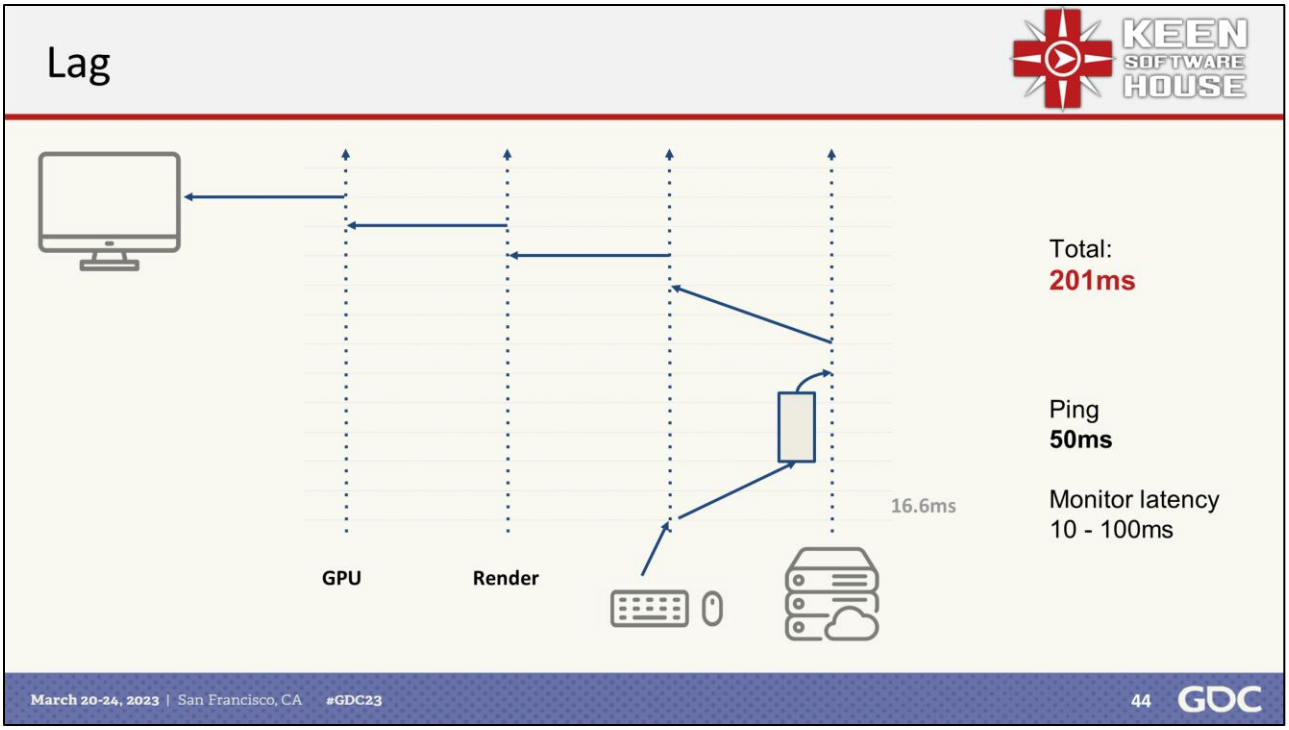
Total:
168ms + 16.6ms

Ping
50ms

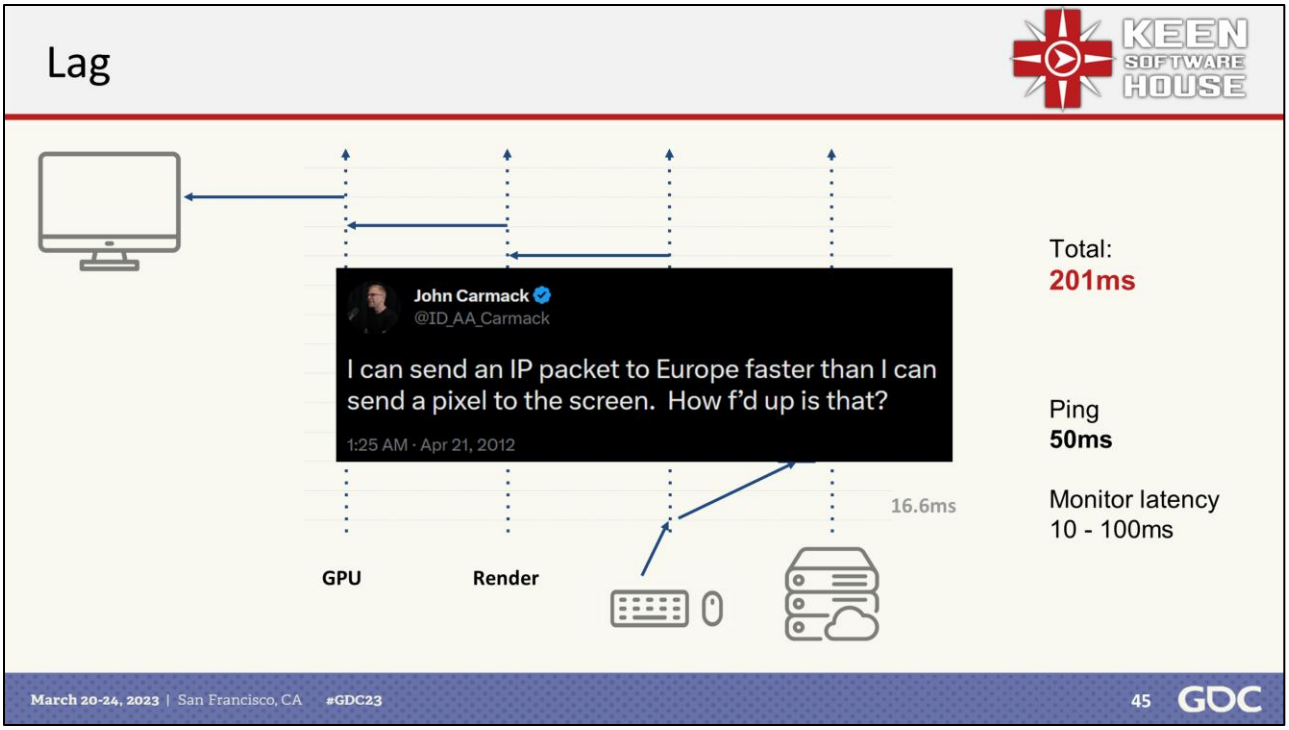
Render processes simulation messages and submits them to GPU, adding another frame latency.



GPU processes command buffers and send the results to the monitor or TV, adding another frame of latency.

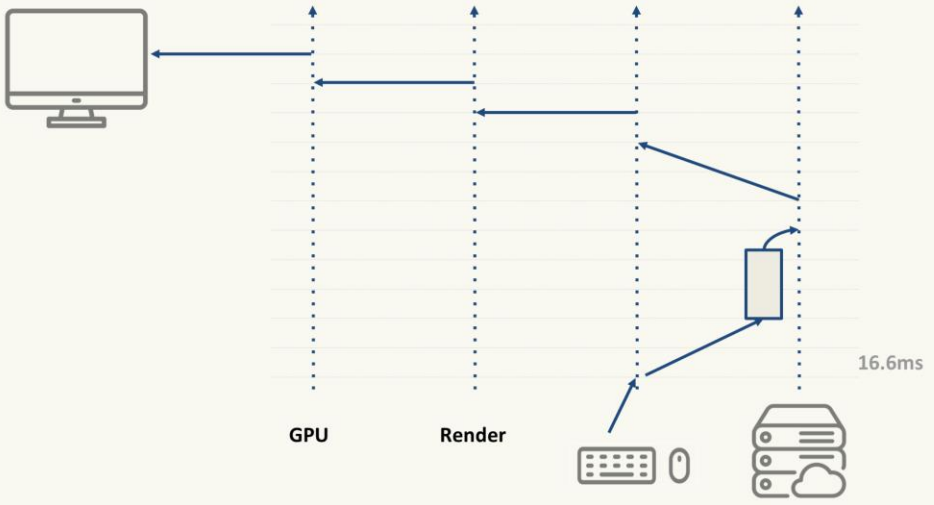


The monitor itself can take some time to display the results, latencies differ greatly from 10 to 100ms, especially for smart TVs with image enhancements.



John Carmack once said: I can send an IP packet to Europe faster than I can send a pixel to the screen.

Lag

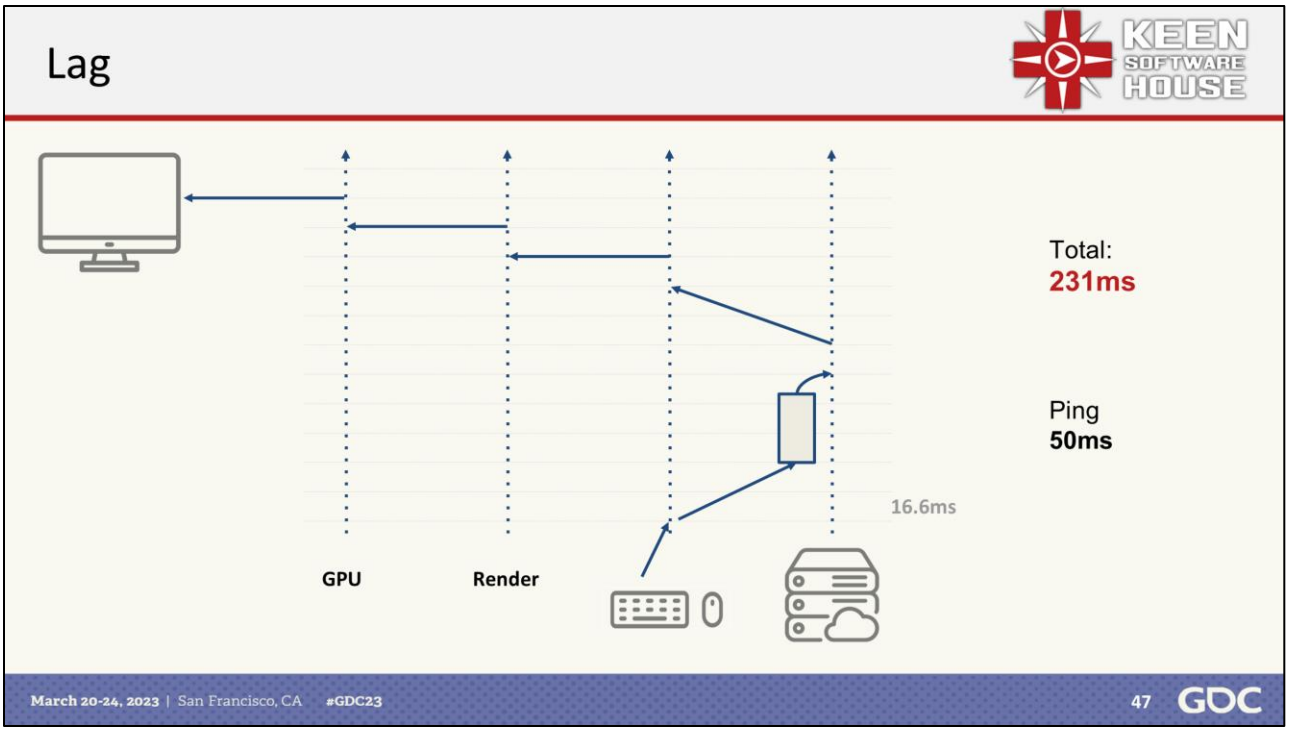


Total:
201ms + 30ms

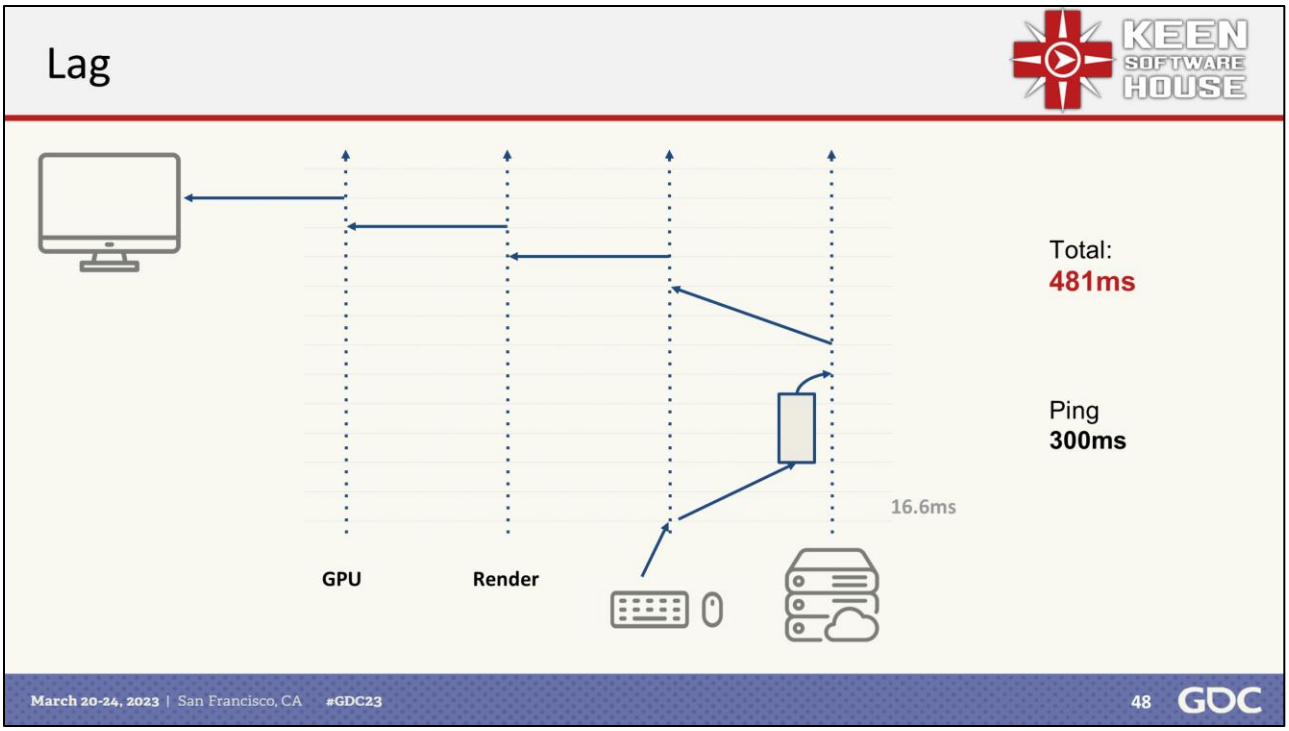
Ping
50ms

Monitor latency
10 - 100ms
30ms
displaylag.com

Let’s pick the average 30ms.



So we are at 231ms in quite optimal conditions. This is without added latencies caused by bugs in wrong frame update loop order, like sampling controls after simulation update.



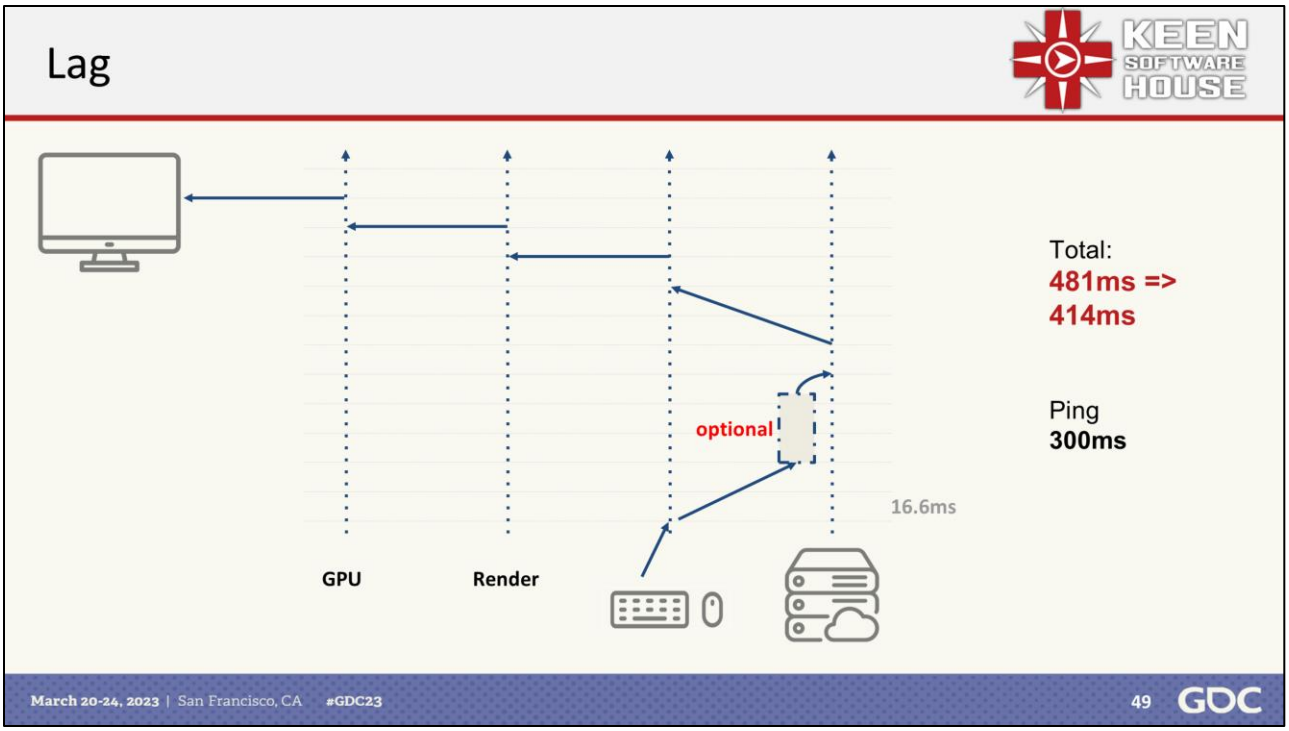
In not-so ideal conditions say, 300ms ping time, it can take **half a second** for the reaction.

And it is still not the worst case. We have 700ms ping players over satellite connection.

You can do your math.

Space Engineers are not competetive shooter, but such a lag is very annoying even in a builder game.

So what can we do about it?

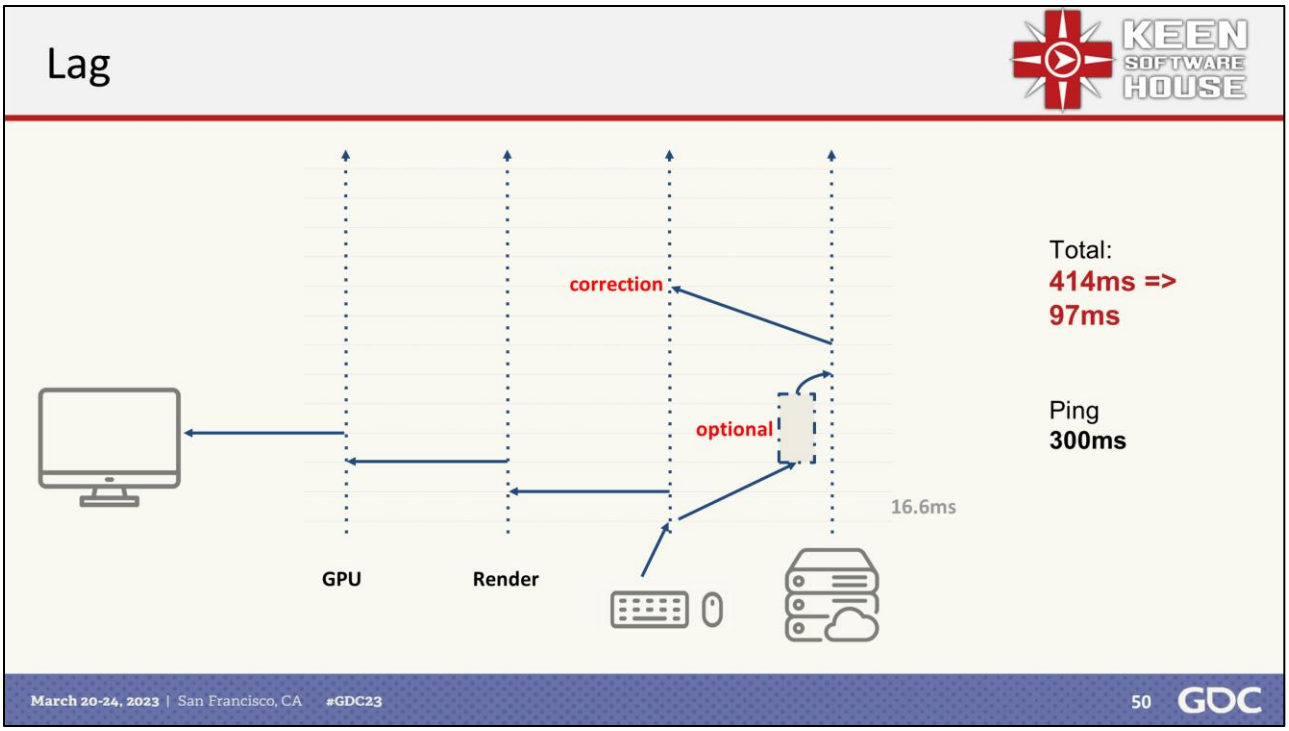


We can make playout delay buffer optional, detect problematic network conditions like out of order packets and enable it when needed.

We can disable it on the fly after packet problems won't happen for some time.

Network issues usually comes in waves - because of internet infrastructure changes like router malfunctioning and rerouting are happening on the fly.

What else can we do better?



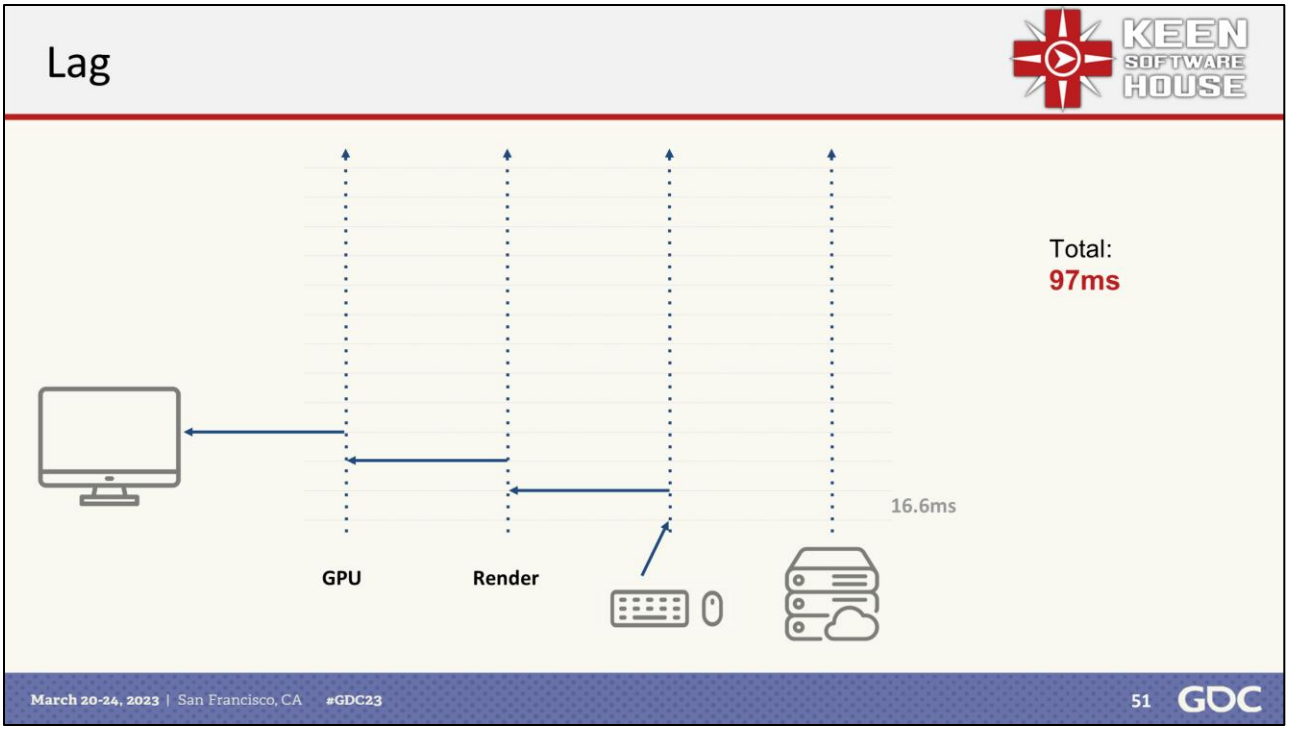
Here we go.

We can predict outcome on the client, instead of waiting for the simulation result from the server.

The benefit is, it cuts huge amount of time needed for network transfer.

The downside is, it adds possibility of wrong outcome of client simulation, so-called MIS prediction.

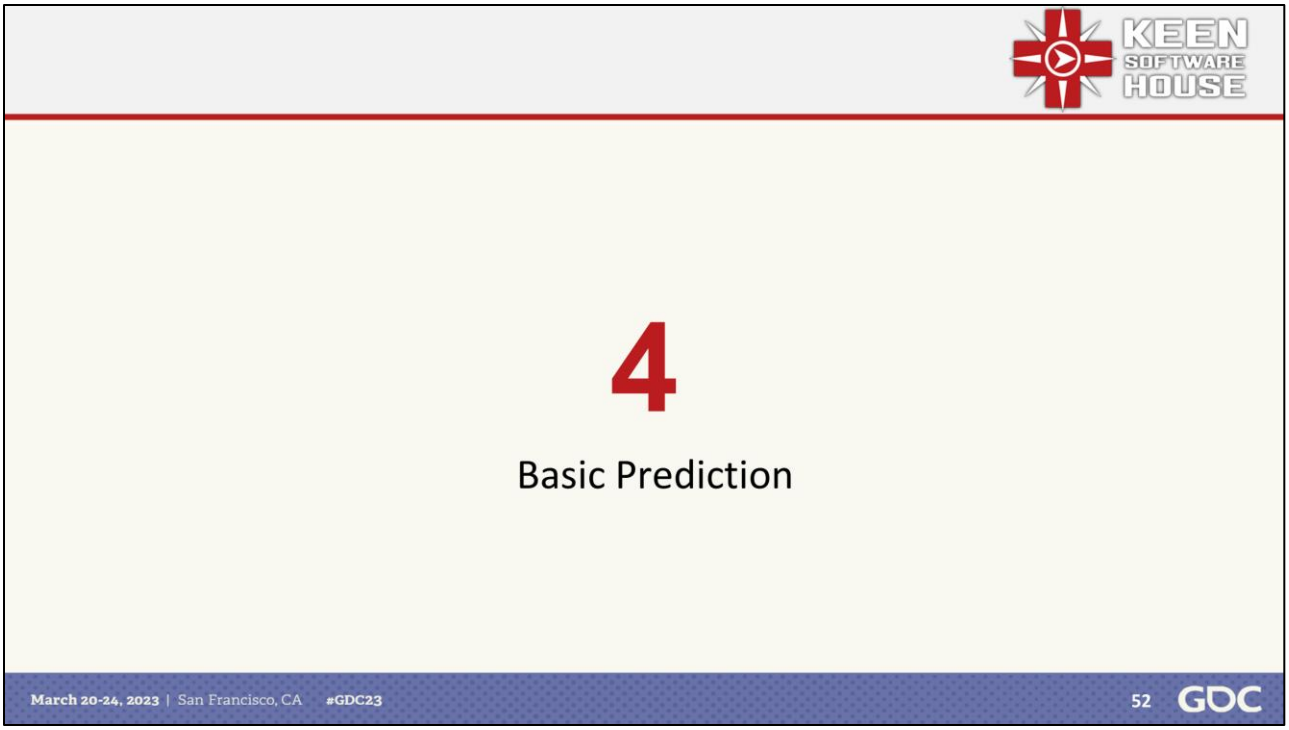
Since we want to obey our server, we have to correct the results later, once we will get them.



We are left with shy of 100ms on the client.

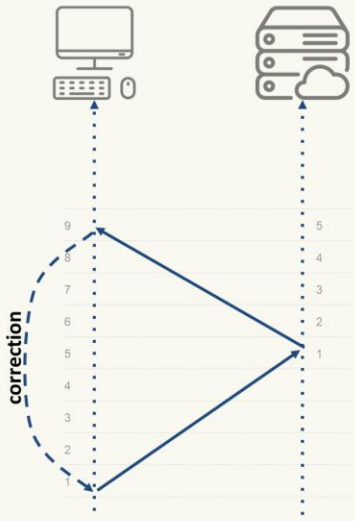
Bare in mind this is a worst case scenario, where you press controls right after sampling it and each frame taking exactly 16.6ms to finish in every stage of the simulation and rendering.

This is rarely the case.



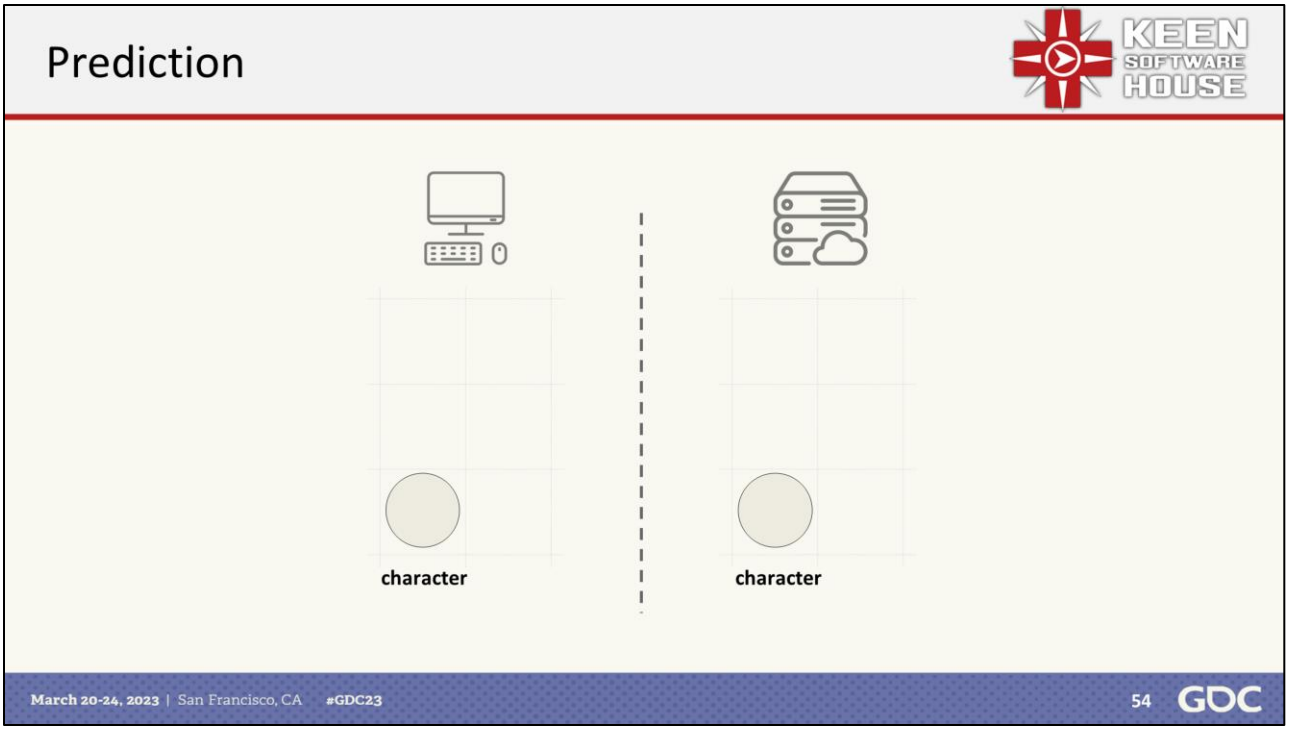
So prediction, how does that work?

Prediction overview



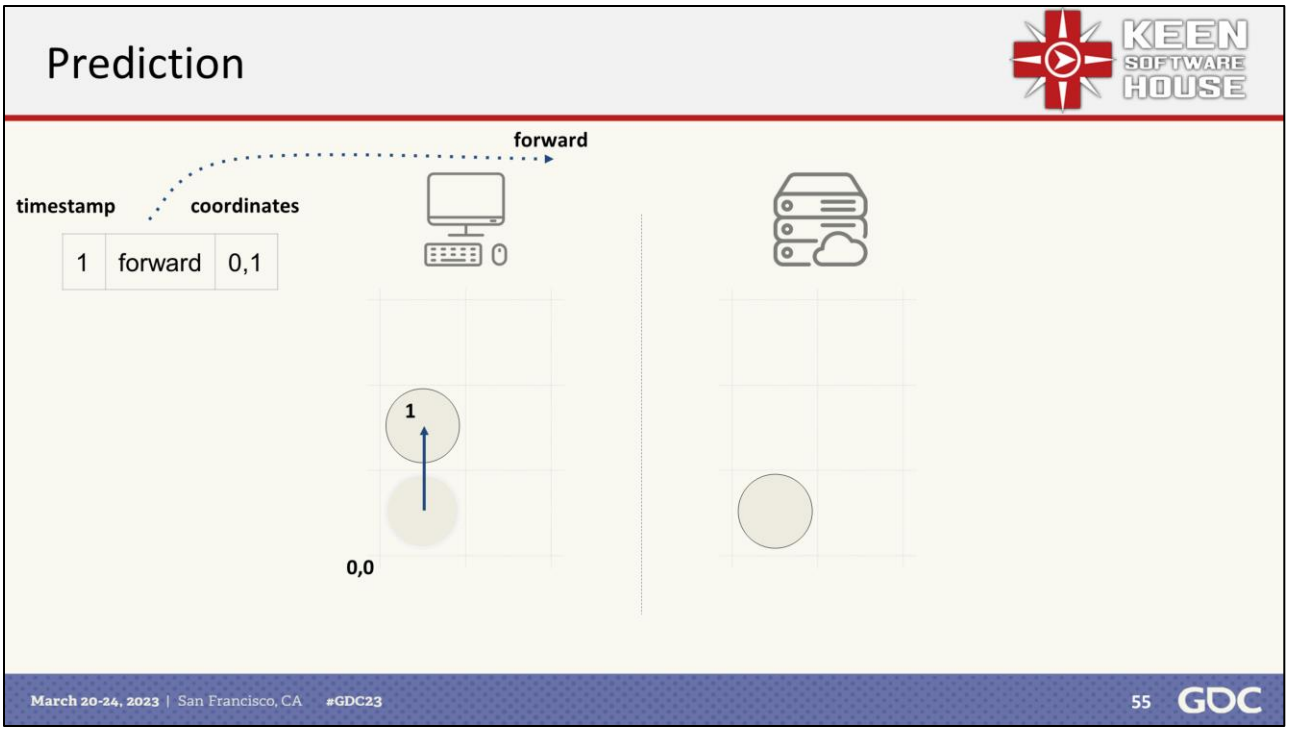
We are recording history of positions simulated on client and comparing the results received from the server later on.

We apply the delta position between client and server not only to the current position, but to the whole list of historical positions.



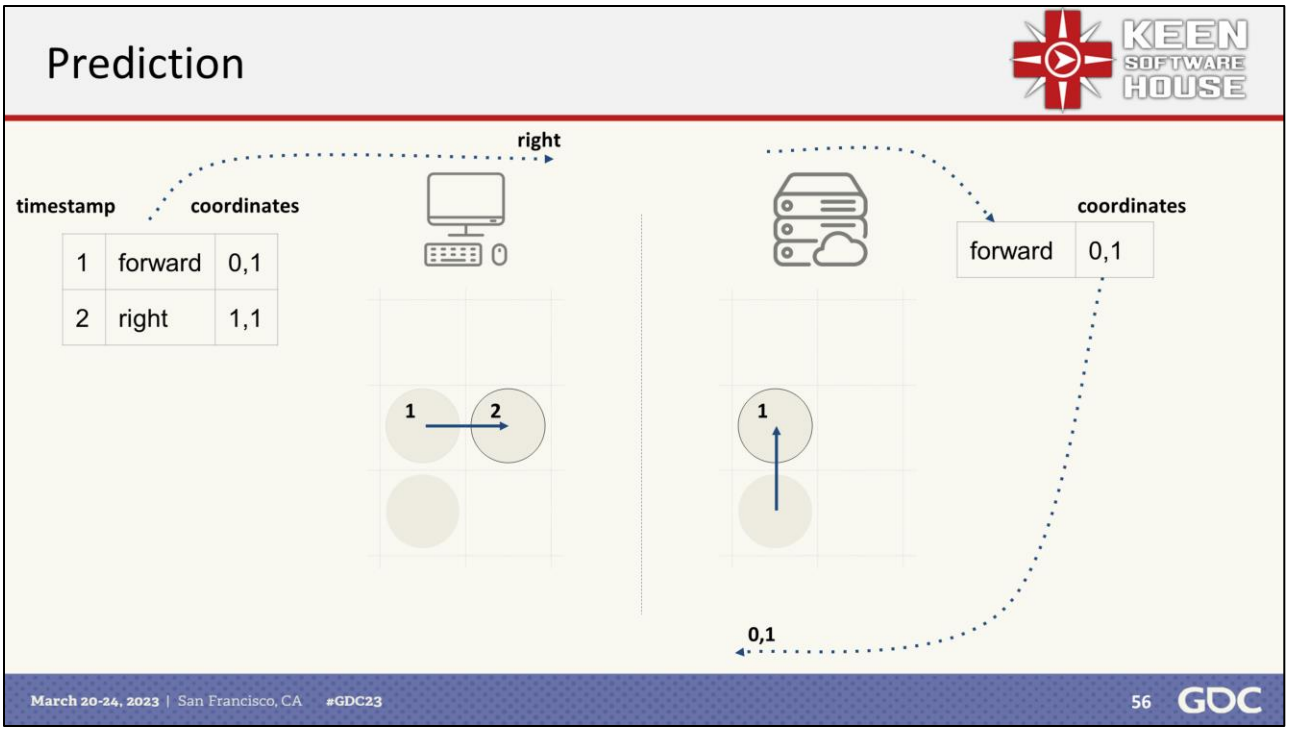
<point>
Let’s start with the same state of the character on client (on the left side) and server (on the right side).

The diagram is from top view in 2 dimensions only for simplicity.



When user presses the forward button

- First, client sends the button press information to the server
- Then, simulates character moving forward from 0,0 coordinates to 0,1
- And last remembers the simulation outcome in the history list

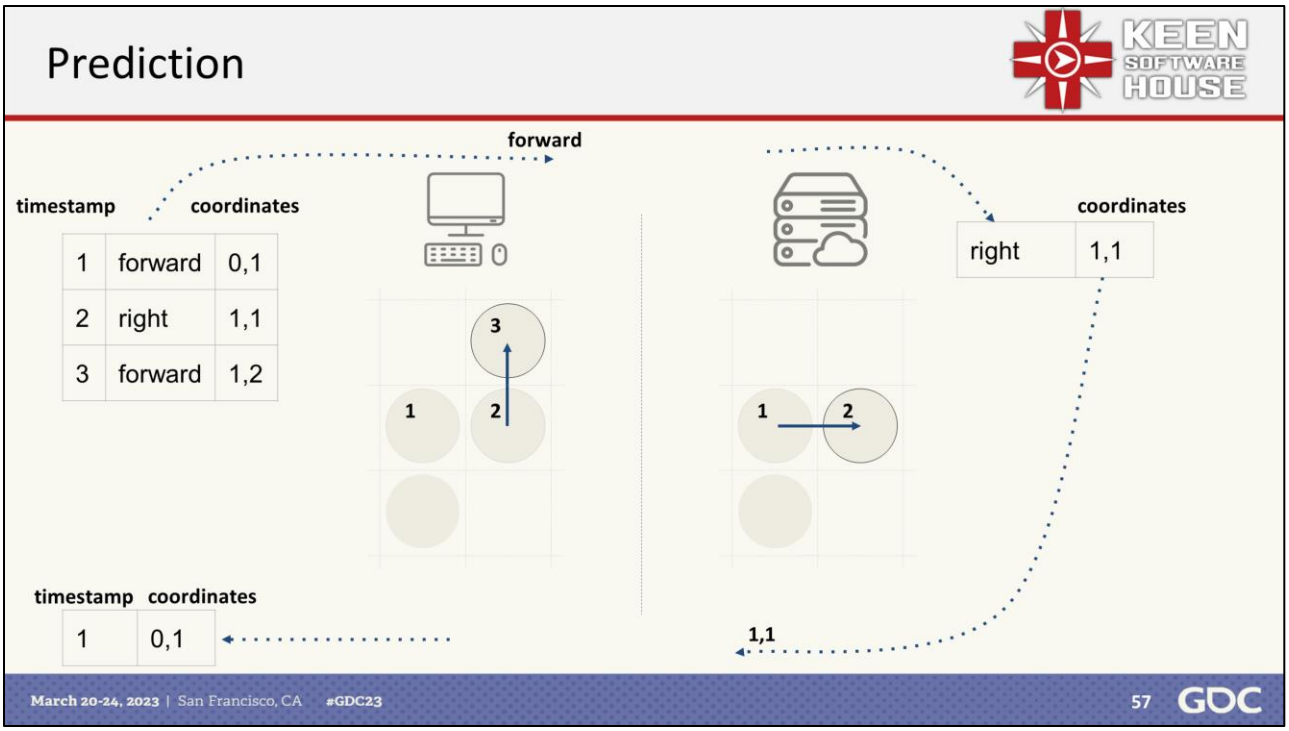


For the next timestep many things happen at once, let's start from the client on the left:

- The steps are very similar to last slide
- User presses the right button
- client sends the button press information to the server
- client simulates character moving forward from last coordinates to 1,1
- And finally client remembers the simulation outcome in the history list

In the meantime, server on the right side

- gets the packet with forward button press from last simulation delivered
- let's say network conditions are good, so we can skip playout delay buffer
- Server simulates the character moving forward to coordinates 0,1
- Server sends the simulation outcome as positions 0,1 to the client

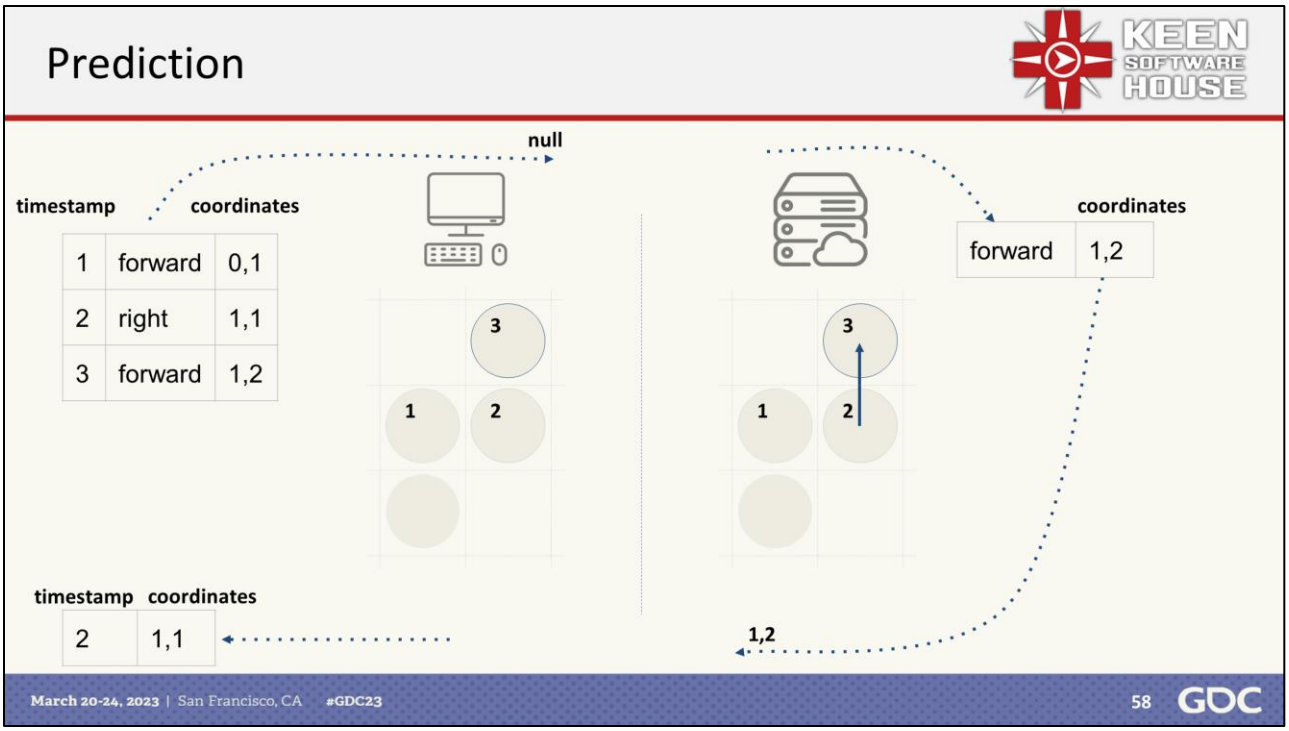


Next frame, let's again look on the client on the left side:

- User presses the forward button
- client sends the button press information to the server
- client receives the simulation outcome from the server and compares it with history timestamp #1
- It correlates, so we can freely continue without any change
- client simulates character moving forward to coordinates 1,2
- And finally client remembers the simulation outcome in the history list

In the meantime, server on the right side:

- gets the packet with button press from the last simulation delivered
- Server simulates the character moving right to coordinates 1,1
- Server sends the simulation outcome to the client

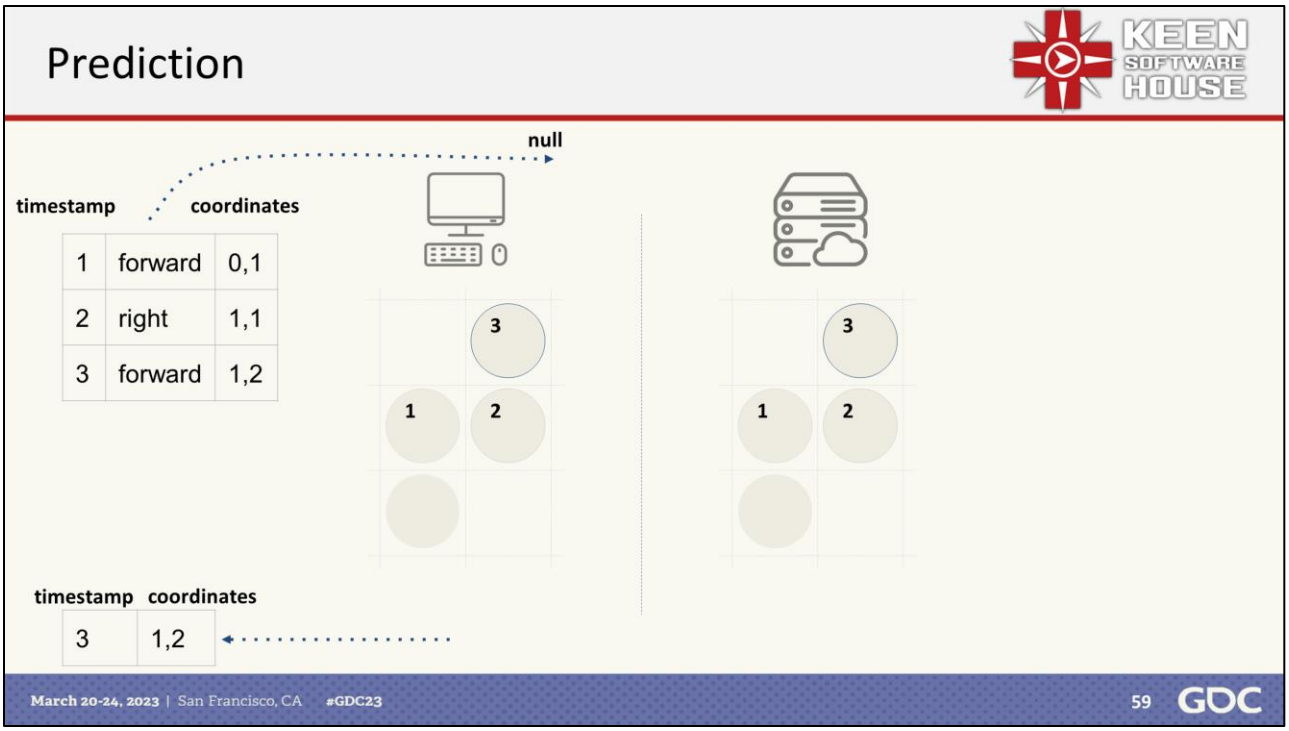


Next frame, again client on the left:

- User won't press any button, null update is sent to server
- client receives the simulation outcome from the server and compares it with history timestamp #2 and it correlates

Server on the right side:

- gets the packet delivered
- Simulates the character moving forward to coordinates 1,2
- And sends the simulation outcome to the client

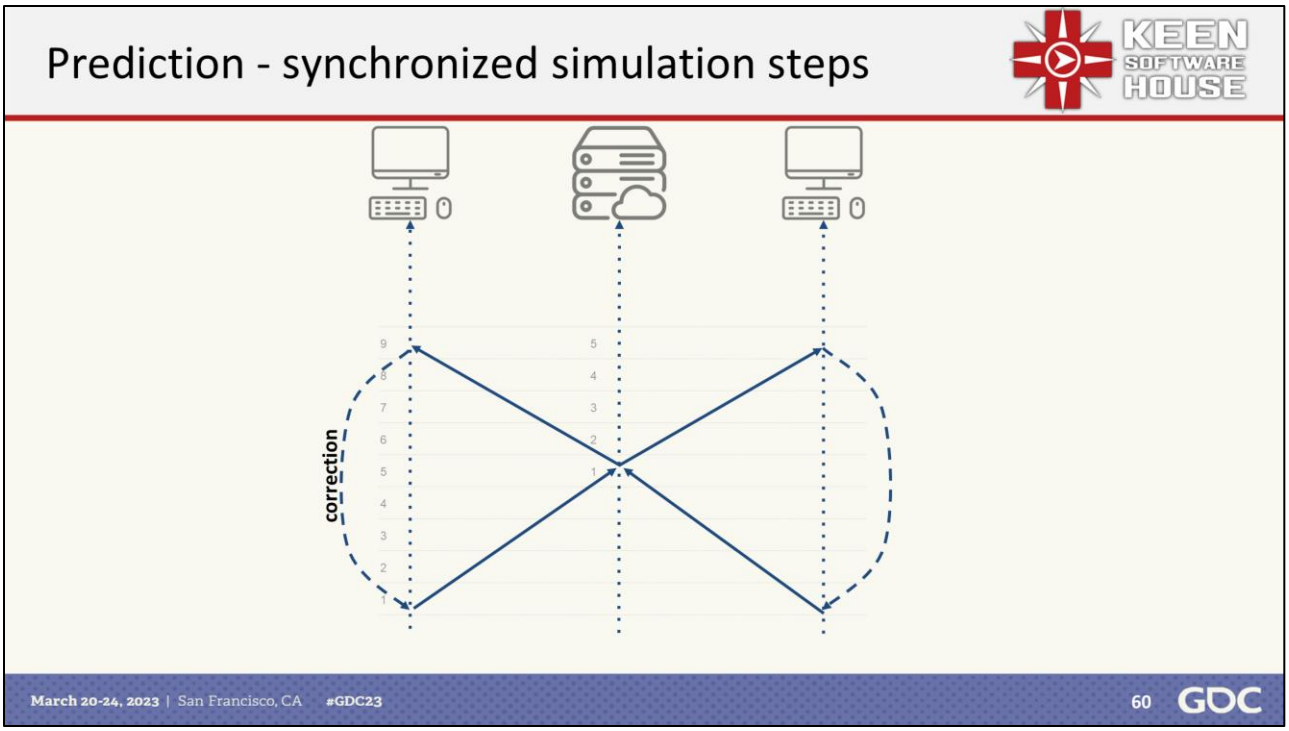


And finally:

- User again won't press any button
- client receives the simulation outcome, compares with #3 timestamp and it correlates

<pause>
This went pretty smoothly, which is not usually the case. In the next chapter, I will show you more realistic case.

But now, let's have a look on the most important prerequisites of prediction, synchronizing simulation steps.



So, synchronizing simulation steps:

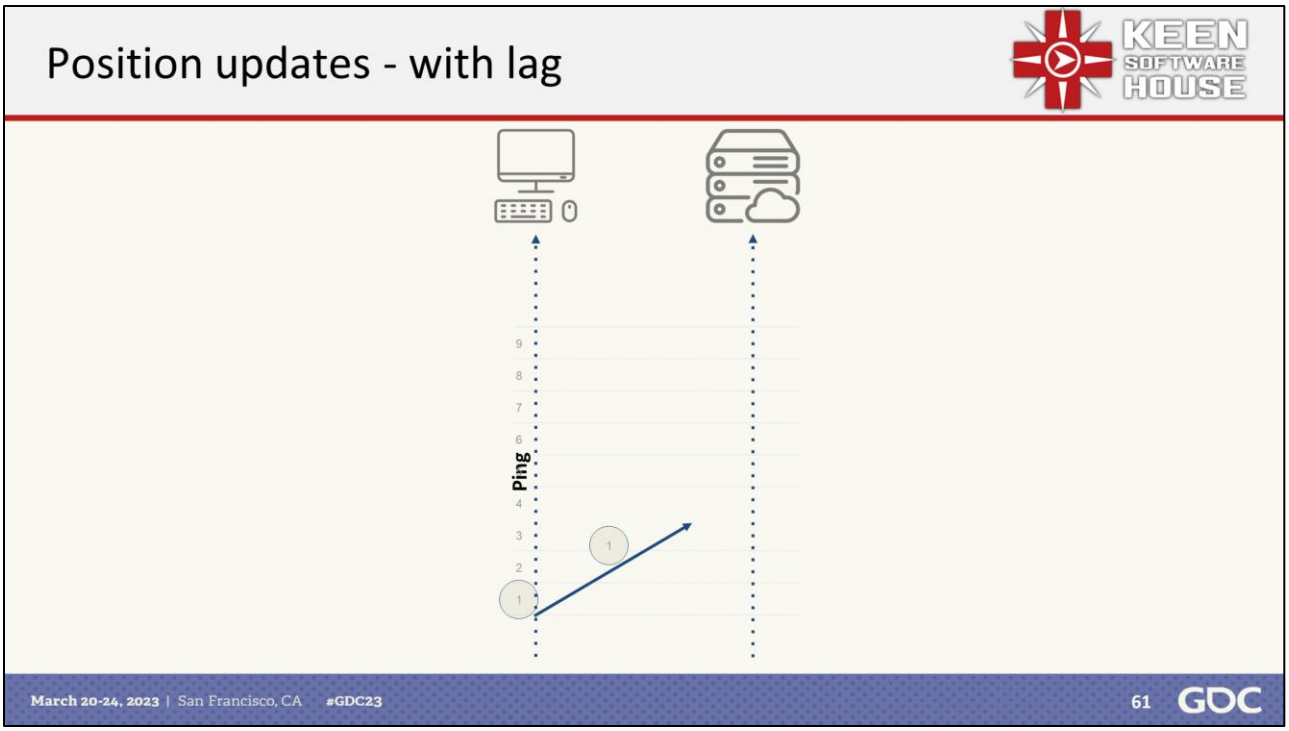
Computers have very precise timing, so why is this actually needed?

Because the client and server frame loops are synchronized until it has just enough work to do in given time frame (let's say 60Hz) and just waits some time at the end.

The second some work takes more time than expected, we get a frame drop, but more importantly for our case, the client and server timings gets out of sync. It can happen on either of the machines.

Optimizing code so it won't happen is very important, but it is impossible to achieve it 100%.

Especially with modding, which our game stands and falls with.

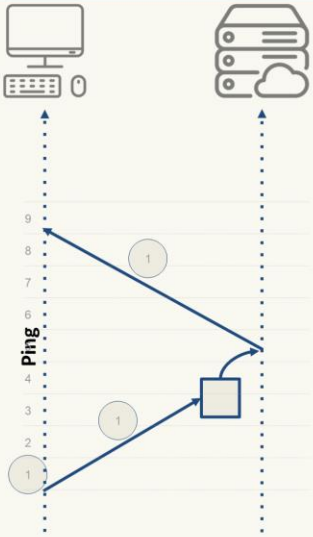


So how do we do it:

We are measuring ping time continuously by stamping control update packets sent from the client with client's simulation time.

Here you can see packet from frame #1 timestamped as packet #1.

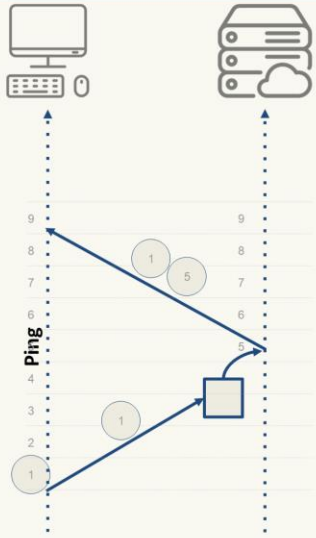
Position updates - with lag



Server, when sending results, hands over this client stamp from the last applied control update.

This way, we know exactly the time it took for the round trip, including sitting in the playout delay buffer and server simulation time.

Position updates - with lag



On top of that, server stamps the packet with its frame #.

Prediction - synchronized simulation steps

- Continuous measurement of the ping
- Client falling behind
  - Speed up
  - >1000ms behind = reset to server
- Client ahead
  - Slow down (wait)
- Hysteresis

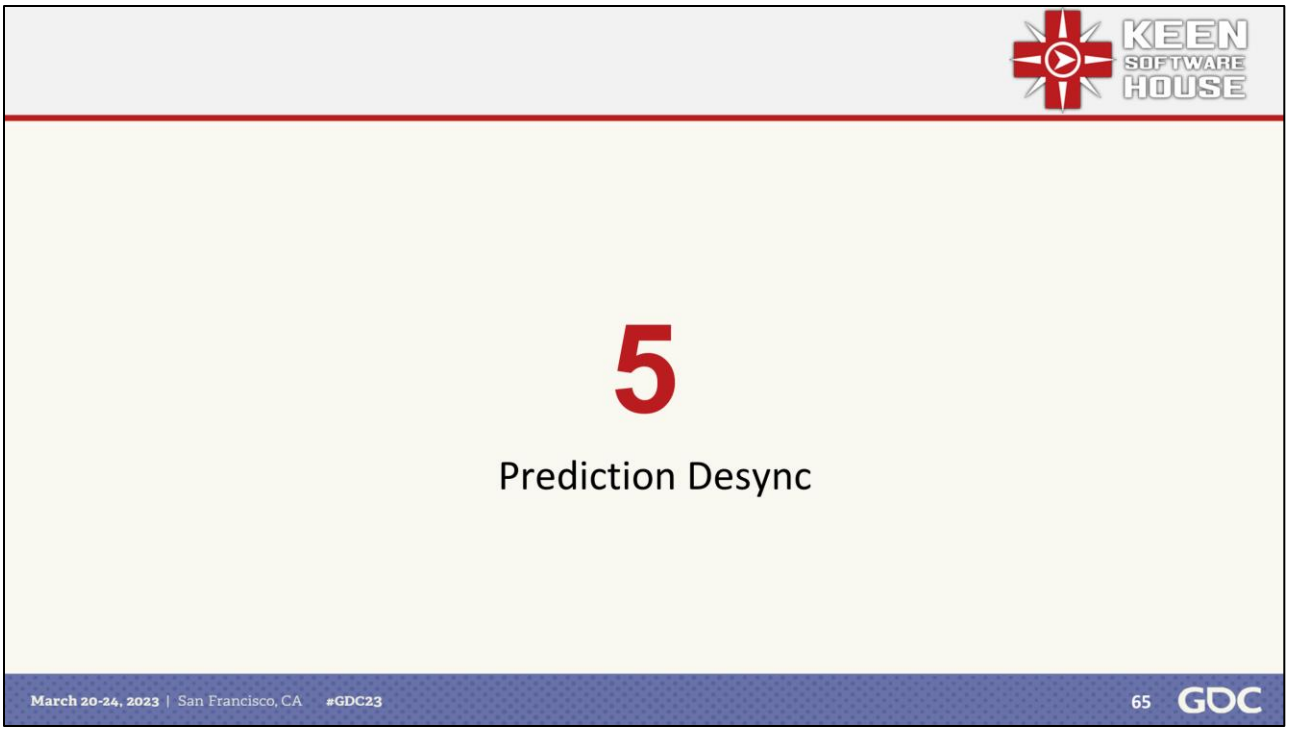
Client is then responsible to keep his prediction simulation time half ping sooner than the server time and interpolation time half ping later.

When client is **falling behind**, it tries to speed up the simulation, if possible - it waits smaller amount of time at the end of the simulation loop.

When there is **too much work** in the frame for speed up to be possible and it falls behind up to 1 second, it skips all frames up to the current server time.

When client is **ahead**, we slow it down by adjusting the wait time at the end of the simulation loop.

Of course, we have to make sure, the timing won't fluctuate and the time changes are subtle enough not to be noticeable by the player, so there is some complicated smoothing and hysteresis going on.

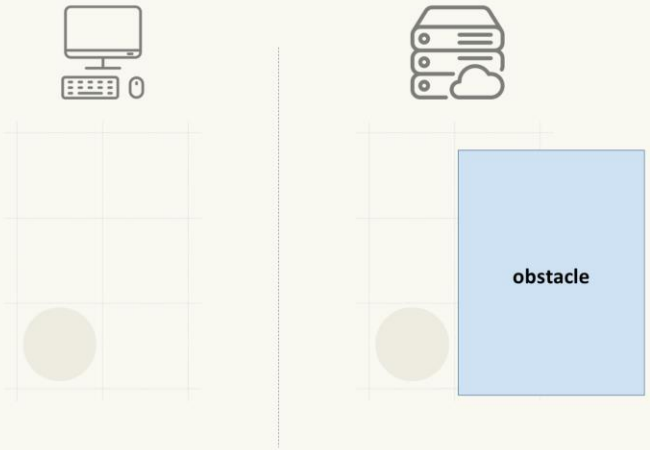


<pause>
So, we had a look on ideal case of client prediction. Let’s now have a look on the less ideal case.

What happens if the predicted entity’s position is different from the server’s results?

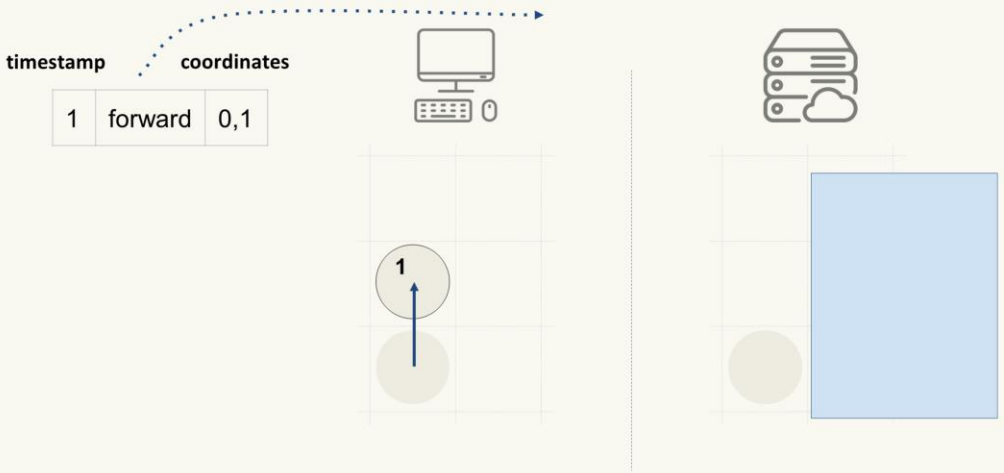
We call it prediction desync or misprediction and trust me, it is quite usual.

Prediction - desync



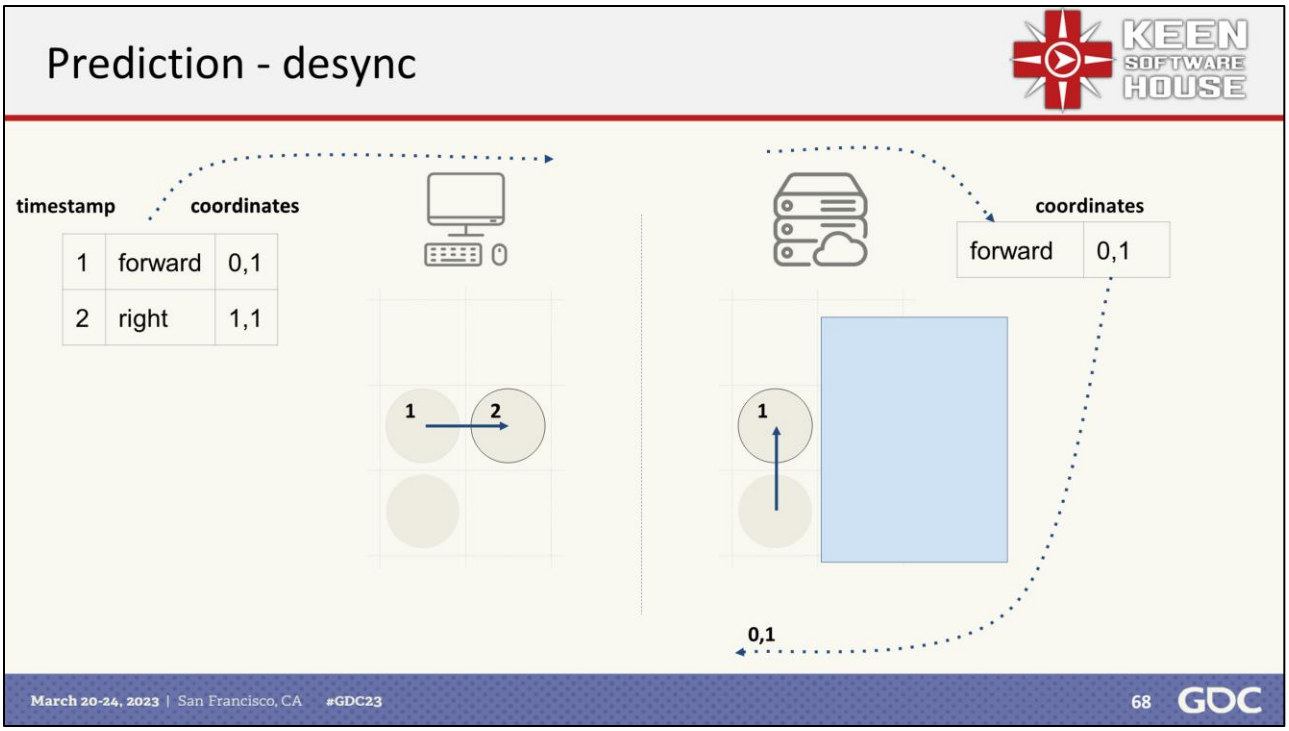
Let’s start with the same state as last time, but having the obstacle on the server side, which is not yet present on the client side. So called desync.

Prediction - desync



<point>
When user presses the forward button

- client sends the button press information to the server
- simulates character moving forward from 0,0 coordinates to 0,1
- and remembers the simulation outcome in the history list

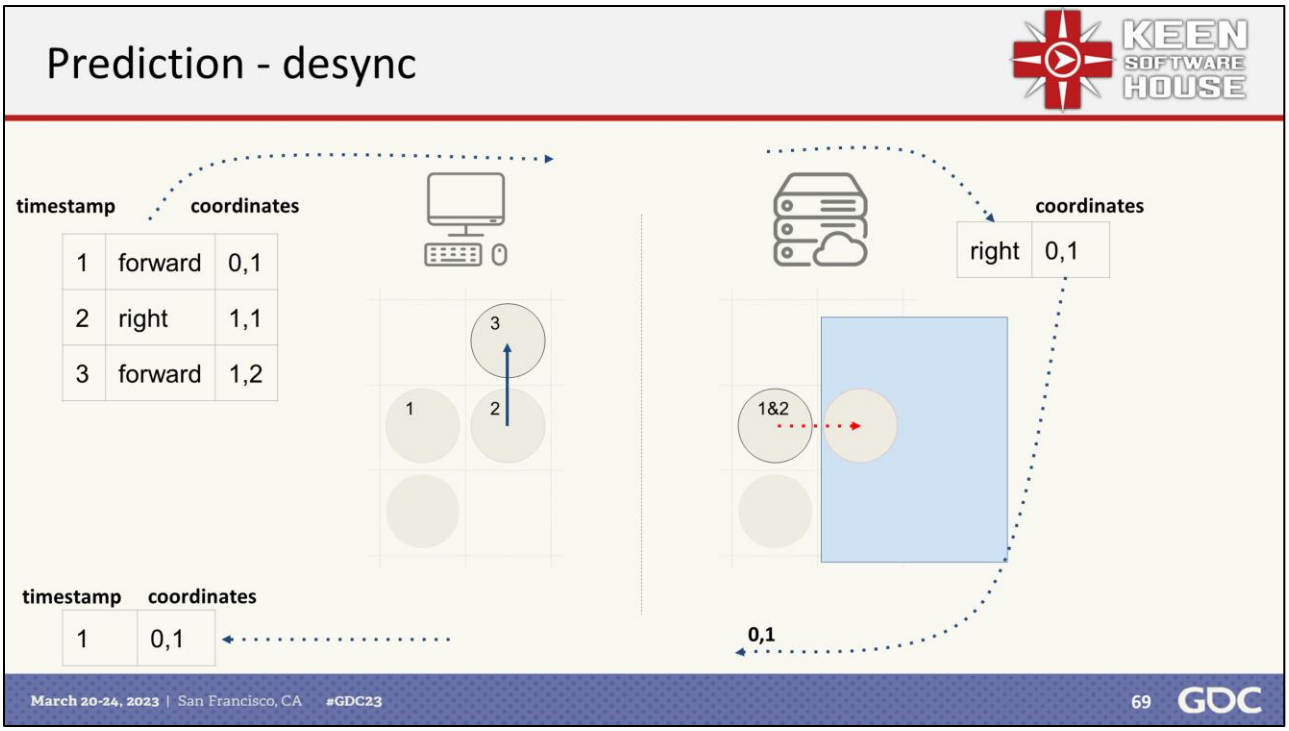


For the next timestep

- User presses the right button
- client sends the button press information to the server
- client simulates character moving right from the last coordinates to 1,1
- And finally client remembers the simulation outcome in the history list

In the meantime, server on the right side

- gets the packet with forward button press from last simulation delivered
- let's say network conditions are good, so we can skip playout delay buffer
- Server simulates the character moving forward to coordinates 0,1
- Server sends the simulation outcome as positions 0,1 to the client

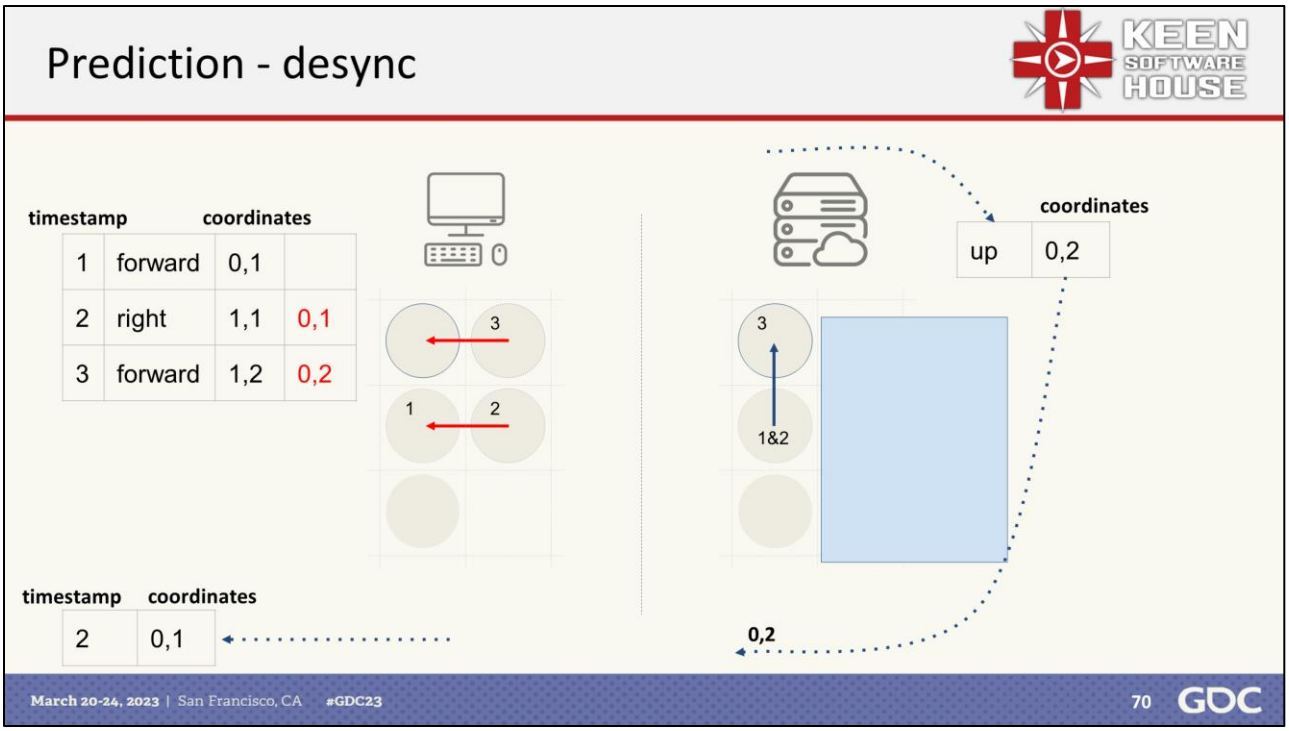


Next frame, on the client:

- User presses the forward button
- client sends the button press information to the server
- client receives the simulation outcome from the server and compares it with history timestamp #1
- It correlates, so we can freely continue without any change
- client simulates character moving forward to coordinates 1,2
- And finally client remembers the simulation outcome in the history list

In the meantime, server on the right side:

- gets the packet with the right button press from the last simulation delivered
- Server simulates the character trying to move right, but hits the obstacle, so the simulation outcome is same coordinate 0,1
- Server sends the simulation outcome to the client



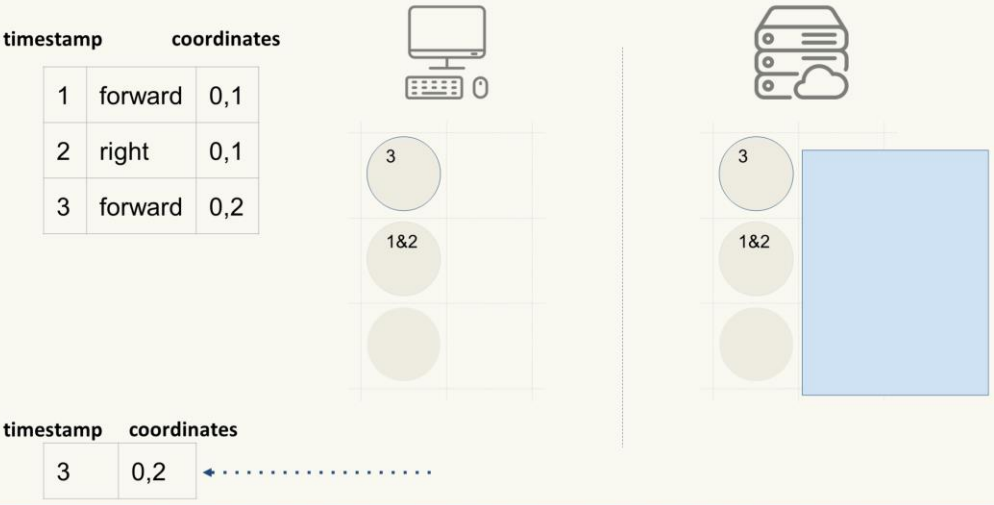
Next frame client on the left:

- client receives the simulation outcome from the server and compares it with history timestamp #2 and it is not correlating, so we have a desync registered, finally
- The correction delta is 0, -1 is applied to both current position and the whole history of positions starting from timestamp #2

Meanwhile, server on the right side:

- gets the packet delivered from the last frame
- Simulates the character moving forward to coordinates 0,2
- And sends the simulation outcome to the client

Prediction - desync

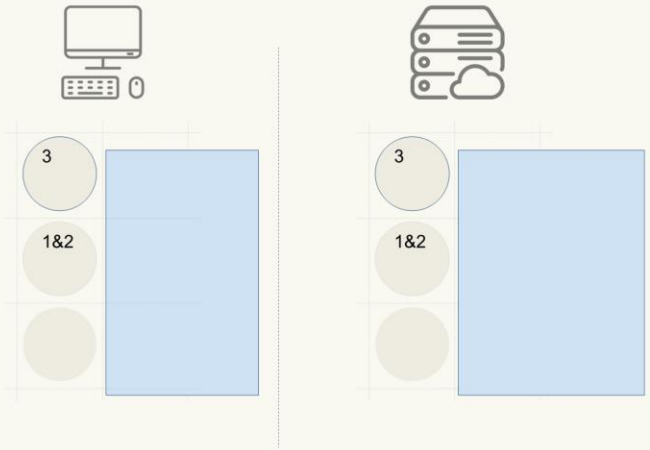


Next:

- client receives the simulation outcome, compares with #3 timestamp and it correlates because we corrected it during previous frame

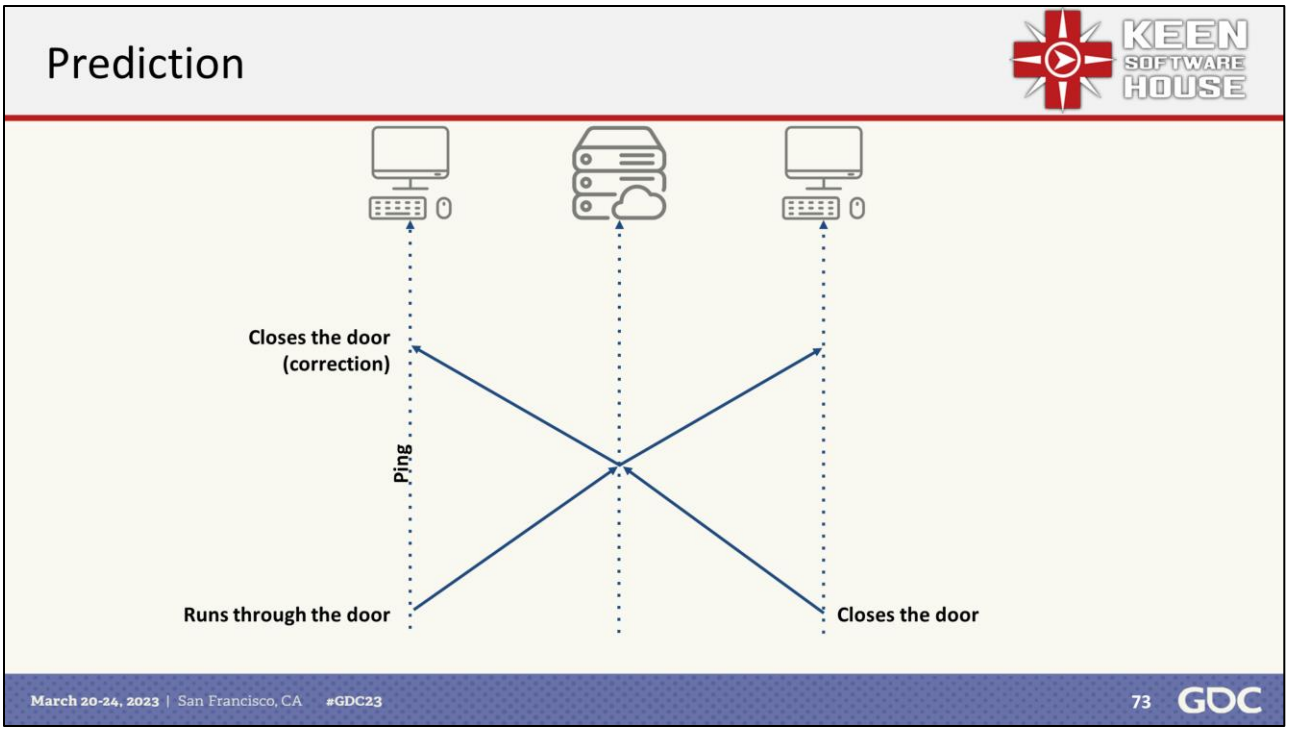
Prediction - desync

timestamp		coordinates
1	up	0,1
2	right	0,1
3	up	0,2



At some point, the other body finishes streaming and appears on the client as well.

This was of course just an artificial case of desync, let’s have a look on some real case.

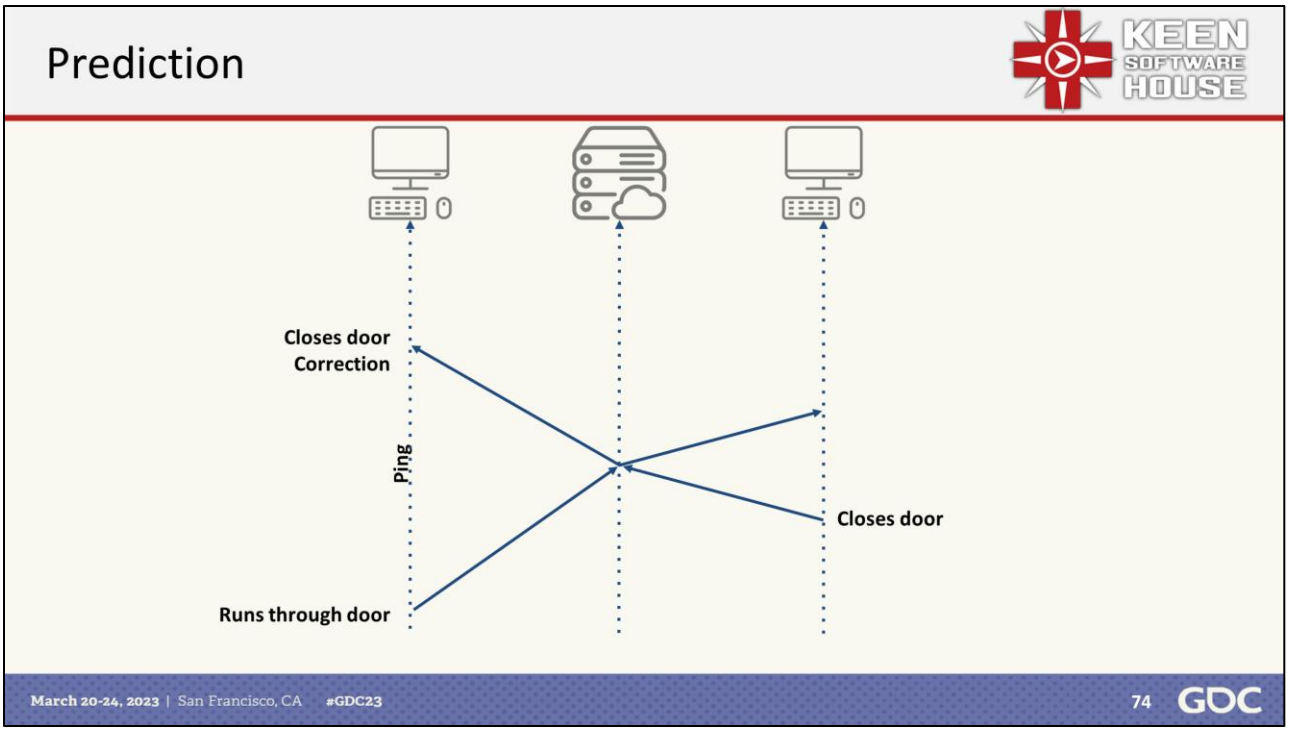


Let’s say, predicted entity on the client runs through the door.

At the same time, other client closes the door.

Server gets both events at the same time and starts closing the door, preventing player from running through them.

Client later gets server’s truth and will have to comply - backing with the client character off the closed room.

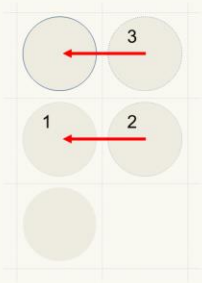


In similar case with better network conditions for one of the clients, other player may even close the door later than and still win.

Life is not fair.

Prediction - desync

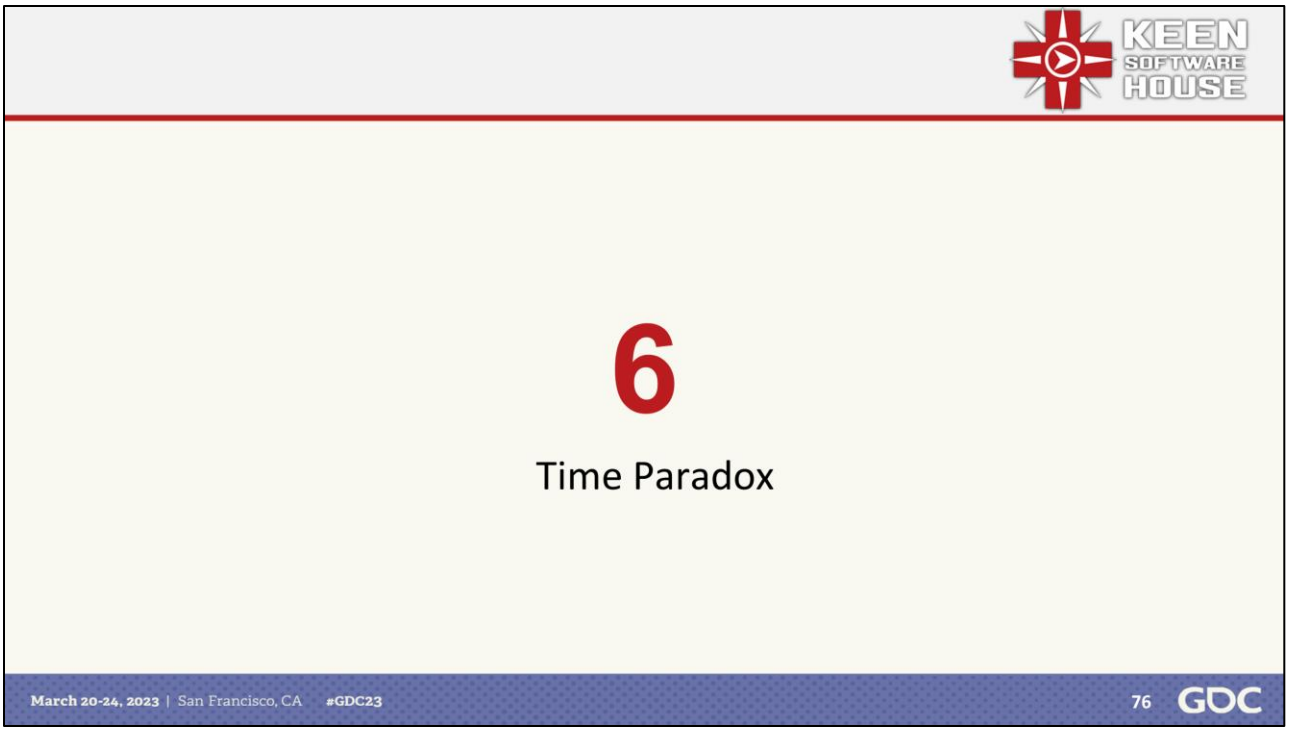
- Determinism not 100%
- Correct only when necessary
- Interpolate corrections



In our examples, we use integers for positions, but in reality, those are floating points and unless you have 100% deterministic simulation, the predicted entity is all the time slightly off.

Messing aroung with the player controlled entity's position can be distracting to the player. We have to make sure to correct only when necessary, in other words when difference is big enough.

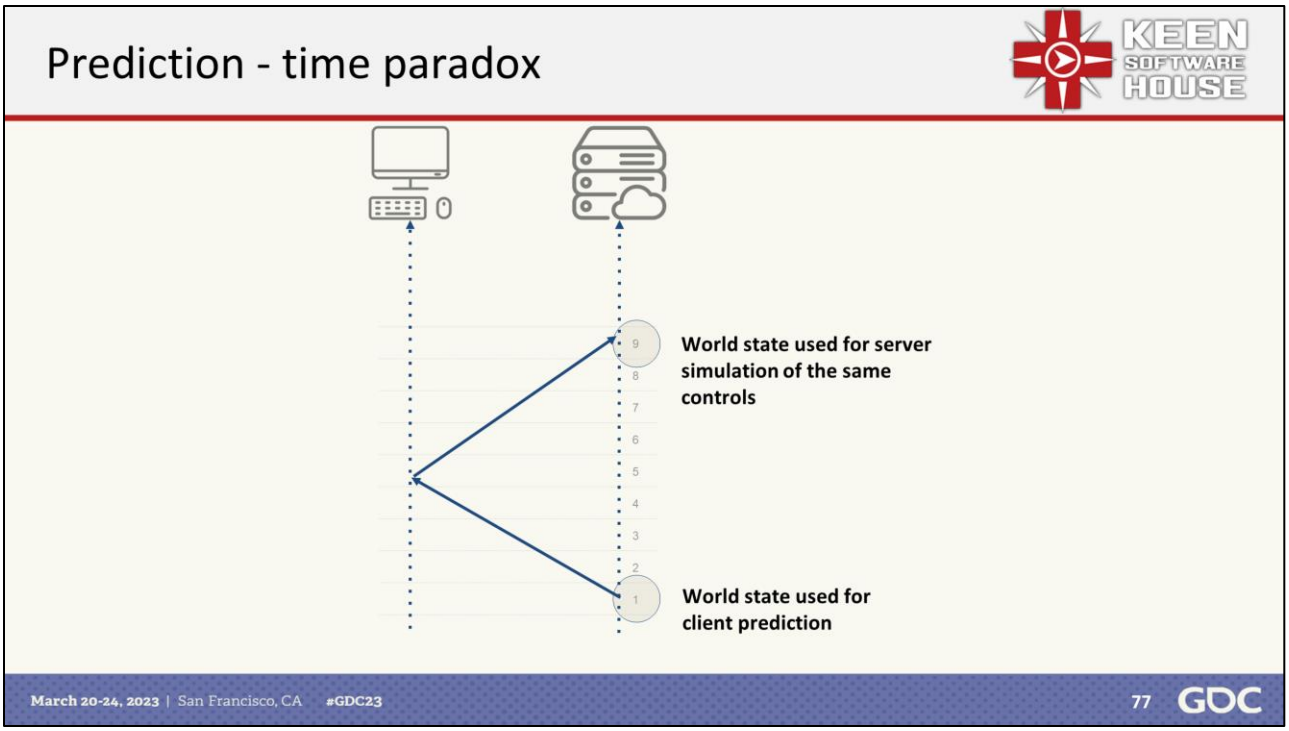
Lastly, the correction should be applied over time with small doses - exponential to its extent.



So far, we were operating in the static world, where only player character is moving.

In our game, it is quite usual the other player is influencing client's simulation. Take an example of player controlling the ship with other players inside.

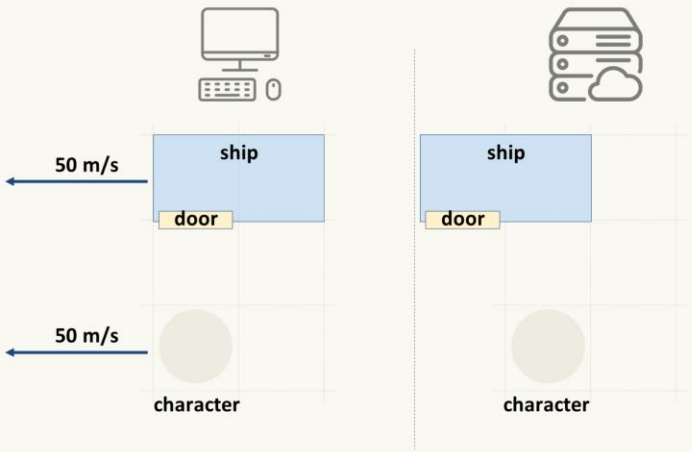
Where the two of these worlds meet, the Time paradox happens.



Clients have to do a prediction based on the current state of the world, but they get positions streamed from the server with latency.

Effectively, server simulates client's controls on a world state, which is one ping time newer than what was the state of the client's world when doing the prediction.

Prediction - time paradox desync



Animated by 2.5m late @ 100ms latency

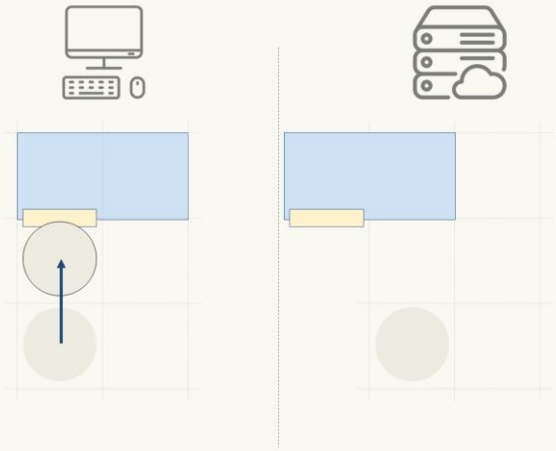
Let’s see an example of character flying alongside of the ship in 50m per second to the left.

Animated ship on client’s machine is 2.5m late to the server’s state.

Character will attempt to board the ship through the door in front of him.

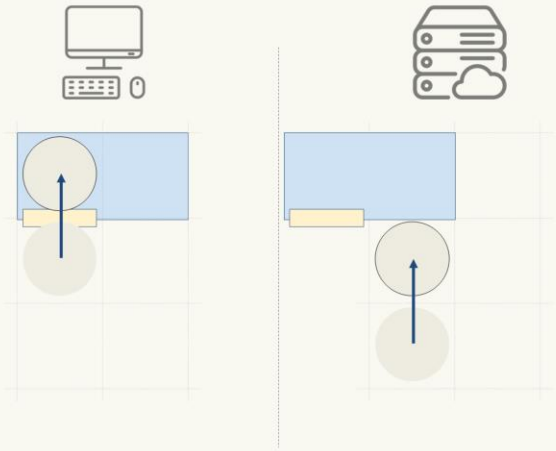
On the left side, you can see the client state, on the right side the server state.

Prediction - time paradox desync



On the client, character will start flying towards the ship.

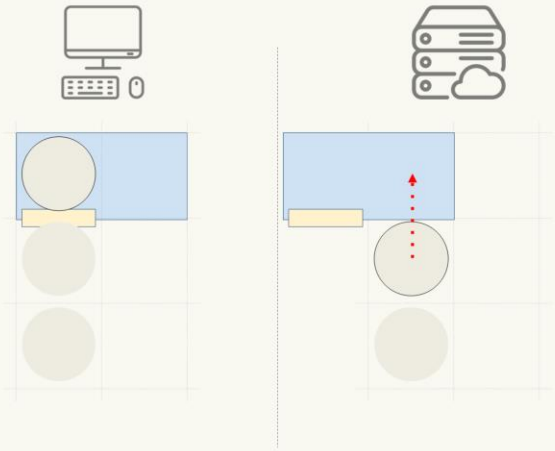
Prediction - time paradox desync



And eventually boards the ship.

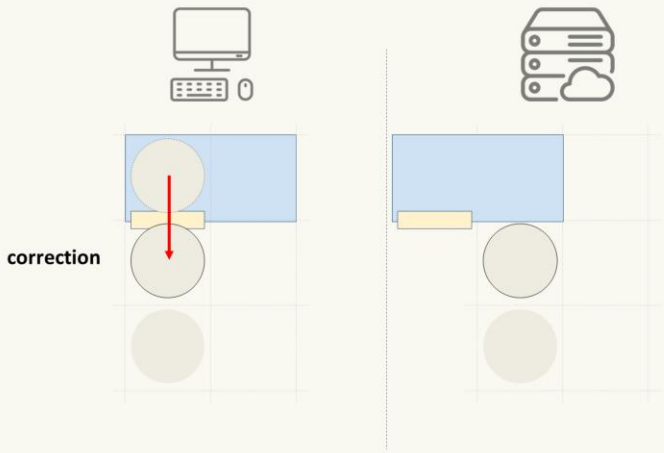
In the meantime on the server, character starts flying towards the ship.

Prediction - time paradox desync

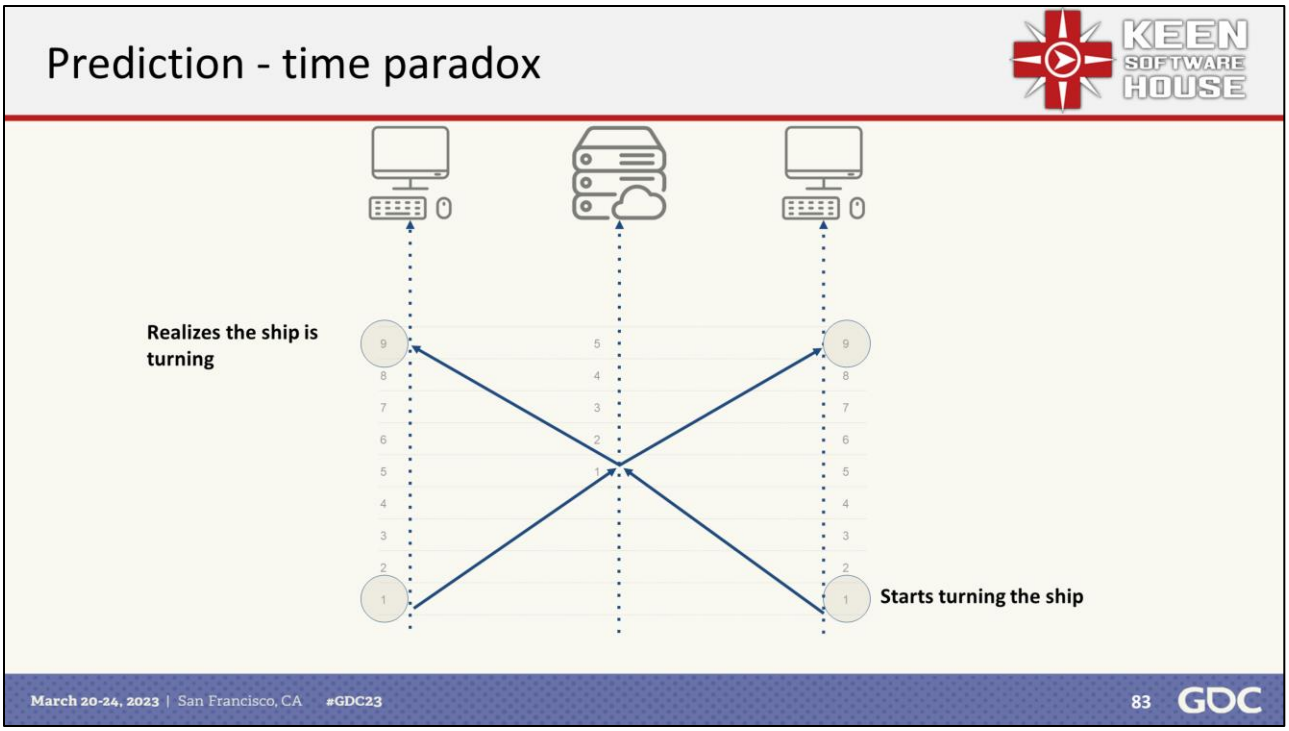


Next frame, it tries to continue in the flight on the server, but hits the wall.

Prediction - time paradox desync



Server sends this outcome to client, which has to back out of the ship to the player's disappointment.

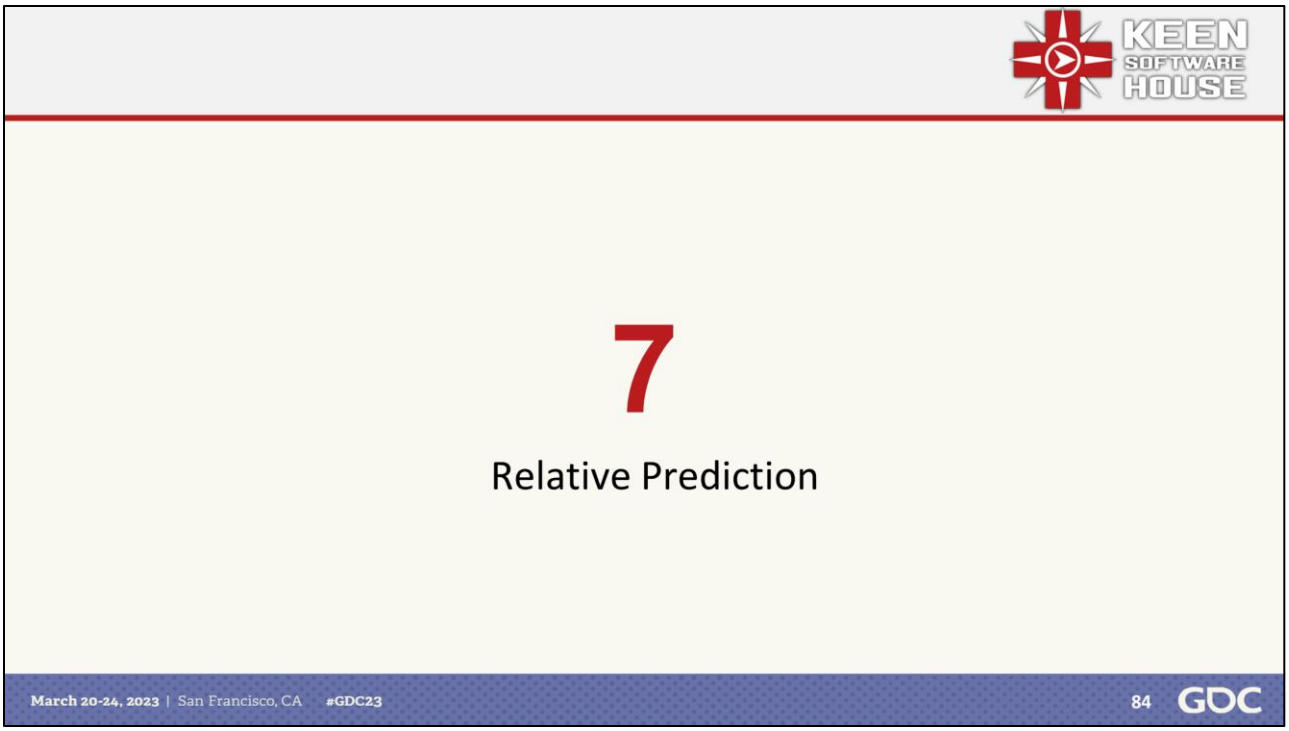


Similar behavior can happen in case of the other player controlling the ship.

The right hand side player starts turning the ship, while left hand side player runs around the ship.

By the time, left hand side player's client realizes
the ship has been turning
many simulation steps has passed.

The client has to start correcting to the different simulation results from the server.



These scenarios are not good for player's experience.

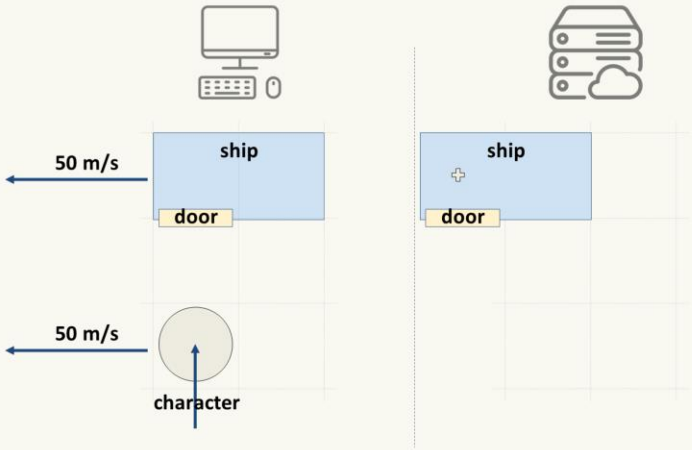
We have to find a way how to be able to operate on the world state as close to server's when it is receiving our controls
including the state changes from other clients.

And that is just impossible as we don't have all the information at hand.

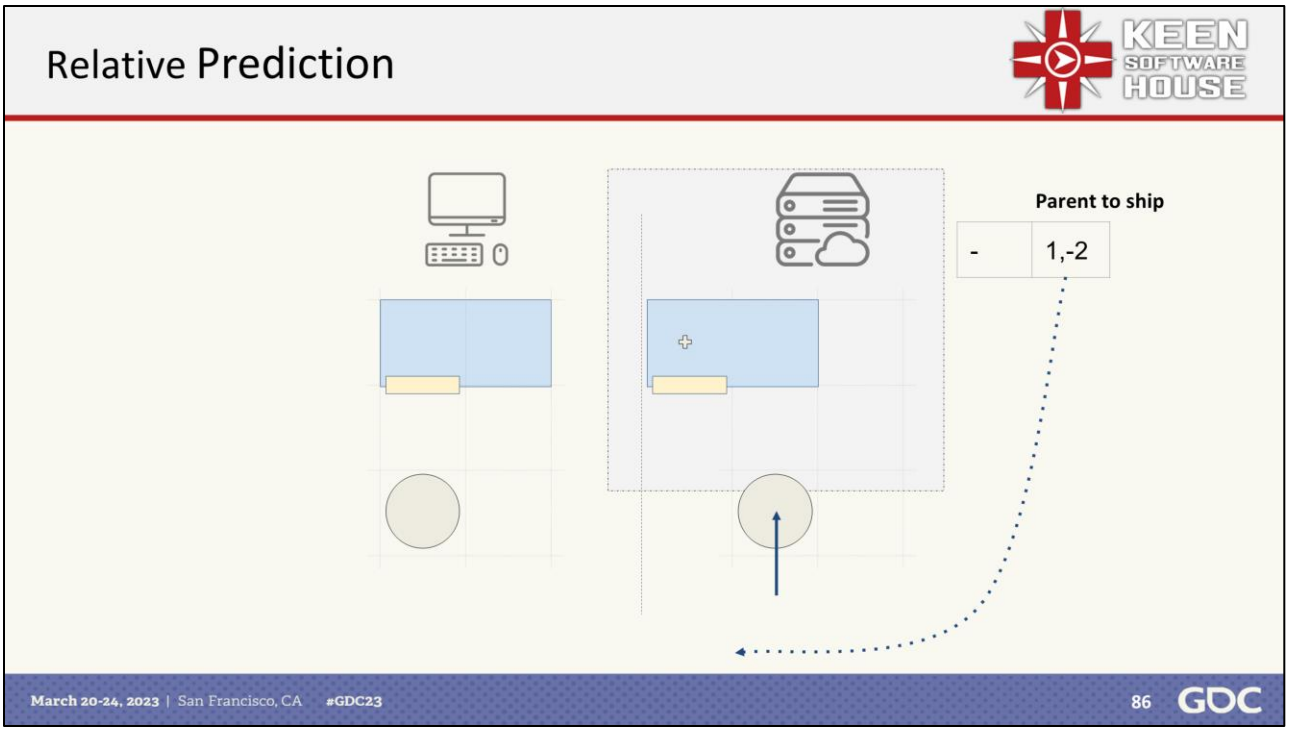
Thankfully, there is a different solution –

And that's **relative prediction**.

Relative Prediction



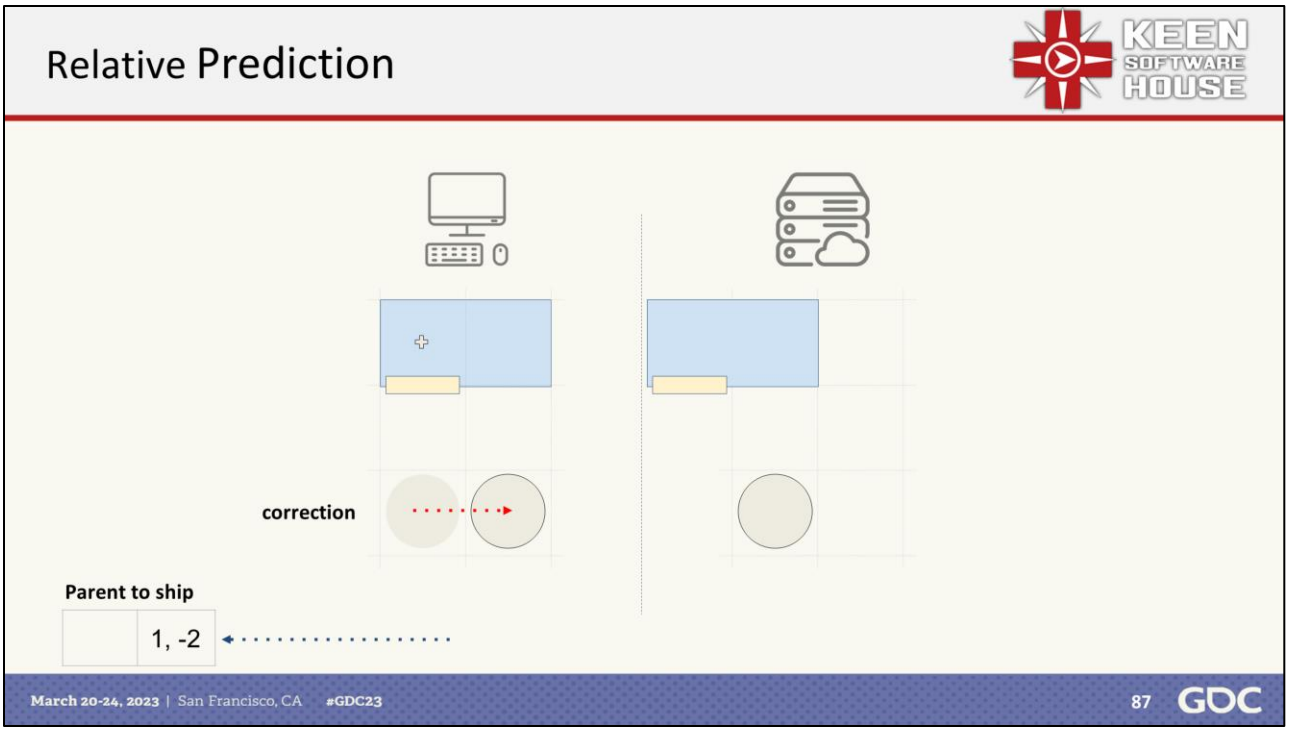
Let's revisit our example of character flying alongside of the ship in 50m per second to the left and character attempting to board it.



When character starts flying towards the ship
- server detects close proximity,
- parents player to it
- And notifies client.

It is very similar concept to the hierarchy I have introduced for the entities connected via physical constraints. The main difference is that this relation is just **logical** for prediction purposes.

From now on, all character's positions will be delivered to the client as **relative** transforms to the ship.

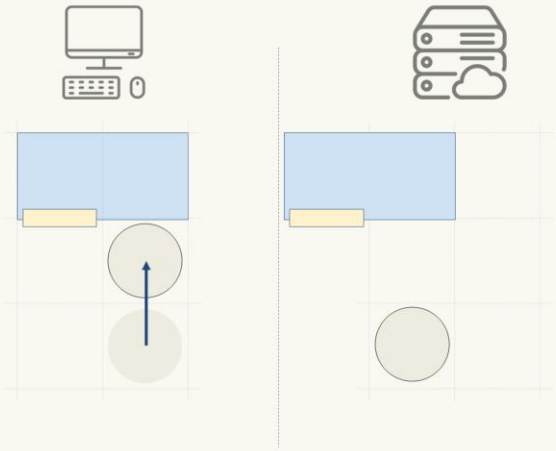


When parented, client corrects player's position to be relatively the same to the parent ship as on the server.

This can be quite distracting to the player as the displacement between animated and predicted entities can be many meters.

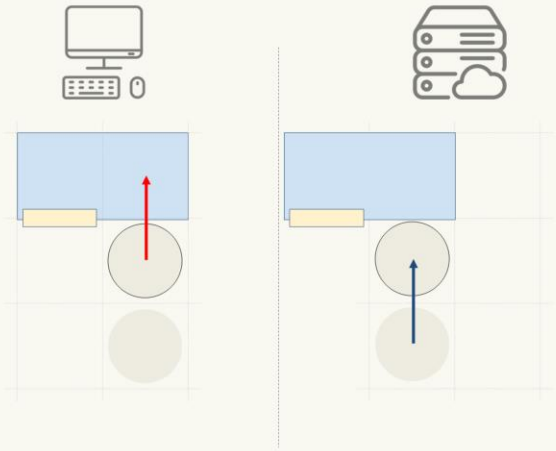
It has to be smoothed out via interpolation.

Relative Prediction



Now, when character proceeds to fly towards the ship.

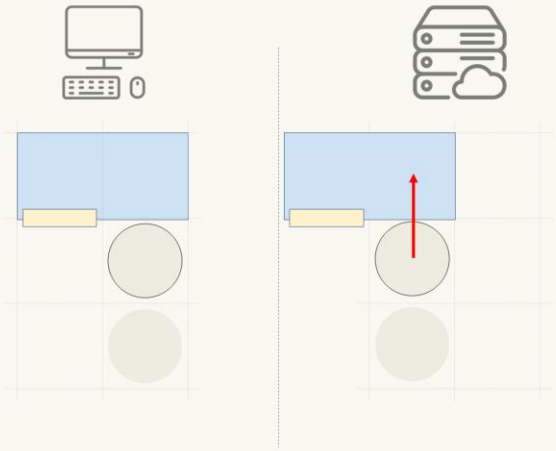
Relative Prediction



It correctly bumps to the wall on the client.

Since player sees the wall he can only blame his poor jetpack controlling skills and not the game.

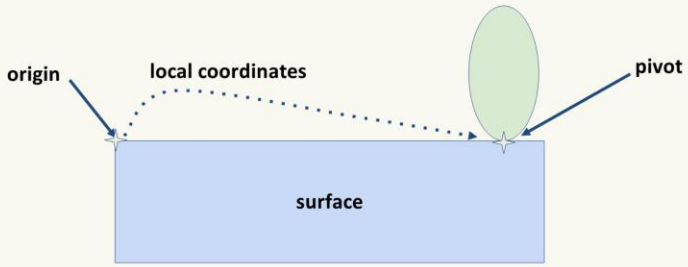
Relative Prediction



Same happens later on the server and no further correction has to be made.

Finding a proper parent

- Character Controller Pivot
  - Transformation is relative



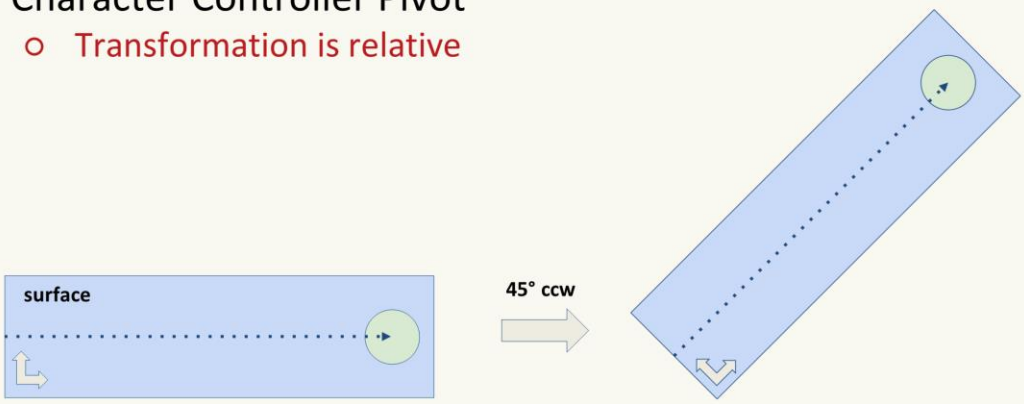
Let’s see some rules on how we are finding proper parents.

Character walking on any surface is parented to that very surface.

From the moment it stands on the surface, its coordinates are streamed to the client relative from surface’s origin to character’s pivot.

Finding a proper parent

- Character Controller Pivot
  - Transformation is relative

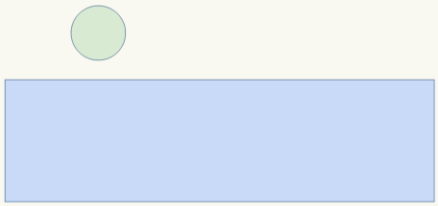


Let’s see an example of different player rotating the ship while our character is walking on it.

We don’t mind it anymore, since our transforms are always relative to the parent, no matter how ship will twist and turn.

Finding a proper parent

- Character Jetpack or Controlling a ship

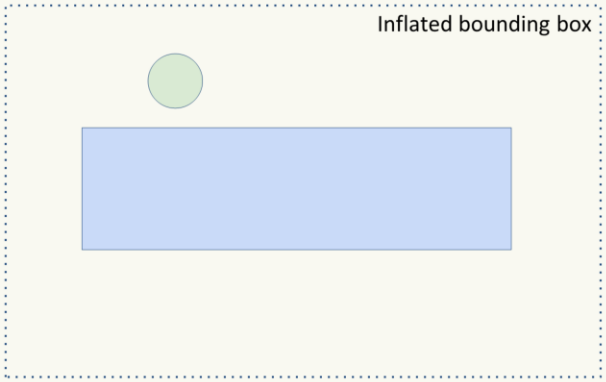


Another case is
- flying character
- or other player controlled entity like small ship

approaching bigger entity.

Finding a proper parent

- Character Jetpack or Controlling a ship
  - Inside inflated bounding box
  - Similar velocity at point



We will inflate the bounding box of the big ship as a hint for parenting.

When velocities get to similar values, it indicates player wants to land or dock to the other ship, thus needs to be parented.

We take velocities into consideration, not to parent to entities which are just passing by in different directions, as it would affect the positions negatively.

In case of multiple overlapping inflated bounding boxes and the similar velocities at point, we will parent to the bigger bounding box.

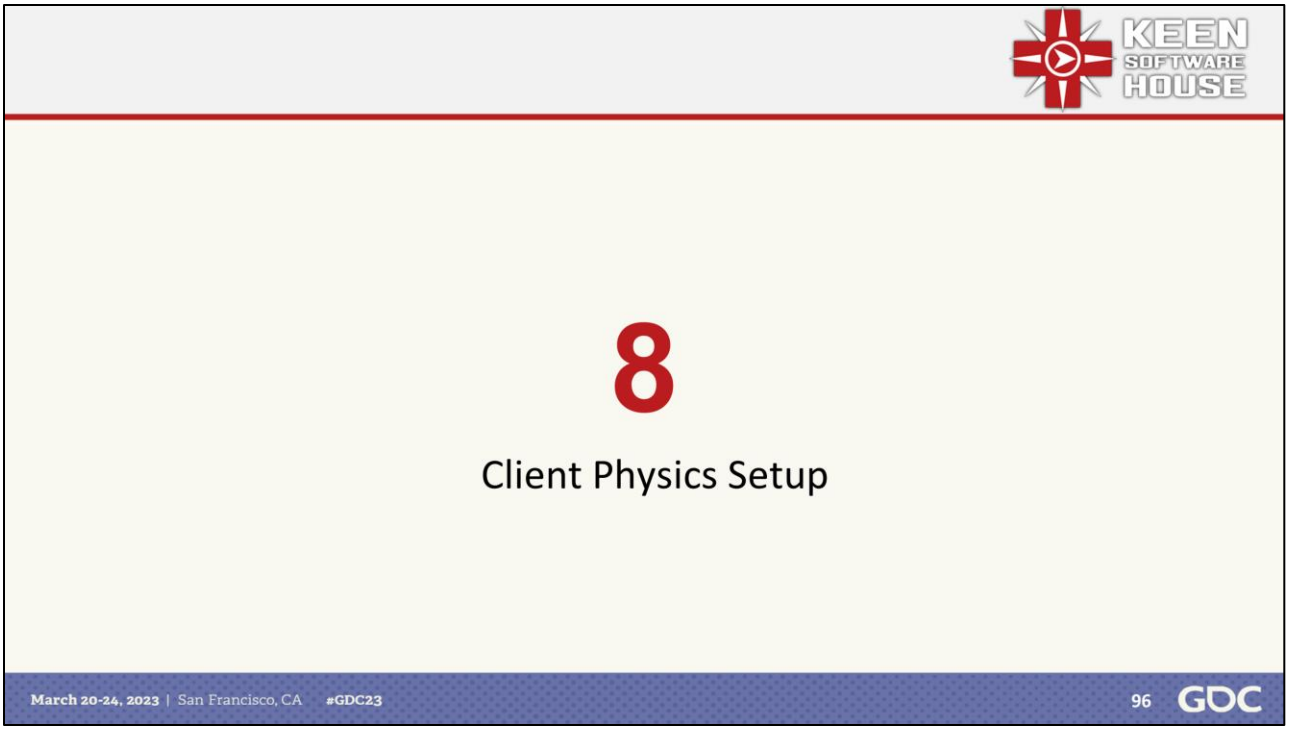
Relative Prediction

- Stream transformations relative to parent
- Corrections in local space
- Convert to world space

To wrap it up:
We stream transforms relative to parent, in his local space.

We apply corrections in local space as well

Only after that we convert to the world space.



Let’s now have a look on some specifics of the physics setup.

Since client is animating all of the world entities but also attempting to simulate the controlled entity,
it warrants special treatment in physics,
very different to how the world is simulated on the server.

Client Physics Setup

- Animated entities as static RBs
- Switch entity to predicted when possessed
  - Dynamic RB
  - Relative velocity
  - Inheriting parent's motion

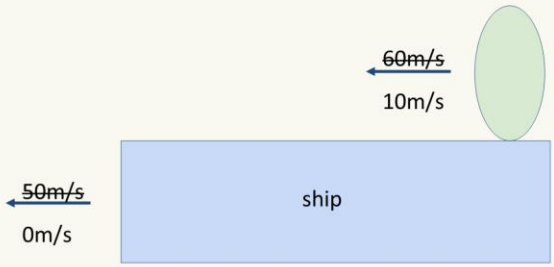
All animated entities are static rigid bodies on the client.

When we start predicting some entity, it switches to dynamic rigid body.

From that moment,
- we start streaming relative transforms and
- Velocities
- from the server
- And we inherit parent's transform changes.

Frame update

- Initial state



Let’s see it on example of steps taken during one frame on the client.

On server, ship is moving 50m/s, while character is running in the same direction at 10m/s, totalling 60m/s velocity.

On client though, static bodies does not have any velocity in the physics simulation, so ship moves 0m/s.

Character as a predicted entity has just its own motion relative to parent, which means 10m/s.

Frame update

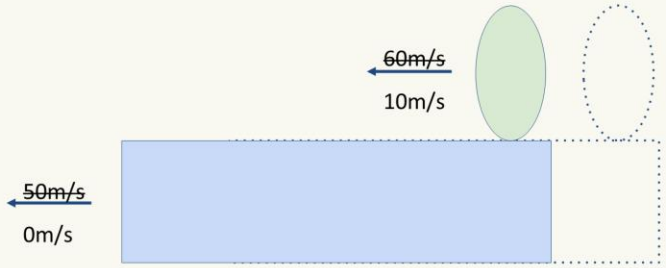
- Entity animation applied before simulation

Client frame update starts with animating all non-predicted entities,

so ship in this case.

Frame update

- Propagate delta through hierarchy

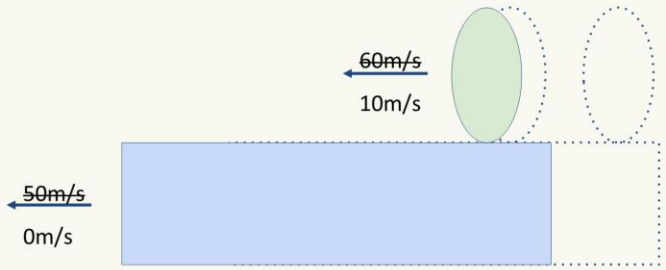


Then, animation delta is propagated to the predicted entity,

so character gets moved as well

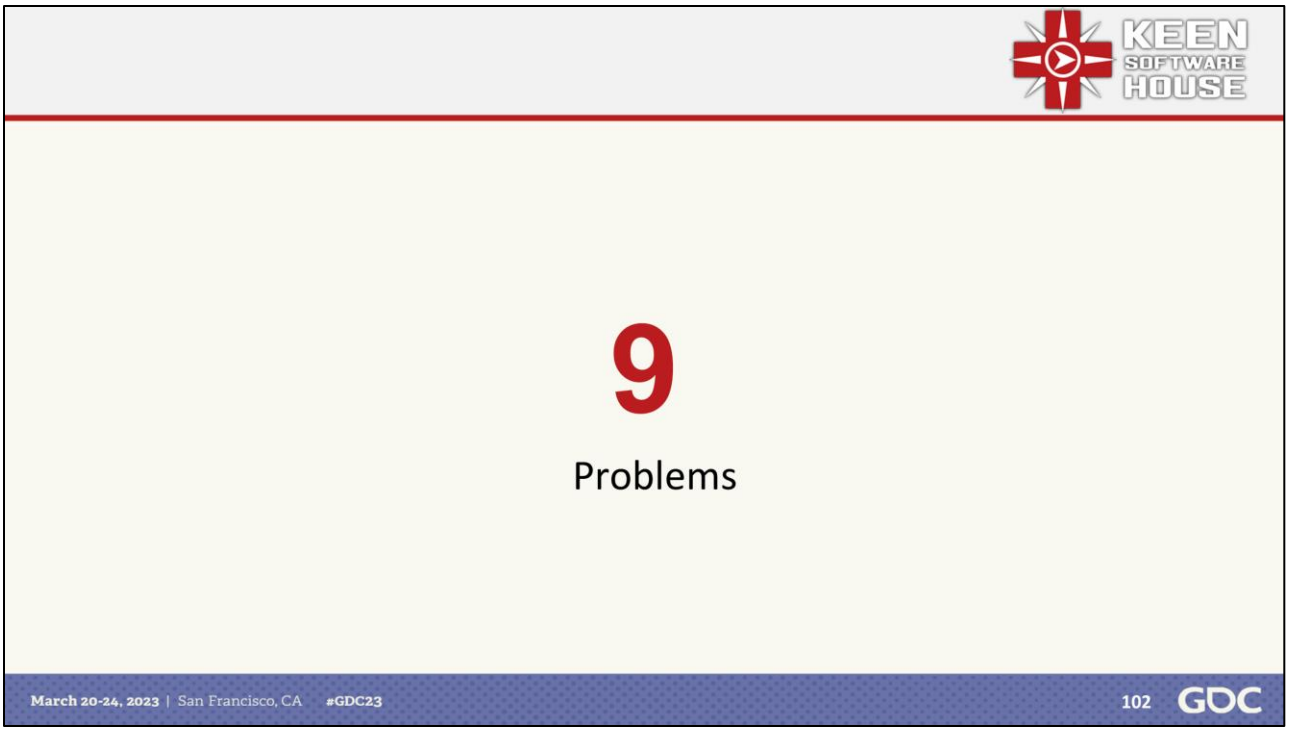
Frame update

- Result of physics simulation step



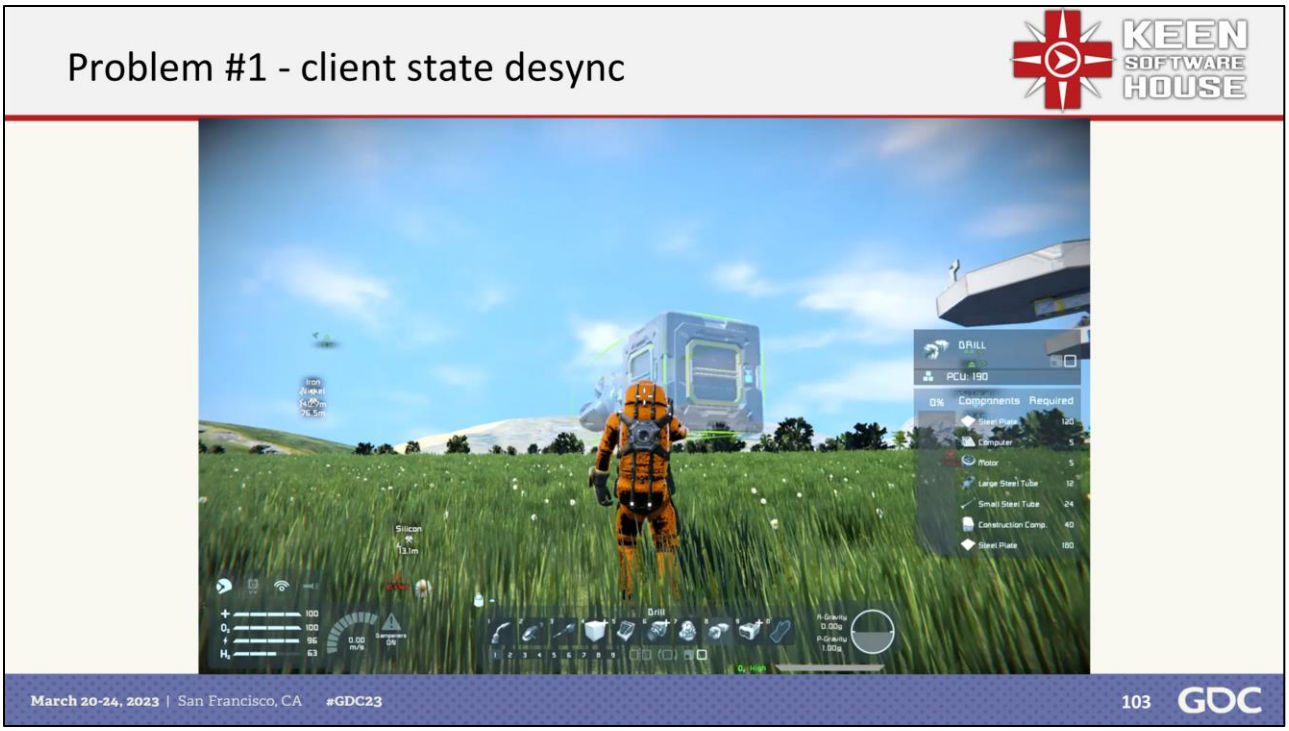
Lastly, physics simulation runs for predicted entity, applying run velocity
<pause>

And that's it, easy.



So we have it figured out nicely and implemented. The theory is sound.

What can possibly go wrong?



So, this is a typical situation of desync between client and server state.

The hole is dug only on client

whereas character on server stands still on the surface.

This causes client’s position being reset periodically to server’s ground truth.

Problem #1 - client state desync

- Bug fix: client state should be in sync with server

Reason is obviously a bug in a code,

client and server world state should be in sync at all times.

Problem #1 - client state desync

In this case, client decides that character should fall down because of slight difference in simulation outcome,

- but server character stays hanging at the edge of the surface.
- As we detect the big desync, we will reset to the server position,
- but the character does not get attached to the surface and
- starts falling down immediately,
- creating neverending fall / reset loop.

Problem #1 - client state desync

- Improvement: sync character physics controller state

The solution is to let the client know the state of the server's character physics controller.

In this case - state of being attached to the surface

and apply it when being reset on the client.

Problem #2 - constraint chains

Next set of problems is caused by players being able to create complex contraptions and control them.

This is a strandbeest-like walker.

It is created using multiple chains of rotors.

Problem #2 - constraint chains

If we try to predict it in multiplayer,

- the prediction error accumulates and
- corrections breaks the physics simulation.

Problem #2 - constraint chains

- New rule: do not predict entities with constrain chains

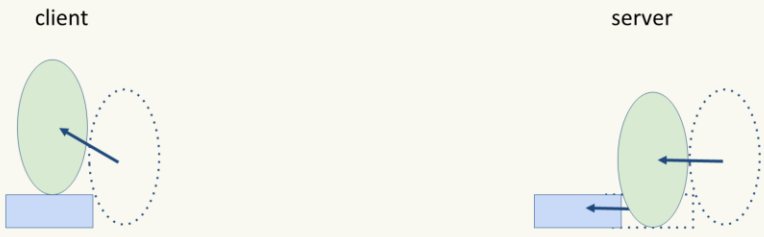
Solution: do not predict constrain chains
- switch to animation.

- As constrain contraptions have latency by themselves,
- the network latency is not a big issue anyway.

Problem #3 - small entities

Next set of issues comes from interaction with small entities.

Problem #3 - small entities



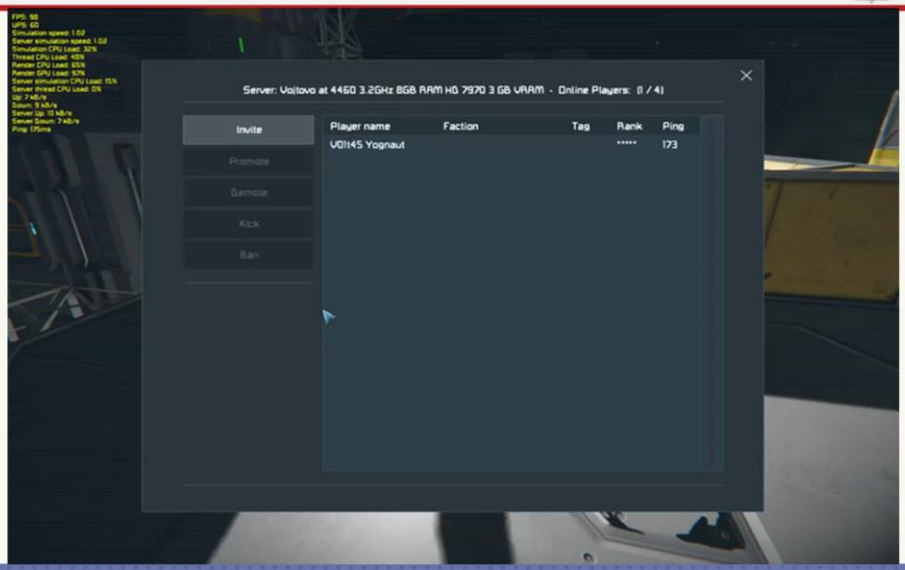
- New rule: parent entity has to be big enough

Let's break down the proble
- (point) the entity on client is static, so character will not push it, but rather walks on top of it.
- (point) server side, the entity gets pushed by character.

This situation will create a desync.

So we have to come up with a new rule:
- Entities affectable by controlled entity should not be used as a parent.
- In other words do not parent to small or light entities.

Problem #3 - small entities



This is similar case, where the character parents to the nearby entity.

We should adjust the rule to parent only to entities we stand on.

Problem #3 - small entities

And of course, during testing period, the definition of what is small and light had to be adjusted,

as QA was coming up with new cases over time.

Problem #4 - physics non determinism

Let’s have a look on another set of problems regarding centrifugal forces.

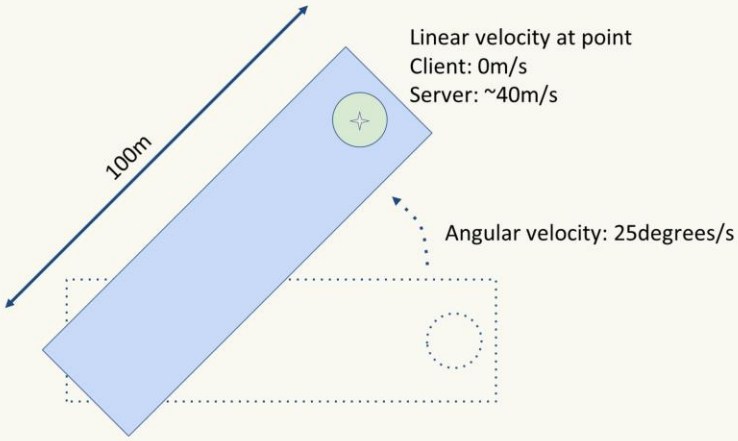
Locally simulated physics does not simulate same centrifugal force,
- since the entity we stand on have zero velocity.

As you can see, it creates major desyncs.

Centrifugal force

- New rule: when in doubt*, animate

* too many corrections



In this case, angular velocity of the ship is 25degrees per second, which means ~40m/s velocity at the position of character on the server.
- which is exactly zero on the client

When player jumps, it has very different outcome.

So, we have to introduce a new rule:
- when client simulation outcomes are sufficiently different to server's, we switch off prediction and resort to animation.

Problem #4 - physics non determinism

The animation has worse latency, but without any desyncs.

Problem #5 - physics non determinism

Another case of physics non determinism comes from interaction of multiple entities.

In this case, the controlled entity is the vehicle.

Since it is controlled, it has dynamic rigid body on both client and server.

(point) There is a small disconnected block inside of it, which is
- dynamic on the server as well
- but it is not controlled on client, so it is static and animated on the client

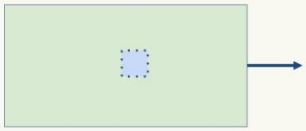
When player starts controlling the vehicle, the small block acts as a hard stop for prediction locally,

creating erratic movement

Dynamic vs static RB

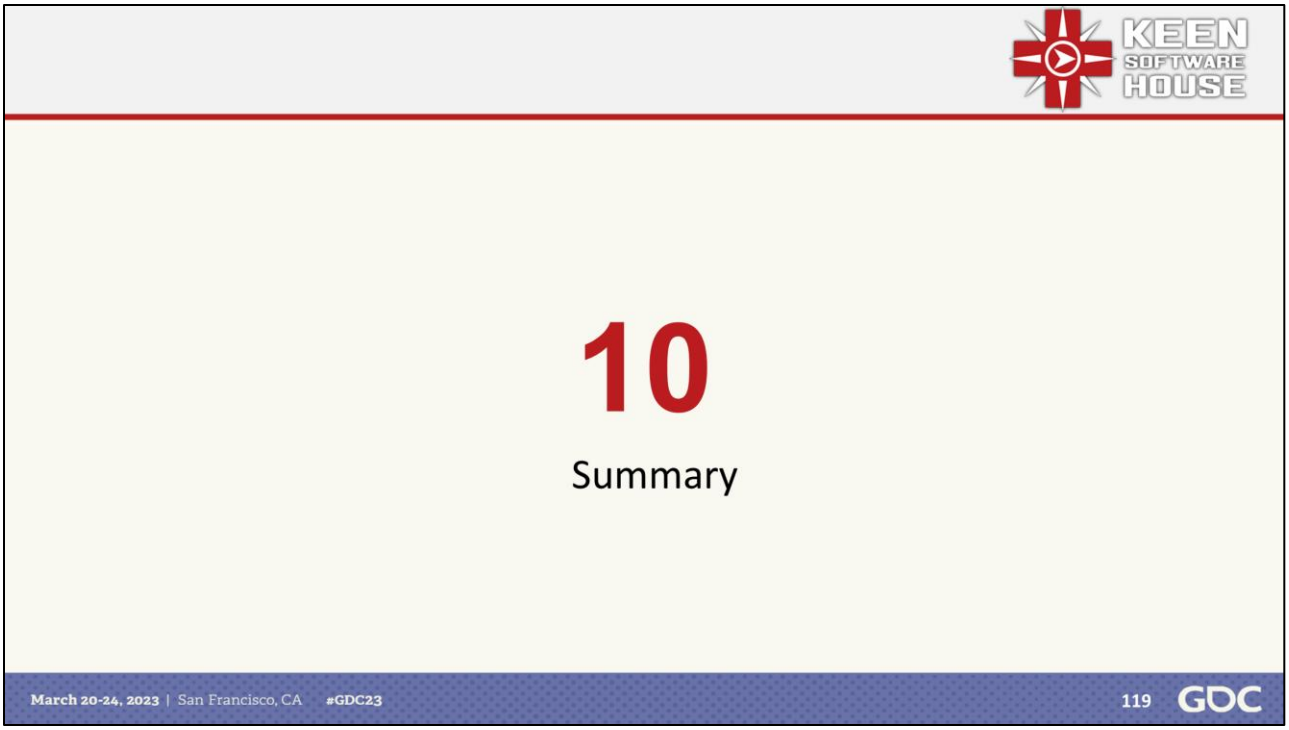
- Reinforced rule: when in doubt*, animate

* too many contacts with supposedly dynamic entities



Solution is to disable prediction when we detect too many contact callbacks between - predicted entity and other entities that are dynamic on the server, but static on the client.

We can switch back to prediction after some time of not penetrating any other surface.



This was just a glimpse of possible problems this setup can bring.

Nevertheless, the solution allowed us to
- reduce latency in common scenarios by significant amount,
- so I believe it is a sound solution, worth invested time.

Let’s bring it all together.

Summary

1. Stream transforms to clients

We are streaming transforms of entities from server to clients

Summary

1. Stream transforms to clients
    a. Hierarchically

And we do it hierarchically for better interpolation outcomes.

Summary

1. Stream transforms to clients
   a. Hierarchically
2. Interpolate / extrapolate on clients

We interpolate or extrapolate transforms on the clients

Summary

1. Stream transforms to clients
   a. Hierarchically
2. Interpolate / extrapolate on clients
   a. As static RBs

In physics on client, all the entities are represented as static rigid bodies.

Summary

1. Stream transforms to clients
   a. Hierarchically
2. Interpolate / extrapolate on clients
   a. As static RBs
3. Predict and correct controlled entities

Except for controlled entities,

which are simulated and

corrected once server delivers its results

Summary

1. Stream transforms to clients
   a. Hierarchically
2. Interpolate / extrapolate on clients
   a. As static RBs
3. Predict and correct controlled entities
   a. Relative to parents

Corrections are delivered relative to close by entities,
so called parents,
to prevent desyncs from having old world state on clients.

Summary

1. Stream transforms to clients
   a. Hierarchically
2. Interpolate / extrapolate on clients
   a. As static RBs
3. Predict and correct controlled entities
   a. Relative to parents
4. Simulate predicted entities physically

Predicted entities are physically simulated on client

Summary

1. Stream transforms to clients
   a. Hierarchically
2. Interpolate / extrapolate on clients
   a. As static RBs
3. Predict and correct controlled entities
   a. Relative to parents
4. Simulate controlled entities physically
   a. As dynamic RBs

As dynamic rigid bodies

<pause>

Questions?

#VRAGE3

- DirectX12
- GPU Driven Pipeline
- Raytraced GI
- Data Oriented Design
- Server - Client by default
- All in one Editor tool

Join our team!

@janhlousek

And that’s it from me. Thank you for listening. And now

- hit me with any questions.

Later I will be available outside for any follow up discussions.

Also, Keen Software House have a booth at GDC Expo near Connect Lounge. Please, come visit us.

- we are working on new generation of our inhouse engine with many improvements you can see on this slide
- Let me know if you would be interested in knowing more on my twitter