Overview

• Goal:
  • To explain how latency impacts the decisions of how to ensure consistency. Latency implies that clients cannot all act the same way because they don’t have consistent information.

• Topics:
  • Synchronising state with latent communications
  • Playout delays, local lag
  • Extrapolation and dead reckoning
Naïve (But Usable) Algorithms

• Most naïve way to ensure consistency is to allow only one application to evolve state at once
• One application sends its state, the others wait to receive, then one proceeds
• Is a usable protocol for slow simulations, e.g. games
  • Not that slow – moves progress at the inter-client latency
• Potentially useful in situations where clients use very different code, and where clients are “un-predictable”
Lock-Step (1)

- If all clients can deterministically on the input data
- Then a more useful form lock-step for NVEs & NGs is that everyone exchange input, proceed once you have all the information from other clients
- But for many simulations, each step is only determined by user input, so can just communicate input
DOOM 1 – iD Software

Doom Client\textsubscript{A}:
- Read Input
- Receive Input
- Simulate
- Rendering

Doom Client\textsubscript{B}:
- Read Input
- Receive Input
- Simulate
- Rendering

Doom Client\textsubscript{C}:
- Read Input
- Receive Input
- Simulate
- Rendering
If the simulation is complex or non-deterministic, use a server to compute the state

Clients are locked to the update rate of the server

Note that own input is delayed
Quake 1
(Pre-QuakeWorld)
Optimistic Algorithms

- Conservative simulations tend to be slowed paced
- Optimistic algorithms play out events as soon as possible
- Of course, this means that they can get things wrong:
  - They may receive an event that happened in the past
  - To fix this they rollback by sending UNDO events
  - For many simulations UNDO is easy (just move something)
Client Predict Ahead

- A form of optimism: assume that you can predict what a server (or another peer) is going to do with your simulation
- Very commonly applied in games & simulations for your own player/vehicle movement
- You assume that your control input (e.g. move forward) is going to be accepted by the server
- If it isn’t, then you are moved back Note this isn’t *forwards in time* but a prediction of the current canonical state (which isn’t yet known!)
Move? P₀ to P₁
Move? P₁ to P₂
Move? P₂ to P₃
Move? P₃ to P₄
Move? P₀ to P₁
Move? P₁ to P₂
Move? P₂ to P₃
Move? P₃ to P₄

Client

Server
Client \( A \)

- \( P_0 \) to \( P_1 \)
- \( P_1 \) to \( P_2 \)
- \( P_2 \) to \( P_3 \)
- \( Q_1 \)

Server

- \( P_0 \) to \( P_1 \)
- \( P_1 \) to \( P_2 \)
- \( P_2 \) to \( P_3 \)
- \( Q_1 \) to \( P_1 \)
- \( P_1 \) to \( Q_1 \)
- \( Q_1 \) to \( P_2 \)
- \( P_2 \) to \( P_3 \)
- \( Q_1 \) to \( Q_1 \)

- Move? \( P_0 \) to \( P_1 \)
- Move? \( P_1 \) to \( P_2 \)
- Move? \( P_2 \) to \( P_3 \)
- Move? \( P_1 \) to \( Q_1 \)
- FailMove \( P_1 \) to \( Q_1 \)
- FailMove \( P_1 \) to \( Q_1 \)
Extrapolation Algorithms

Because we “see” the historic events of remote clients, can we predict further ahead (i.e. in to their future!)

This is most commonly done for position and velocity, in which case it is known as dead-reckoning

You know the position and velocity at a previous time, so where should it be now?

Two requirements:

- Extrapolation algorithm: how to predict?
- Convergence algorithm: what if you got it wrong?
Dead Reckoning: Extrapolation

- 1\textsuperscript{st} order model
  \[ P_1 = P_0 + (t_1 - t_0).V_0 \]

- 2\textsuperscript{nd} order model
  \[ V_1 = V_0 + (t_1 - t_0).A_0 \]
  \[ P_1 = P_0 + (t_1 - t_0).V_0 + \frac{1}{2}.A_0.(t_1 - t_0)^2 \]
When to Send Updates

- Note that if this extrapolation is true you never need to send another event!
- It will be wrong (diverge) if acceleration changes
- BUT you can wait until it diverges a little bit before sending events
- The sender can calculate the results as if others were interpolating (a ghost), and send an update when the ghost and real position diverge
1st Order Model
2^{nd} Order Model
Convergence Algorithm

- When they do diverge, you don’t want the receiver to just jump: smoothly interpolate back again
- This is hard:
  - Can linearly interpolate between old and new position over time, but vehicles don’t linearly interpolate (e.g. could mean slipping or even going through obstacles)
Convergence Appears as Sliding Motion

Blending between the old ghost and new ghost over several frames
Interpolation

- Extrapolation is tricky, so why not just interpolate?
- Just delay all received information until there are two messages, and interpolate between them
- Only adds delay equal to the time between sending packets
Interpolation & Playout Delays

• Extrapolation is tricky, so why not just interpolate?
• Just delay all received information until there are two messages, and interpolate between them
• Note that jitter is not uniform, you need to be conservative about how long to wait (if a packet is late you have no more information to interpolate, so the object freezes)
• NVEs and NGs thus sometimes use a _playout delay_
• Note that if you use a playout delay on the clients own input, then all clients will see roughly the same thing at the same time!
Playout Delay

Maximum latency

Interpolate $P_0 \rightarrow P_1$

Playout delay
Non-Linear Interpolation

- Need to consider several aspects
- Object movement is not linear, so could use quadric, cubic, etc. by keeping three or more updates
- Note that this causes more delay
- However, if update rate is fast, the trade off is that movement is apparently a lot smoother
Non-Linear Interpolation
• You can’t beat latency, so you need to deal with the consequences
• Over LAN you can just do a lock-step or simple synchronisation scheme
  • Server can calculate all behaviours
• Over a WAN you can’t live with the implied delays, so it’s comes to use optimistic schemes
• Alongside that, one might delay playouts and interpolate historic events to ensure that every site see a similar state at the same time.