eecs489 Computer Networks

Lecture 34:
Multiplayer Gaming

Networking in Games

Differentiate between in-game networking and backend infrastructure

Backend infrastructure:
- lobby where gamers meet
- authentication and key checking
- accounting and billing
- ranking and ladder
- reputation and black list
- buddy lists, clans, and tournaments
- mods and patches management
- virtual economy
- beware of DDoS

Issues: scalability, adapting to failure, security

In-game Networking Topics

Topology:
- client-server or peer-to-peer

Computational model:
- distributed object vs. message passing

Bandwidth requirement

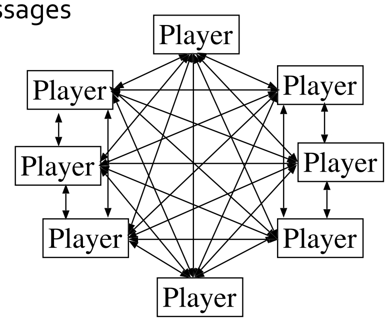
Latency requirement

Consistency

Which transport protocol to use?
- TCP, UDP, reliable UDP

Peer-to-Peer

Peer-to-peer with $O(N^2)$ unicast connections:
- each player is connected directly to all other players
- each player simulates the whole world
- advantages: reduced latency, no single point of failure
- disadvantages: easier to cheat, not scalable: each client must send and receive N-1 messages



Client-Server

Two flavors:

- ad-hoc servers: death match
- dedicated servers: MMOG

Two types of clients:

- clients simulate world, server has authoritative state: allows for client-side dead reckoning (Quake III/Half-Life)
- clients as dumb terminal, all simulations at server: useful for thin clients, e.g., cell phones, and persistent-world MMOG

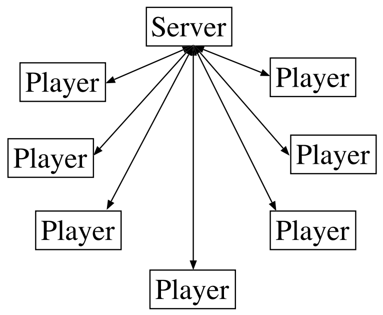
Client-Server

Advantages:

- each client sends only to server, server can aggregate moves
- dedicated servers makes cheat-proofing easier
- server can be better provisioned
- persistent states (for MMOG)

Disadvantages:

- longer delay
- server bottleneck
- single point of failure
- needs server management

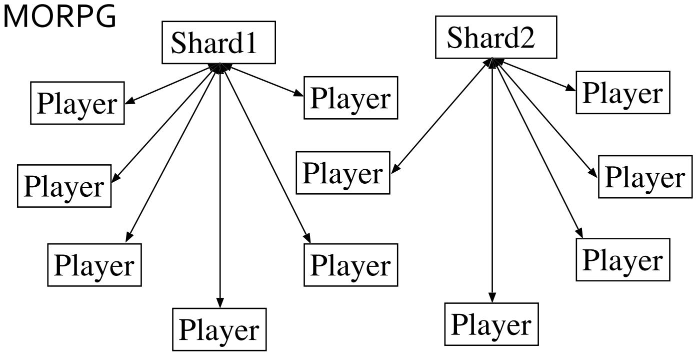


Server Architecture: Replicated

The world replicated at each server (shard)

- each shard contains an independent world
- players go to specific shard

Most MMORPG

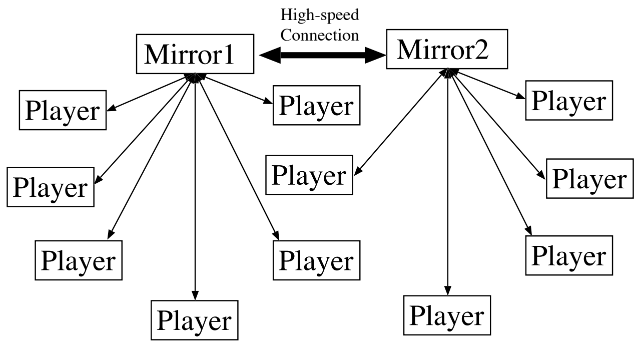


Server Architecture: Mirrored

The world replicated at each server (mirror)

- all the worlds are synchronized
- players see everyone across all mirrors

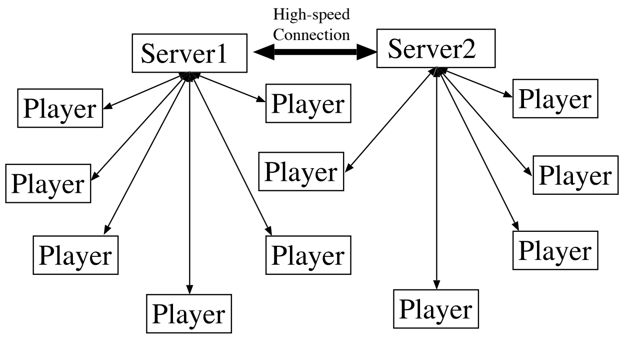
Mirrors must be kept consistent



Server Architecture: Partitioned

The world is split into regions
- each region is hosted by a different server
- example: *Second Life*

Servers must be kept consistent



Distributed Computing Model

Game companies have their preferred computing model and would provide high-level libraries to implement the model

Two common models:
- distributed objects
- message passing

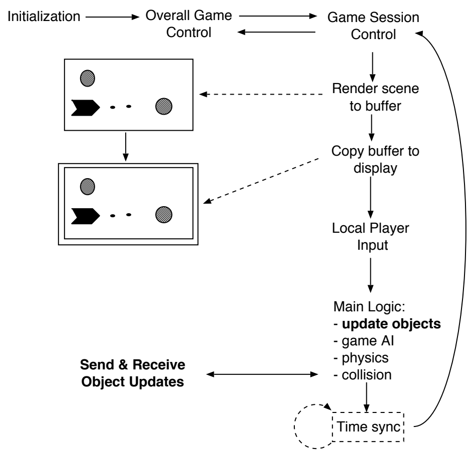
Distributed Objects

Characters and environment maintained as objects

Player inputs are applied to objects (at server)

Changes to objects propagated to all players at end of game loop

Object update usually implemented as one or more library calls

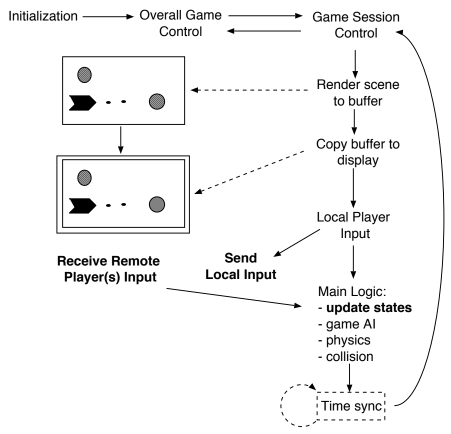


Message Passing

Player inputs (either button pushes or higher-level movements) are sent to other players (or server)

All players update their own game state

Or server updates the global game state and sends it out to all players



Multiplayer Gaming Traffic

What information is sent in a multiplayer game?

- depends on your computing model
- distributed objects: game state, e.g., coordinates, status, action, facing, damage
- message passing: user keystrokes, e.g., commands/moves For RTS: every sec, up to commands/sec during battles (but some of these are redundant and can be filtered out)

In-game Networking Topics

Topology:
- client-server or peer-to-peer

Computational model:
- distributed object vs. message passing

Bandwidth requirement

Latency requirement

Consistency

Which transport protocol to use?
- TCP, UDP, reliable UDP

Bandwidth Requirement

Bandwidth requirement has been HIGHLY optimized
- even with audio chat, takes up at most Kbps
- so, bandwidth is not a big issue
  - but note the asymmetric nature: for N players, you receive N-1 times the amount of bytes you send out
- must be continually vigilant against bloat

However, with player-created objects and worlds, bandwidth becomes an issue again: use streaming, levels of details, and pre-fetching

Latency Requirement

How is latency different from bandwidth?

Tolerable round-trip latency thresholds:
- RTS: $\leq$ ms not noticeable, 250-500 ms playable, $>$ 500 ms noticeable
- FPS: $\leq$ 150 ms preferred
- car racing: $<$ 100 ms preferred, 100-200 ms sluggish, $\geq$ 500 ms, car out of control

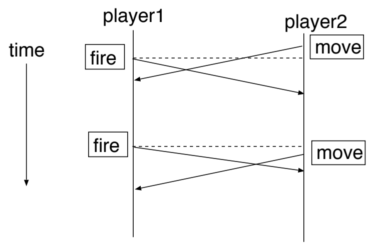
Players' expectation can adapt to latency
- it is better to be slow but smooth than to be jittery

Latency Effect: Consistency

Problem statement:

Case 1:

Case 2:



In both cases:

- at Player 1: Player2's move arrives after Player1's fire
- at Player 2: Player1's fire arrives after Player2's move

Should Player2 be considered shot in both cases?
Or only in the second case?

Synchronization

Synchronization ::=
order moves by their times of occurrence

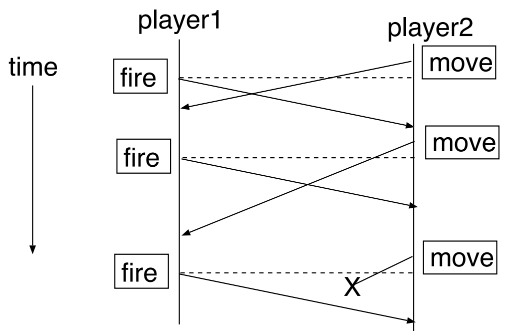
Assume globally synchronized clocks

Out-of-synch worlds are inconsistent

Small inconsistencies not corrected can lead to large compounded errors later on (deer not speared means one less villager means slower barrack build, etc.)

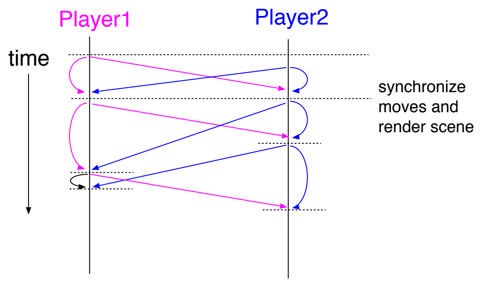
When to Render a Move?

How long do you have to wait for the other players' moves before rendering your world?



Lock-Step Protocol

Algorithm: each player receives all other players' moves before rendering next frame

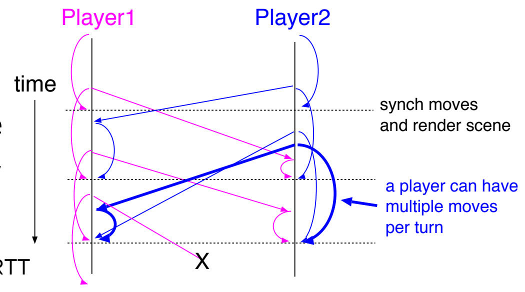


Problems:

- long latency on the Internet
- variable latencies ⇒ jittery game speed
- game speed determined by the slowest player

Bucket Synchronization

Algorithm:
- buffer both local and remote moves
- play them in the future
- each bucket is a round, say of about 200 ms
- bucket size can be adapted to measured RTT


Player1
Player2
time
synch moves and render scene
a player can have multiple moves per turn
X

Smoother play, but:
- game speed (bucket size) still determined by slowest player
- what if a move is lost or late?

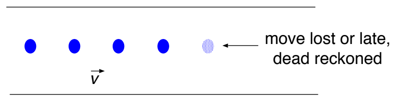
Pessimistic Consistency

Every player must see the exact same world
- each player simulates its own copy of the world
- all the worlds must be in synch
- uses bucket synchronization
- each player sends moves to all other players
- dropped packets are retransmitted
- a designated host collects measured RTTs from all players and set future bucket sizes

Problems of bucket synchronization:
- variable game speed if lost packets must be retransmitted
- speed determined by the slowest player

Dead Reckoning and Roll-back

Dead reckoning, a.k.a. client-side prediction
- extrapolate next move based on prior moves
  - e.g., compute the velocity and acceleration of objects to dead reckon



  - players can help by sending velocity and acceleration along
  - obviously, only works if velocity and acceleration haven't changed

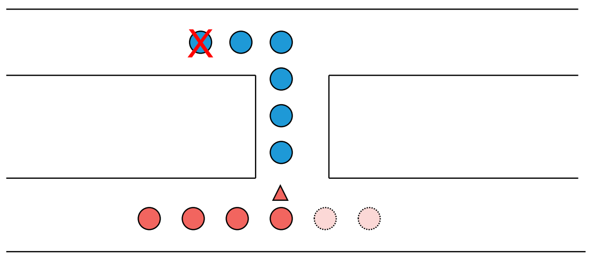
In case of inconsistency:
- server assumed to always have authoritative view
- when clients correct (roll-back) inconsistent views, players may experience "warping"

Optimistic Consistency with Roll-back

Observation: dead reckoning doesn't have to be limited to lost packets!

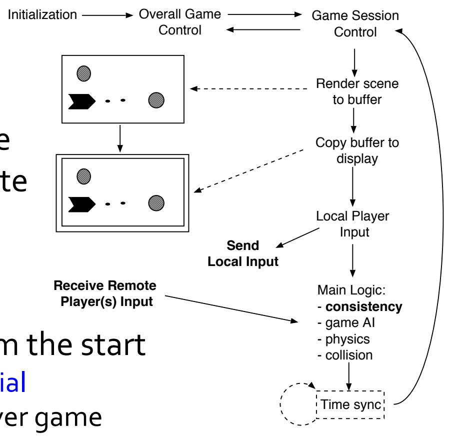
Half-Life:
- each client plays back its own moves immediately and sends the moves to server
- each client also dead reckons the other players' moves
- server computes world and sends its authoritative version to all clients
- clients reconcile dead reckoned world with server's version
- only need to synchronize important events, but must be careful that dead reckoning errors don't get compounded over time
- can result in some jerkiness and perception of "shooting around corner"

Shooting Around Corner



Consistency: Correctness

For consistency all user input must pass through the synchronization module



Be careful with random number generators: isolate the one used for game-state updating from other uses (ambient noise etc.)

Design for multiplayer from the start
- single-player becomes a special case of single-client multiplayer game

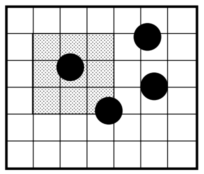
Consistency: Smoothness

For smoother playback, decouple bucket size from frame rate

Immediately render local moves

Modify game design to allow for latency and loss, e.g.,
- make player wait for elevator
- teleportation takes time
- require multiple hits per kill, even snipers can miss
- let bullet/missile have flying time
- build in inertia, don't allow sudden changes in facing

Reducing Consistency Check



Do area-of-interest management (a.k.a. relevance filtering):
- aura: how far you can be sensed (ninjas and cloaked ships have aura of 0)
- nimbus: how far you can sense (empath and quantum-sensor have large nimbus)

Perform consistency check only when B is within A's nimbus and A is within B's aura

Aura and nimbus are defined for a given set of "game technology" (e.g., cloaking device, quantum sensor, etc.)

Which Transport Protocol to Use?

Gaming requirements:

- late packets may not be useful anymore
- lost information can sometimes be interpolated
- (though loss statistics may still be useful)

Use UDP in game:

- can prioritize data
- can perform reliability if needed
- can filter out redundant data
- use soft-state
- send absolute values, not deltas
- or if deltas are used, send ``baseline'' data periodically
- must do congestion control if sending large amount of data