NEXON
DEVELOPERS
CONFERENCE

Game Sync Technics through CAP Theorem

+ Server-side Rewinding and Determinism

Nexon OC Studio
Technical Director

Seungmo Koo

NEXON
DEVELOPERS
CONFERENCE

Agenda

1. CAP Theorem and Eventual Consistency
2. Multiplayer Game Sync Techniques 101
3. Server-side Rewinding
4. Best Practices

Things not covered in this session

- Various dead reckoning techniques*
- Consensus algorithms and state replication protocols*
- Extension theorem for CAP like PACELC*
- Consistency Models*

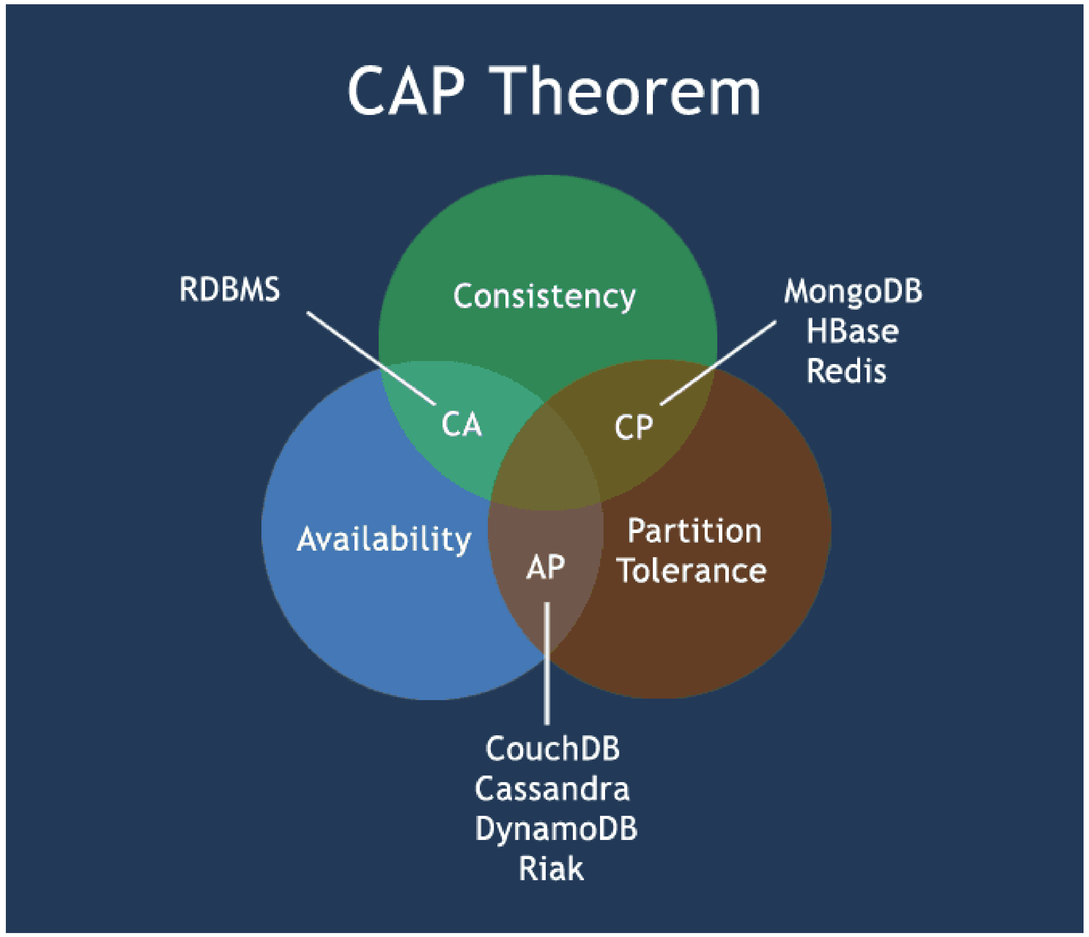
* For more information, please refer to the last slide (references) …

NEXON
DEVELOPERS
CONFERENCE

CAP Theorem and Eventual Consistency

CAP Theorem

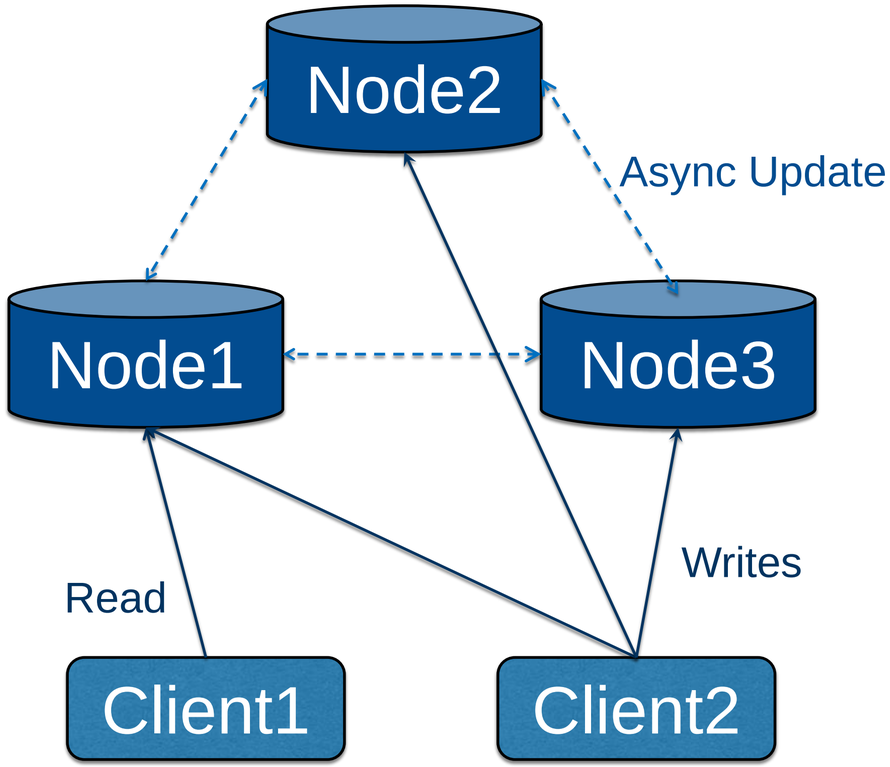
- Eric Brewer, PODC 2000*
- Theorems to help you make technical choices when designing robust distributed systems
  - Consistency
  - Availability
  - Partition Tolerance
- No system satisfies C+A+P at the same time
- Not directly related to NoSQL
  - Just for easy understanding



IMG From: https://docs.deistercloud.com/Technology.50/NoSQL/index.xml

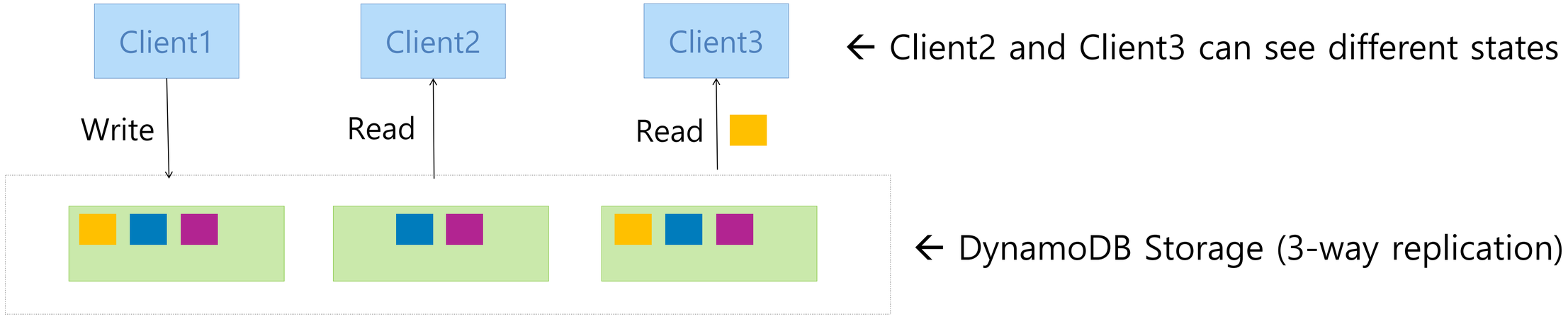
NoSQL

- Working with a huge quantity of data
  - Simple to use but deliver high performance and high availability
- Schemaless
  - Key-Value, Key-Document
  - Internally, the storage structure is sharded and replicated
- Commonly used in practice*
  - MongoDB, Cassandra, DynamoDB, …



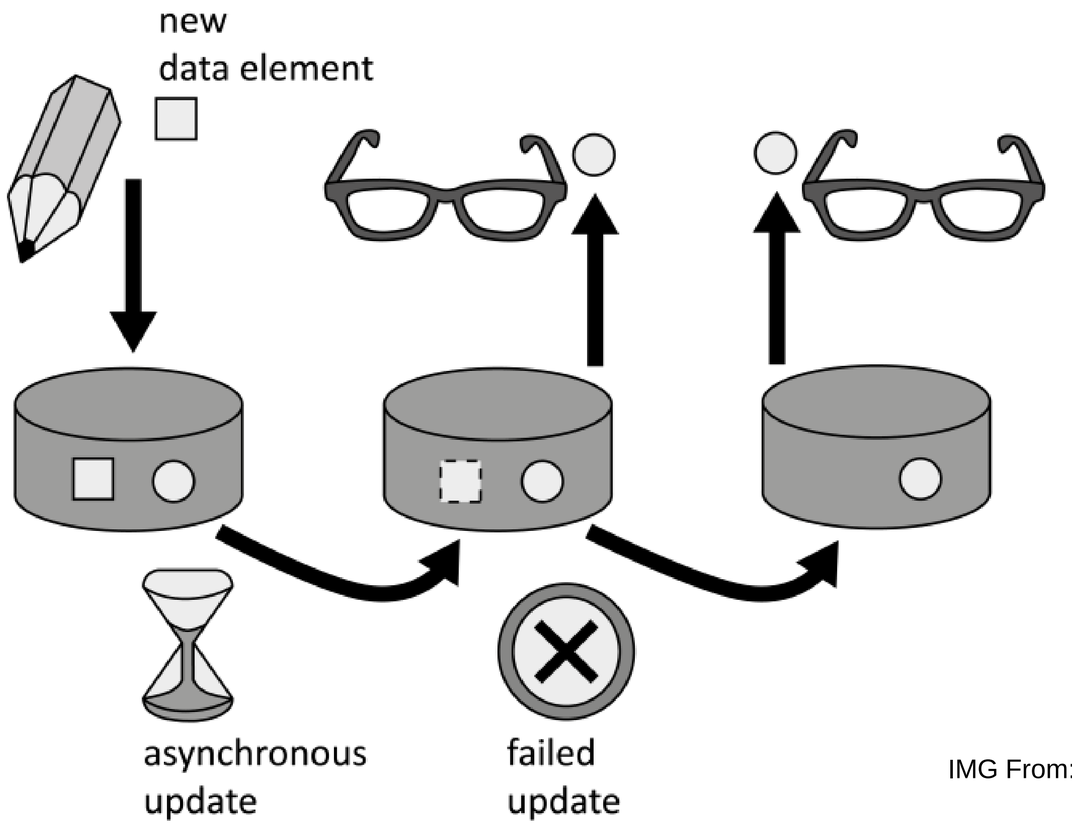
AWS DynamoDB Example

- Data Read/Write on DynamoDB
  - Write: saves data with 3 node replication
  - 2 options for Read operation
    - Default Read: read from one of the three nodes that responds first. In this case, it is possible to read past data. (Availability)
    - Consistent Read: reads after all node's data are synchronized (Consistency)



Eventual Consistency

- Use when Availability is more important than consistency*
- Certain clients can instantly see the state of the data in the past, but it achieves consistency, eventually.



IMG From: http://www.cloudcomputingpatterns.org/eventual_consistency/

Consistency in Multi-threaded environment

- Singe-threaded environment
  - C + A: Consistency + Availability (as long as there is no blocking due to I/O)

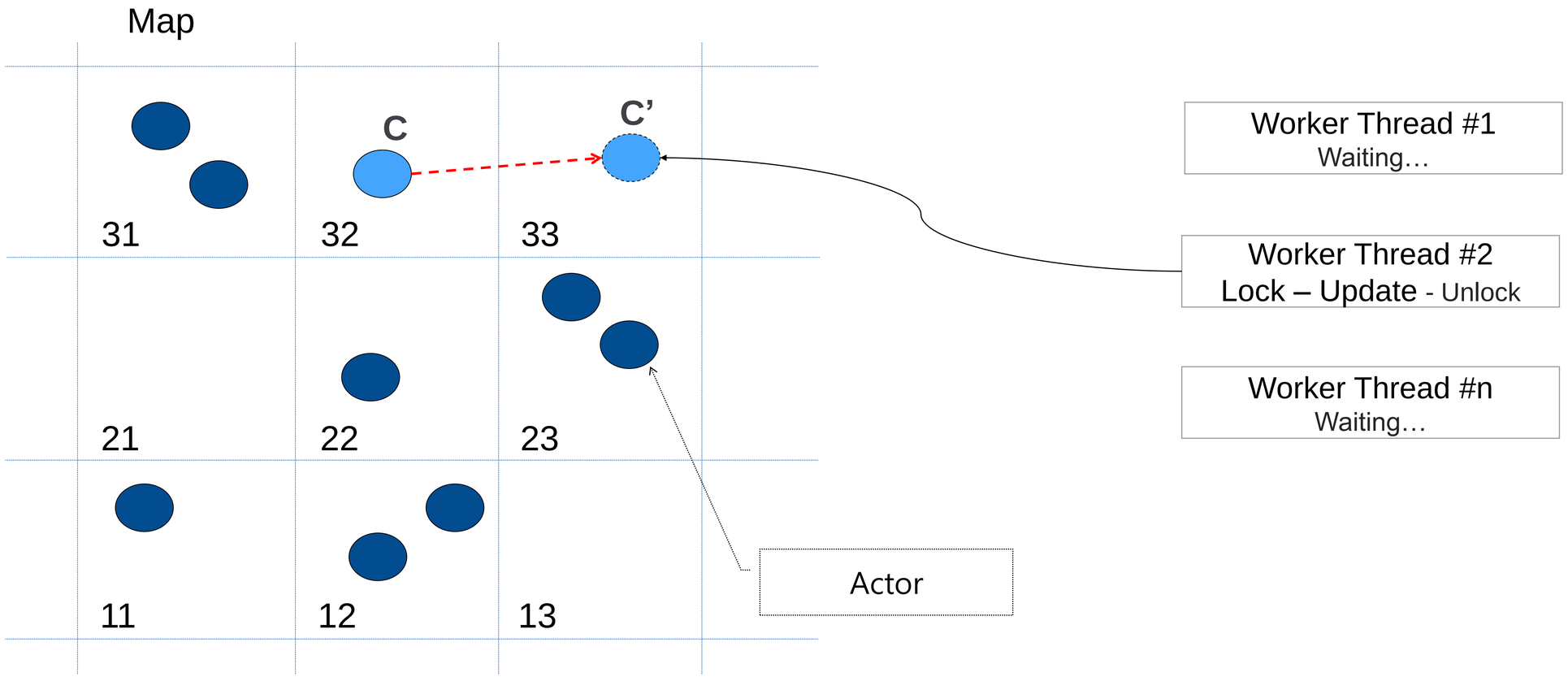
- Multi-threaded environment that synchronizes with Lock
  - C + P: Consistency through locking (C) + Multi-threads (P)
  - Threads can always see the same shared resource state (C)
  - Does this method guarantee availability?
    - If one thread enters the critical section, all other threads must wait. (It means not available to all other threads)
  - If there is a wait in the critical section due to I/O? Lock Convoying…

Improving performance with eventual consistency

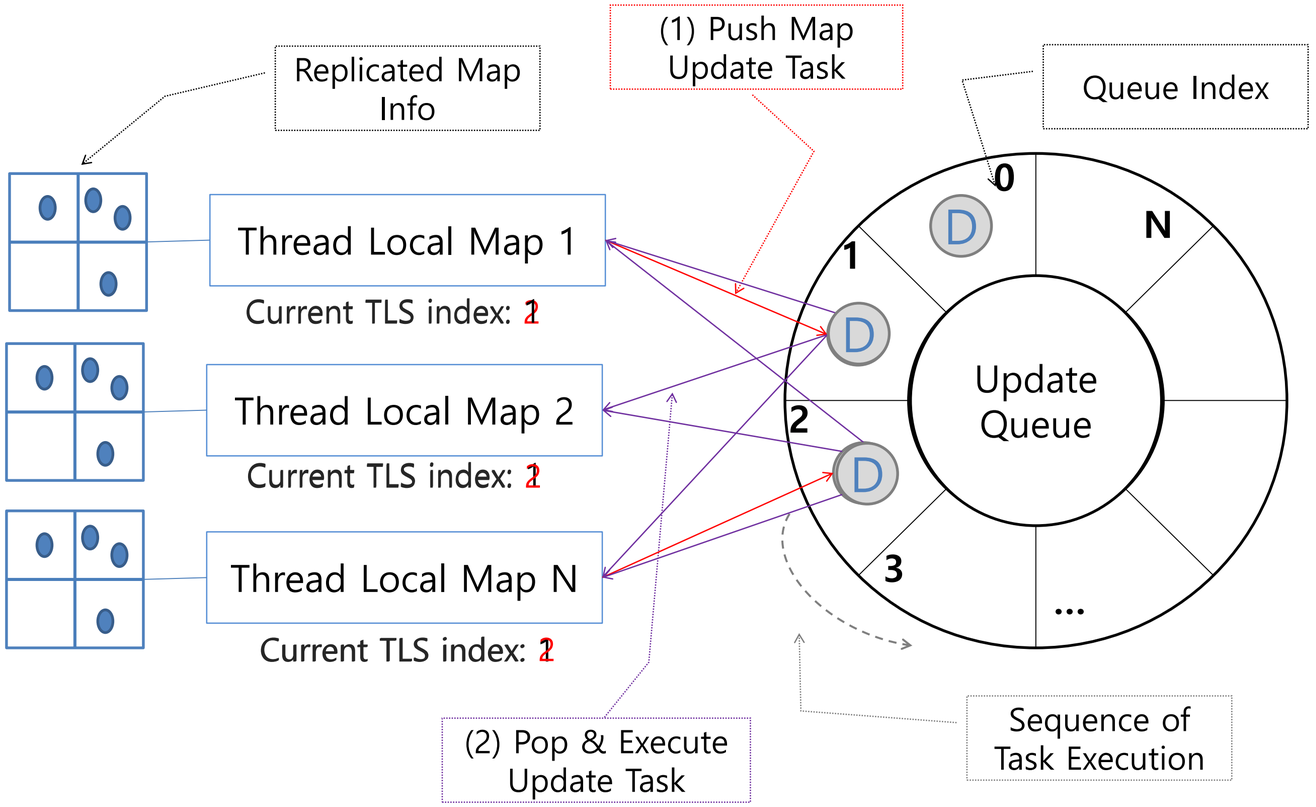
- Not keeping strong consistency
  - Ensuring that a thread does not wait for a shared resource to be accessed, even if a particular thread has only seen its past state
  - Choose C+A instead of C+P
    - Instead, it is needed to implement all threads to be in the same state eventually
- How-to
  - Replicate data structures by thread local storage
  - Data structure updates are made on a thread-by-thread basis using a update queue (lock-free or wait-free)
  - Read is always available because it is a thread-local access
  - Example: https://github.com/zeliard/Dispatcher/blob/LB_version/JobDispatcher/LoadBalancer.h

TERA Case Study

- Initial design
  - There is a single shared map (terrain information) and multiple game logic threads compete. (ReadLock for read, WriteLock for update)
  - Although all threads can see the same map state → CPU 100% !!



Change to eventual consistency model using TLS



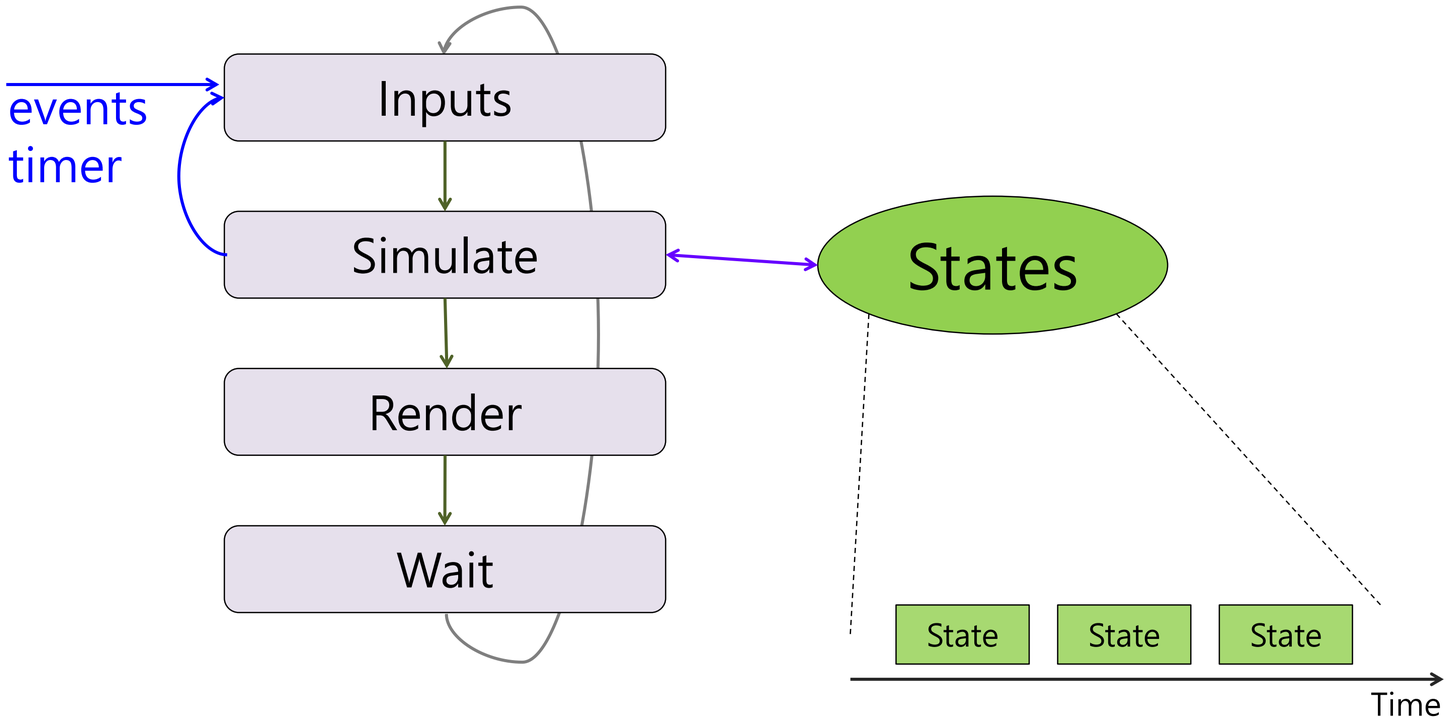
Instantly, a particular thread can see the state of the past, but no lock contention
→ From a thread's point of view, the map data structure is always available → Performant!

NEXON
DEVELOPERS
CONFERENCE

Game Sync Techniques 101

Single Player Game

C + A System



Frame Per Second (FPS) → How many times does this loop run per second?

Multiplayer Game Sync Types

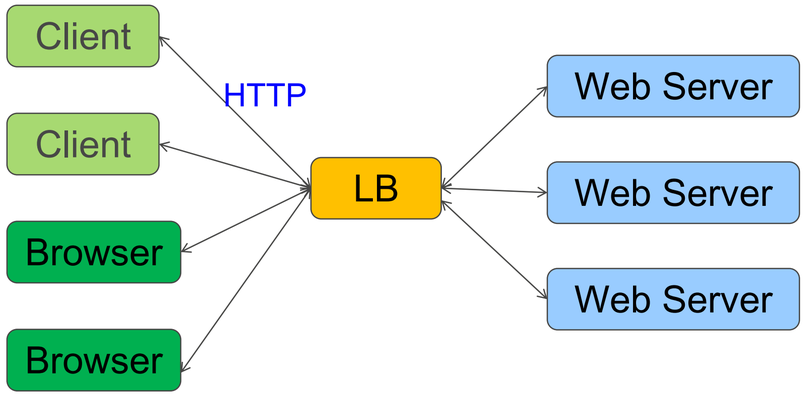
Asynchronous

Synchronous Type 1

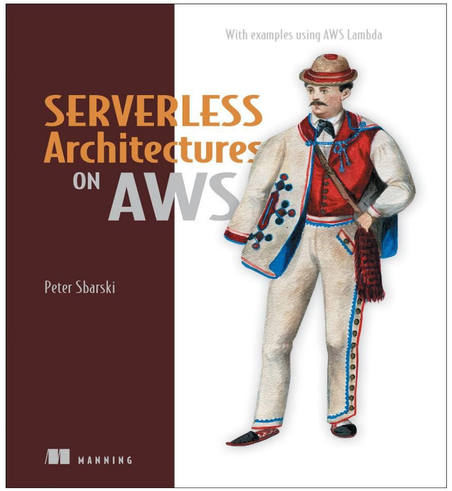
Synchronous Type 2

Asynchronous Type

- Web-service style backend
  - 2-tier: web servers + data store
  - 3-tier: web front + app servers + data store
  - HTTP / HTTPS
  - No need to maintain the connection continuously
  - Suitable for mobile games and social gaming genres



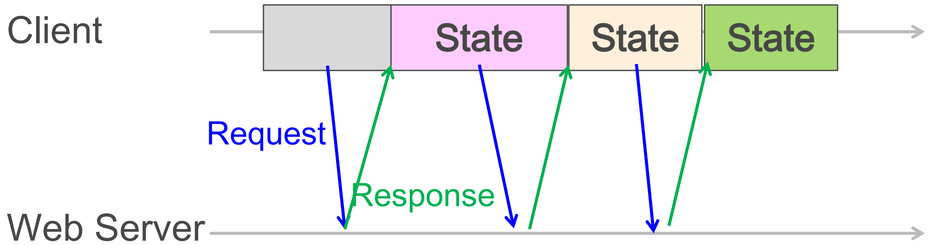
- (Trends) Backend can be built in Serverless form
  - API Gateway, Cloud function, Lambda, …




352,314
Add Coins & Cash
1
305855
57
Barbara's Farm
3
Red Hay Bale
FUEL
+2134
Add Neighbor
Rebecca 3
78
Angela 14
3229
Sandra 15
4245
Jay 18
6872
Chris 18
7365
Lauri 20
8909
James 20
9278
MARKET
33

How the asynchronous game works

- Request/Response
  - Inheriting the characteristics of the web
    - Atomic, Stateless



  - When ordering between players is required
    - What if my friend first harvested an apple?
    - Directly synchronized from cache server or Database

  - Passive: No server-initiated action
    - No active NPC action like monster's preemptive attack

Synchronous Type

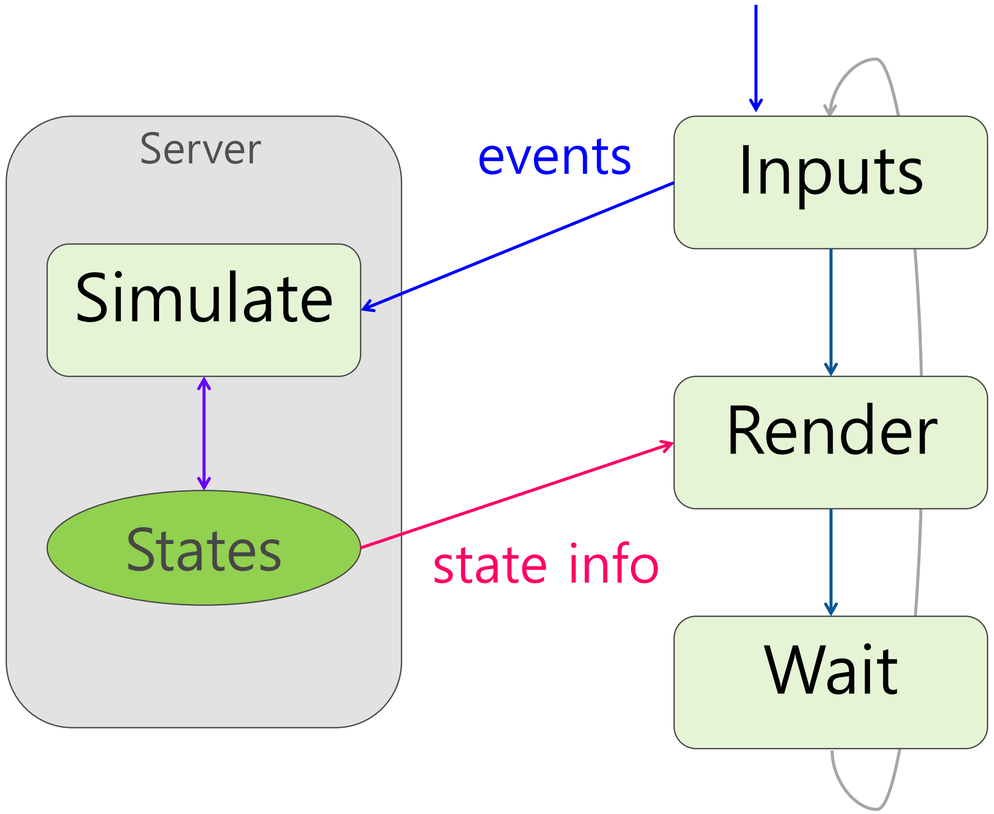
- Realtime Sync

- Multiplayer games can also be considered as a distributed system
  - CAP theorem can be applied anywhere in distributed systems
  - Sync by remote replication

- Choose between consistency and availability depending on a game
  - Network game means a distributed system → P is essential
    - Choice: availability (A) vs consistency (C)

Type 1: Server Authoritative Model

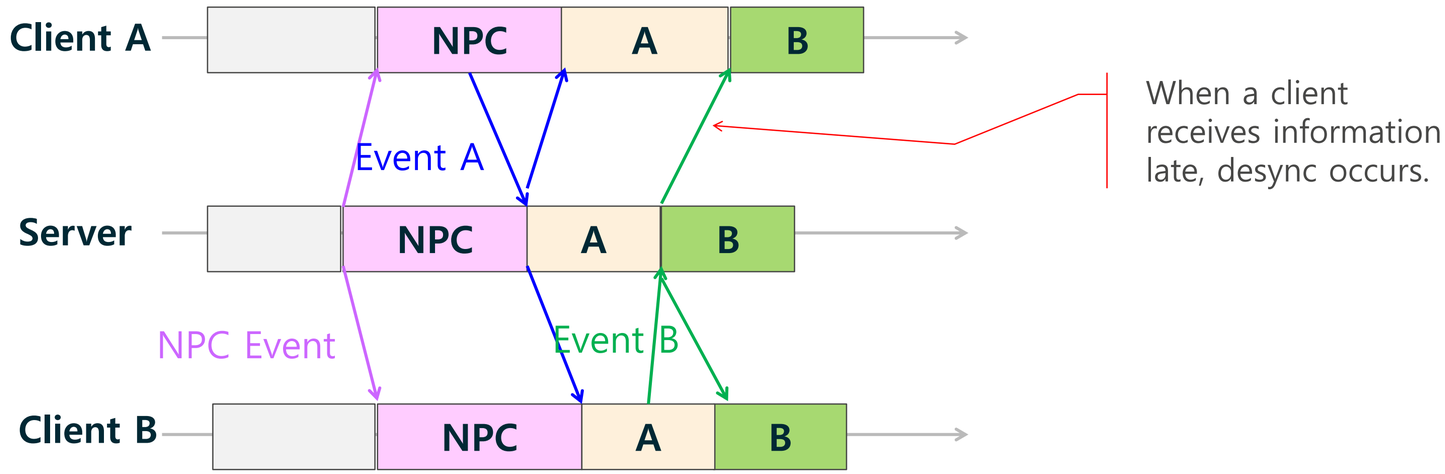
- Server is responsible for game logic simulation and synchronization
  - Clients connects indirectly through the server
  - Server can actively drive NPCs
  - Mainly used genres: FPS, MMORPG, MOBA, Sports




하급기사 Zerocore
<이나중탁구부>
파수병 미러
<Knights of Azeroth>
<For Honor>
<FreeStyle>
<The Darkness>
<Knights of Azeroth>

Mechanism for Server Authoritative Model

- Client-Server architecture
  - Server collects events from clients, calculates them, and broadcasts them to the clients
- Capitalism style / Optimistic way
  - Server proceeds first → Clients follows the server → The faster the network speed is, the fewer the gap with the server (better!)
  - Clients should do Dead Reckoning*
  - A + P System

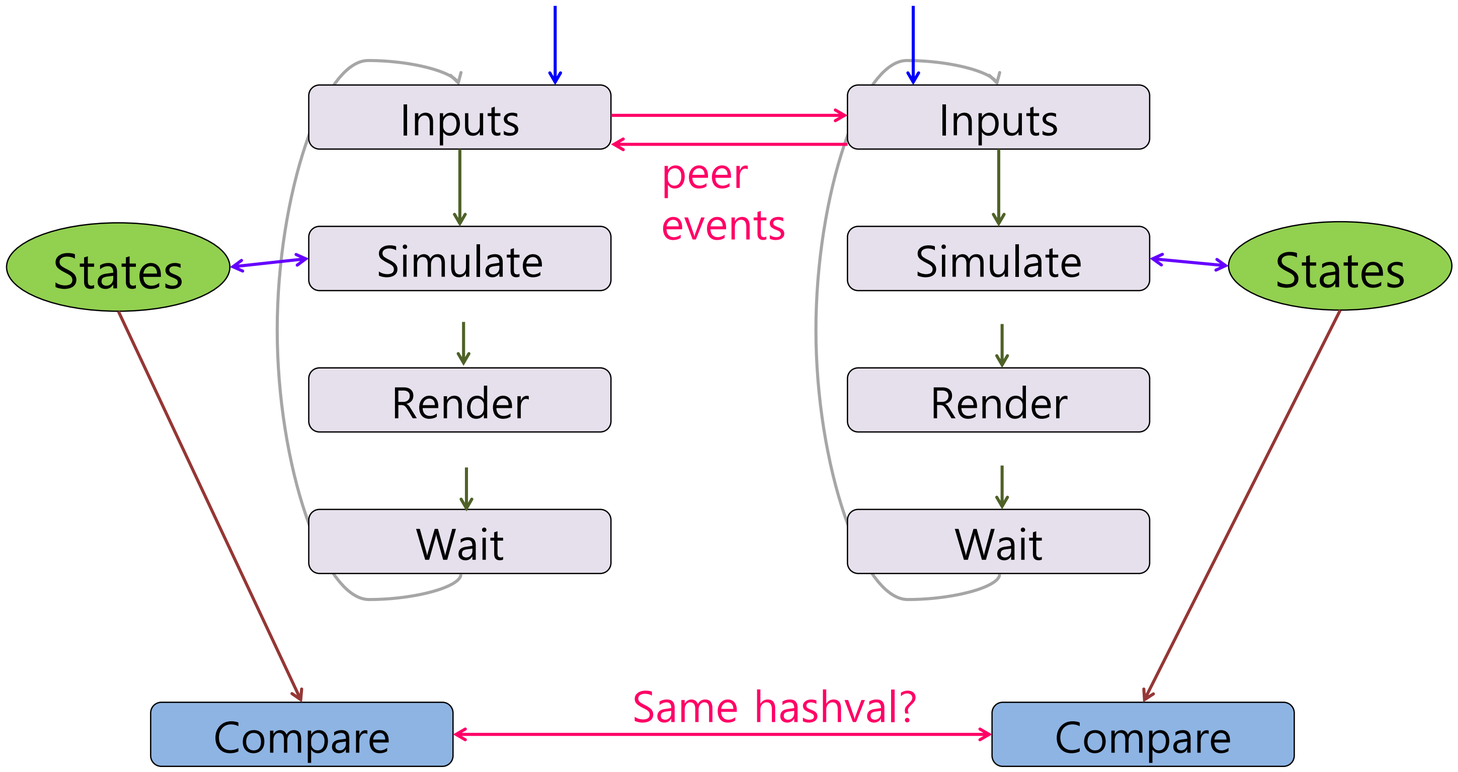


Rubber Banding?

IMG From: http://m.post.naver.com/viewer/postView.nhn?volumeNo=11513471&memberNo=16036253

Type 2: Lock-Step Model

- Collect events from the opponent players periodically and simulate them individually
- Mainly used in RTS and past AOS genres

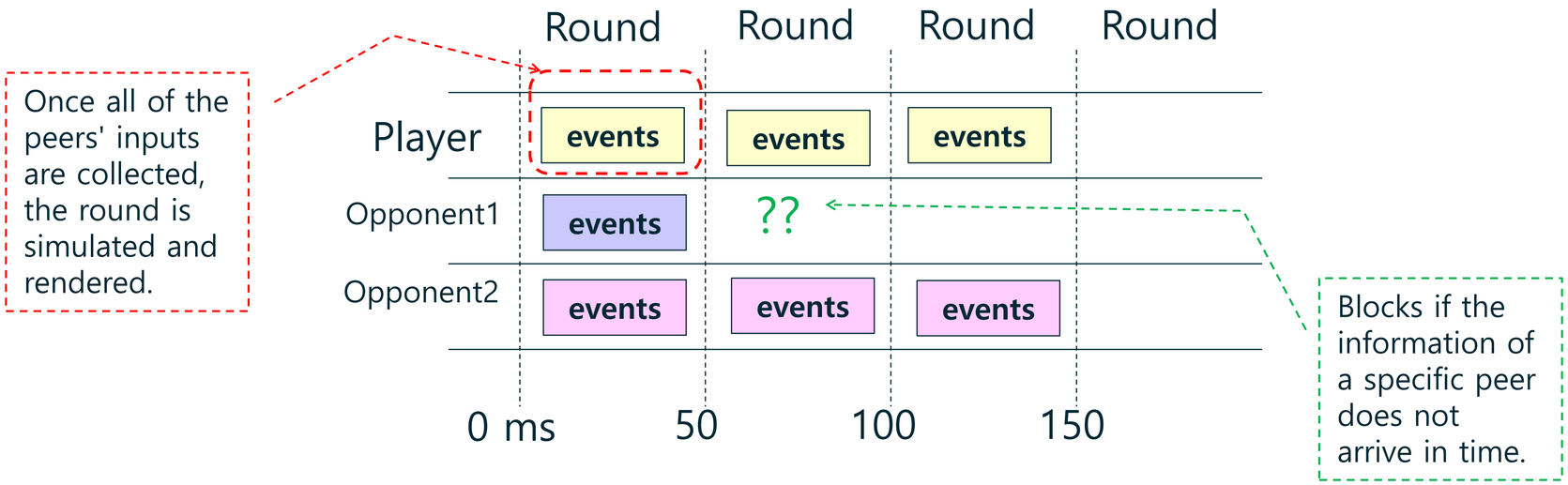


Dropped from the game?

IMG From: http://starcraft.burningblade.org/stories/index.html

Mechanism for Lock-Step Model

- Each client maintains an event queue like a below picture
- Communism style / Pessimistic way
  - Clients all go together → They will all wait if one stops
  - Unlike the server authoritative model, there is no desync situation
  - Lag hiding technics between Rounds*: (e.g.) Starcraft Marine Pre-Anim & Sound play
  - C + P System



Waiting for players?

Re:CAP

- Server Authoritative Model
  - A+P system: to make system always available in order to give players as much as possible a lag-free experience

- Lock-Step Model
  - C+P system: to make system always consistent in order to give players a fair experience

NEXON
DEVELOPERS
CONFERENCE

Server-side Rewind Technique

Why compensate Lag?

- Internet is faster, but…
  - Global Multiplay
  - Global one-build and player battle with bigger distance
  - Paradoxically, players are more exposed the lag

- Eventually we need handle the lag
  - Forcing players to match each other in close distance
  - OR, ignoring the lag
  - OR, using a trick to correct the lag which deceives players

Trends

- If we cannot avoid the lag anyway…
  - To wait for someone to be consistent in an unpleasant experience
    - CP systems are very limited (e.g. round cycle of 500ms or more)
      - Assuming that even in the worst case the response comes within this cycle
  - Using a way not to wait someone (AP system)
    - Best effort for maximum fairness through eventual consistency

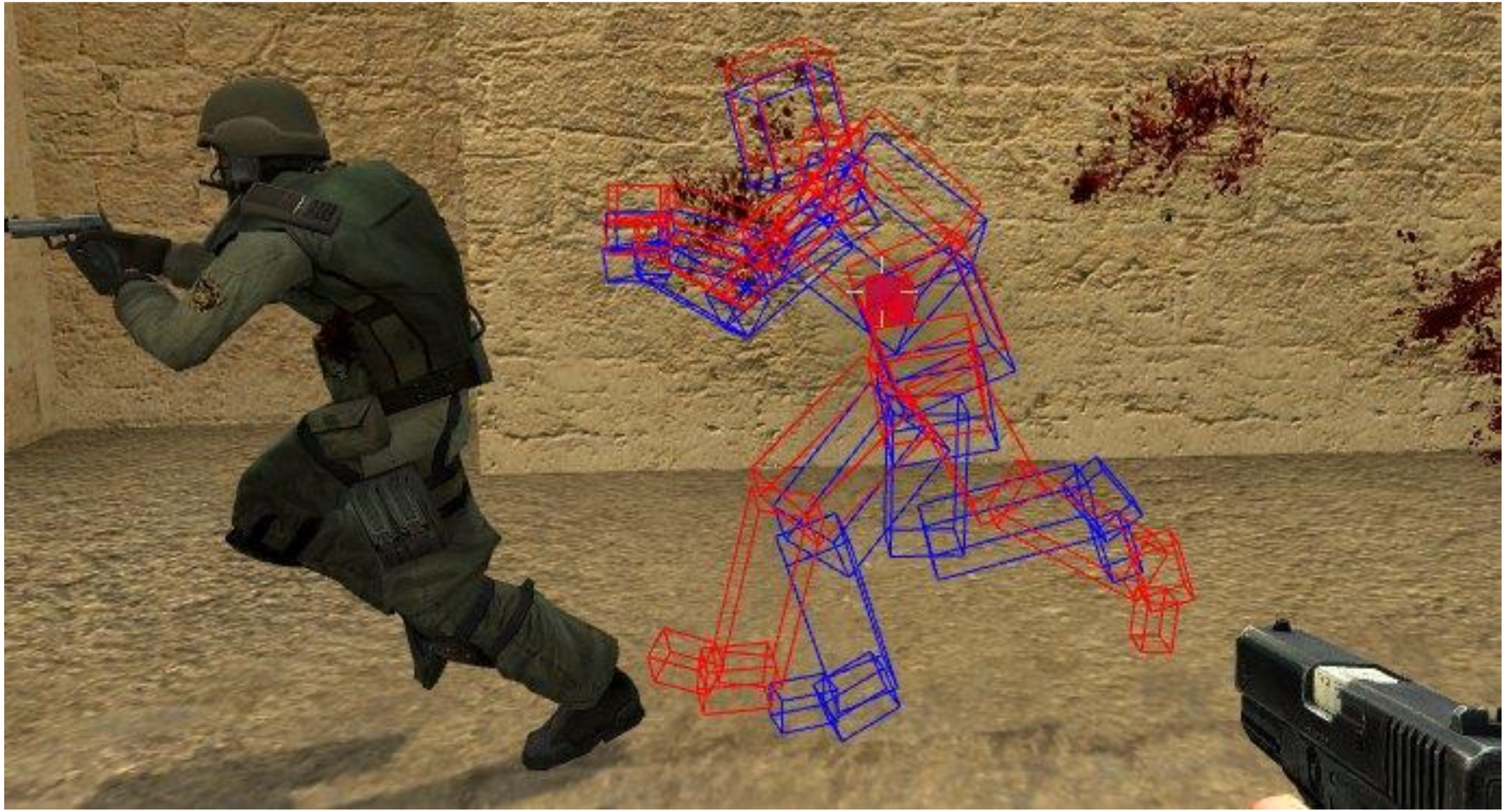
- Compensate the lag to give a pleasant experience and a fair play
  - Sync by using a server-side rewinding*

Server-side Rewinding Technique

- Server rewinds to the past to judge a hit
  - Used by Quake, Duke Nukem, Call of Duty, …
  - Game logic simulation is done on both server and client
    - Clients send input and self simulation results to server
    - Simulate by rewinding to the point where the client inputted at the server
    - If it is the same as the result of self-simulation from client, it is OK
      - Server tells the client if it is different from the results of the simulation
    - When the final result from the server arrives, the client calibrates

- Clients proceed the simulation through predicting and send inputs to the server

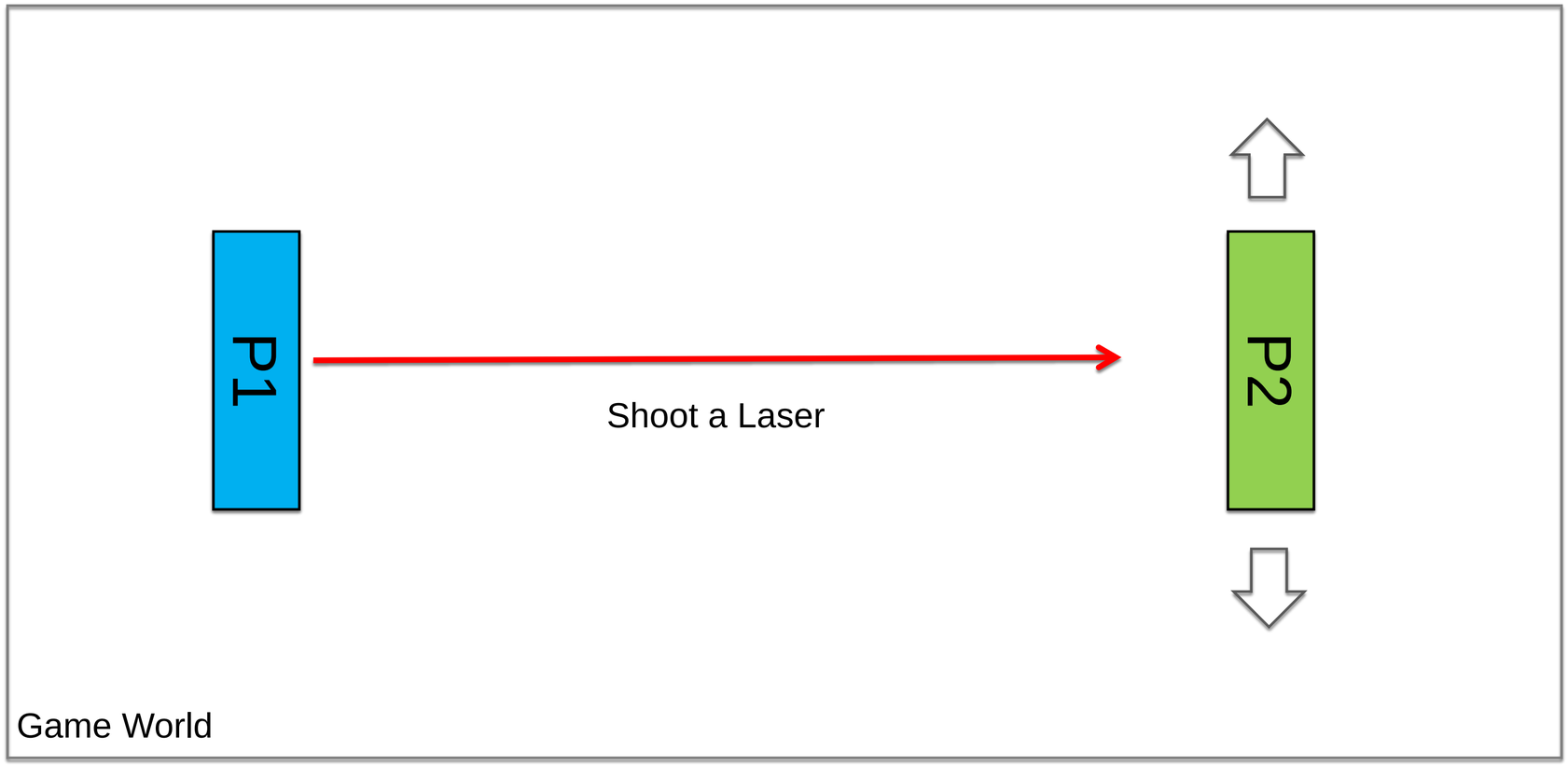
Server-side Rewinding Visualization



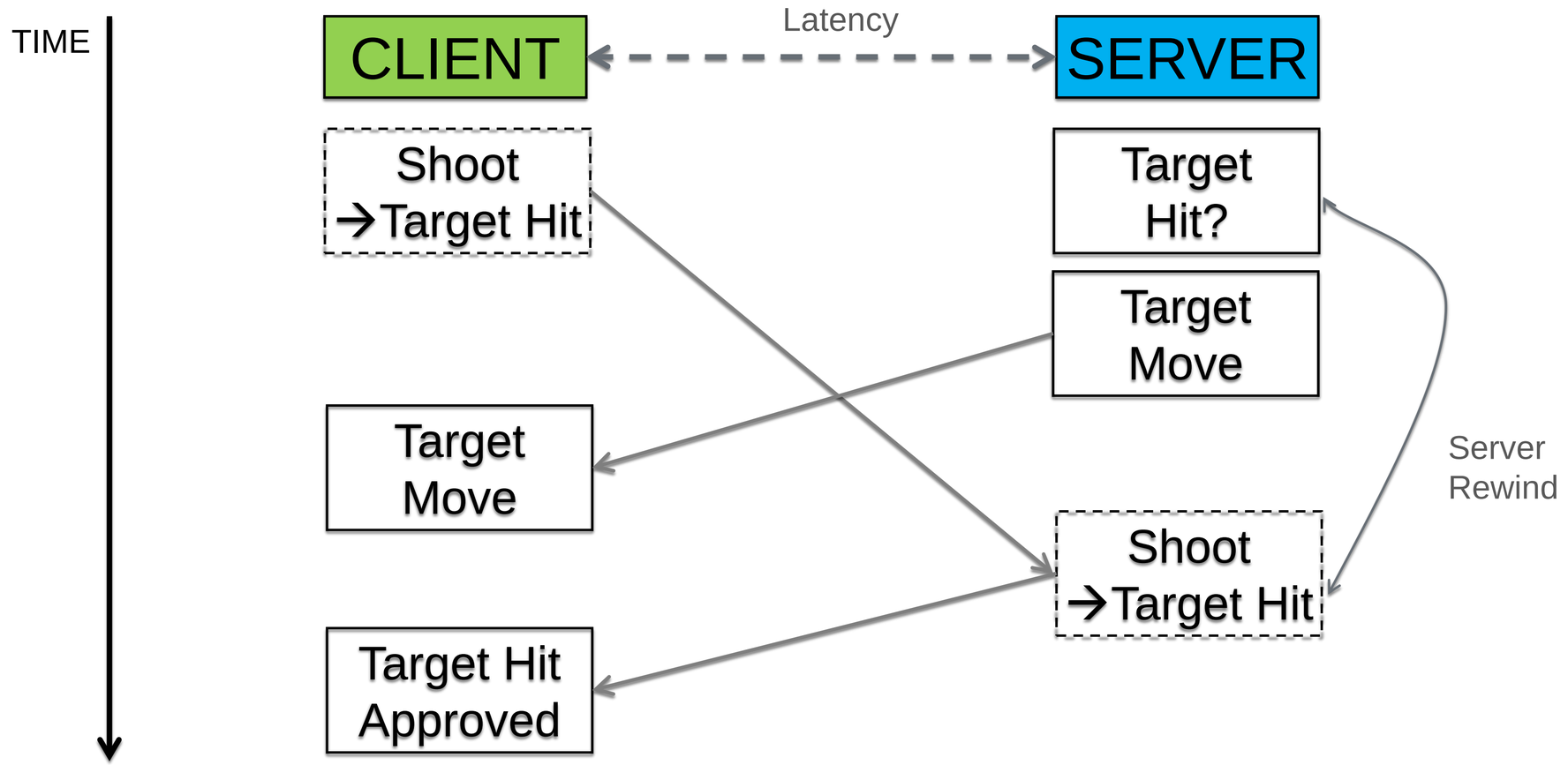
IMG From: https://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking

Very Simple Easy Example

- Pong Laser Game



Basic Principle



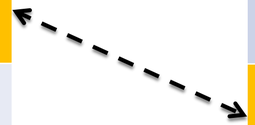
State Change Table

- States of the game world that the server sent to a client

WorldFrame	TARGET PosY
1	2.0
2	3.0
3	**5.0**
4	8.0

- Input changes sent to the server from a client

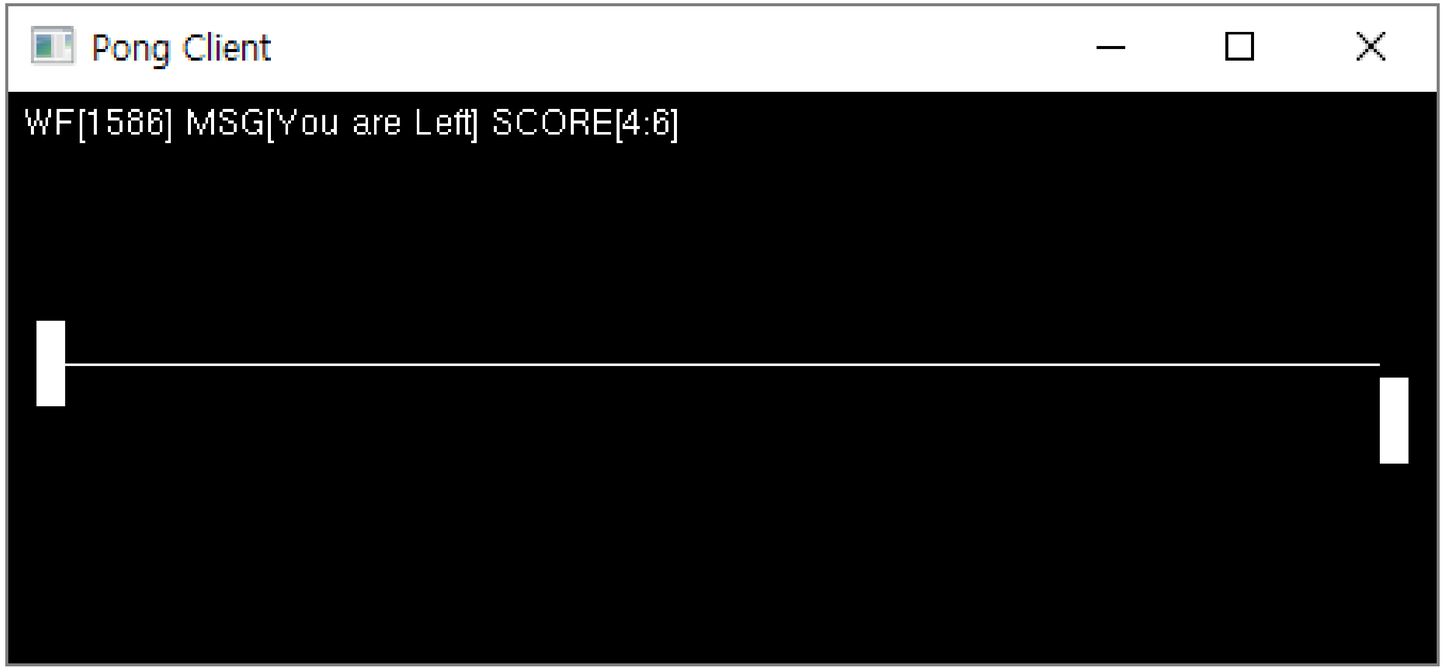
WorldFrame	Client Input FR	Shoot
1	1	No
2	2	No
2	3	No
3	4	**Yes**



The server rewinds the state based on the WorldFrame that the client knows

Real Code Sample

- Pong Laser
  - https://github.com/zeliard/PongRewind
    - Created a Desync situation by deliberately adding a random delay to both sides of the server and client



Pseudo Code Level

- Client → Server

  Client Input (Request)

```
PlayerType mPlayerType;
unsigned int mRecentWorldFrame;
unsigned int mInputFrame;

float mPosDiff;
bool mIsShoot;

unsigned __int64 mStatusHash;
```

- Server → Client

  Game Status (Broadcast)

```
unsigned int mWorldFrame;

float mLeftRacketPosY;
float mRightRacketPosY;

int mLeftShoot; /// True if val> 0
int mRightShoot;

int mLeftScore;
int mRightScore;
```

Maintaining World State History for Server Rewind

- Keep state by WorldFrame on the server side
  - Snapshots by WorldFrame (server frame) for objects that require rewinding
  - Note: This example is so simple that it keeps the whole world object
    - In practice, it is recommended to keep only limited objects that require rewinding

```
RewindObject<GameStatus, MAX_WORLD_FRAME_HISTORY> mFrameHistory;
unsigned int mCurrentWorldFrame;
```

Client Logic

- Client Update Loop
  1. Get the input of the player
  2. Simulate based on the player input
  3. Get a hash of the updated state
  4. Send player's input and the hash value to the server
  5. Record client input history
     - For replaying inputs in case of desync situation

```
std::map<uint32_t, InputElem> mInputHistory;
uint32_t mRecentInputFrame;
```

https://github.com/zeliard/PongRewind/blob/master/PongClient/PongClient/GUIController.cpp#L145-L164

Client Logic

- Client Receive Handler
  1. Apply the lastest state that arrived from the server
     - It means forced update to the latest server state, but you can choose whether to reflect directly on the rendering or to calibrate more smoothly (dead reckoning) on the client.

     https://github.com/zeliard/PongRewind/blob/master/PongClient/PongClient/GUIController.cpp#L233

- Client Resync Handler
  - Invoked when the hash sent by the client differs from the hash computed by the server. (Server tells the client the desynced Client Input Frame)
    1. Replay & Re-apply from the desynced Client Input Frame

    https://github.com/zeliard/PongRewind/blob/master/PongClient/PongClient/GUIController.cpp#L275

Server Logic

- Server Update Loop
  1. Get the state of the previous WorldFrame
  2. ++WorldFrame
  3. Update the world status
  4. Save updated world status to current WorldFrame history
  5. Broadcast updated status to clients

https://github.com/zeliard/PongRewind/blob/master/PongShared/GameLogic.cpp

Server Logic

- Server Receive Handler
  - Judge a hit or so at the WorldFrame time received from a client
    - Simulate after rewinding the server to that point
    - When the simulation results is different from the client by comparing the hash, the server notifies the client to request a resync.
  - Process the client input

https://github.com/zeliard/PongRewind/blob/master/PongServer/PongServer/GameManager.cpp#L57

Side Effect of Server-side Rewinding

- Discomfort due to late judgement as much as rewind time
  - (e.g.) As soon as the victim hides behind the cover, the bullet is hit.
  - Delays can not be removed, and you have to hide them elsewhere
    - Who would you like to feel better? Attacker or Attackee?
    - In general, the FPS has an absolutely greater number of shooting than the number of being hit.
      - So, modern FPS games mainly choose to make shooters more feel better

Re:Wind

- Sync Technique by using server-side rewind
  - It is a technology that has been widely adopted in FPS games.
  - Basically, it is based on the server authoritative model (AP system)
  - To minimize the unpleasantness caused by lag, it abandons strong consistency. Instead, it adopts a way to emulate eventual consistency to give players best effort fairness

NEXON
DEVELOPERS
CONFERENCE

Best Practices

And the problems to think about

Best Practices for Server-side Rewinding

- Do we rewind all of the world state?
  - Rewind only relevant objects (mainly actors)
- How much history will we keep?
  - Usually within the maximum affordable latency
- Applying to weapons that are fast (judgement within RTT/2)
  - Completely different way for Projectiles*
- Specify Unrewindable Properties
  - Already committed on the server (e.g. Alive, Score, …)
- Desync Handling
  - It is better to include it in the update packet broadcast periodically rather than send it as a separate Resync request packet

TickRate for Server Authoritative Model

- TickRate frequency for server-side simulation and broadcasting
  - Higher TickRate is more advantageous for server rewinding as well as more accurate hit-judgement
  - However, it requires more CPU and network bandwidth (this means higher infrastructure costs)

Considerations for Broadcasting

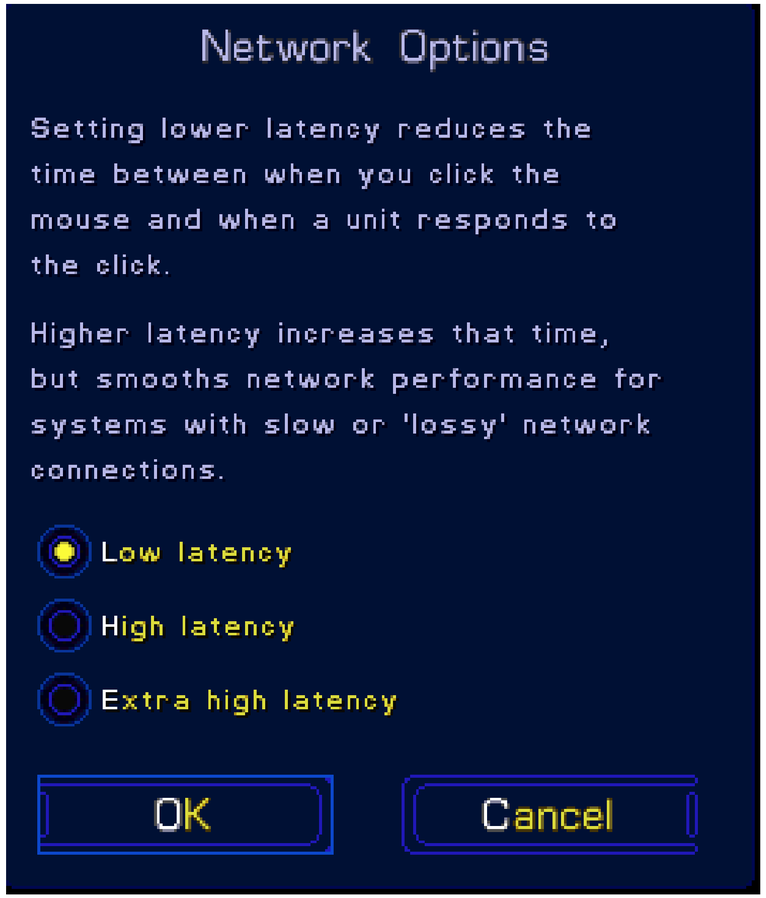
- Sending a delta for the world state?
  - Or, sending the entire data with compression?
    - Higher CPU utilization but lower network bandwidth

      (For personal experience, packet compression is often meaningless if less than 600 bytes)

- Notes by protocol
  - TCP: Most of the time you need to turn off the Nagle algorithm
  - UDP: The larger the packet size, the higher the probability of loss
    - If the size is small enough (within 100 bytes), retransmission can be avoided by including redundant data in the previous frame.
    - Consider bit packing
    - Consider compression if the packet size is likely to be close to MTU boundary

TickRate for Lock-Step Model

- Affects lock-step synchronization period (Round length)
- Starcraft example: Low/High latency option
  - Low: more likely to see the “Waiting for players”, but better responsive
    - Higher TickRate, shorter Round length
  - High: smoother but less responsive
    - Lower TickRate, longer Round length
- Clash Royale example
  - Global one server area (AWS US region)
  - 1sec Round: latency to half a circle of earth is less than 0.5s



Security and Broadcast Coverage

- Lock-Step Model
  - Vulnerable to hack: the entire world state is known to each client
  - Any client can see the client memory (maphack/helper)
- Server Authoritative Model
  - Area Of Interest management: only the information that a client needs to know
  - The narrower AOI range, the more unpleasant the player experience (narrower field of view)
  - The larger AOI range, the more network bandwidth (less security)
    - (e.g.) PUBG: because of sniping, far away enemy info must be synced
      - It is difficult to apply AOI Occlusion Culling because of the enemy protruding behind the wall.

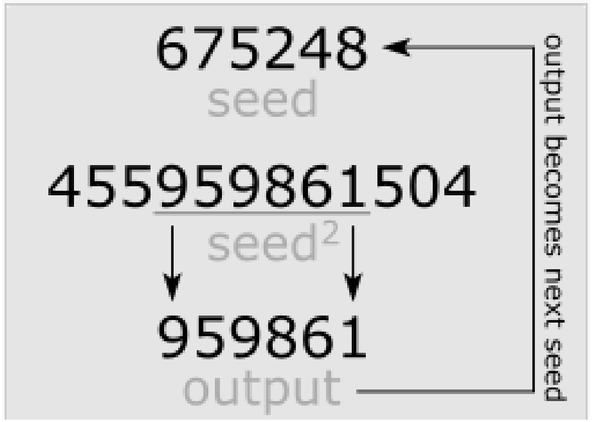
Determinism

- Deterministic game logic
  - Game logic that always produces the same result for the same input
    - Matching inputs ensures the same result
    - Synchronization design can be easy
    - (e.g.) deterministic physics*

- What if random elements exist?
  - (e.g.) What if Starcraft marine rifle damage is 6~10 random?
    - What if a target Hydra HP is 8?
    - In some cases, Hydra lives, and in some cases it dies
    - Is this non-deterministic?

Randomness & Determinism

- Random number generated by a computer is Pseudo Random Number
  - Generated through a predetermined mechanism and a certain seed value




```
m_w = <choose-initializer>;    /* must not be zero */
m_z = <choose-initializer>;    /* must not be zero */

uint get_random()
{
    m_z = 36969 * (m_z & 65535) + (m_z >> 16);
    m_w = 18000 * (m_w & 65535) + (m_w >> 16);
    return (m_z << 16) + m_w;  /* 32-bit result */
}
```

- Pseudo Random Number Generator (PRNG)
  - If the initial SEED value is the same, the same random number is generated
  - Deterministic!

Seed Synchronization

- PRNG has a state
  - The state changes each time PRNG is called
  - Use some of the random numbers generated just before as SEED
  - In other words, the number of PRNG calls must be equal to ensure determinism
  - Using a specific server frame number as a random SEED makes random number synchronization easier
  - Random numbers cannot be synchronized when we use multi-threaded game logics

Non-Determinism (Multi-thread)

- Multiple game logic in the form of multi-threaded?
  - Even if the same input is given, simulation results may be different
  - We cannot predict exactly what order to execute in multi-threads
    - NP-Hard problem ~~(Welcome to the Hell)~~
    - Non-determinism
    - Lock-Step model and server rewind technique cannot be used

- What if the game logic must be multi-threaded?
  - A game that needs to simulate tens of thousands of actors simultaneously? MMORPG?
    - In situations where you have to use the A+P system extensively

Floating-point determinism

- If it is the same physical calculation formula, but the other player machine has different result?
  - The input and logic are the same, but the results are different?
    - A scary situation
  - Floating-point determinism may be broken if any of the following occurs:
    - CPU architecture, OS, compiler and optimization level
    - Build type (debug/release)
  - Especially when creating cross-platform multiplayer games, you need to be careful.

Memory & Determinism

- Unintended nondeterminism due to memory address value
  - Sorting issues within C++ STL containers
  - Default sort does not preserve the original order (cf: stable_sort)
  - Occurs when comparing mainly based on pointer value
    - Be cautious when using pointer values as keys
    - If you synchronize something by iterating the container based on the sorted information, there will be a problem
- Unintended nondeterminism due to a memory alignment
  - Filled value in space between class members is different
    - To clean up the padding, use memset()
  - Be careful when sorting by direct binary comparison

Summary

- Eventual consistency is one of the useful tools
  - But, no Silver Bullet

- Depending on game characteristics, Choose between C vs A
  - There is no case to satisfy both conditions at the same time.
  - If there is a large amount of physical state to transfer, a lock-step model that can be synchronized by input is advantageous.
  - Otherwise, the server authoritative model is most advantageous.

- When designing synchronization mechanisms, determinism should always be considered.

References

- Dead Reckoning: https://en.wikipedia.org/wiki/Dead_reckoning
- Raft: https://raft.github.io/
- Paxos: https://en.wikipedia.org/wiki/Paxos_(computer_science)
- CAP theorem: https://people.eecs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf
- PARCELC: https://en.wikipedia.org/wiki/PACELC_theorem
- NoSQL: https://www.slideshare.net/akleinbe/the-rise-of-nosql
- Eventual Consistency: https://en.wikipedia.org/wiki/Eventual_consistency
- Eventual Consistency for TERA : http://download.enmasse.com/documents/201205-gdm-tera.pdf
- Consistency Model: https://en.wikipedia.org/wiki/Consistency_model
- Lag Compensation: http://www.gamedonia.com/blog/lag-compensation-techniques-for-multiplayer-games-in-realtime
- NetCode for Overwatch: https://www.youtube.com/watch?v=vTH2ZPgYujQ https://www.youtube.com/watch?v=H0zbpPCdhGk
- Server Rewind for FPS Games: https://kotaku.com/5869564/networking-how-a-shooter-shoots
- Server Rewind for Projectiles: https://www.gamasutra.com/blogs/NeemaTeymory/20160906/280377/Why_Making_Multiplayer_Games_is_Hard_Lag_Compensating_Weapons_in_MechWarrior_Online.php
- Deterministic Physics: https://gafferongames.com/post/deterministic_lockstep/
- Floating-point Determinism: https://gafferongames.com/post/floating_point_determinism/
- Rollback Networking for Inversus: http://blog.hypersect.com/rollback-networking-in-inversus/

Thank you.

NEXON COMPANY