Peer-to-Peer Architecture
Peer-to-Peer Architecture

Role of clients
- Notify clients
- Resolve conflicts
- Maintain states
- Simulate games
Latency
Robustness
Conflict/Cheating
Consistency
Accounting
Scalability
Complexity
Lower Latency
No Single Point of Failure
MiMaze from INRIA, France
Age of Empire Series

Without a server with authority, we can easily get into inconsistent states.
Problem: order of messages received are wrong.
**Idea:** If we synchronize the clocks of all players and timestamp each message, we can know the right order of execution.
**Problem:** When to execute? (Is there another message generated earlier than this that is still on its way here?)
(old) **Idea:** Delay processing of messages
Bucket Synchronization
Make the game into a turn-based game ..
.. with very fast turns.
The game is divided into rounds (e.g. 25 rounds per second).
Players are expected to send an update in each round (e.g. 25 updates per second).
(and therefore is only suitable if we have a small number of players..)
Players are expected to send an update in each round (e.g. 25 updates per second).
Every round has a “bucket” that collects the update messages.
Messages generated in the same rounds goes into the same bucket of a future round.
We know which rounds a message is generated based on time-stamp.
When it is time to execute a round, the messages in the bucket is processed (in order of time-stamp).
Which future bucket to go into depends on the latency among the players. (Hopefully not too far in the future else responsiveness will suffer).
If messages from another player is lost (or late), we can predict its update (e.g. using dead reckoning) when possible.
Inconsistency still arises due to prediction.
Alternative is to ensure every update is received before executing the bucket.
Stop-and-Wait Protocol
Synchronized Simulations
Every player sees exactly the same states (but maybe at different time)
Players can tell each other their processing time and latency among players, so that turn length and lag can be adjusted.
Cheating
Look-Ahead Cheat
Player C (or a bot) can peek at A’s and B’s actions first, before deciding his/her moves.
Dealing with cheaters:

1. Prevent cheats (hard)
2. Detect cheats (easier)
Detecting Look-Ahead Cheats
“Mmm... player C always the last one that make its move”
Time-stamp Cheat
Player C (or a bot) can put in an earlier timestamp in its messages.
Suppress-Update Cheat
If dead reckoning is used, Player C can stop sending update and let others predict its position. C then sends an update at appropriate time to “surprise” other players.
C stops sending update.
A predicts C’s position.
C stops sending update.
A predicts C’s position.
C stops sending update.
A predicts C’s position.
C sends an update and shoots A.
Cheater!

Not me! My packets were dropped..
Cheat-Proof Protocol
Lock Step Protocol
One-Way Function $f$:
Given $x$, we can compute $f(x)$ easily. Given $f(x)$ it’s hard to find out $x$ if $x$ is random.
Lock Step Protocol
Two stages needed for each round of stop-and-wait updates.

Stage 1. Everyone decides on its move $x$, and send $f(x)$ to each other.
Stage 2. After $f(x)$ from every other player is received, sends $x$ to each other.
How does lock-step prevent:

look ahead cheat?
timestamp cheat?
suppress-update cheat?
f(x) is known as commitment to x.

A player, once committed to its move, can’t change it.
Problem: Lock-step protocol is slow.
Idea: Use Interest Management

Players only engaged in lock-step protocol when they influence each other. Otherwise their games proceed independently.
This is known as
Asynchronous Synchronization
or Asynchronous Lock-step
Let’s call the two stages in lock-step protocol as commit and reveal stages.
We may also stagger these two stages to improve responsiveness. Multiple commitments can be sent out before we reveal the actions.
1. A player can reveal its action in round $i$ once it receives all commitment of round $i$ from other players.
2. A player can make $p$ moves (send $p$ commitments) without engaging in lockstep.
This is known as Pipelined Lock-step
\(l_{\text{max}} = \text{maximum latency}\)
\(r_{\text{max}} = \text{maximum frame (round) rate}\)

**Lockstep Protocol**

\[
l/r = \max\{2l_{\text{max}}, 1/r_{\text{max}}\}
\]
\[ l_{\text{max}} = \text{maximum latency} \]
\[ r_{\text{max}} = \text{maximum frame (round) rate} \]

**Pipeline Lockstep Protocol**

\[ \frac{1}{r} = \max\{\frac{2l_{\text{max}}}{p}, \frac{1}{r_{\text{max}}}\} \]
\( l_{\text{max}} = \text{maximum latency} \)
\( r_{\text{max}} = \text{maximum frame (round) rate} \)

We can pick optimal \( p \) as

\[ p = 2 \ l_{\text{max}} \ r_{\text{max}} \]
Cheating in Pipelined Lock-step
C makes its 4-th move after seeing the first p moves from A. A’s first six moves is not based on C’s move.
C makes its 4-th move after seeing the first p moves from A. A’s first six moves is not based on C’s move.
But it’s fair if all players do the same.
Player C can’t peek at extra moves if $C_{4,C}$ is received within $2I_{AC}$ of sending $M_{1,A}$.
We can detect late commit if we know the latency between players.
\[ \text{l}_{\text{max}} = \text{maximum latency} \]
\[ \text{r}_{\text{max}} = \text{maximum frame (round) rate} \]

**Pipeline Lockstep Protocol**
*without late commit*

\[ \frac{1}{r} = \max\left\{ \frac{\text{l}_{\text{max}}}{p}, \frac{1}{\text{r}_{\text{max}}} \right\} \]
\[ p = \text{l}_{\text{max}} \text{ r}_{\text{max}} \]
Player can lie about latency!
C can pretend to be on a slow network when measurement of $l_{AC}$ is done (e.g. using ping).
If fake $l_{AC}$ is larger than $l_{\text{max}}$, then we increase $p$, limiting the cheat to one round.
You Are Here

- CS4344
  - Client/Server Architecture
    - Synchronization Protocols
  - Interest Management
- Peer-to-Peer Architecture
  - Cheat-proof Synchronization Protocol