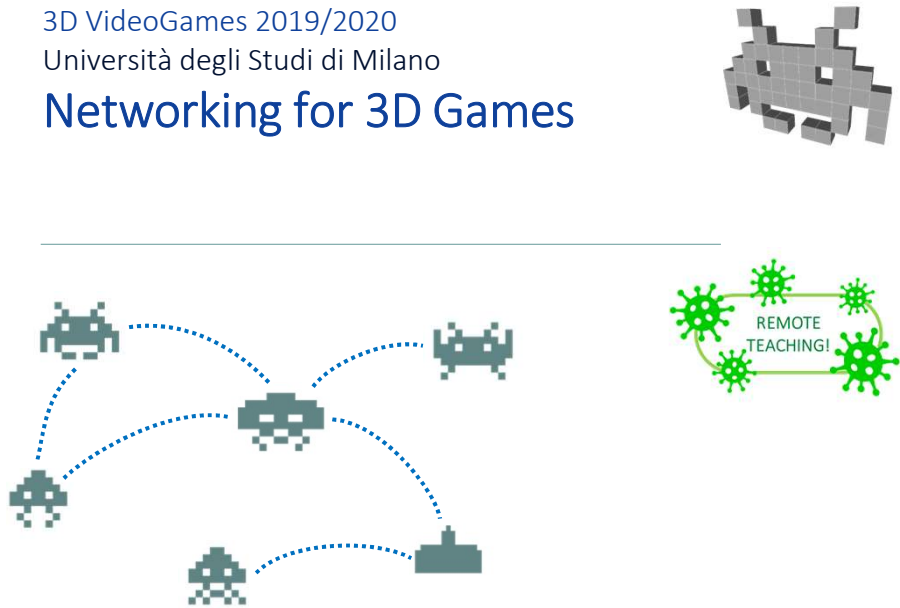


3D VideoGames 2019/2020
Università degli Studi di Milano

Networking for 3D Games



1

Course Plan

- lec. 1: Introduction ●
- lec. 2: Mathematics for 3D Games ●●●●●
- lec. 3: Scene Graph ●
- lec. 4: Game 3D Physics ●●● + ●●●
- lec. 5: Game Particle Systems ▶
- lec. 6: Game 3D Models ●◀
- lec. 7: Game Textures ●●
- lec. 8: Game 3D Animations ▶●●
- lec. 9: Game 3D Audio ●
- lec. 10: Networking for 3D Games ●◀
- lec. 11: Artificial Intelligence for 3D Games ●
- lec. 12: Game 3D Rendering Techniques ●●

2


Player 2 has joined the game

- **Multiplayer** game types, according to **gameplay**
 - collaborative
 - competitive
 - versus
 - teams...
- *How much* multiplayer?
 - no: single player
 - 2 players?
 - 10 players?
 - >100?
 - > 100000? } («massively» multiplayer, **MMO**)

3

Player 2 has joined the game

- Types of multiplayer games
 - Hot-seat
 - players time-share
 - Local multiplayer (Side-to-side)
 - e.g. Split screen
 - players share a terminal
 - **Networked**
 - each player on a terminal
 - terminals connected...
 - ...over a LAN
 - ...over the internet



4

Networking in Games

(main course: [Online Game Design](#))

- One task of the Game Engine
 - in practice, many engines leave much to do to game programmers
- Different scenarios:
 - [number of players](#)? (2, 10, 100, 100.000?)
 - game [pace](#)? (real time action != chess match)
 - [joining ongoing games](#) : allowed?
 - [cheating](#) : must it be prevented?
 - [security](#): is it an issue (e.g. DoS attacks)
 - [medium](#) : LAN only? internet too?
Latency tolerance? Bandwith tolerance?

5

Networking in 3D Games

Objective: all players *see* and *interact with* a [common](#) 3D virtual world



how can this illusion be achieved?

6

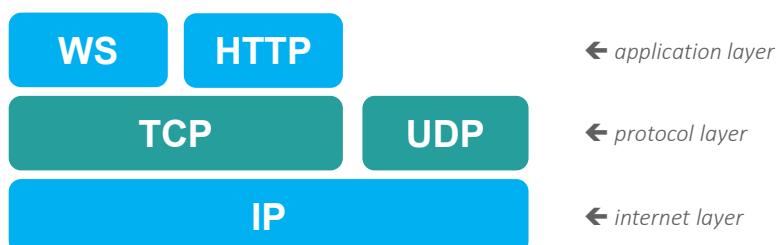
A few choices of a networked-game dev



- **What** to communicate?
 - complete status, status changes, inputs...
- How **often** ?
 - at which rate
- Over which **protocol** ?
 - TCP, UDP, WS ...
- Over which **network architecture** ?
 - Client/Sever, Peer-To-Peer
- How to deal with networking problems
 - **latency** ("lag") <= one main issue
 - limited **bandwidth**
 - connection loss


7


Reminder: Protocols



8


Protocols



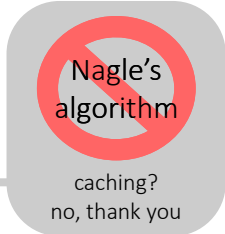
TCP sockets	UDP sockets
<ul style="list-style-type: none">● Connection based● Guaranteed reliable● Guaranteed ordered● Automatic breaking of data into packets● Flow control● Easy to use, feels like read and write data to a file	<ul style="list-style-type: none">● <i>What's a connection?</i> ● No reliability● No ordering● Break your data yourself● No flow control● Hard. Must detect and deal with problems yourself.

9

UDP vs TCP



- Problem with **TCP**
 - too many strong guarantees
 - they cost in terms of latency (==>lag)!
 - no good for time critical application
 - (if they have to be used, at least enable the option `TCP_NODELAY`)
- Problem with **UDP**
 - not enough guarantees
 - guarantees: "packets arrives all-or-nothing". The end.
 - no concept of connection
 - no timeouts, no handshake, a port receives from anyone
 - no guarantees: packets can arrive...
 - ...out of order :-O , ...not at all :-O , ...in multiple copies :-O



Nagle's algorithm

caching?
no, thank you

10

UDP vs TCP



- Problem with **TCP**
 - too many costly guarantees
- Problem with **UDP**
 - not enough guarantees
- The hard way:
 - use **UDP**,
but *manually re-implement a few guarantees*
 - best, for the most challenging scenario
 - fast paced games, not on LAN



11

Virtual connections over UDP: how-to (notes)



- add **connection ID** to packets
 - to filter out unrelated ones
- **time out** on prolonged silence (e.g. few secs)
 - declare “connection” dead
- add **serial number** to packets
 - to detect when one went missing / is out of order / is duplicate
 - (warning: int numbers *do* loop – solutions?)
- give **ack** back for received packets
 - optimize for lucky (& common) cases!
 - N (say 100) received msg == 1 ack (with bitmask)
 - resend? only a few times, **then give up (data expired)**
- congestion avoidance: measure **delivery time**
 - tune send-rate (packets-per-sec) accordingly
- obviously: **NON blocking** receives!

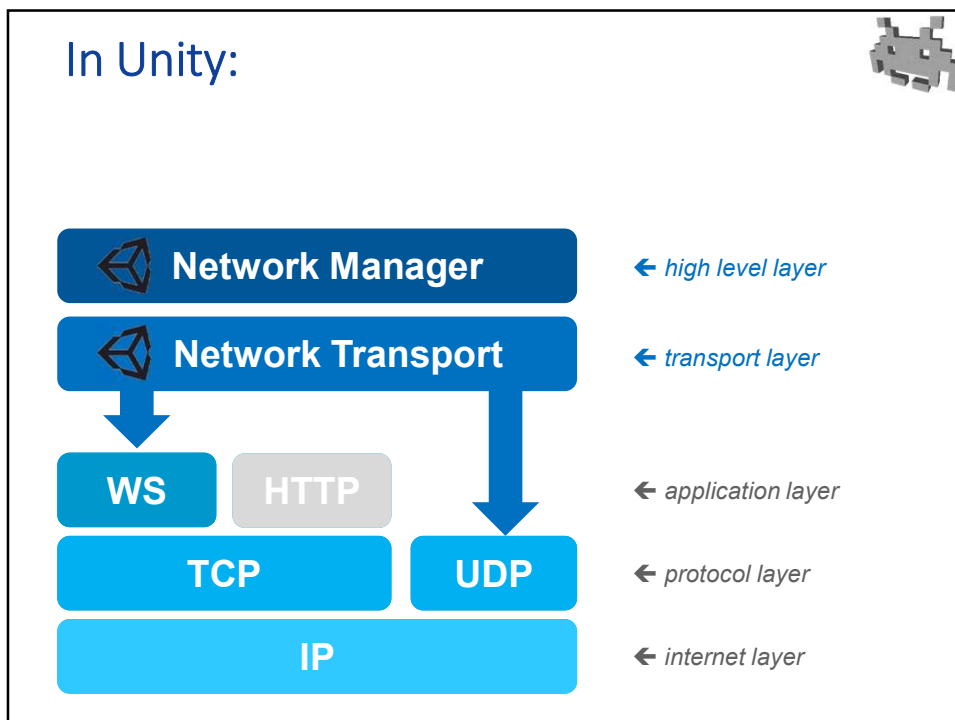
what **TPC**
doesn't
understand

12

Choosing a Protocol

- In summary, it is a question of pacing
 - fast paced game?
 - action games, FPS, ...
 - (sync every 20-100 msec)→ UDP necessary (unless LAN only)
 - slow paced game?
 - RTS, RPG...
 - (sync every ~500 msec)→ can get away with TPC
 - slower paced games?
 - MMORPGs, cards ...
 - (sync every few sec)→ why not just HTTP
 - traditional turn based ?
 - chess, checker
 - (sync every hour/day)→ may as well use EMAIL

13



14

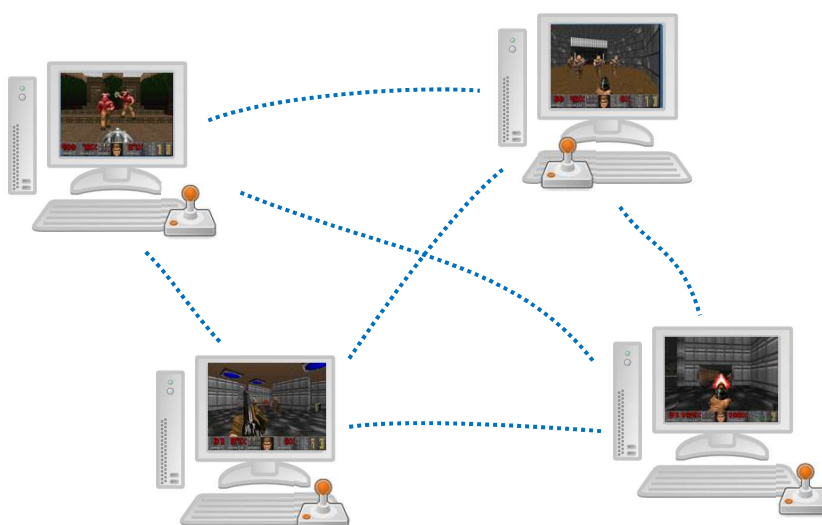
In Unity:



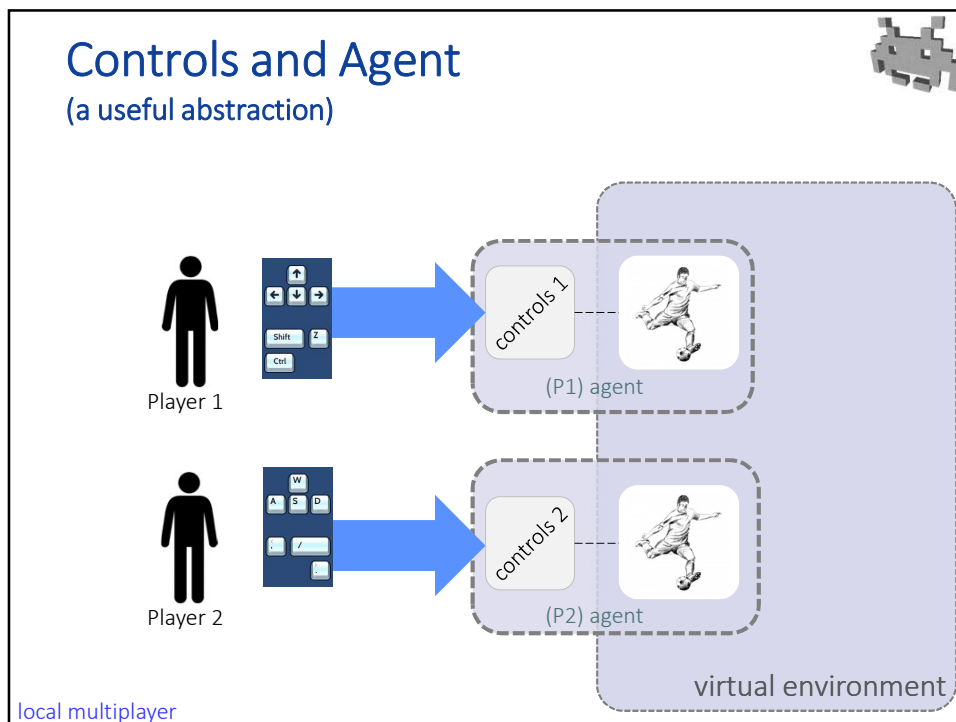
- Low level: **Transport Layer**
 - Builds up guarantees over UDP (connections)
 - Easy to use as TCP, but optimized for games
 - see how-to list above
 - Can work over WS instead UDP (abstracts the differences)
 - WS needs be used for web / WebGL games
- Hi level: **Network Manager**
 - presets network connectivity
 - standard “client hosted” games
 - server is also a player
 - controls shared state of the game
 - deals with clients
 - sends remote commands

15

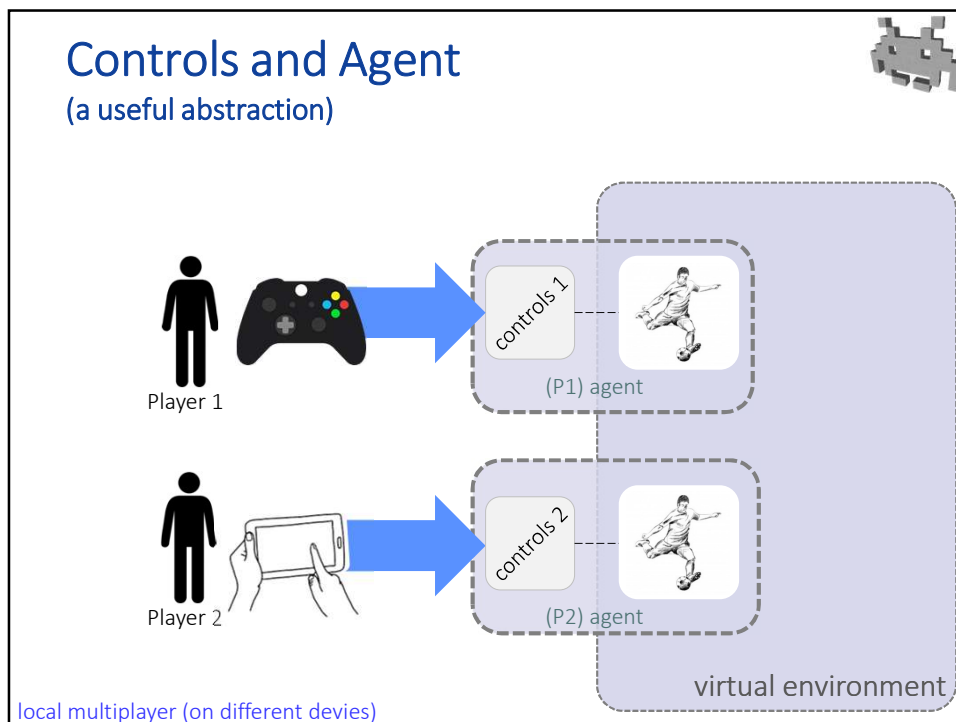
Network structure: Peer-to-Peer



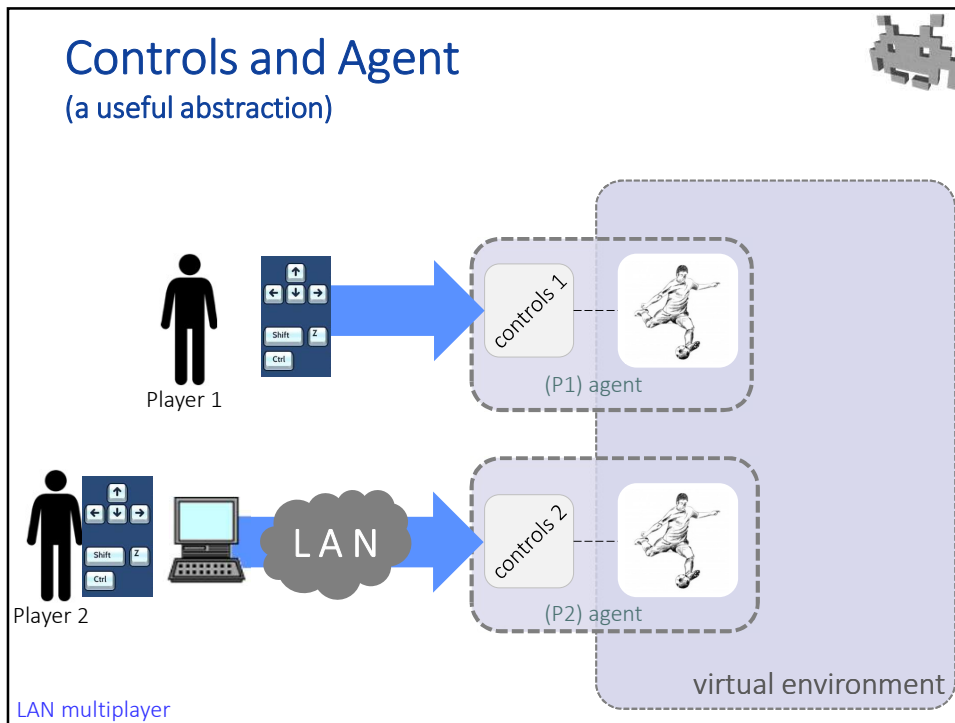
16



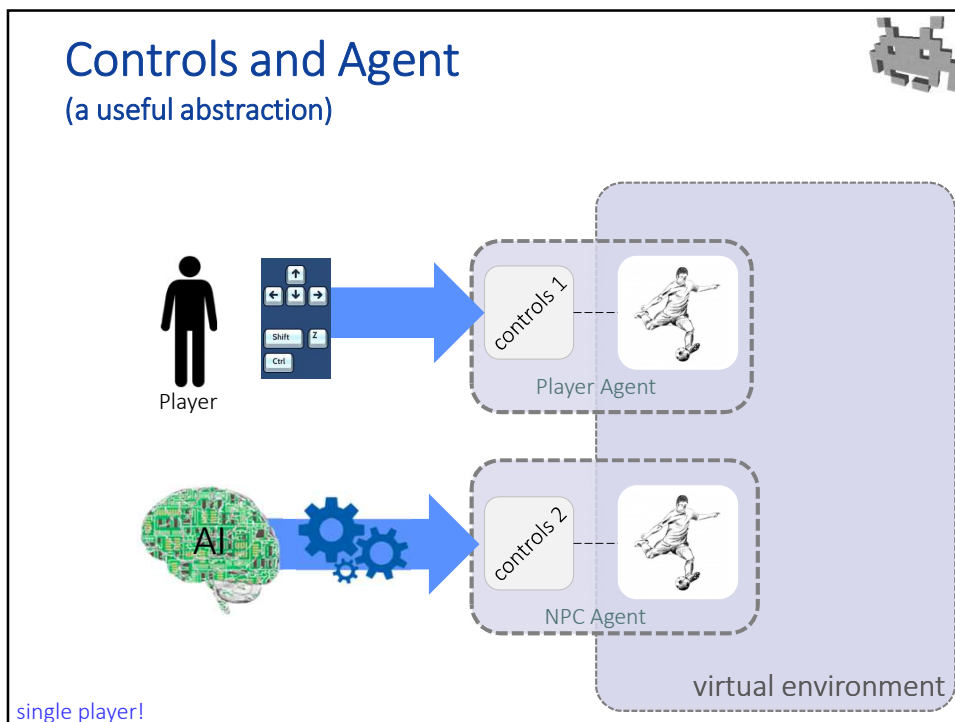
17



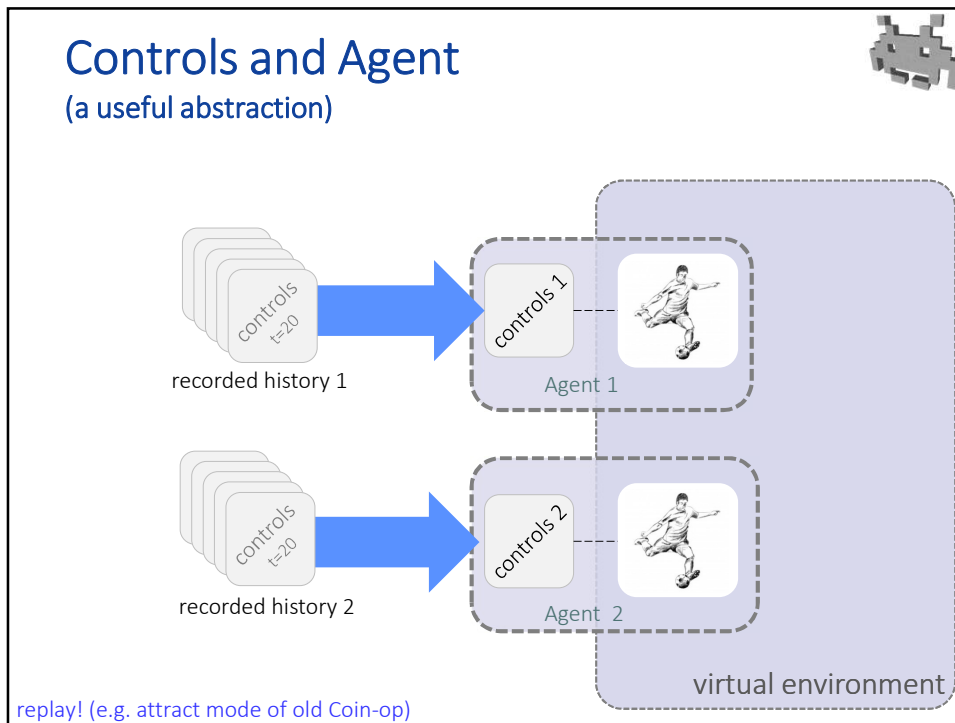
18



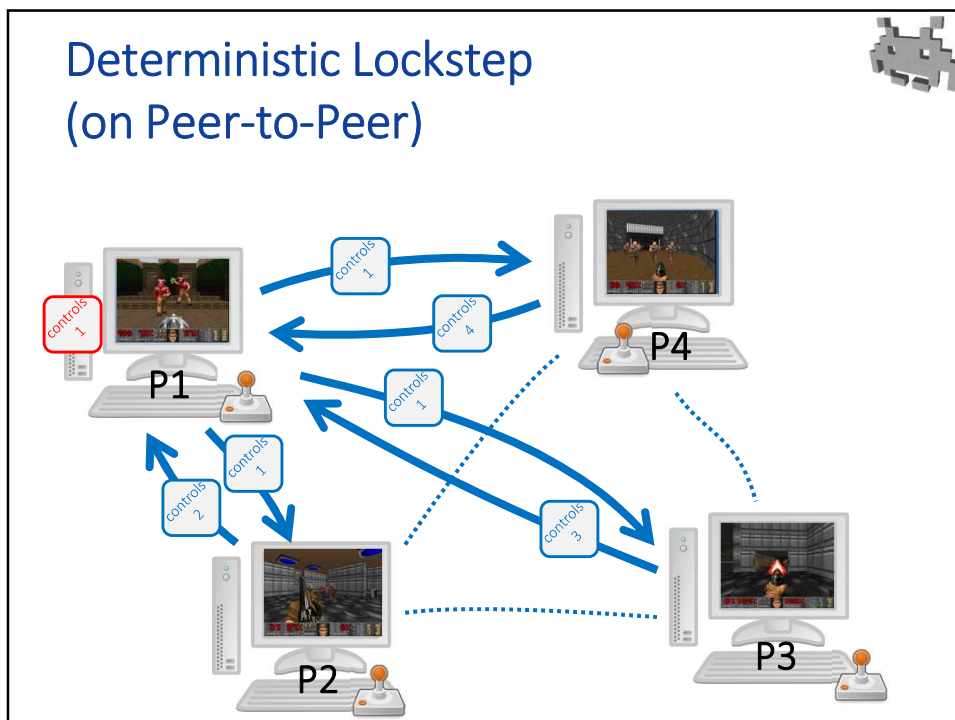
19



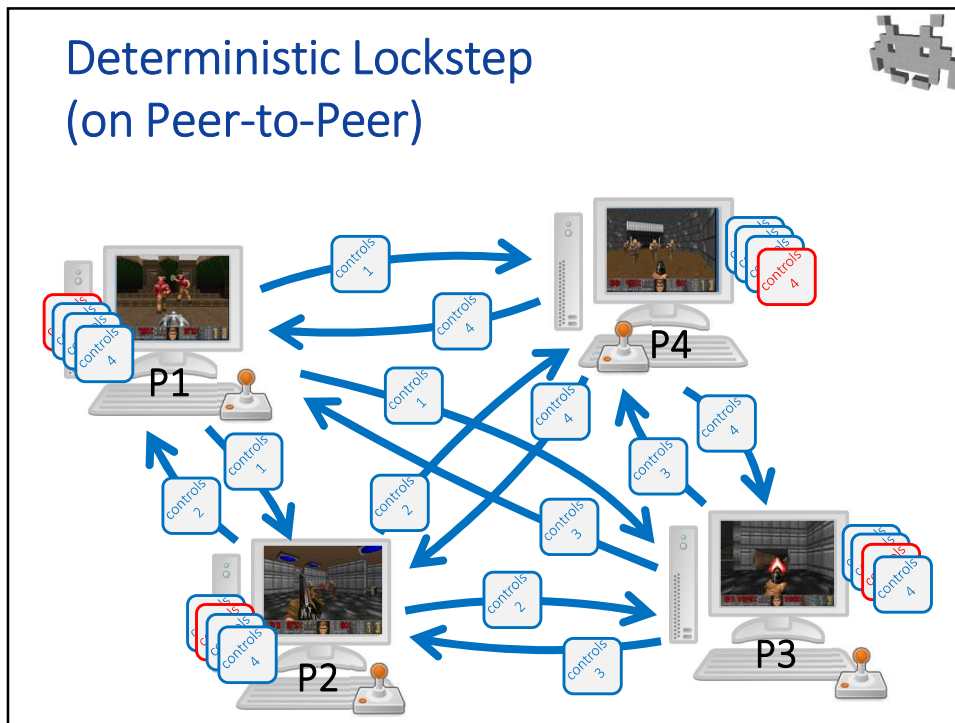
20



21




22



23

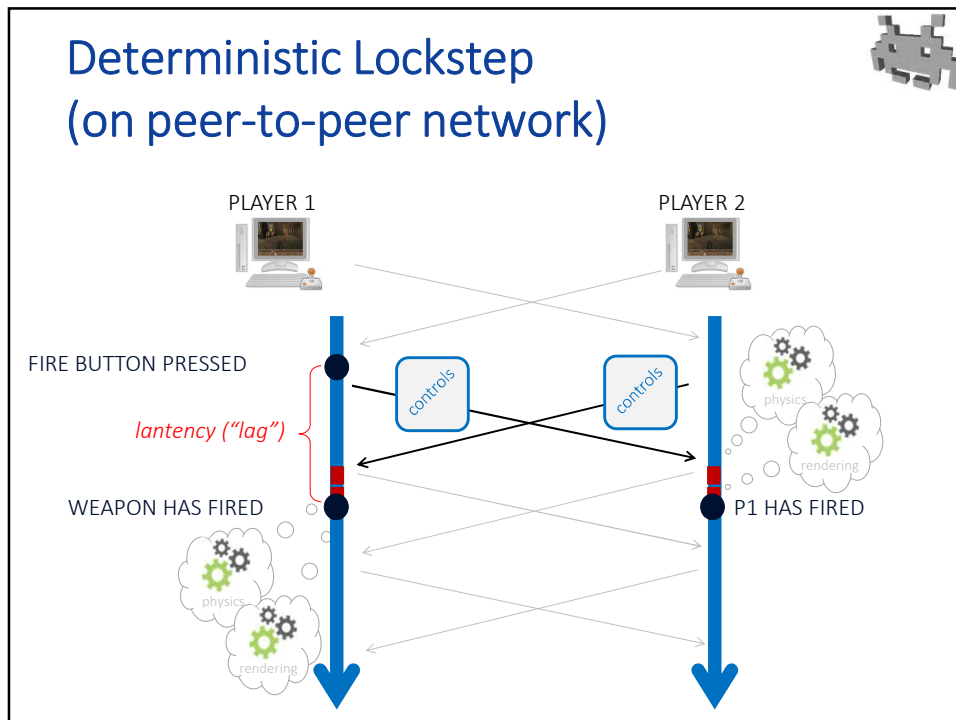
Deterministic Lockstep (on Peer-to-Peer)



- Game evolution = sequence of “turns”
 - e.g. physics steps (fixed dt !)
- Each node sends its current *controls* (inputs)
 - to everybody else
- After all controls are received, each node computes its own evolution
 - deterministically:
same input \rightarrow same result

← even if independently computed

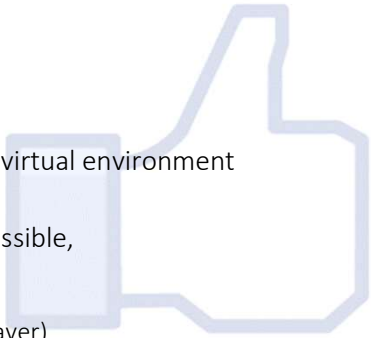
24



25

Deterministic Lockstep: the good

- elegant and simple! 😊
- minimal **bandwidth** needed
 - only sent data = controls
 - compact! (e.g. a bitmask)
 - does not depend on complexity of virtual environment
- **cheating**: inherently limited
 - but a few ways to cheat are still possible, e.g.:
 - aimbots (unlawful assist from AI)
 - x-rays (unlawful reveal of info to player)
- mixes well with:
 - non-cheating AI, replays, player performance recording...
- can use simple **TCP connections**
 - because we need 0% packet loss anyway (but...)



26

Deterministic Lockstep: can as well use TPC instead of UDP ?



- why yes:
 - TPC is so simple!
 - takes care of everything
 - works well, when no packet loss
 - (with loss, we need resend it anyway: let TPC do that)
 - makes little sense to use UDP and then... try to re-implement all TPC over it
 - at the beginning of dev, UDP is a (premature) optimization
- why not:
 - to degrade better with lost packets
 - e.g.: use redundancy – instead of resend-on-failure
 - controls are small: send 100+ controls in every packet
 - keep resending until ack received

27

Deterministic Lockstep

- Common, e.g., in:
 - RTS
 - controls = orders
 - can be fairly complex
 - but game status = much more complex
 - first generation FPS
 - controls = [gaze dir + key status]

...why not anymore?



Starcraft Blizzard 1998-2015



Command and Conquer
EA / Westmany et al
1995..2012





Age of Empires
Ensemble Studios et al,
1998..2015



Doom ID-soft, 1998

29


Deterministic Lockstep (on peer-to-peer): the bad



- **responsiveness:**
 - input-to-response delay of 1 x delivery time (even locally!)
 - (you cannot act immediately even on your own local input)
- **does not scale** with number of players
 - quadratic number of packets
 - 2P ok, 100P not ok
- **input rate = packet delivery rate**
- **delivery rate** = as fast as the *slowest* connection allows
- connection problems (anywhere): everybody freezes!
- joining ongoing games: difficult
 - needs sends full game state to new player
- assumes full agreement on initial conditions
 - ok, that is fairly easy to get
- **assumes complete determinism!**

30

Determinism: traps

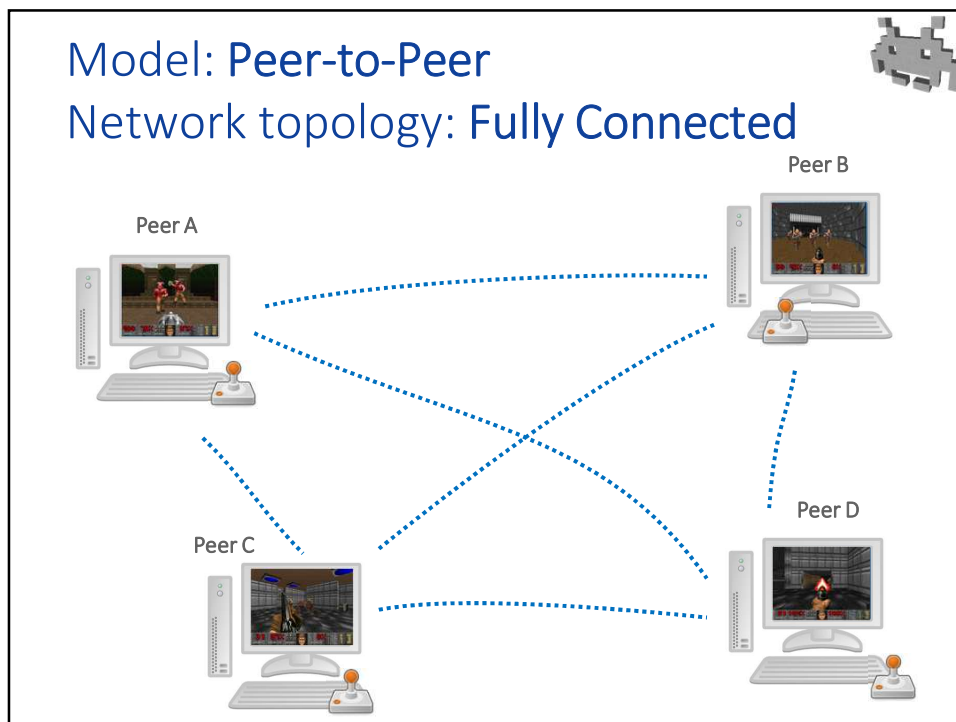


- Pseudo-**Random**? → not dangerous
 - fully deterministic (just agree on the seed)
- ⚠ **Physics:** many preclusions and traps
 - ⚠ variable time step? **bad**
 - ⚠ time budgeting? **bad**
 - ⚠ hidden threats:
 - order of processing of particles/constraints
- ⚠ anything that depends on **clock**?
→ **poison** to determinism
- ⚠ GPU computations? **very dangerous**
 - slightly different outcome on each card
- ⚠ **floating point** operations?
 - **many hidden dangers**,
e.g. different hardwired implementations
 - best to assume very little (**fixed point** is much safer)
- ⚠ NOTE: 99.999% correct == **not correct**
 - virtual world is faithful to reality enough to be chaotic → butterfly effect:
the tiniest local difference == completely different outcomes soon

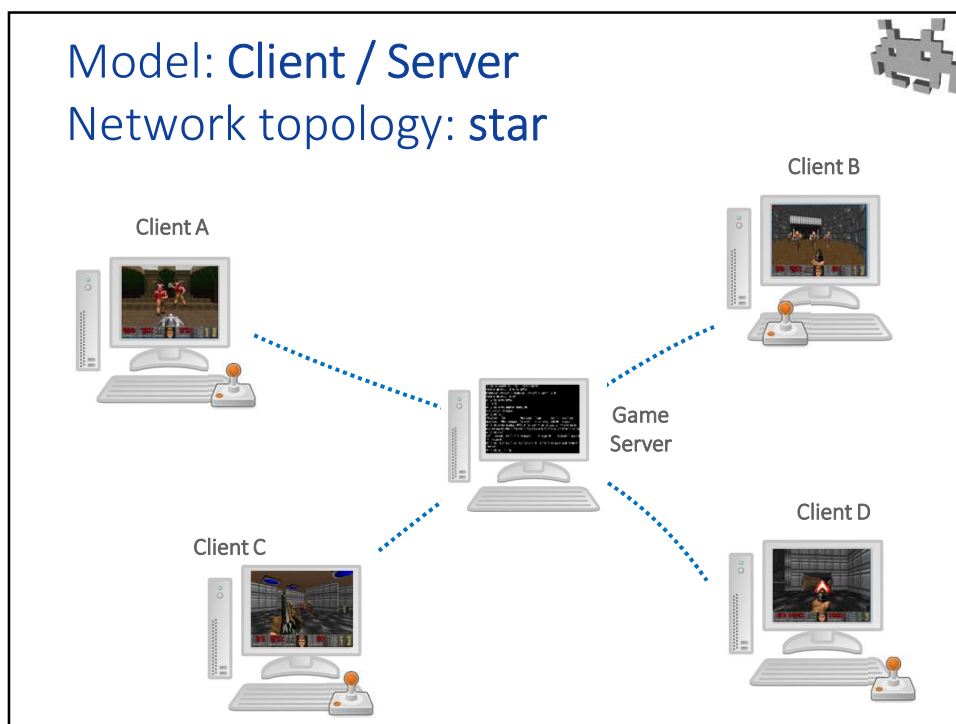
The entire game system must be designed from the start with “determinism” in mind ...

...and it still difficult to get (and debug)

31



32



33

Deterministic lockstep with Star-shaped network



- Server sits on central node
- Protocol
 - Each client sends his controls to server
 - Server collects all controls and sends back to clients
- Advantage:
 - scalability:
number of packets becomes linear (not quadratic)
- Cost:
 - **responsiveness:**
latency = 2 x delivery time :-O
- Bonus: the server can now be made **authoritative**
 - Many new options available. For example...



hurts
gameplay!

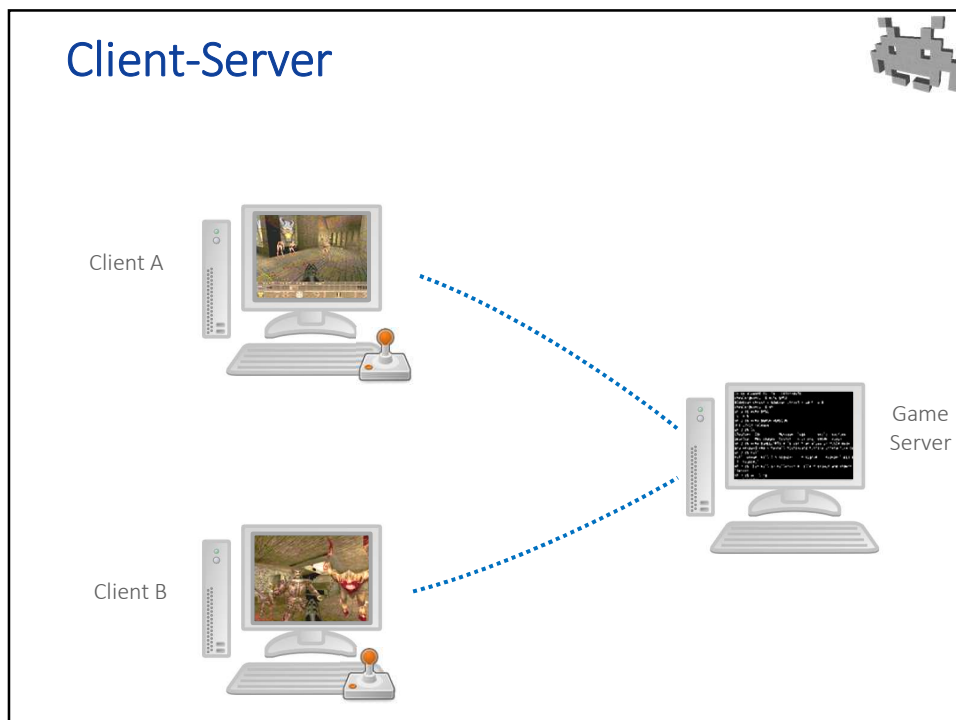
34

“Server is the man” (authoritative server)

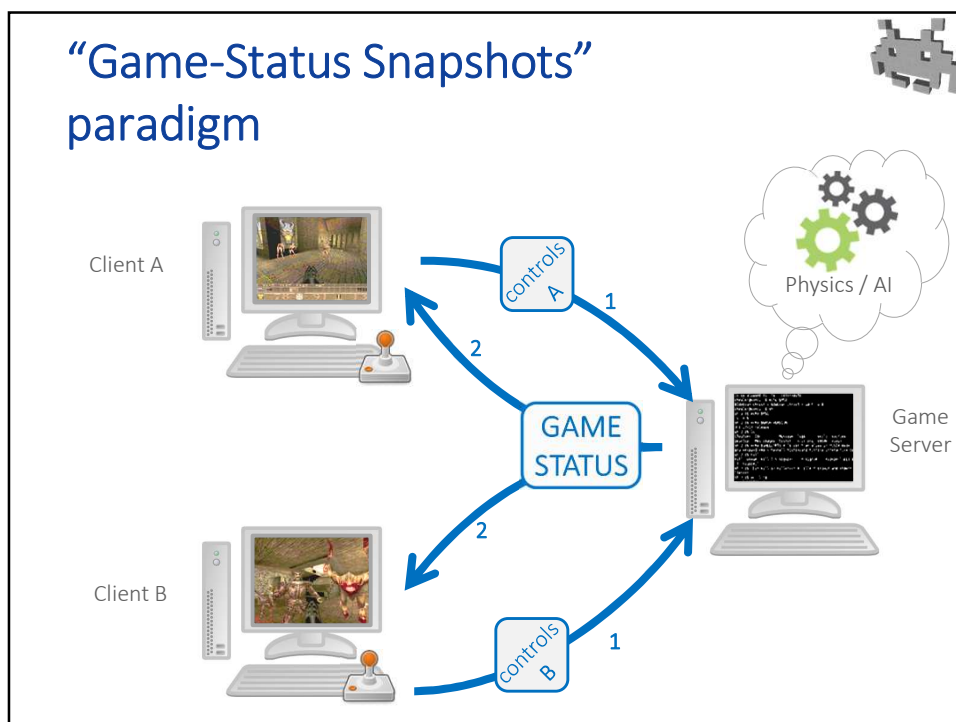


- The server: it has the last word
- For example:
 - Packet loss from player 3?
Server makes up control for player 3
(instead of waiting for them)
 - Note: server *defines* what player 3 eventually did,
not player 3 itself!
 - i.e. clients take server's word even for its own actions
 - Packet loss affects one player only

35





36

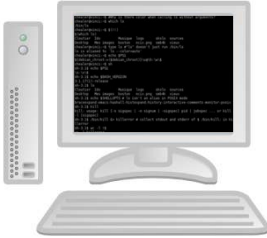



37

Game-Status Snapshots




- Client:
 - just a remote visualizer of the current status
 - status is "read only"
 - (+ remote input collector)
- Server:
 - computation of the evolving status
 - (including physics)
 - it's where the "real game" runs

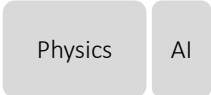
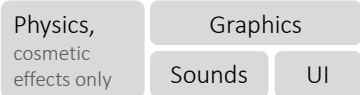


39

Game-Status Snapshots



- Client:
 - connected: to server only
 - captures input
 - sends controls
 - receives game status
 - or relevant portions of it
 - renders it
 - using all relevant assets
- Server
 - connected: to all players
 - receives all controls
 - (missing? doesn't matter)
 - updates game status
 - physical simulations, etc
 - sends current status
 - to all



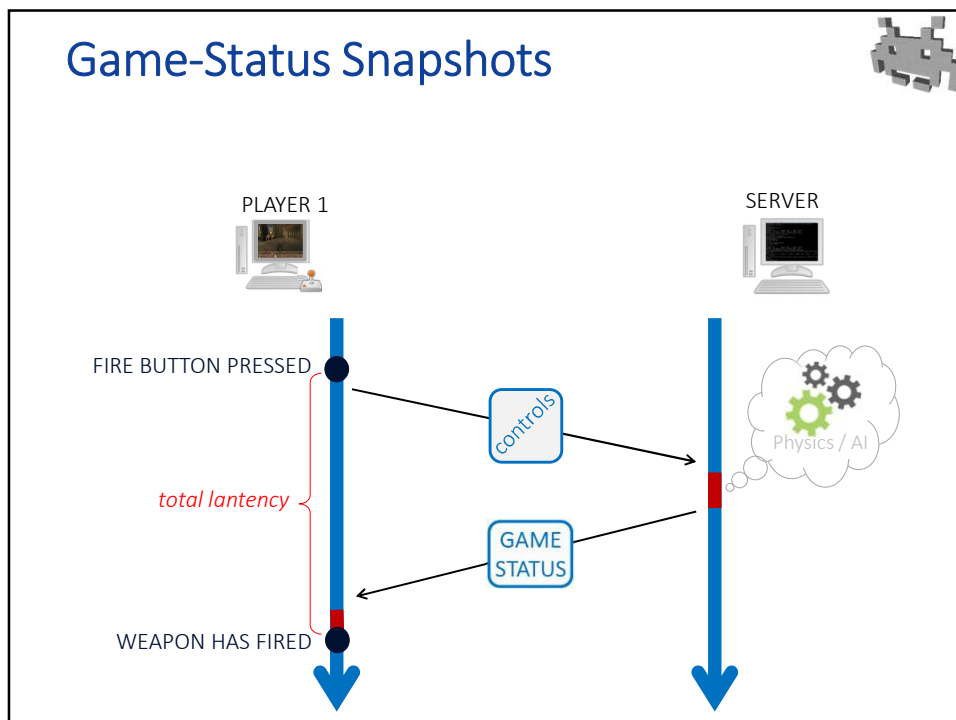
40

Game-Status Snapshots

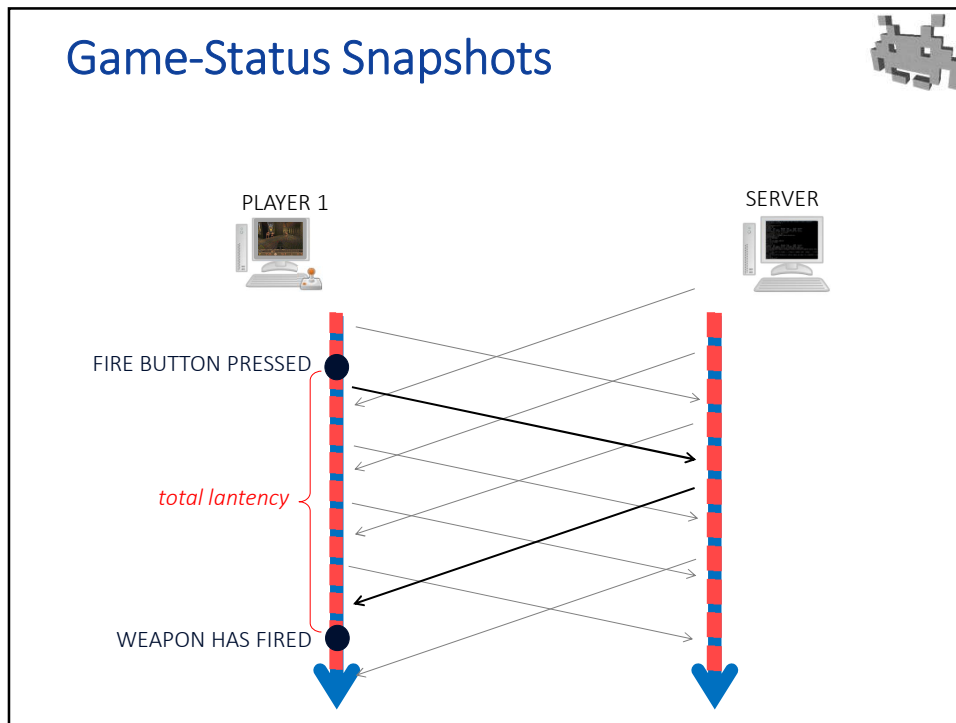
- the gains:
 - determinism: no longer needs be assumed
 - joining ongoing games: trivial now
 - packet loss: bearable (hurts the player *only*)
 - to profit: **UDP**
 - slower connection: bearable (affects that player *only*)
- the losses:
 - packet size: a lot bigger!
 - optimizations, to counter this:
 - compress world status
 - send to each client only the portions which interest its player
 - **responsiveness:**
from input to effect = delivery time :-(
from input to visual = 2 x delivery time :-O

hurts gameplay!

41



42

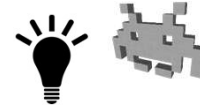


Game-Status Snapshots: with Interpolation: the idea

- World “Snapshot” contains:
 - data needed for 3D rendering:
(position-orientation of objects, plus anything else needed)
- Problem:
 - large snapshot size! (even with optimizations)
 - ==> few FPS (in the physical simulation)
 - ==> “jerky” animations
- Solution 1: client-side **interpolation**
 - client keeps last two snapshots in memory
 - last received one + the previous one
 - interpolates between them,
 - client lags behind server by even more!
 - gain: smoothness (high FPS with low packet - rate)
 - loss: responsiveness (increased latency) **oh noes!**

44

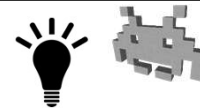
Game-Status Snapshots: with Extrapolation: the idea



- World “Snapshot” contains:
 - data needed for 3D rendering:
(position-orientation of objects, plus anything else needed)
- Problem:
 - large snapshot size! (even with optimizations)
 - ==> few FPS (in the physical simulation)
 - ==> “jerky” animations
- Solution 2: client-side **extrapolation**
 - clients keeps last two snapshots in memory
 - last received one + the previous one
 - extrapolates between them, i.e. shows the expected “future”
 - i.e. it shows an attempted prediction to the next snapshot
 - NOTE: this prediction is often wrong: glitches.
 - gain: responsiveness
 - loss: accuracy - lots of glitches. :-(

45

Client-side Game Evolution (aka *distributed physics*): the idea

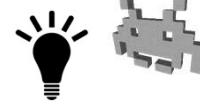


- Each client:
 - in charge for game evolution
 - including physics
 - communicates to others a reduced game-status snapshot
 - describes only status of own player
(e.g. positions + ori, its flying bullets)
 - receives other partial snapshots
 - merges everything up
 - (updates statuses of other players)
- Simple, zeroed latency
 - immediately responsive to local player controls
 - remote agents updated according to “what their client says”
- Problem: can still need determinism
 - (who keeps NPCs / environment in sync?)
- Problem: **authoritative clients**: prone to **cheating!!!**

to server,
or , in a P2P network,
to each other peers

46

Client-Side Prediction: the idea



- Client:
 - get Commands from local inputs
 - sends Commands to Server
 - computes game evolution (the prediction)
 - maybe “guessing” other players commands (which it ignores)
 - zero latency!
- Server:
 - receives Commands (from all clients)
 - computes game evolution (the “reality”)
 - “*Server Is The Man*” - Tim Sweeney (Unreal Engine, EPIC)
 - prevents cheating!
 - sends Snapshot back (to all clients)
- Client:
 - receives Snapshot (the “real” game status)
 - corrects its prediction, *if needed*

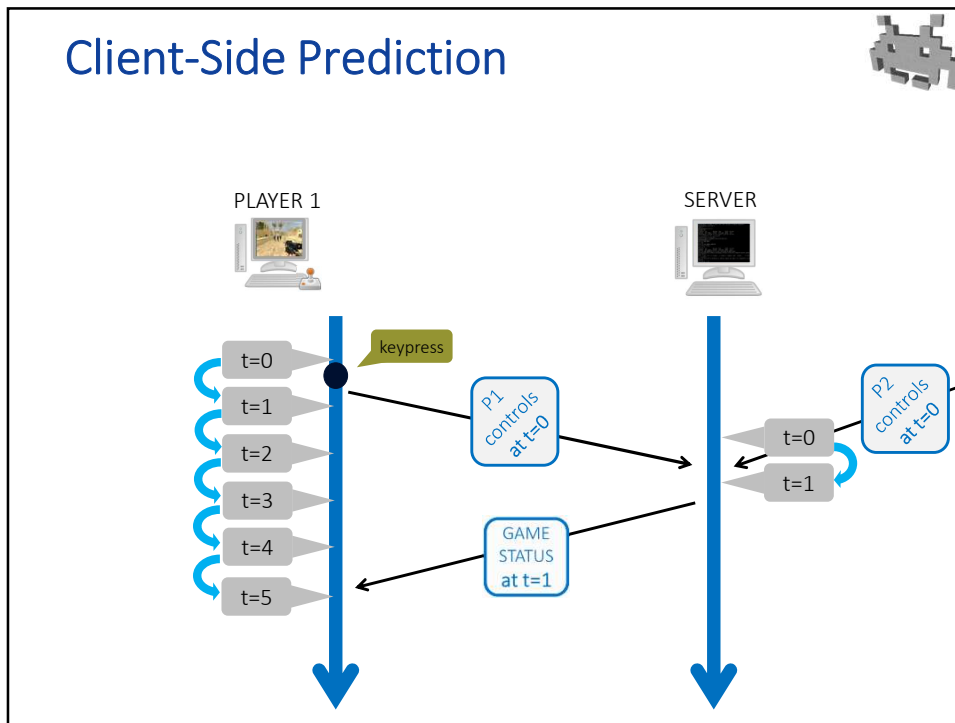
47

Client-Side Prediction: correction from the server

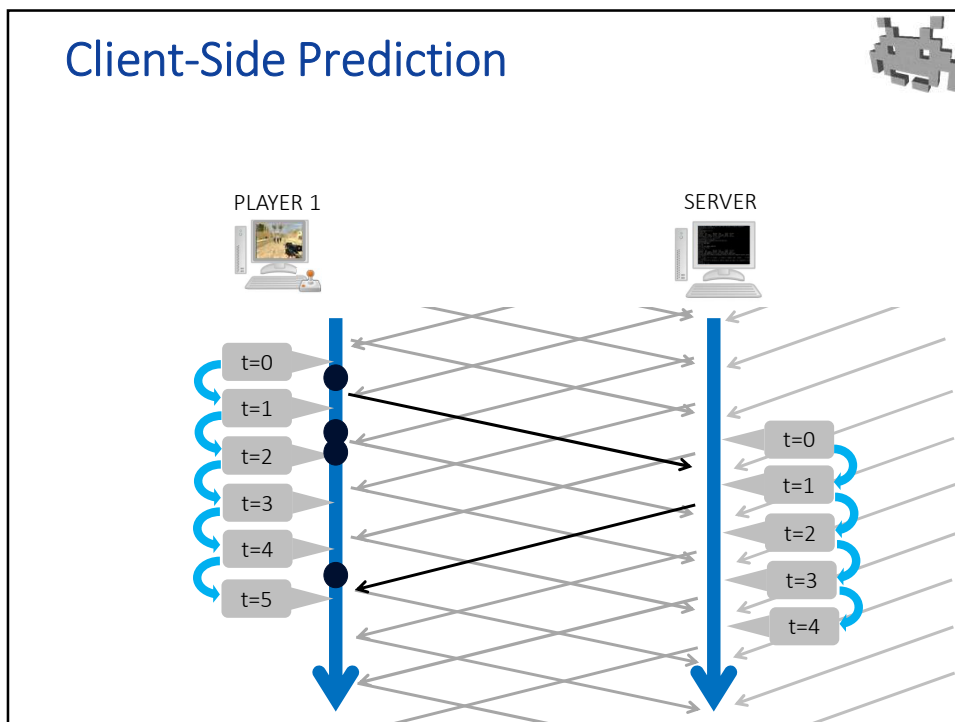


- The server-side “real” simulation lives k msecs in the past of the client-side “predicted” one
 - k = deliver time
 - remember: virtual time != real world time
- When server correction arrives to client, it refers to $2k$ msecs ago (for the client)
- Q: how to correct... *the past?*

48



49



50

Client-Side Prediction: correction from the server



- Q: How to correct... *the past*?
- A:
 - keep last N statuses in memory
 - including own controls
 - “real” status (the correction) of the past arrives from server
 - compare it with corresponding past status:
 - does it **match**?
nothing to do
 - does it **mismatch**?
discard frame and following ones,
rerun simulation to present (reusing stored controls)

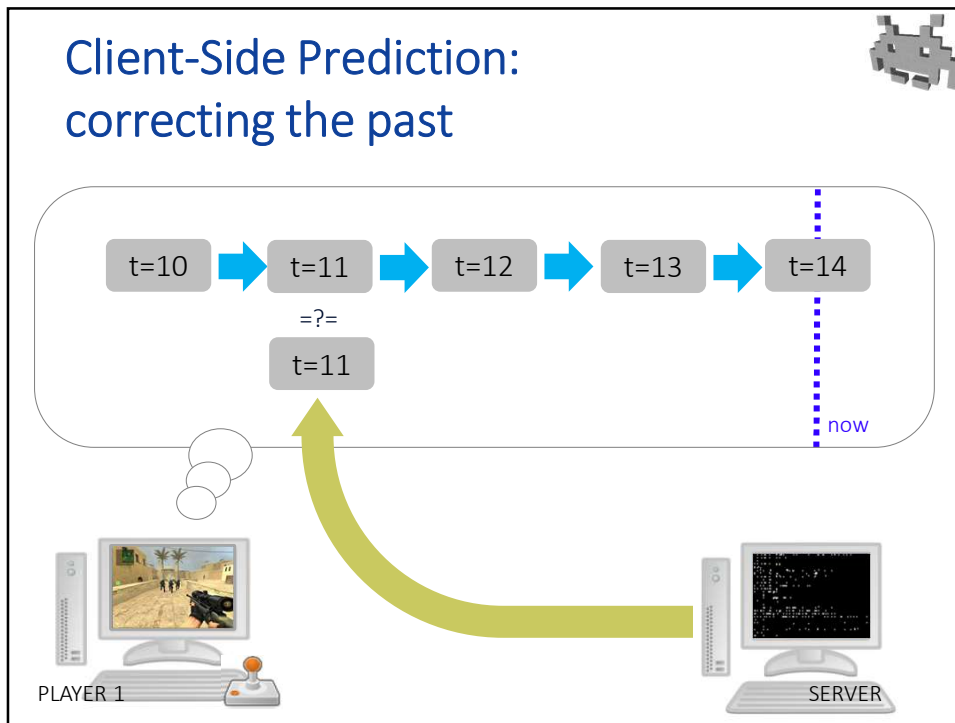
51

Re-running physical simulation

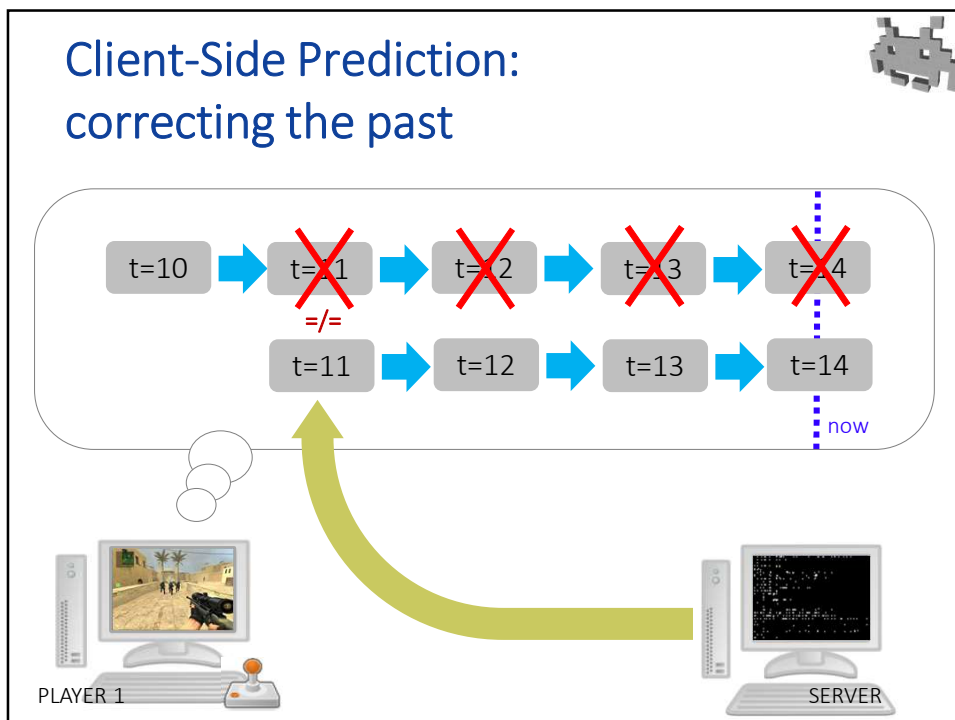


- Just need to catch up with the present
- Physics and AI only... at full speed
 - no graphics, no sound rendering,
no cosmetic particle system...
- Can use larger dt if necessary
 - Compromises accuracy
- Must reuse same controls of own player
 - Which are also cached
- Note: player is never shown these intermediate steps. Only the final result
- Glitches when going from current present to a different (corrected) present

52



53



54

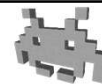
Client-Side Prediction: what causes mispredictions?



- Lack of **determinism**
 - e.g. physics was approximated – “soft real time”
 - see above for more possible causes of this
 - (minor/rare issue)
- Didn't account that *own* controls were **not received** by server (in time)
 - server: “actually, you didn't jump, back then”
 - authoritative server – server *defines* the truth, (even when the client is in a better position to know)
 - (minor/rare issue)
- Didn't account for **other players' controls**
 - (the biggest issue)
- Note: none of the above breaks the game (hopefully)
 - it just causes minor / temporary glitches (maybe)

55

Client-Side Prediction: optimizations 1/2



- reduce snapshots size
(==> to increase packet frequency)
 - partial snapshots: refresh more often the parts which are most likely to be predicted wrong / or which changed
 - drastic space reductions!
 - but make sure that every part is eventually refreshed
- reduce correction computation
(==> so to make corrections quicker)
 - partial physic steps:
 - update only the parts affected by the error
 - use bigger dt (fewer steps to get to present)

56

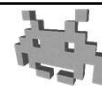
Client-Side Prediction: optimizations 2/2



- tentatively predict also unknown data
(==> so to reduce correction frequency)
 - e.g. also predict other player's controls
 - easiest prediction: players do what they did last frame
- trigger correction only when status differ *enough*
(==> so to reduce correction frequency)
 - e.g. when any spatial position difference > epsilon
 - tolerate small discrepancies
 - (warning: discrepancies tend to explode exponentially with virtual time – because Chaos)

57

Client-Side Prediction: notes



- A snapshot = includes physical data
 - (not just for the 3D rendering, also to update physics)
 - can be small, when optimized!
- 😊 No **latency**: immediately react to local input
 - client proceeds right away with next frame
 - *when prediction is correct*: seamless illusion
 - *otherwise*: (minor?) glitches
- 😊 **Determinism**: not assumed
- 😊 **Cheating**: not easy (server is **authoritative**)

58

Summary : rules of thumb



- How to choose the network layout
 - **peer-to-peer** :
 - ☺ reduced latency
 - ☹ quadratic number of packages (with number of players)
 - **client-server** :
 - ☹ doubled latency
 - ☺ linear number of packages (with number of players)
 - *REQUIRED*, for any solution with **authoritative server**
 - *REQUIRED*, for num players $\gg 4$

59

Summary : rules of thumb



- How to choose the network paradigm
 - **Deterministic Lockstep**, if
 - determinism can be assumed
 - few players (up to 4-5)
 - fast + reliable connection (e.g. LAN)

← **RTS**
most common option !

} or, slow paced game
 - **Game-status Snapshots**, if
 - game status not overly complex
 - a little latency can be tolerated
 - **Client-side evolution**, if
 - preventing cheating not important
 - **Client-side prediction + server correction**, if
 - game status not overly complex

↓ **FPS**
most common option !

60

Summary: classes of solutions

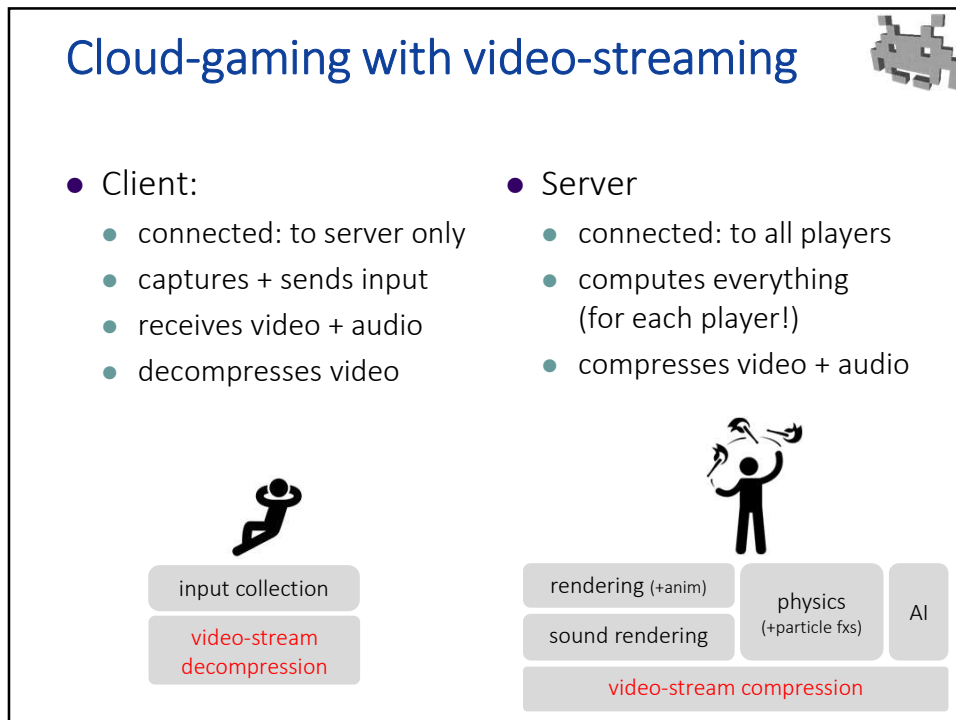
- Who computes game evolution? (incl. physics)
 - **deterministic-lockstep** : clients
 - there may be no server at all: peer-to-peer
 - independent computation, same result
 - **game-status snapshots** : server
 - clients are just visualizers
 - maybe with interpolation / extrapolation
 - **(distributed physics** : both clients and server)
 - clients in charge for own agent(s)
 - server in charge for env. / NPCs
 - **client-side predictions** : both clients and server
 - clients “predict” (just for local visualization purposes)
 - server “corrects” (it has the last word!)

61

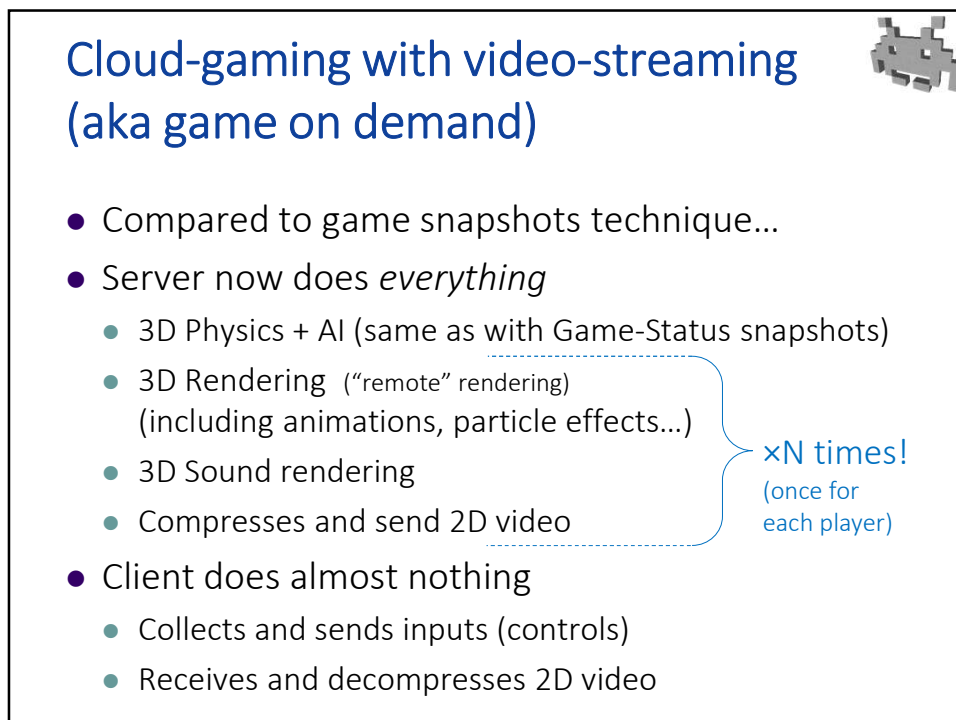
Cloud-gaming with video-streaming (aka game on demand)



62



63



64

Cloud-gaming with video-streaming



- Advantages: client is “thin”
 - client does (almost) nothing
 - client needs nothing (no asset, no storage)
 - total: client capabilities can be extremely limited (a pad)
- Challenges:
 - Server must be very powerful
 - High bandwidth required (high-res video + audio)
 - Latency!!! It now includes two-ways trip, plus compression by server, plus decompression by client
 - Video resolution: now makes thing more difficult

Luckily, video-on-demand technologies can be reused

65

Cloud-gaming with video-streaming (aka game on demand)



- Fast growing technique
- Latency = maybe 80-100 ms
 - Is this acceptable?
- Bandwidth = min 25-50 mbits/s
- Will it work for hard-core games?



66