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*)
Player 2 has joined the game

- Multiplayer game types
  - 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
      - ...

Networking in Games

Objective: all players see and interact with a common virtual world

how can this illusion be achieved?
Networking in Games

- 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: needs be prevented?
  - medium: LAN only? internet too? bandwidth?

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…
- Over which network architecture?
  - Client/Server, Peer-To-Peer
- How to deal with networking problems
  - latency (“lag”) <= main problem
  - limited bandwidth
  - connection loss
Reminder: Protocols

Protocols

TCP sockets
- 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

UDP sockets
- What's a connection?
- No reliability
- No ordering
- Break your data yourself
- No flow control
- Hard. Must detect and deal with problems yourself.

HTTP
TCP
UDP
IP

(application layer)
(protocol layer)
(internet layer)
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

The hard way:
- use **UDP**, but *manually re-implement a few guarantees*
- best, for the most challenging scenario
  - fast paced games, not on LAN
Virtual connections over UDP:

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 dupl
  - (warning: int numbers do loop – solutions?)
- give **ack** back for received packets
  - optimize for lucky (i.e. 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!

Choosing a Protocol

- In summary, it is a question of pacing
  - fast paced game?
    - action games, FPS, ...
    - (sync every 20-100 msec)
  - slow paced game?
    - RTS, RPG...
    - (sync every ~500 msec)
  - slower paced games?
    - MMORPGs, cards ...
    - (sync every few sec)
  - tradit. turn based?
    - chess, checker
    - (sync every hour/day)

  - **UDP necessary** (unless LAN only)
  - can get away with **TPC**
  - why not just **HTTP**
  - may as well use **EMAIL**
Network structure: Peer-to-Peer

Controls and Agent
(an useful abstraction)
Controls and Agent
(an useful abstraction)

LAN multiplayer

Player 1
(P1) agent

Player 2
(P2) agent

controls 1
controls 2

virtual environment

Controls and Agent
(an useful abstraction)

Player

Player Agent

NPC Agent

controls 1
controls 2

virtual environment

(single player!)
Controls and Agent
(an useful abstraction)

(recorded history 1)
Controls 1
Agent 1

(recorded history 2)
Controls 2
Agent 2

(replay!) virtual environment

Deterministic Lockstep
(on Peer-to-Peer)

P1  P2  P3  P4
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 → same result

**even if independently computed**
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 v. limited
    - but, e.g.
      - aimbots (unlawful help from AI)
      - x-rays (unlawful reveal of info to player)
    - still possible
  - mixes well with:
    - non-cheating AI, replays, player performance recording…
  - can use simple **TCP connections**
    - because we need 0% packet loss anyway (but…)

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 reemployment 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 the last 100 control in every packet
    - go on resending until ack received
Peer-to-Peer Deterministic Lockstep

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

...why not anymore?

Peer-to-Peer Deterministic Lockstep

- The bad:
  - responsiveness:
    - input-to-response delay of 1 x delivery time (even locally!)
  - does not scale with number of players
    - quadratic number of packets
    - 2 ok, 100 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!
Determinism: traps

- Pseudo-Random? → not dangerous
  - fully deterministic (just agree on the seed)

⚠ Physics: many preclusions and traps
  ▲ variable time step? bad
  ▲ time budget? 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: much better)

⚠ NOTE: 99.999999% correct == not correct
  - virtual world is faithful to reality enough to be chaotic → butterfly effect:
    the tiniest local difference == completely different outcomes soon

Network structure: Client-Server

The entire game system must be designed from the start with this objective in mind …

… and it still difficult to get (and debug)
### Client-Server: Game-Status Snapshots

1. **Client:**
   - just a remote visualizer of the current status
   - "read only"
   - (+ remote input collector)

2. **Server:**
   - computation of the evolving status
   - (including physics)
   - it’s where the “real game” runs

---

```
Client A
```

```
Game Server
```

```
Client B
```
Client-Server: 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

---

We gained:
- determinism: no longer assumed
- number of packets: linear now
- joining ongoing games: trivial now
- packet loss: bearable (poor player)
  - to profit: UDP
- slow connection: affects that player only :D

We lost:
- packet size: a lot bigger! ==> fewer packets
  - optimize:
    - compress world status
    - send to each client only the portions which interest its player
- responsiveness:
  - latency = 2 x delivery time :-O

Hurts gameplay!
Game-Status Snapshots (on client-server network)

PLAYER 1
FIRE BUTTON PRESSED
WEAPON HAS FIRED

total lantency ≤

SERVER
GAME STATUS

Game-Status Snapshots (on client-server network)

PLAYER 1
FIRE BUTTON PRESSED
WEAPON HAS FIRED

total latency ≥
Deterministic Lockstep (on peer-to-peer network)

Game-Status Snapshots: with Interpolation

- World “Snapshot” contains:
  - data needed for 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
  - clients keeps last two snapshots in memory
    - current one, last one
    - interpolates between them, i.e. shows “the past”
    - lagging behind server by one more snapshot!
  - gain: smoothness
  - lose: responsiveness (increased latency) oh noes!
Game-Status Snapshots: with Extrapolation

- World “Snapshot” contains:
  - data needed for 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
    - current one, last one
  - extrapolates between them, i.e. shows the expected “future”
    - NOTE: this crude prediction is often wrong: glitches.
  - gain: smoothness
  - lose: accuracy. Lots of glitches.

Client-side Game Evolution (aka distributed physics)

- 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 client: prone to cheating!!!
**Client-Side Prediction: the idea**

- **Client:**
  - get Commands from local inputs
  - sends Commands to Server
  - computes game evolution (the prediction)
    - “guessing” other players commands
    - 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*

**Client Side Prediction: correction from the server**

- The server-side “real” simulation lives $k$ ms in the past of the client-side “predicted” one
  - $k = \text{deliver time}$
  - recall: virtual time $\neq$ real world time

- When server correction arrives to client, it refers to $2k$ ms ago (for the client)

- Q: How to correct… the past?
Client Side Prediction

Player 1

- \( t=0 \)
- \( t=1 \)
- \( t=2 \)
- \( t=3 \)
- \( t=4 \)
- \( t=5 \)

KeyPress

Player 2

- \( t=0 \)
- \( t=1 \)
- \( t=2 \)
- \( t=3 \)
- \( t=4 \)

GAME STATUS (at \( t=1 \))

Client Side Prediction

Player 1

- \( t=0 \)
- \( t=1 \)
- \( t=2 \)
- \( t=3 \)
- \( t=4 \)
- \( t=5 \)

Player 2

- \( t=0 \)
- \( t=1 \)
- \( t=2 \)
- \( t=3 \)
- \( t=4 \)
Client Side Prediction: correction from the server

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

Client Side Prediction: correction in the past

```
t=10  t=11  t=12  t=13  t=14
```

$\Rightarrow$ t=11

PLAYER 1 $\Rightarrow$ SERVER

now
**Client Side Prediction: correction in the past**

```
<table>
<thead>
<tr>
<th>t=10</th>
<th>t=11</th>
<th>t=12</th>
<th>t=13</th>
<th>t=14</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>/\</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Client Side Prediction: optimizations**

- reduce snapshots size
  
  (==> increase packet frequency)

- partial snapshots:
  
  refresh more often the parts which are most likely to be predicted wrong

- drastic space reductions!

- reduce correction overload
  
  (==> quicker corrections)

- partial physic steps:
  
  update only the parts affected by the error

- better prediction of unknown data
  
  (==> less frequent corrections)

- e.g. of other player’s controls
Client Side Prediction:
notes

- A snapshot = includes physical data
  - (not just for the 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)

Summary : rules of thumb

- How to choose the network layout
  - peer-to-peer :
    - ☺ reduced latency
    - ☺ quadratic number of packages
      (with number of player)
  - client-server :
    - ☺ doubled latency
    - ☺ linear number of packages
      (with number of player)
    - REQUIRED, for any solution with authoritative server
    - REQUIRED, for num players >> 4
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)
  - **Client-side evolution**, if
    - cheating not a problem
    - game status not overly complex
    - determinism, or,
      - no NPCs / complex environment elements
  - **Client-side prediction + server correction**, if
    - game status not overly complex

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 of own agent(s)
    - server in charge of env. / NPCs
  - **client-side predictions**: both clients and server
    - clients “predict” (just for local visualization)
    - server “corrects” (it has the last word!)