# 10 Networking in Games

Tvorba a dizajn počítačových hier

Návrh a vývoj počítačových hier

# Motivation

- Online statistics & analytics
- Online storage of user state
- Online software and content updates
- Communication with 3rd parties
- Multiplayer
  - Asynchronous (e.g. Clash of Clans)
  - Turn-based games (Chess, Hearthstone)
  - Simple real-time games (Clash Royale)
  - Fast action games (CS:GO, Fortnite, Overwatch…)
  - MMO games (World of Warcraft…)
- We will focus on the hardest of these, multiplayer

# Multiplayer – Basic idea

- Player input modifies game state
- Game state is synchronized across the network
- Each player modifies only a subset of the game state
- The usual approach uses client-server communication
  - The whole level has one server – could be one of the players or a **dedicated server**
  - Other players are connected to the server

# Client-Server model

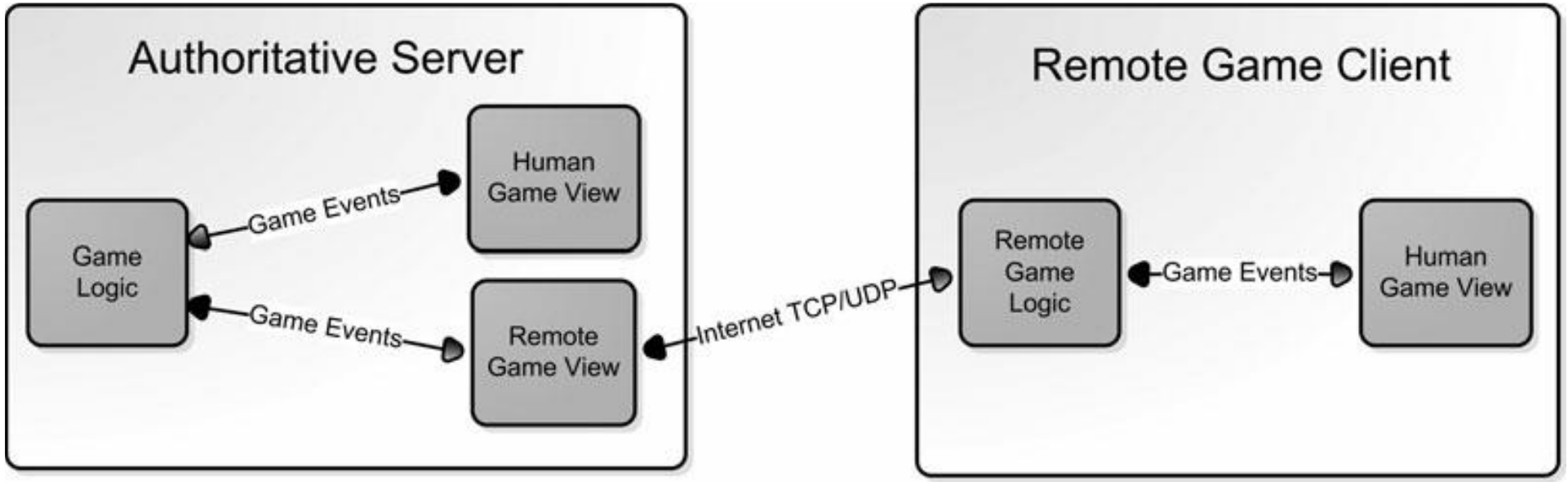
- Connection-based approach
- **Server** – provides a service (game state)
- **Client** – requests the service from a server (access/modify game state)
- Multiple clients communicate with one server, never between each other
- Centralized system, high demands on server resources
  - Especially for MMO games such as World of Warcraft
- Low player numbers can be handled by a normal PC and internet connection
  - Counter-strike, Starcraft, …

# Relation to the Observer pattern

- Game state on the server (subject) is being observed by clients (observers)
- Clients get notified on state changes
- Clients send their data to modify the state
  - But the server decides what to modify (to avoid cheating)
- **Both parties receive data with a delay**

# Networked Game through Views

- We implement two additional classes to enable networking
  - Remote game view
  - Remote game logic
  - Proxy design pattern



# Remote Game View

- On the server, a remote player is just like an AI agent
- What happens inside the game view is totally different
- Game events are packed up and sent via TCP or UDP to a client
  - Need to compress data, select only important game events
  - No need to send “object moved” if several such events occurred since the last packet was sent, we send only the most recent
- Receives game commands from the client
  - Should not trust these entirely
  - Need sanity checks to prevent hacking
  - After filtering impossible commands, they are passed on to the game logic

# Remote Game Logic

- The game logic is an authoritative server, it represents the real game state
  - Clients need a copy of the game state to be able to present the player the game correctly
  - They also need to account for network delays and network errors
  - This is the job of Remote Game Logic
- It is similar to the server game logic
  - There is a need to simulate without receiving events from the server
  - Saves bandwidth
  - Increases responsiveness
- Allow for “against the rules” corrections when the server sends the correct data

# Typical Client-Server interaction

- Client sends a request to create a new connection for it on the server
- Server receives the request and prepares a connection, then notifies the client
- The client sends data through the connection (over and over)
- Whenever the server receives data from a client, it is notified and responds based on the data
- The client is finished (player turned off the game) and notifies the server to abort the connection
- If any of the sides does not respond within a reasonable time frame, a time-out occurs, and the connection is lost
  - Handle the disconnect automatically

# Peer-to-peer networking

- Alternative to client-server
- Decentralized system
- All nodes are equal, there is no server
- Distributed resources, part of resources available on each node
- Very rarely used in games
  - Not reliable communication between multiple players
  - More vulnerable to cheating attempts

# Multi-server network architectures

- Used for MMOs
- Multiple distributed servers that communicate between each other and each has several clients connected
- The servers connect nearby clients, clients that are in the same location, …
- Complicated load balancing
- Can sustain millions of players (World of Warcraft)
  - But not all interacting at the same time
  - In-game locations have their limits on the number of players

# Low-level protocol

- UDP – User Datagram Protocol
  - Connection-less communication
  - Packets are always sent to a specific IP address and port
  - Unreliable protocol
  - Order of packets is not guaranteed, need to split your data into packets
  - Delivery of packets is not guaranteed (1-5% loss)
  - Duplicated packets can occur
- TCP/IP – Transmission Control Protocol
  - Connection-based communication
  - Guaranteed order and reliability
  - Splits your data into packets automatically
  - Flow control
  - Easy-to-use, just like writing/reading data to files

# Which one is used in real-time multiplayer games? UDP or TCP?

# Problems with TCP/IP

- Input and output streams (“files”) are buffering data on both sides and decide when to send the data
  - Lots of small-sized data (such as player commands) might be buffered for seconds before sending
- Possible fix is using TCP_NODELAY to send data immediately
- If packets are lost or come out of order, they are sent again or re-ordered
  - If it happens again, it tries again
- This may cause huge latency problems (seconds)
- TCP is basically UDP with added overhead, that splits your data into packets, numbers them, and then checks them on the other side

# High latency in TCP

- If we have problems reconstructing the packets, stalls occur
- Depends on the ping between both sides
- On packet loss, stalls might reach duration of latency*3
- If new data was sent during that time, TCP **forces** it to wait for the old data
- Therefore, most real-time games use UDP
  - But with added features
- TCP can used for games that do not need fast real-time communication
  - Turn-based games
  - Online statistics, analytics
  - Saving user data
  - Important communication – players buying something, …

# Problems with network games

- High latency
- “Lag” – Round trip time (RTT)
  - How long it takes to send a packet to the other network node and receive an answer
  - Latency = 0.5 * RTT
- “Jitter” – Fluctuation of latency between packets
- Packet loss – as mentioned before

# Avoiding problems

- Reducing the distance between end nodes
  - 12.000km distance, light travels at 300.000km/s => 40ms latency
  - This is the lowest bound, since we are limited by the speed of light
- Getting a better internet connection ☺
  - Not really an option, since players want to play in their current setup
- Not much we can do to reduce these
- We can improve by:
  - Sending consistently large packets to avoid jitter
  - Reducing packet size – sending only essential data
  - Not requiring packet ordering (UDP in favor of TCP)
  - Split data based on importance and required latency (combining UDP & TCP)

# Latency compensation techniques

- *“Bandwidth problems can be cured with money. Latency problems are harder because the speed of light is fixed – you can’t bribe Physics.”*
- There is nothing we can do about latency, so let’s try to compensate
- Prediction techniques
- Manipulation of game time to equalize gameplay
- Data compression
- We need to be careful, since we might be opening doors for cheating
  - Cheat detection

# Prediction

- The client predicts the server response and presents the game state as it is
- The game responds immediately to user input
- The game state might diverge from the actual (server) game state
- Higher latency = bigger differences in states
- Repairing of game state when the server message is received
- Player “prediction”
  - Let the user interact, expect what will be received concerning only this player
- Opponent prediction
  - Predict positions (and lots more…) of entities not controlled by the player

# Player “prediction”

- We need to keep the game state *reasonable* while we wait for the server to respond
- The player is allowed input, however some of their actions might be undone later, when the client repairs the game state according to the server
- Introducing tears and “teleports” is a negative side-effect, however not so bad as stalling until new information is received from the server
- Can be in the form of calculating correct physics, allowing player movement and actions
- More responsive ⇒ less consistent
- More consistent ⇒ less responsive

# Opponent prediction

- We take the opponents' entities, their last known position and where they are heading
- Predict position based on direction and speed
- Take it as the truth if we do not receive an update
  - which will be retrospective!
- We might need synchronized Random Number Generators (RNGs)
  - An opponent fires a gun, it does between 15 and 30 damage
  - RNG can make the difference between a unit dying or surviving
  - Pseudo-RNG are initialized with the same seed value on all clients and the server
  - Usually used in RTS games (Warcraft 3 does this)
  - Avoiding RNG in game design is also valid

# Opponent prediction in FPS games

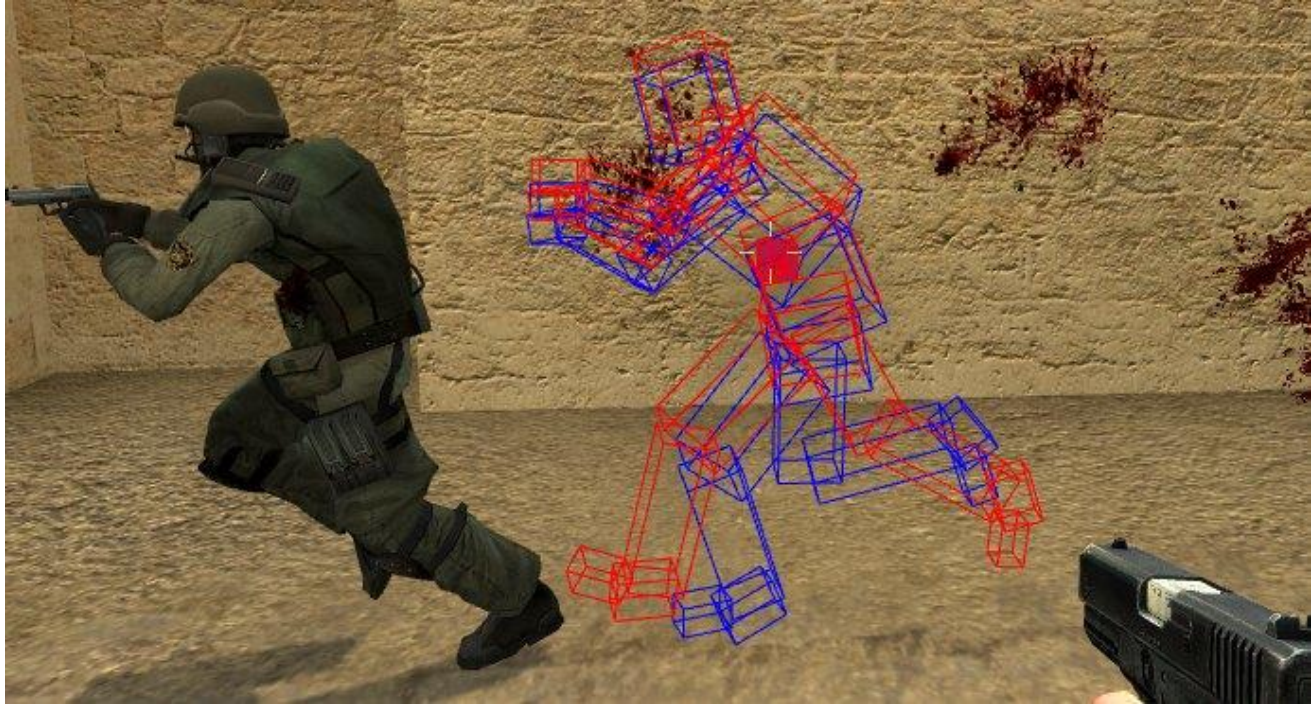
- Position $P$, Velocity $\boldsymbol{v}$
- Last received packet was at time $t_0$, current time is $t$
- Very simple linear prediction:
  - $P(t) = P + \boldsymbol{v} * (t - t_0)$
  - We additionally use the physics engine to avoid displaying the enemies running through walls
- Unsolvable problem
  - Firing at the opponent at a predicted position
  - The server receives only your position and direction when you shoot
  - You clearly see that you are firing directly at them
  - The server knows better, and the opponent does not die

# Adjust send rate

- Do not send all data every frame
- If game is running at 200 FPS, update just a small amount of necessary data every frame
- CS:GO – most servers run at 64 updates per second (“tick rate”)
- CS:GO tournaments = 128 tick rate

- Different data in-game can have different send rate
- Should not depend on framerate
  - All network communication usually runs on a separate thread

# Time warping



- The client has fired a shot
  - At $t_0 = 0$
- The server received the message
  - At $t_1 = 100ms$
  - The enemy moved left in the 100ms
- Red boxes – Show the position of the enemy where it was on the client in time $t_0 = 0$
- Blue boxes – Position estimated by the server after a time warp
  - The server rolls time back
  - Using the game time from the client

# Opponent prediction in RTS games

- Each unit has received a command
- Clients receive only a list of commands for a list of units
- Units can keep executing their commands
  - Must be deterministic
- More consistent than FPS games
  - Commands for one unit do not change as often as the position or rotation of the player
- Example:
  - Max actions per minute for RTS players is around 600
  - 10 actions per second ⇒ 1 action/packet every ~100ms
  - Compared to 128 tick servers in CS:GO ⇒ 1 packet every ~8ms

# Data compression

- Lossless compression
- Opponent prediction
- Delta compression
  - Send only changes of the state, not the whole state
- Interest management
  - Send only info that the user can see
- (Peer-to-peer)
- Update aggregation
  - Group multiple messages (can be from different moments in time) into one
  - Problematic with real-time games

# Cheating

- We cannot trust packets we receive from the clients
  - Someone might be altering them
  - Or it might not be a game client at all, but a custom-made program trying to mess up our server
  - We need some “correctness” detection on the server
    - Ignore packets that update position of players too rapidly
    - Ignore unnatural Interest management packets
- Most typical cheats
  - Wall-hack
  - Map-hack
  - Speed-hack – unlimited run speed, sends fake “move player” packets
  - Aim-bot – usually combined with wall-hack, sends fake “mouse move” messages to game

# Wall-hacking/map hacking

- Wall-hacking: seeing through walls in FPS games
- Map-hacking: ignoring fog of war and showing units in fog of war

- The hack alters the game’s window and renders additional objects on top of the game’s output image
- Monitors memory/packets to identify position of units
  - All unit info is received over the network – man-in-the-middle attacks
  - Special algorithms to find the data in memory
  - Or a directly altered executable

# Speed-hack

- Alters memory of the process
- Send “impossible” movement data to the server
- If the server is not validating it, it just accepts incorrect data

# Aim-bot

- Read positions of enemies from process memory or network
- Directly alter rotation of player in memory
- Or generate fake mouse movement

# Anti-cheating

- Network encryption
  - SSL or another encryption
- Monitor memory reading/writing attempts
  - Usually by a different anti-cheat process
- Encrypt data in memory
- Protect executable from cracking attempts
  - Denuvo

- CheatEngine is a simple tool for reading/writing memory
  - Use at your own risk
  - Never to cheat online
  - Removing challenges makes the game less fun

# Networking in Unity

- Simple communication over HTTP
  - UnityWebRequest, C# HttpClient/HttpRequest
- UDP/TCP communication
  - C# TcpClient, TcpListener, UdpClient, Socket
- UNet – deprecated, high-level API (pre-made components), low-level API
  - No longer part of Unity (2019.4 was the last version for LLAPI)
- Unity NetCode – part of DOTS, in preview for 2+ years
- Mirror – based on UNet, offers different low level transports
  - free & open-source, but you need your own server
- Photon – several different solutions

# Photon

- Has several solutions for networking, each with specific use-cases
  - Quantum – deterministic engine, high cost
  - Realtime – cross-platform, for various engines, lower-level
  - PUN (Photon Unity Networking) – easy to use, similar to Unet, built on top of Realtime
  - Bolt – for Unity, relatively easy to use, built on top of Realtime
  - Fusion (in preview) – new, Unity-only, has various improvements over PUN and Bolt
    - allows different network architectures
- Has free 20 CCU for testing

- Games build with Photon (that we know of)
  - Humankind, Robocraft, Phasmophobia, VRChat, Golf Clash, Ylands, Outward, Prison Architect, Descenders

| | PUN | Bolt | Fusion |
|---|---|---|---|
| | 2011 | 2014 | 2021 |
| Target Player Count | 32 | 32-50 | 200 |
| **Core-Features** | | | |
| Tick based simulation | ✗ | ✓ | ✓ |
| Client Side Prediction | ✗ | ✓ | ✓ |
| Lag Compensation | ✗ | ✓ | ✓ |
| Snapshot Interpolation | ✗ | ✓ | ✓ |
| **Replication System** | | | |
| Delta Snapshots | ✗ | ✗ | ✓ |
| Eventual Consitency | ✗ | ✓ | ✓ |
| **Performance** | | | |
| Allocations (Runtime) | / | / | zero |
| Performance (Benchmarks) | + | + | +++ |
| Bandwidth | / | + | +++ |
| **Bespoke Prebuilt Functionality** | | | |
| Area of Interest (AOI) | ✗ | ✗ | ✓ |
| Network Animator | ✗ | ✗ | ✓ |
| Network Character Controller (KCC) | ✗ | ✗ | ✓ |
| Auth. Rigidbodies w/ CSP | ✗ | ✗ | ✓ |

# References


- Armitage, G., Claypool, M., and Branch, P. *Networking and Online Games: Understanding and Engineering Multiplayer Internet Games*. Wiley, 2006.
- McShaffry, M., and Graham, D. *Game Coding Complete, Fourth Edition*. Course Technology PTR, 2012.
- Hall, R., and Novak, J. *Game Development Essentials: Online Game Development*. Cengage Learning, 2008.
- Novak, J. *Game Development Essentials: An Introduction*. Cengage Learning, 2011.