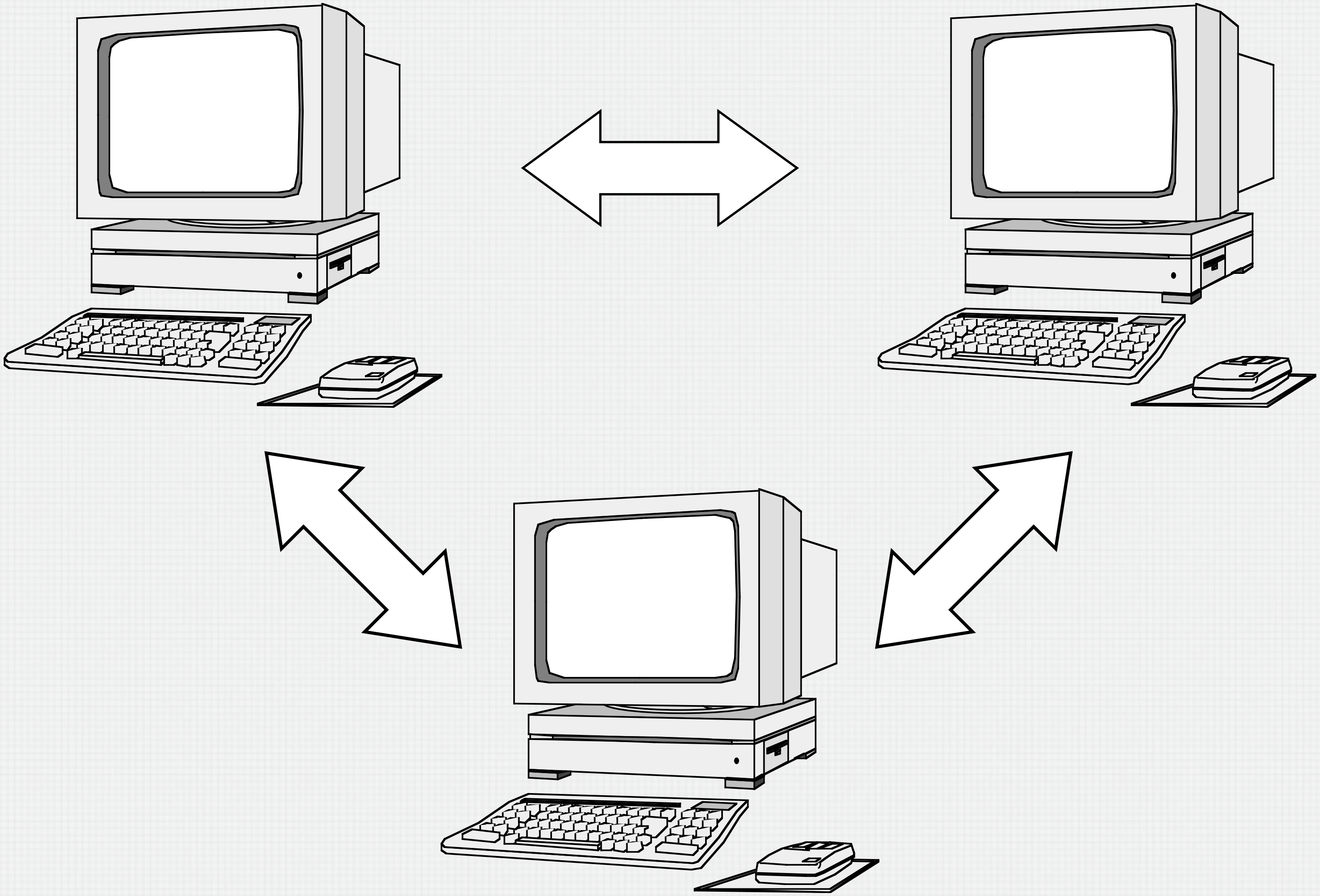
Distributed Game Architectures

Jörg Kienzle

Distributed Game Architectures Overview

- Distributed Architectures
  - Client-Server
  - Peer-to-Peer
  - Startup
    - Centralized
    - Decentralized
- Object-Oriented Communication over the Network
  - Remote Commands

Peer-2-Peer Model



Direct communication between Players

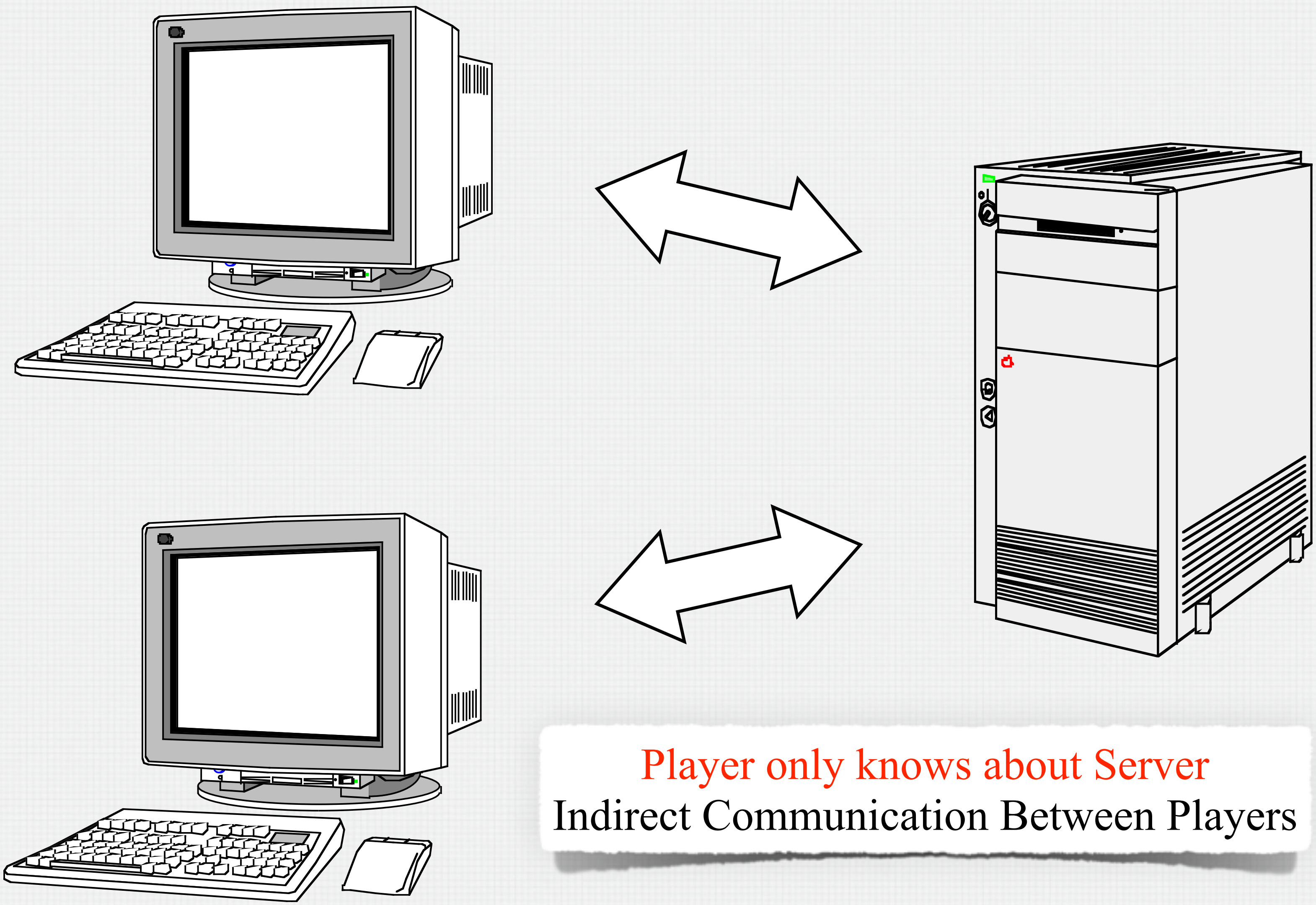
Peer-2-Peer (2)

- Both computers run the same application
- Main loop
  - If it is the turn of the local player
    - Get move from GUI
    - Verify correctness
    - Apply move to local state
    - Send move over the network
  - If it's the turn of the remote player
    - Get move from the network
    - Apply move to local state

Peer-2-Peer Comments

- Advantages
  - Only one application to develop
    - Symmetric architecture
  - The GUI has access to the entire game state, if needed
  - Game state is local, which increases performance
- Disadvantages
  - One application
  - No authoritative game state
    - Game must be deterministic
  - Startup has to be asymmetric (see slide on distributed system startup)

Client-Server Model



Player only knows about Server
Indirect Communication Between Players

Client-Server (2)

- Client
  - Graphical User Interface
  - Network Interface
- Main loop
  - If it is the turn of the local player
    - Get move from GUI
    - Send move over the network to server
  - Wait for game state from server

- Server
  - Game State
  - Game Behaviour
  - Network Interface
- Main loop
  - Wait for move from player
  - Verify move
  - Apply move to game state
  - Send updated game state to clients

Client-Server Comments

- Advantages
  - Clear separation of concerns between GUI and Game Logic
  - The authoritative game state is at the server
  - Saving and loading games is easy
- Disadvantages
  - Two applications to develop
  - Game state is remote, which decreases performance
    - Caching of game state at the client can help
  - GUI does not have access to entire game state
    - Caching of game state at the client can help
  - If caching is used, then cache must be kept up to date

Client-Server, Thick vs. Thin Clients

- Thin GUI
  - No local game state
    - At the limit, the GUI does not even know what game is being played!
  - Sends commands directly to the server
  - Server sends information to be displayed back to the client
  - Example: onlive game streaming
- "Intelligent" GUI
  - Some local state
    - How much state is enough?
  - Can do verification of correctness of move without contacting the server, based on local information
  - Can provide user guidance / interactive help that understands the game semantics

What if the entire game state is replicated on the client?

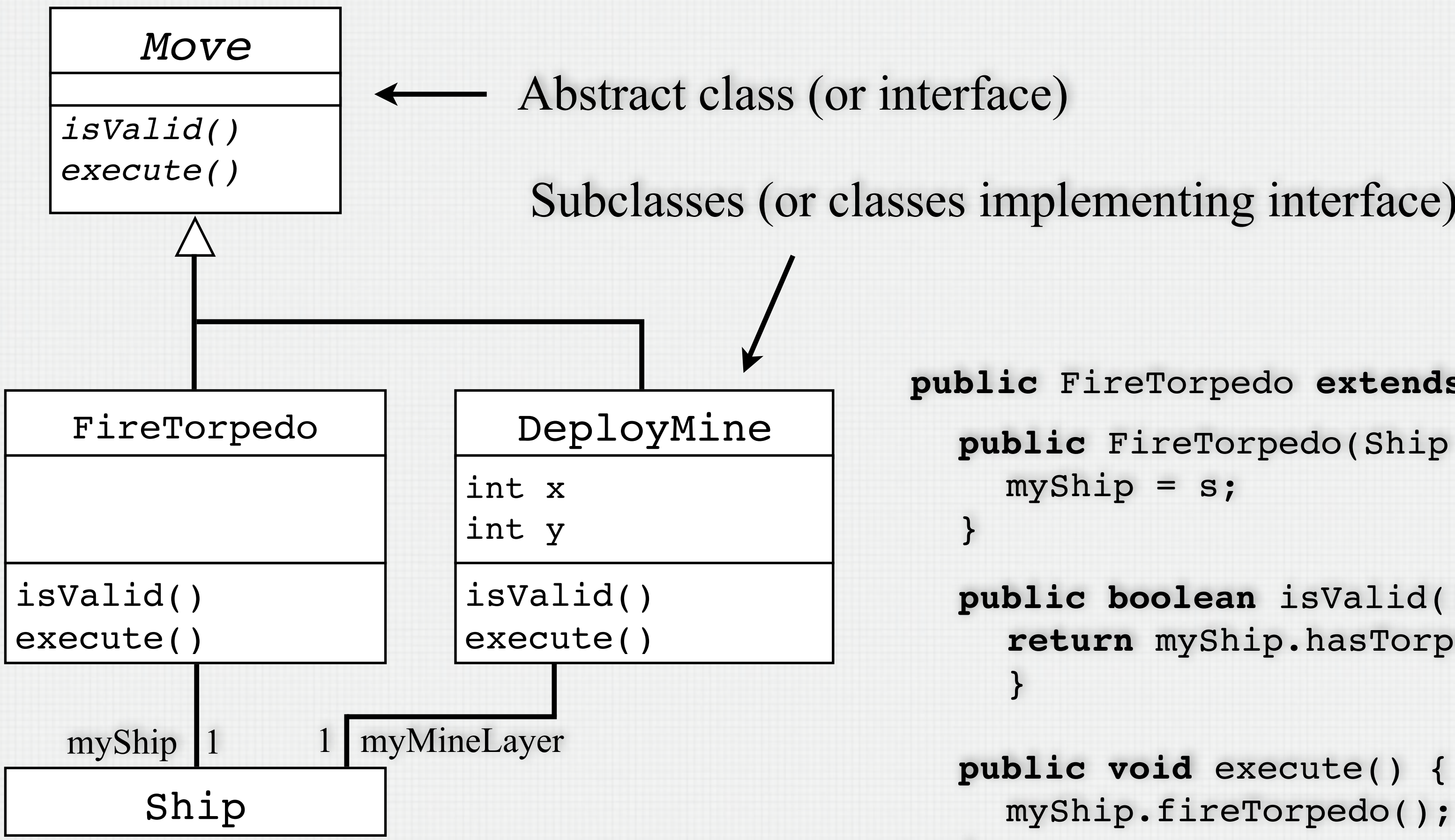
Distributed System Startup

- Somehow your (player) machine has to connect to other (player) machines
- Always asymmetric
- Centralized startup (à la server)
  - Dedicated “startup machine” (or set of machines) that is assumed to be always running
    - Location of this machine / set of machines is often hard-coded, or set in a configuration file
  - When player machine starts, it connects to the startup machine to announce its presence
  - The startup machine forwards addresses of other machines to the player machine on a need-to-know basis
- Decentralized startup (à la P2P)
  - Start one (player) machine (registration peer), remember network information
  - The other peers are provided with the network information of the first peer, typically in the login / startup window
  - When contacted, the registration peer forwards network information of all other registered peers
  - Sometimes also the registration peer provides a broadcast functionality that allows new peers to announce themselves

Networking and Turn-Based Games

- Movements of players have to be sent over the network
  - From client to server, or
  - From peer to peer
- Object-oriented Solution
  - Remote Command pattern
  - Define a class hierarchy of move actions
  - Each action knows how to validate and execute itself, which results in updating the game state

Move Hierarchy




```java
public FireTorpedo extends Move {
  public FireTorpedo(Ship s) {
    myShip = s;
  }
  public boolean isValid() {
    return myShip.hasTorpedo();
    }

  public void execute() {
    myShip.fireTorpedo();
  }
}
```

Move Execution Peer-2-Peer

- On current player's computer
  - GUI handles player input until it determined what move the player wants to execute
  - GUI instantiates the corresponding move object
  - GUI verifies if move is valid by calling `isValid`()
    - `isValid`() calls the appropriate verification methods on the model (i.e. package / classes containing the game state)
  - GUI gives move to the move executor
    - Executor executes move on the game state by calling `execute`()
  - Move is sent to the other players' computers (using serialization)
- On other computers
  - Move instance is read from the network and given to executor
  - Executor executes move on the game state by calling `execute`()

Move Execution Client-Server

- On current player's client machine
  - GUI handles player input until it determined what move the player wants to execute
    - Optional verification (only possible if the client knows about relevant game state)
  - GUI instantiates the corresponding move object
  - Move object is sent to server
- On server
  - Move instance is read from the network and given to executor
  - Executor validates move by calling `isValid`() (if not already done on the client)
  - If validation succeeds, executor executes move on the game state by calling `execute`()
    - "move effect" (i.e. updated game state) is sent to all players
  - If validation fails, exception is sent back to the current player's machine
- On all player's client machines
  - Move effects are displayed

Questions?

?

?

?

?

?

?

?

?

?