DISTRIBUTED SYSTEMS SS 2008

Some basics of multiplayer network games

[content partly taken from Stefan Zickler (szickler@cs.cmu.edu)]

OUTLINE

- Multiplayers Games (MG)
  - Why multiplayer?
  - Which types of MGs exist?
- How to program network-based MG
- Example project with demonstrations

MULTIPLAYER GAMES

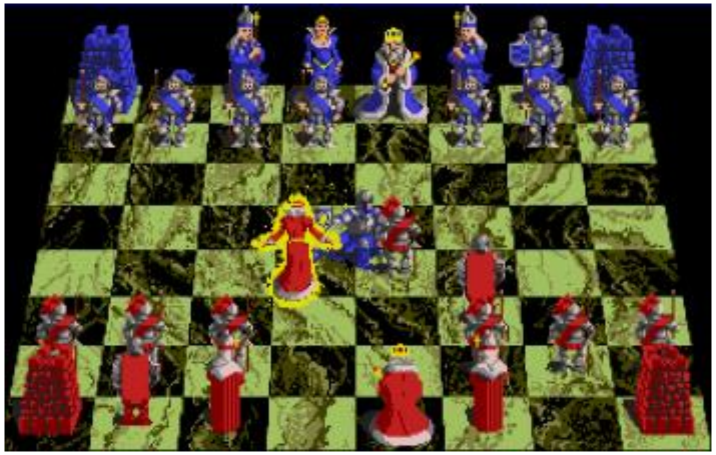
- Why Multiplayer Games?
  - Humans are better at most strategy than current Ais
  - Humans are less predictable
  - Can play with people, communicate in natural language
  - Add social aspect to computer games
  - Provide larger environments to play in, with more characters
  - Make money as a professional game player

Types of Multiplayer Games

- Turn-based
- Fully distributed
  - (Split)/Shared-Screen
  - Small group multiplayer games
  - Massive Multiplayer Online Games (MMOGs)

TURN-BASED MULTIPLAYER GAMES

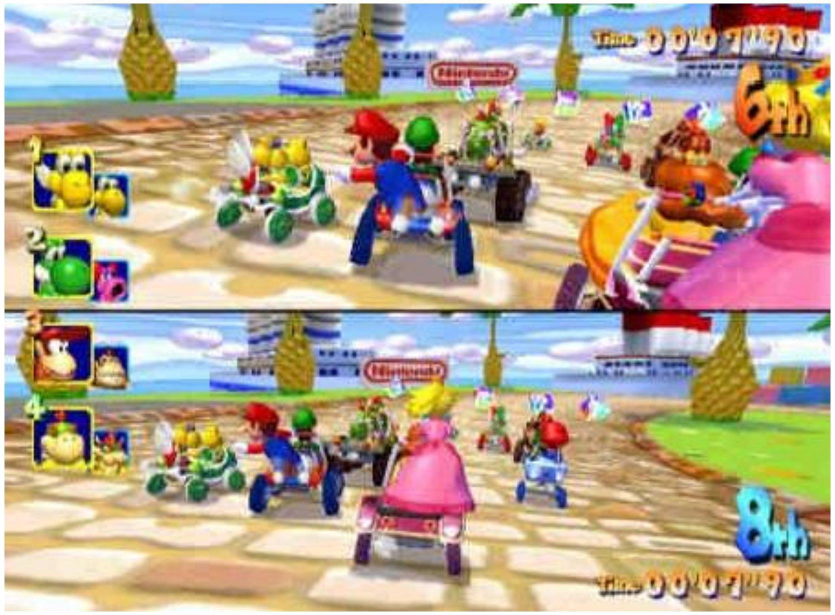
+ Easy to implement
– No real fun since players need to wait for their turn


Battle Chess (1988)

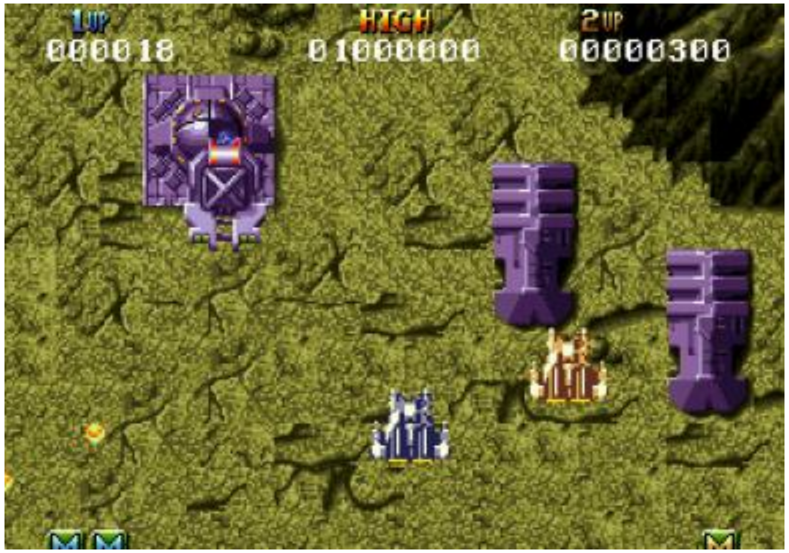
Whirlwind Snooker (1991)

SPLIT-SCREEN AND SHARED-SCREEN MULTIPLAYER

+ No latency issues, simple and even funny
– Scalability, physical location



Mario Kart Double Dash (split-screen, 2003)



Battle Squadron (shared-screen, 1992)

Small Group Multiplayer Games

- Ad-hoc, short-lived sessions
- Any client can be server
- Typically limited number of players
- E.g.
  - Shooters
  - RTS
  - Racing

Counter Strike (1999)

Massive Multiplayer Online Games (MMOGs)

- Persistent
  - World state
  - (Character state)
- Thousands of simultaneous players
- Most MMOGs are *Massive Multiplayer Online Role Playing Games* (MMORPGs)

MMORPGs

- **Ultima Online (1997)**
  - Isometric view
  - 3 years development
  - 1998: > 100,000 players
  - 2003: 225,000 players

MMORPGs

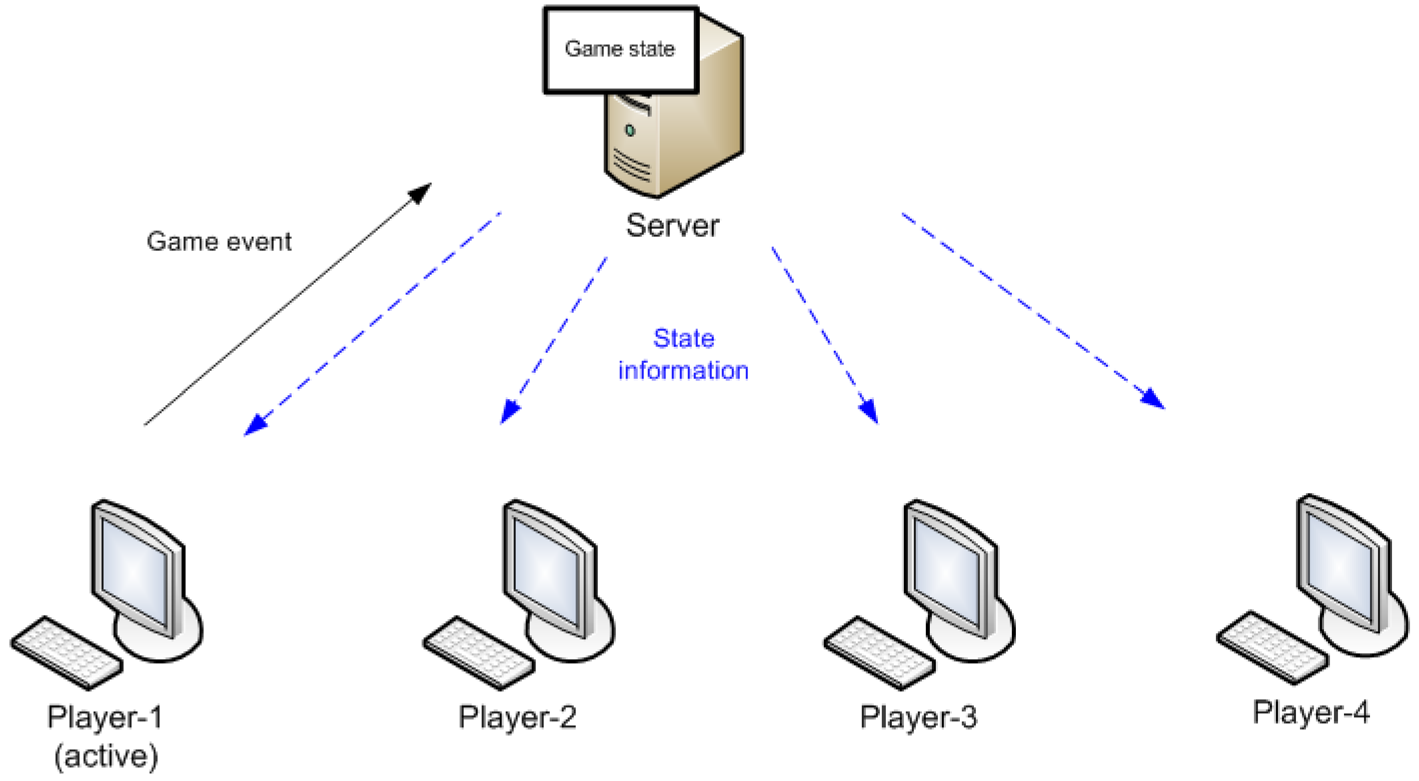
- **World of Warcraft (2004)**
  - Dec 2005: 5 million players
  - Dec 2006: 8.3 million players
  - Nov 2007: 9.3 million players
  - Jan 2008: 10 million players
    - 2.5 million in North America
    - 2 million in Europe
    - 5.5 million in Asia

NETWORK ARCHITECTURE FOR MULTIPLAYER GAMES

ARCHITECTURES

- Client-Server
- Peer-to-Peer
- Hybrid

Client-Server



CLIENT-SERVER

- Server handles all important decisions
- Server computes game state only
- Game state is broadcasted to all clients (e.g. every 10ms)
- Every event (even local) must go through server

+ Conceptually simple
+ Easy to implement (e.g. collision detection)
+ Harder to cheat
– Server becomes a bottleneck
– Bad response/reaction time (even of local player)

Client-Server

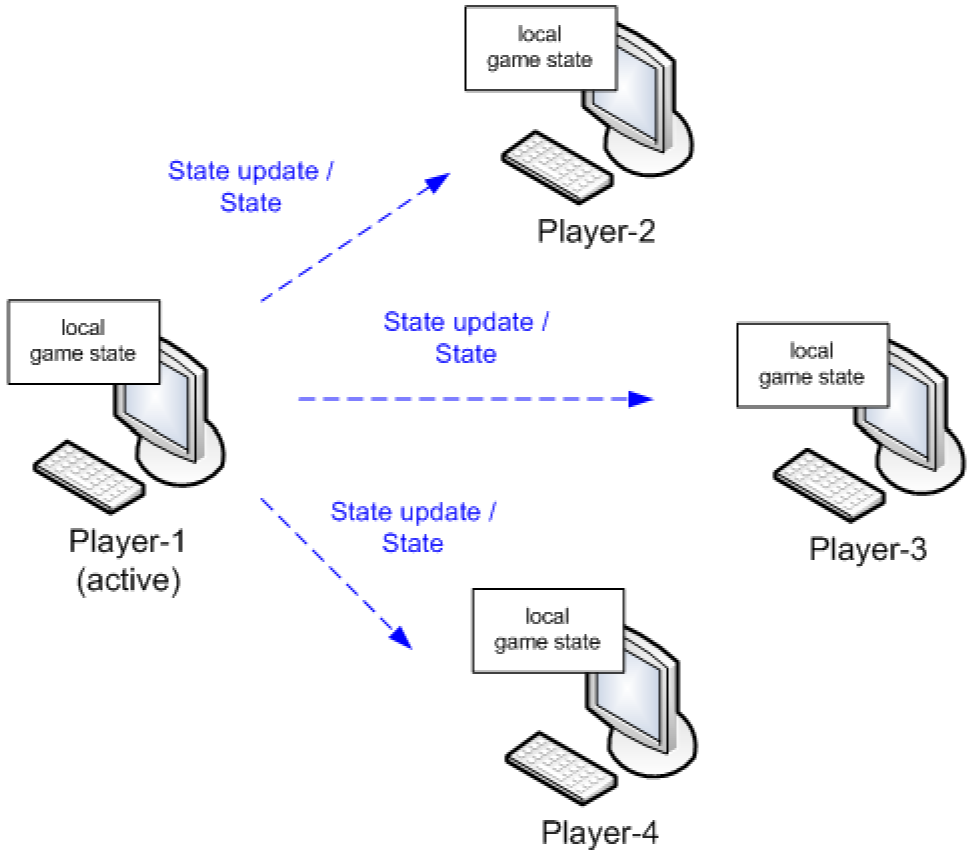
Server‘s main loop

```
while not done
   for each player in world
      if input exists
         get player command
         execute player command
         tell player of the results
   simulate the world
   broadcast to all players
```

Client‘s main loop

```
while not done
   if player has typed any text
      send typed text to server
   if output from server exists
      print output
```

PEER-TO-PEER


local game state
State update / State
Player-2
local game state
State update / State
local game state
Player-1 (active)
State update / State
Player-3
local game state
Player-4

PEER-TO-PEER

- Every client computes its own state locally
- Own game state is reported to others
- → Load is spread among all players
- Works better if local state prediction (and compensation) of other players is used

+ Lower latency (one-way-delay)
+ Better response/reaction time (particularly with prediction)
+ No single point of failure
– Game state synchronization is more difficult (e.g. collision detection)
– Can be easily cheated

PEER-TO-PEER

Client‘s main loop

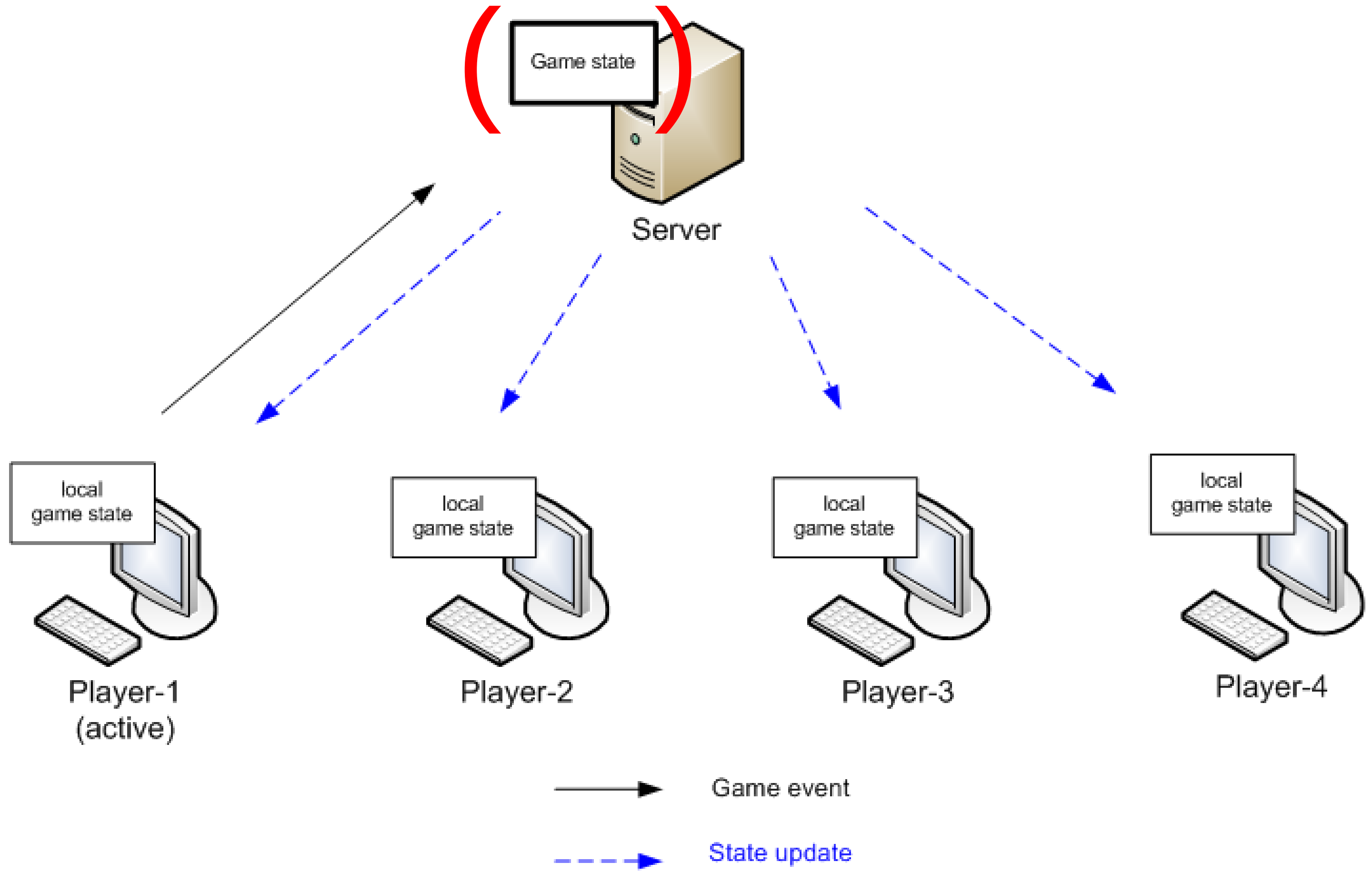
```
while not done
   collect player input
   collect network input about other players
   simulate player
      update player's state if informed it's been hit
      inform other clients if they've been hit
      move player
      update other player state (ammo, armor, etc.)
   report player state to other clients
```

Can be easily chated!

How to cheat:

1) Ignore incoming hit message
2) Tell others they are hit, although they aren‘t
3) Move player at arbitrary speed
4) Infinite ammo, armor, ….

HYBRID


(Game state)
Server
local game state
Player-1 (active)
local game state
Player-2
local game state
Player-3
local game state
Player-4
Game event
State update

HYBRID

- Combining Client-Server with P2P approach
  - Load distribution
  - Better responsiveness
  - Local computation of own state
  - Report state through server (harder to cheat!)
  - Severy may compute game state as well
    - E.g. used for collision detection
- Mostly used in practice
  - Typically with client-side prediction of game state and game state compensation

FURTHER ASPECTS

- Many other issues…
  - Different performance/latency of participating machines
  - Measure latency and consider it in state prediction too
  - Randomness
  - Avoid cheating
    - Why? Because otherwise no one would play the game!
  - Design your own protocol (typically based on UDP)
    - Own sequencing, retransmission of important packets, etc.
  - Implement many other components
    - Lobby, Copy protection, Updating, …

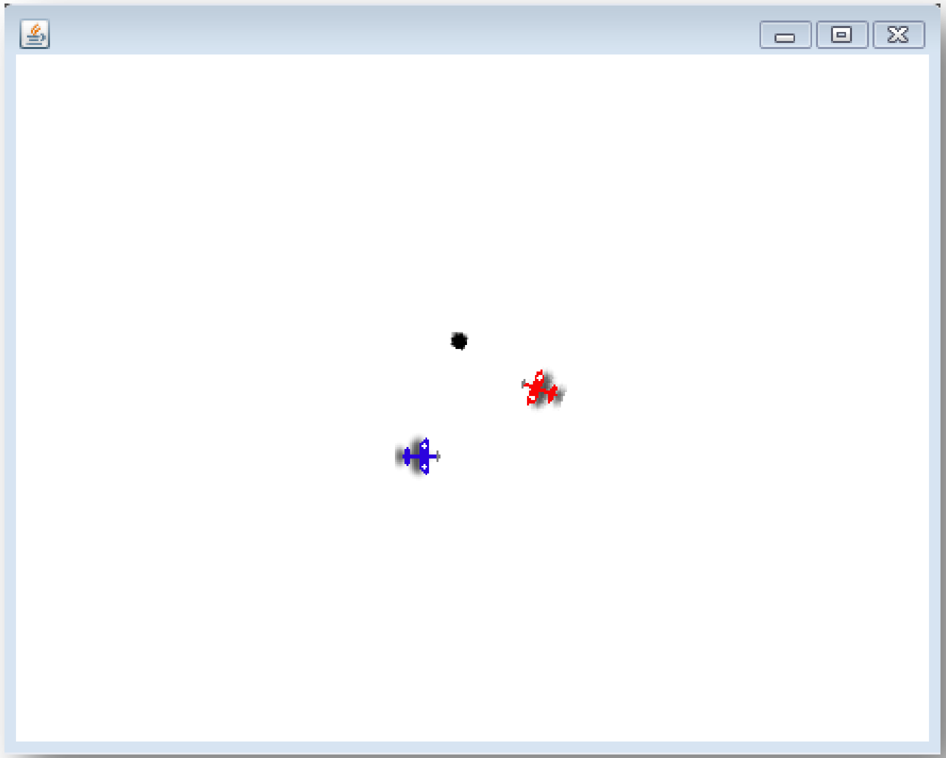
TYPICAL LATENCY REQUIREMENTS

- RTS
  - 250 ms not noticable
  - 250-500 ms playable
  - > 500 ms noticeable
- FPS
  - < 150 ms preferred
- Car racing
  - < 100 ms preferred

EXAMPLE PROJECT

EXAMPLE PROJECT (RMI BASED)

- Objects moving in a rectangular area
  - Can be rotated
  - Can accelerate and brake
  - Can fire a cannon (ball up to 3 times reflected by a wall)
  - More a demo than a real game
    - No collision detection
    - No one can win the game ☺
  - Single player version
    - 430 LOC (with comments)
  - Last multiplayer version
    - 620 LOC (with comments)



EXAMPLE PROJECT – GAME STATE

```
//constants
static int PWIDTH = 500; // size of panel
static int PHEIGHT = 400;
final static float maxSpeed = 25; //maximum speed of robot
final static float minSpeed = 0; //minimum speed of robot
final static int SPEED_DIV = 3; //speed is divided by this value
final static int MAX_WALLS = 3; //maximum number of reflections of cannon on a wall

 int botImgWidth;
 int botImgHeight;
 int botID;
 static int botcount = 0;

 boolean acceleratePressed = false;
 boolean brakePressed = false;
 boolean leftPressed = false;
 boolean rightPressed = false;
 boolean firePressed = false;

 //game state
 boolean cannonActive = false;
 int botX = 200, botY = 200, botRot = 0;
 int cannonX, cannonY, cannonRot;
 float speed = minSpeed;
 float cannonSpeed;
 int cannonWallCount;
```

EXAMPLE PROJECT – INPUTS

```
private void initKeyListener()
{
  addKeyListener( new KeyAdapter() {

    public void keyPressed(KeyEvent e)
    {
      int keyCode = e.getKeyCode();

      if (keyCode == KeyEvent.VK_UP) {
        botstate[0].acceleratePressed = true;
      }
      //...
    }

    public void keyReleased(KeyEvent e)
    {
      int keyCode = e.getKeyCode();

      if (keyCode == KeyEvent.VK_UP) {
        botstate[0].acceleratePressed = false;
      }
      //...
    }
  });
}
```

Example Project – Game Loop

```
long period = 20;


public void run()
/* Repeatedly update, render, sleep */
{
  long gameStart = System.currentTimeMillis(), gameSimTime = 0;

  running = true;
  while(running) {
    long gameTime = System.currentTimeMillis() - gameStart;
    while (gameSimTime < gameTime) {
      gameUpdate();
      gameSimTime += period;
    }

    gameRender(); // render to a buffer
    paintScreen(); //draw buffer to screen
    Thread.yield();
  }
  System.exit(0); // so enclosing JFrame/JApplet exits
} // end of run()
```

EXAMPLE PROJECT – GAME LOOP

```
private void gameUpdate()
{
  updateCounter++;

  for(BotState bs : botstate) {
    bs.handleRotationChange();
    bs.updateBot();
    bs.handleAccBrake(updateCounter);
    bs.handleFire();
    bs.updateActiveCannon();
  }
}
```

MULTIPLAYER V1

- Client-Server architecture
- Game state simulation is performed on server
  - gameUpdate()
  - run()
- Input events are sent to server
  (and processed on server)
- Server periodically broadcasts state of all players to all players
- → „dumb“ clients

MULTIPLAYER V1

- Server Interface (RMI)

```
public interface IServer extends Remote {

  public enum Key {Left, Right, Up, Down, Fire};

  int join(IClient client) throws RemoteException;
  void keyEvent(int clientID, Key key, boolean pressed) throws RemoteException;
  void leave(IClient client) throws RemoteException;
}
```

- Client Interface (RMI)

```
public interface IClient extends Remote {

  void stateUpdate(BotState[] state) throws RemoteException;
}
```

State of all players

MULTIPLAYER V1

```
public class BroadcastThread extends Thread {

  static int LATENCY = 100;
  IServerImpl server;

  public BroadcastThread(IServerImpl server){
    this.server = server;
  }

  public void run() {
    while (true) {
      server.broadcast();
      try {
        sleep(LATENCY);
      } catch (InterruptedException e) {}
    }
  }
}
```

MULTIPLAYER V1

- Problems?
  - Even local animation might be slow
- Demonstration

MULTIPLAYER V2

- Hybrid architecture
- Naive approach:
  - Send input events through server to all clients
  - Game state simulation is performed locally (on every client)
  - → „dumb“ server

MULTIPLAYER V2

- Server Interface (RMI)

```
public interface IServer extends Remote {

  public enum Key {Left, Right, Up, Down, Fire};

  int join(IClient client) throws RemoteException;
  void keyEvent(int clientID, Key key, boolean pressed) throws RemoteException;
  void leave(IClient client) throws RemoteException;
}
```

- Client Interface (RMI)

```
public interface IClient extends Remote {

  void stateUpdate(BotState[] state) throws RemoteException;
}
```

MULTIPLAYER V2 - CLIENT

```
private void initKeyListener()
{
  addKeyListener( new KeyAdapter() {

    public void keyPressed(KeyEvent e) {
        int keyCode = e.getKeyCode();

        if (keyCode == KeyEvent.VK_UP) {
          server.keyEvent(clientID, Key.Up, true);
        }
        //...
    }

    public void keyReleased(KeyEvent e) {
        int keyCode = e.getKeyCode();

        if (keyCode == KeyEvent.VK_UP) {
          server.keyEvent(clientID, Key.Up, false);
        }
        //...
    }
  });
}
```

MULTIPLAYER V2 - SERVER

```
public void keyEvent(int clientID, Key key, boolean pressed) {
  if (pressed) {
    if (key == Key.Up)
      botstate[clientID].acceleratePressed = true;
    //...
  } else {
    if (key == Key.Up)
      botstate[clientID].acceleratePressed = false;
    //...
  }
  broadcast();
}

void broadcast() {
  for (IClient c : clients) {
    try {
      c.stateUpdate(botstate);
    } catch (RemoteException e) {}
  }
}

public void leave(IClient client) {
  //...
}
```

MULTIPLAYER V2

- Server performs no game state simulation
- Just input events are stored and broadcasted
- Problems ?
- → Will finally result in different simulations due to different network delays to the server
(and different speed of machines)
- Demonstration

MULTIPLAYER V3

- Hybrid architecture
- Every client performs game state simulation
- Own state is broadcasted to other players (via server)

- A kind of („very simple“) game state prediction
- However, no compensation

MULTIPLAYER V3

- Server

```
public interface IServer extends Remote {

  public enum Key {Left, Right, Up, Down, Fire};

  int join(IClient client) throws RemoteException;
  void stateUpdate(IClient client, int clientID, BotState state) throws RemoteException;
  void leave(IClient client) throws RemoteException;
}
```

- Client

```
public interface IClient extends Remote {

  void stateUpdate(int clientID, BotState state) throws RemoteException;
}
```

State of <u>one</u> player

MULTIPLAYER V3 - CLIENT

```
public class ClientBroadcast extends Thread {

  static int LATENCY = 100;

  GamePanel gp;
  IServer server;

  public ClientBroadcast(GamePanel gp, IServer server) {
    this.gp = gp;
    this.server = server;
  }

  public void run() {
    while (true) {
      try {
        server.stateUpdate(gp.iclient, gp.clientID, gp.botstate[gp.clientID]);
        sleep(LATENCY);
      } catch (RemoteException e) {
        System.out.println("Error with broadcast!");
      } catch (InterruptedException e) {}

    }
  }
}
```

MULTIPLAYER V3 - SERVER

```
public void stateUpdate(IClient client, int clientID, BotState state)
    throws RemoteException {
  for (IClient c : clients)
  {
    if (!c.equals(client)) {
      try {
        c.stateUpdate(clientID, state);
      } catch (RemoteException e) {}
    }
  }
}
```

MULTIPLAYER V3 - CLIENT

```
public class IClientImpl extends UnicastRemoteObject implements IClient {

  GamePanel gp;

  public IClientImpl(GamePanel gp) throws RemoteException
  {
    this.gp = gp;
  }

  public void stateUpdate(int clientID, BotState state)
  {
    gp.botstate[clientID] = (BotState)state.clone();
  }
}
```

Just overwrite state
(no compensation)

MULTIPLAYER V3

- Still problems?
- Yes: collision detection
- Solution
  - Perform game state simulation on server as well
  - Use game state on server for collision detection and inform clients
- Demonstration

MORE RESOURCES

- Introduction to Multiplayer Game Programming
  http://trac.bookofhook.com/bookofhook/trac.cgi/wiki/IntroductionToMultiplayerGameProgramming
- Unreal Networking Architecture
  http://unreal.epicgames.com/Network.htm
- The Quake3 Networking Model
  http://trac.bookofhook.com/bookofhook/trac.cgi/wiki/Quake3Networking
- Latency Compensating Methods in Client/Server In-game Protocol Design and Optimization
  http://developer.valvesoftware.com/wiki/Lag_Compensation
- Source Multiplayer Networking
  http://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking