Overview

Application side:
- Game Network Architectures
- Networked Virtual Environments
- Game Communication Protocols

Game Engine side:
- Networking support in SAGE
- Protocol Stacks
- Java Sockets
- UDP / TCP client and server tasks

Game Network Architectures

<table>
<thead>
<tr>
<th>type</th>
<th>advantage</th>
<th>disadvantage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Peer-to-peer</td>
<td>sometimes simpler, no server needed</td>
<td>limited by slowest machine, easier to cheat/hack</td>
</tr>
<tr>
<td>(all talk to each other)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Client/Server</td>
<td>more secure, centralized control</td>
<td>powerful server needed, single point of failure</td>
</tr>
<tr>
<td>(one server per game)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Floating Server:
peer-to-peer, but one “peer” is the server

Distributed Server:
multiple services for managing a very large world

Networked Virtual Environments (NVE)

“A computer-based artificial world of 3D spaces and objects visited by geographically dispersed users who interact and collaborate with each other and with objects/entities local to the world” [1]

Common terms: NVE, MMG, MMOG, MMORPG, Virtual World

Many different types:
- Gaming
- Military
- Business / Training (“Serious Games”, “performance engineering”)
- Education

Examples: Star Wars The Old Republic, World of Warcraft, Everquest, League of Legends, etc...

mpogdi.com (multi-player online game directory) lists thousands of MMO’s.

Client-Server Organization

Client game-loop with networking:
```java
while (!gameOver)
{
    handleNetworkInput();
    handlePlayerInput();
    update();
    render();
    sendStateToNetwork();
}
```

Game Communication

What to send?
- Entire world state is usually too much
- Can usually just send user actions
- also impacted by NPCs (we will see later)

“Fat Client”
(each client runs world simulation)
Advantages:
- Fast local updates
- Server handles message-switching
Drawbacks:
- Code/data duplicated on each client
- Need to synchronize client worlds
- Clients must be deterministic

“Thin Client”
(server runs world simulation)
Advantages:
- Client can be non-deterministic
- Synchronization easier
Drawbacks:
- More network traffic
- Server bottleneck for all activity
Example Game Protocol

Messages from Client to Server:
- **CREATE (name, position)**
  - Informs server of a new world participant
- **MOVE/ROTATE (amount)**
  - Informs server about a change in a client avatar
- **DETAILSFOR (addr, port, position, orient)**
  - Informs server of local avatar position/orientation (intended for forwarding to another client)
- **BYE (name)**
  - Informs server that client is leaving

Messages from Server to Client:
- **CREATE (name, position)**
  - Informs client that a new remote avatar exists
- **MOVE/ROTATE (senderName, amount)**
  - Informs client of a change in status of a remote avatar
- **WANTSDETAILS (addr, port)**
  - Informs client that a remote client wants a local status update
- **DETAILSFOR (senderName, position, orientation)**
  - Provides client with updated status of a remote avatar
- **BYE (name)**
  - Informs all clients with the name of a client who quit

Example Protocol Processing

Server Handling:
- **CREATE (name, position)**
  - Save name and corresponding IP/port
  - Forward CREATE message to all other clients
- **WANTSDETAILS (addr, port)**
  - Forward WANTSDetails to all other clients
- **MOVE/ROTATE (name, amount)**
  - Look up sender name
  - Send MOVE/ROTATE message to all other clients
- **DETAILSFOR (addr, port, position, orientation)**
  - Look up sender name
  - Send DETAILSFOR to <addr, port>

Client Handling:
- **CREATE (name, position)**
  - Create Ghost avatar
- **MOVE/ROTATE (senderName, amount)**
  - Update Ghost avatar for sender
- **WANTSDetails (addr, port)**
  - Get local avatar position/orientation
  - Send DETAILSFOR (addr, port, mypos, myorientation)
- **DETAILSFOR (senderName, position, orientation)**
  - Update Ghost avatar for sender

Sample Protocol Sequence
Networking Support in SAGE

server side:
- extend GameConnectionServer
- override constructor, calling parent class constructor with port and type (typically UDP)
- if the message represents a first message from a new client, use addClient(...), getting the client’s info using createClientInfo(...)
- To send messages to clients, use sendPacket() or sendPacketToAll(...), etc.

client side:
- extend GameConnectionClient
- override constructor, calling parent class constructor with address, port and type.
- override processPacket(...), implementing desired protocol. This method is called each time another node (typically the server) sends it a message. The message is contained in a java Object passed as a parameter (typically a string).
- The game’s update() method should call client.processPackets()

Networking in SAGE

Creating Unique Names

Class UID (“Unique ID”)
- Unique with respect to the current host
- Components:
  - Time, VM ID, Counter

Class UUID (“Universally Unique ID”)
- 128-bit value, guaranteed unique world-wide
- Components:
  - Time, VM ID, Counter, MAC address
- Potential security issue: publishing MAC addresses
- Partial solution: random UUIDs
  
  ```java
  UUID newID = UUID.randomUUID();
  ```

Synchronization issues

- Latency can cause “freezing”
- Solution: data prediction
  - Clients retain position “history” for ghosts
  - Extrapolate “likely next position” from history (typically use a quadratic polynomial)

Correcting Prediction Errors

Remote avatar changes
- speed and direction here

Position Migration

- Actual Position at T_3
- Extrapolated Position at T_3
- Likely Position at T_3

NVE issues

Naming
- Need some way to identify clients uniquely

Server Discovery
- Clients need a way of finding servers

Synchronization
- Need to keep the various simulations “in step”
Additional Synchronization Issues

Potential inconsistencies in code
- Different handling of user actions
- Different (pseudo-) random sequences

Divergence in worlds
- Small errors can lead to big divergence

Game protocol must allow for this
- (or assume the code is “perfect”...)
  - Example: “resynchronize” protocol
    Causes “jumps”, uses “smoothing algorithm”

Server Discovery

Client must know how to contact server

Various approaches:
- “Hard-coded” in client
- Player-provided to client at startup
  - Command line
  - Interactive prompt
- “Multicast” groups

Multicast Groups

Networking Internals

Protocol Stacks

Connection Abstraction: SOCKETS

- Socket: end-to-end connection
  - Can operate in either TCP or UDP mode
- Programs read/write socket
  - Can ignore underlying mechanism

Client-Server Socket Usage
Java Socket Support – TCP

- **Class ServerSocket**
  - Automatically listens for connections
  - Accepts connections and returns a new Socket

- **Class Socket**
  - Automatically connects to a listener (server)
  - Supports input and output streams

**Simple (TCP) Client-Server**

**Server main:**
```
ServerSocket serverSock = new ServerSocket(port);
loop {
    Socket clientSock = serverSock.accept();
    handleClient(clientSock);  //talk to client until done
    clientSock.close();
}
```

**Client main:**
```
Socket sock = new Socket(hostIP, port);
InputStream inStream = sock.getInputStream();
OutputStream outStream = sock.getOutputStream();
//talk to server until done
loop {
    inStream.read(data)
    process data;
    outStream.write(data);
} until done;
sock.close();
```

Threaded Servers

**Server main:**
```
ServerSocket serverSock = new ServerSocket(port);
loop {
    Socket clientSock = serverSock.accept();
    Thread t = new ClientHandlerThread(clientSock);
    t.start();
}
```

**Class ClientHandlerThread**
```
public ClientHandlerThread(Socket clientSock) {
    ...code here in constructor to save socket
}
public void run() {
    handleClient(clientSock);  //talk to client until done
    clientSock.close();
}
```

Input & OutputStream data are binary
(wrap them in BufferedReader/Writer)

Java Socket Support – UDP

- **Class DatagramSocket**
  - Defines a UDP-based socket
  - Supports send/receive of datagrams (packets)

- **Class DatagramPacket**
  - Defines a packet in UDP format
  - All packets contain sender or dest addr

**UDP–Based Server**

**Server main:**
```
DatagramSocket serverSock = new DatagramSocket(PORT);
loop {
    //create an empty packet
    byte data[] = new byte [BUFFER_SIZE];
    DatagramPacket recvPacket = new DatagramPacket(data, data.length);
    //fill the packet from the input channel
    serverSock.receive(recvPacket);
    processPacket(recvPacket);
}
```

**processPacket(DatagramPacket recvPacket)**
```
...code here to process message data
if (replyRequired) {
    InetAddress replyAddr = recvPacket.getAddress();
    int replyPort = recvPacket.getPort();
    byte replyMsg[] = ...
    DatagramPacket replyPacket = new DatagramPacket(replyMsg, replyMsg.length, replyAddr, replyPort);
    serverSock.send(replyPacket);
}
```

**UDP–Based Client**

**Client main:**
```
get server address and port;
DatagramSocket serverSock = new DatagramSocket();
loop {
    byte data[] = new byte [BUFFER_SIZE]; //create an empty packet
    DatagramPacket recvPacket = new DatagramPacket(data, data.length);
    serverSock.receive(recvPacket); //fill packet from input channel
    processPacket(recvPacket);
}
```

**sendMessage(InetAddress addr, int port, byte [] mesg)**
```
DatagramPacket sendPacket = new DatagramPacket(mesg, mesg.length, addr, port);
serverSock.send(sendPacket);
```