Game Engine Development II

Week 14

Hooman Salamat

Multiplayer Systems

Network Programming

Objectives

- Define a network socket
- Examine a client-server architecture
- Create a protocol for communication
- Apply multiplayer concepts to a game
- Evaluate problems with latency and cheating

Objectives

- A Local Area Network (LAN) is a multiplayer mode allowing for cooperation or conflict over a cable or wireless network
- Over time, the Internet became more powerful, allowing for online multiplayer

- We will be implementing a fully-networked concept

Sockets

- We need to understand how computers communicate with each other over a network
- A socket is a gateway for data
  - It is a link between two applications
  - With any concept in SFML, we are provided a class to manage them
  - First, let's examine the two main ways that computers communicate

TCP

- Transmission Control Protocol
  - Uses the Internet Protocol (IP)
  - Your computer can "speak" to another computer so long as they understand by using the same protocol (TCP/IP)
  - For example, some websites will be hosted on Linux machines and some on Windows, but they all use TCP/IP to communicate

TCP (cont'd.)

- SFML provides two classes for using TCP sockets:
  - `sf::TcpSocket` and `sf::TcpListener`
  - `TcpSocket` initiates TCP connections while `TcpListener` listens on a certain port for an incoming connection
  - Before sending and receiving data, the computers "shake hands" and create a pathway of information between them
  - A network port is an integer typically ranging from 0 to 65535 which defines that gateway
    - Do not choose a reserved port such as 80 or 1024

UDP

- User Datagram Protocol
  - Pushes an array of bytes to the network
  - No pre-established connection so data is sent over the network but there is no notification that it has been received
  - SFML provides `sf::UdpSocket` and `sf::UdpSocket.bind()` in order to bind to a port
  - Then to send data or check if it was received, use `sf::UdpSocket::send()` and `sf::UdpSocket::receive()`

Custom Protocols

- SFML also supports the HTTP and FTP protocols
  - `sf:Http` and `sf:Ftp`
  - With HTTP for example, you can send high score data to a website to display on a score board

Data Transport

- Data over the network is just a stream of bytes
- SFML has a `sf::Packet` class to turn any data into a byte stream
- Here's a sample implementation:

```
sf::Packet packet;
std::string myString = "Hello Sir!";
sf::Int32 myNumber = 20;
sf::Int8 myNumber2 = 3;
packet << myString << myNumber << myNumber2;
mySocket.send(packet);
```

Data Transport (cont'd.)

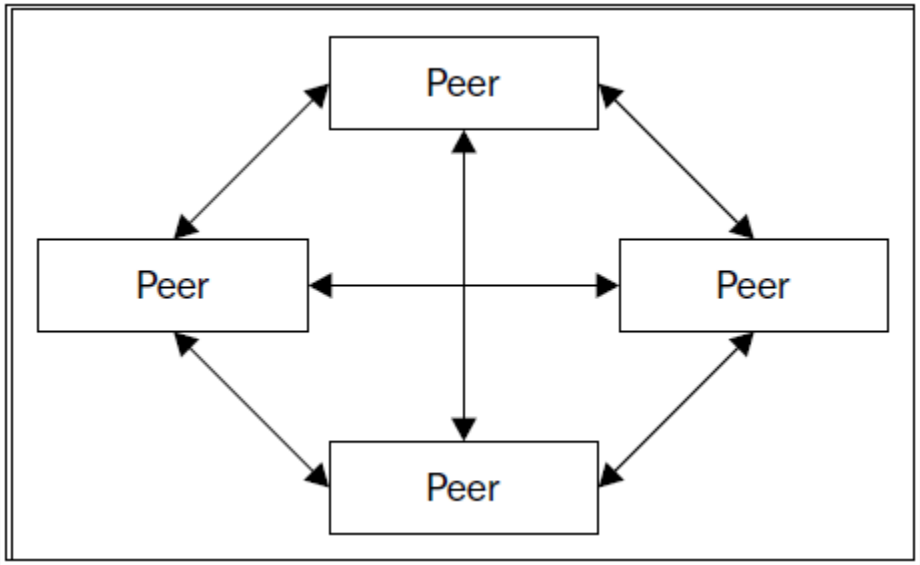
- At the other end, we implement code to receive and unpack that data into variables
- Here's a sample implementation:

```
sf::Packet packet;
std::string myString;
sf::Int32 myNumber;
sf::Int8 myNumber2;
mySocket.receive(packet);
packet >> myString >> myNumber >> myNumber2;
```

- Looks easy enough, doesn't it?

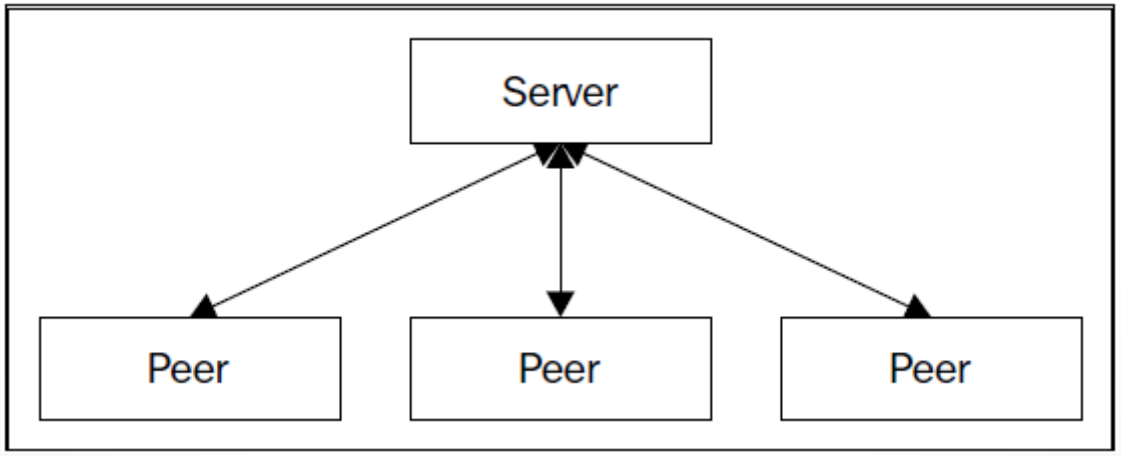
Network Architectures

- Let's look at two of the common network architectures in data communication
- **Peer-to-peer**
  - Each client is linked (connected) to each other
  - One computer becomes the "server"



Network Architectures

- **Client-server**
  - One computer becomes the server or the bridge between computers
  - One peer doesn't know the address of another
  - Could be considered more secure
  - Server does handle a heavy load, however



Setting up Multiplayer

```
class MultiplayerGameState : public State
{
public:
    MultiplayerGameState(StateStack& stack, Context context, bool isHost);
    virtual void draw();
    virtual bool update(sf::Time dt);
    virtual bool handleEvent(const sf::Event& event);
    ...

private:
    void updateBroadcastMessage(sf::Time elapsedTime);
    void handlePacket(sf::Int32 packetType,
    sf::Packet& packet);

private:
    typedef std::unique_ptr<Player> PlayerPtr;
```

Setting up Multiplayer (cont'd.)

```
private:
    World mWorld;
    sf::RenderWindow& mWindow;
    TextureHolder& mTextureHolder;

     std::map<int, PlayerPtr> mPlayers;
     std::vector<sf::Int32> mLocalPlayerIdentifiers;
     sf::TcpSocket mSocket;
     bool mConnected;
     std::unique_ptr<GameServer> mGameServer;
     sf::Clock mTickClock;

     std::vector<std::string> mBroadcasts;
     sf::Text mBroadcastText;
     sf::Time mBroadcastElapsedTime;
    ...
}
```

Server Thread and Loop

```
void GameServer::executionThread()
{
    initialize();
    while(!timeToStop) loop();
    shutdown();
}

…

handleIncomingPackets();
handleIncomingConnections();
while (stepTime >= stepInterval)
{
    updateLogic();
    stepTime -= stepInterval;
}
while (tickTime >= tickInterval)
{
    tick();
    tickTime -= tickInterval;
}
sf::sleep(sf::milliseconds(50));
```

Remote Peers

```
struct RemotePeer
{
   RemotePeer();
   sf::TcpSocket socket;
   sf::Time lastPacketTime;
   std::vector<sf::Int32> aircraftIdentifiers;
   bool ready;
   bool timedOut;
};


// Structure to store information about current aircraft state
struct AircraftInfo
{
   sf::Vector2f position;
   sf::Int32 hitpoints;
   sf::Int32 missileAmmo;
   std::map<sf::Int32, bool> realtimeActions;
};
```

Accepting New Clients

```
if (mListenerSocket.accept(mPeers.last()->socket) ==
    sf::TcpListener::Done)

// order the new client to spawn its own plane (player 1)
mAircraftInfo[mAircraftIdentifierCounter].position =
    sf::Vector2f(mBattleFieldRect.width / 2,
    mBattleFieldRect.top + mBattleFieldRect.height / 2);
sf::Packet packet;
mAircraftInfo[mAircraftIdentifierCounter].hitpoints = 100;
mAircraftInfo[mAircraftIdentifierCounter].missileAmmo = 2;
packet << static_cast<sf::Int32>(Server::SpawnSelf);
packet << mAircraftIdentifierCounter;
packet << mAircraftInfo[mAircraftIdentifierCounter].position.x;
packet << mAircraftInfo[mAircraftIdentifierCounter].position.y;
mPeers[mConnectedPlayers]
    ->aircraftIdentifiers.push_back(mAircraftIdentifierCounter);
broadcastMessage("New player!");
informWorldState(mPeers[mConnectedPlayers]->socket);
notifyPlayerSpawn(mAircraftIdentifierCounter++);

mPeers[mConnectedPlayers]->socket.send(packet);
mPeers[mConnectedPlayers]->ready = true;
mPeers[mConnectedPlayers]->lastPacketTime = now(); // prevent initial
timeouts
mAircraftCount++;
```

Handling Disconnections

```
FOREACH(sf::Int32 identifier, (*itr)->aircraftIdentifiers)
{
   sendToAll(sf::Packet() << static_cast<sf::Int32>
        (Server::PlayerDisconnect) << identifier);
   mAircraftInfo.erase(identifier);
}
mConnectedPlayers--;
mAircraftCount -= (*itr)->aircraftIdentifiers.size();

itr = mPeers.erase(itr);

// Go back to a listening state if needed
if (mConnectedPlayers < mMaxConnectedPlayers)
{
   mPeers.push_back(PeerPtr(new RemotePeer()));
   setListening(true);
}
broadcastMessage("An ally has disconnected.");
```

Incoming Packets

```
sf::Packet packet;
packet << static_cast<sf::Int32>(identifier);

namespace Server
{
    enum PacketType
    {
          BroadcastMessage,
          SpawnSelf,
          ...
    };
}

namespace Client
{
    enum PacketType
    {
          PlayerEvent,
          PlayerRealtimeChange,
          ...
    };
}
```

Incoming Packets (cont'd.)

```
bool detectedTimeout = false;
FOREACH(PeerPtr& peer, mPeers)
{
    if (peer->ready)
    {
        sf::Packet packet;
        while (peer->socket.receive(packet) == sf::Socket::Done)
        {
            // Interpret packet and react to it
            handleIncomingPacket(packet, detectedTimeout, *peer);
            peer->lastPacketTime = now();
            packet.clear();
        }
        if (now() >= peer->lastPacketTime + mClientTimeoutTime)
        {
            peer->timedOut = true;
            detectedTimeout = true;
        }
    }
}
if (detectedTimeout)
    handleDisconnections();
```

Incoming Packets (cont'd.)

```
sf::Int32 packetType;
packet >> packetType;

switch (packetType)
{
   ...
}
```

The Client

```
sf::IpAddress ip;
if (isHost)
{
    mGameServer.reset(new GameServer());
    ip = "127.0.0.1";
}
else
{
    ip = getAddressFromFile();
}
if (mSocket.connect(ip, ServerPort, sf::seconds(5.f)) ==
    sf::TcpSocket::Done)
    mConnected = true;
else
    mFailedConnectionClock.restart();

mSocket.setBlocking(false);
...
```

The Client (cont'd.)

```
sf::Packet packet;
if (mSocket.receive(packet) == sf::Socket::Done)
{
   sf::Int32 packetType;
   packet >> packetType;
   handlePacket(packetType, packet);
}

updateBroadcastMessage(dt);

mPlayerInvitationTime += dt;
if (mPlayerInvitationTime > sf::seconds(1.f))
   mPlayerInvitationTime = sf::Time::Zero;
```

The Client (cont'd.)

```
if (mTickClock.getElapsedTime() > sf::seconds(1.f / 20.f))
{
   sf::Packet positionUpdatePacket;
   positionUpdatePacket << static_cast<sf::Int32>(
   Client::PositionUpdate);
   positionUpdatePacket << static_cast<sf::Int32>(
   mLocalPlayerIdentifiers.size());
   FOREACH(sf::Int32 identifier, mLocalPlayerIdentifiers)
   {
        if (Aircraft* aircraft = mWorld.getAircraft(identifier))
             positionUpdatePacket << identifier
                  << aircraft->getPosition().x
                  << aircraft->getPosition().y;
   }

   mSocket.send(positionUpdatePacket);
   mTickClock.restart();
```

Game Actions

```
namespace GameActions
{
    enum Type { EnemyExplode };
    struct Action
    {
          Action();
          Action(Type type, sf::Vector2f position);
          Type type;
          sf::Vector2f position;
    };
}

class NetworkNode : public SceneNode
{
public:
    NetworkNode();
    void notifyGameAction(GameActions::Type type,
    sf::Vector2f position);
    bool pollGameAction(GameActions::Info& out);

private:
    std::queue<GameActions::Action> mPendingActions;
```

Game Actions (cont'd.)

```
Command command;
command.category = Category::Network;
command.action = derivedAction<NetworkNode>(
[position] (NetworkNode& node, sf::Time)
{
    node.notifyGameAction(GameActions::EnemyExplode, position);
});

GameActions::Action gameAction;
while (mWorld.pollGameAction(gameAction))
{
    sf::Packet packet;
    packet << static_cast<sf::Int32>(Client::GameEvent);
    packet << static_cast<sf::Int32>(gameAction.type);
    packet << gameAction.position.x;
    packet << gameAction.position.y;
    mSocket.send(packet);
}
```

Latency

- Latency is the delay a network packet takes to reach its destination
  - We have interpolation tricks to "fill in" the gaps until the network can re-synchronize if there's a delay in the delivery of packets

```
if (aircraft && !isLocalPlane)
{
   sf::Vector2f interpolatedPosition = aircraft->getPosition() +
     (aircraftPosition - aircraft->getPosition()) * 0.1f;
   aircraft->setPosition(interpolatedPosition);
}
```

Cheating Prevention

- A user might be able to modify their client to change their movement to appear at another location and the data would get sent back to the server
- We could fix it by doing the following:
  - On the server side, check to see if the new client data is valid or even possible given velocities and other statistics
  - If the new move isn't possible, you can ban or kick or log the action
  - Too many improbable actions could result in a kick or ban
  - If you think offline coding requires a lot of checks, just wait until you start network programming!
- As a golden rule, always check to see if what a client is requesting is possible