Weight: 18%

FULL SAIL UNIVERSITY

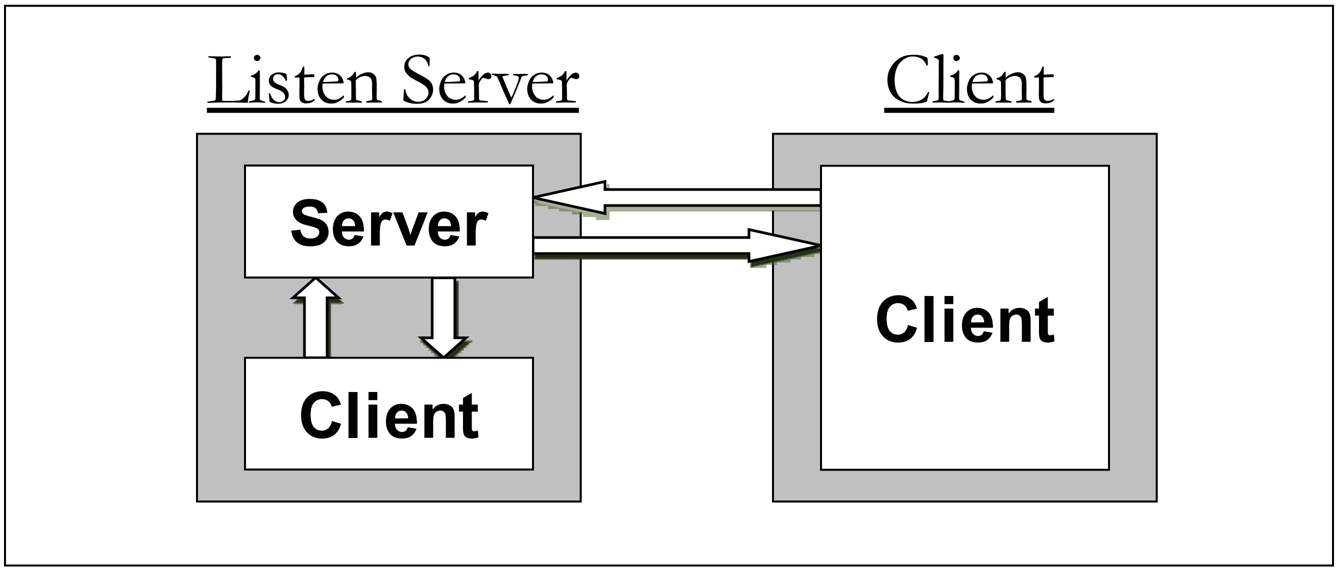
Meatball Tennis

Table of Contents

Table of Contents

Overview

In this lab assignment, we will learn to use the UDP/IP networking interface. Using UDP datagram sockets, you will be implementing a client/server "pong" application. The overall architecture that you should have in mind for this is shown in the **figure**:



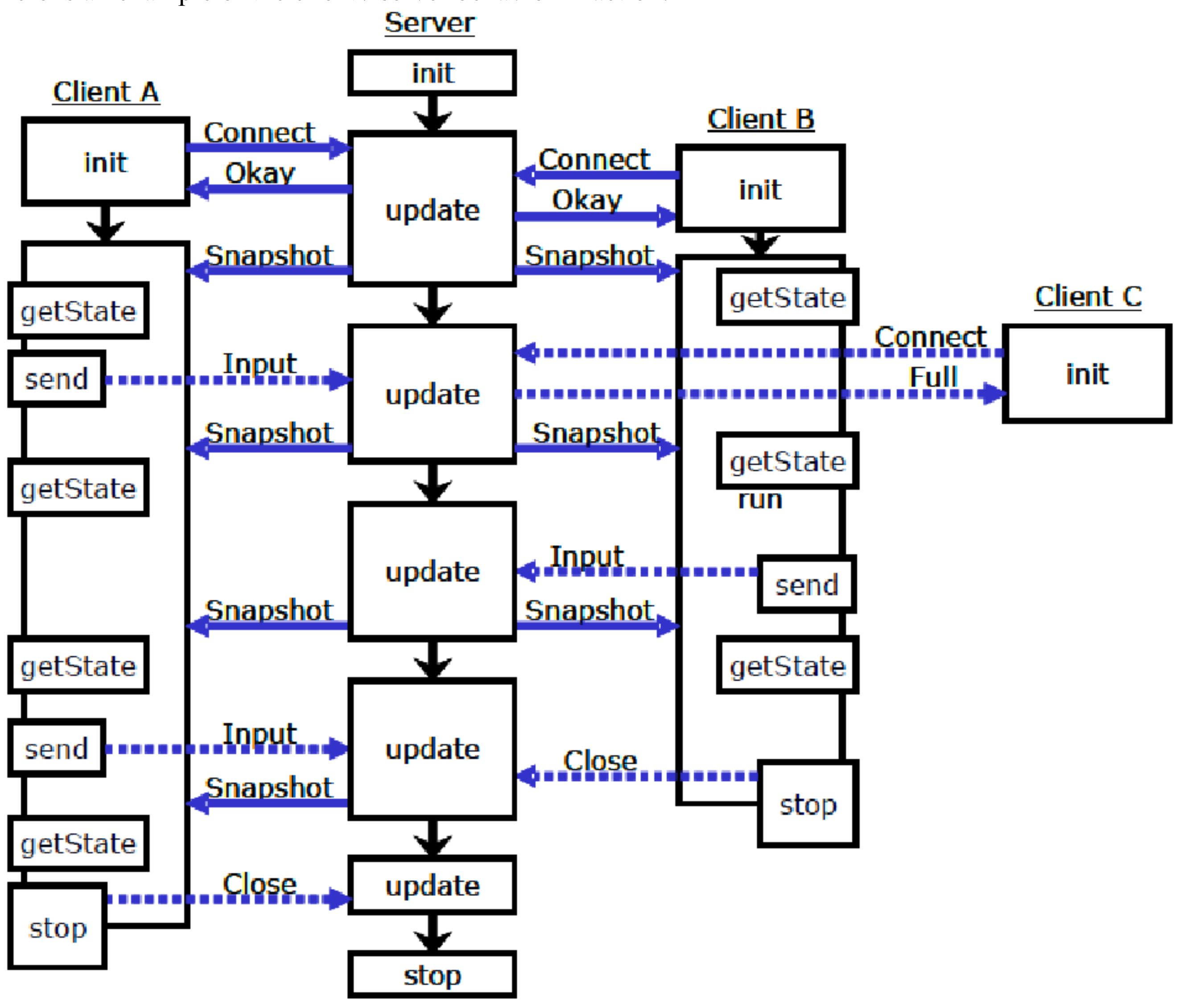
Layout of communication in Meatball Tennis

The server accepts connection requests from up to two clients. When a client connects to the server, it requests to either be the server-side client (on the listen server) or the standalone client. This dictates whether the client is Player 1 (left) or Player 2 (right). Clients send input messages to the server, and the server sends a snapshot every game tick. This lab consists of C++ networking classes and a C# GUI frontend.

As part of this assignment students must determine how to use UDP and conserve bandwidth. The exact method of doing so in code is left up to the student, but some guidance and suggestions are given later in this document.

Client / Server Behavior

Here is an example of the client / server behavior in action:



- Server initializes and sets update timer. It begins sending regular snapshots to all clients.
- Client A connects to the server (this is a listen server client); after being accepted, the client enters its run() method and begins receiving regular snapshots.
- Client B then connects to the Server and is accepted, following the same process as Client A. Upon receiving Client B's list request, the Server begins the game.
- Client C tries to connect as a client but is told by the Server that it is full.
- Client A sends the Server an input change, which the Server processes on its next update (tick.) Likewise, Client B send the server an input message that is processed during the next tick.
- Client B disconnects, and the Server returns to waiting mode.

- Finally, Client A's stop method is called (by a separate thread) and it disconnects. As a result, the listen server initiates the Server shutdown.

Below is a list of the general behavior of the client and server:

Client

- Client initiates connection. Initial message is (CL_CONNECT) followed by the player number.
- Within run() method, client should loop, receiving messages from the server and parsing them.
  (a) If the client receives a "close" (SV_CL_CLOSE) message, the client should terminate and return SHUTDOWN. It should also set the gamePhase to DISCONNECTED.
  (b) On reception of "snapshot" (SV_SNAPSHOT) the client should update the game state from message's contents. Every 10 snapshots client should send an "alive" (CL_ALIVE) message.
- The client's sendInput() method should send an "input" (CL_KEYS) message containing the changes to that client's input.
- The client's getState() method should set the state referenced by the passed pointer to the current game state.

Server

- The server responds to a client's connection request as follows:
  (a) If there is room on the server, it sends an "accept" (SV_OKAY) message.
  (b) If the server is full, the server sends a "server full" (SV_FULL) message.

The client should shutdown if the server is full.

- In update, which is called on every tick, the server will:
  (a) Receive and parse incoming messages.
    - Parse player "input" (CL_KEYS) messages (and update the key states.)
    - Parse player "alive" (CL_ALIVE) messages (and reset the player timer.)
    - Disconnect any players who have sent a "close" (SV_CL_CLOSE) message.
    - Disconnect any players from whom an "alive" message hasn't been received for 50 ticks.
  (b) Update the game state (by calling updateState() method.)
  (c) Send the most up to date snapshot to each connected client.

Note: You should be sensitive not to allow the client or server to arbitrarily delay any message.

General Approach

As with the chat system lab project, modularization is the key to making our project easier to approach. Following is a breakdown of a general approach that students might find useful in approaching Meatball Tennis.

Modularization

As with previous projects, an attempt will be made to break the project down into smaller, more easily digested parts. This is already partly done through the division between client and server classes and the current methods.

This can be further broken down by message, both those that may be received and those to be sent. In other words, for each message that might be received or sent, a separate private method could be created to deal with that particular message. This will minimize duplication of code as well.

The Client

The client's most complicated methods are **`init()`** and **`run()`**, so they will be the focus below.

Initialization

A client initialization method might look something like this:

```
init()
{

    Initialize both player key input states.
    Allocate a buffer into which messages can be received.

    Create a socket.
    Optionally, call “connect” (we'll only communicate with the server.)
    Send connect message, along with local player number (passed in.)

    Wait on the server's response message.
    If server responds “okay”, connection was successful.
    Otherwise, shutdown the client.
}
```

Main Run Loop

A client's main run loop method might look something like this:

```
run()
{
    Establish a snapshot counter.
    Allocate a buffer (NetworkMessage) into which messages can be received.

    While the client is connected:
    {
        Receive the next message.

        If message is a close message:
            Shutdown the client and set the game phase to DISCONNECTED.

        If the message is a snapshot:
            Update the current game state.
            Every 10 snapshots, send an “alive” (CL_ALIVE) message.
    }
}
```

Sending a Specific Message

After we've received a snapshot, it is often useful to have a separate method for parsing:

```
sendFooBar(args)
{
    Allocate a buffer (NetworkMessage) for building an outgoing message.
    Build the message into the buffer according to the arguments.
    Send the message to the server.
}
```

As you can see, it is useful to create private methods for the various outgoing messages so that they can be called whenever necessary. Optionally, one could also create a different method for each receivable message (though in this case there are few enough that it may not be that beneficial.)

The Server

In the server, the most complicated methods are **`init()`** and **`update()`**.

Initialization

The server's initialization method might look something like this:

```
init(server port)
{
    Initialize the game state.
    Create a listening socket.
    Bind the socket to the port.
}
```

Game Update

The server's update routine might look something like this:

```
update()
{
    Allocate a buffer into which messages can be read.
    Allocate a socket address to store the source of messages.

    While there are messages waiting to be read:
    {
        NOTE: DO NOT BLOCK ON RECEIVE IN THE SERVER.
        Read the next incoming message into the buffer.
        If there is an error caused by a network problem, return DISCONNECT.
        If there is an error caused by a server shutdown, return SHUTDOWN.
        If successful, parse the new message.
    }

    If any player has not sent an "alive" message in the last 50 ticks:
        Disconnect that player.

    Update the state of the game.
    Send the latest snapshot to each client.
}
```

Message Parsing

The server might use a routine like this to parse the messages:

```
parseMessage(source, buffer)
{
    If the message is a "connect", see if that player slot is available.
    If so, send an "okay" (SV_OKAY) message to the source.
    If not, send a "full" (SV_FULL) message to the source.

    Otherwise, determine which player the message is from (source.)

    If the message is an "input" message:
        Apply the key changes to the proper player's input state.
    If the message is an "alive" message:
        Reset the proper player's player timer.
    If the message is a "close" message:
        Disconnect the proper player.
}
```

Be careful not to block on receives within the server's methods! The server must serve two clients and blocking will prevent other client from being served.

Classes & Functions

Meatball Tennis's functionality is broken down into two classes, the Client class and the Server class. The Client class connects to a GUI front-end that allows displaying of the game state & collecting input.

Supporting Classes & Functions

In addition to the main classes, completed classes and functions are provided to aid students.

NetworkMessage Class

The **`NetworkMessage`** class is a class designed to help serialize UDP messages. They come in two types: INPUT and OUTPUT. An INPUT message is intended to be received and read; an OUTPUT message is intended to be written to and sent. There are also helper methods for sending and receiving these messages.

Here are some examples of receiving and reading from / writing and sending a NetworkMessage:

```
void receiveMessage(SOCKET socket)
{
    NetworkMessage message(INPUT);

    if (recvNetMessage(socket, message) <= 0)
        dealWithError();

    type = message.readByte();
    switch (type)
    {
        case TEST_MESSAGE:
            char testString[MAX_DATA];
            message.readString(testString, MAX_DATA);
            cout << testString << endl;
    }
}

void sendTestMessage(SOCKET socket)
{
    NetworkMessage message(OUTPUT);
    message.writeByte(TEST_MESSAGE);
    message.writeString("This is the data of the test message.");

    if (sendNetMessage(socket, message) <= 0)
        dealWithError();
}
```

The NetworkMessage class has the following methods:

```
NetworkMessage(IO _type)
```
Constructor. Builds a new network message of the specified type (INPUT or OUTPUT). An INPUT message may only be received / read; an OUTPUT message may only be written / sent.

```
int read(char* data, int offset, int length)
```
Reads **`length`** bytes into **`data`** from stream, starting at **`offset`**. Returns bytes read.

```
int readString(char* data, int length)
```
Reads **`length`** bytes into **`data`** from message up to a null terminator. Returns bytes read.

```
int8_t readByte()
int16_t readShort()
int32_t readInt()
```
Reads appropriate data-type from the message buffer and returns it.

```
int write(char* data, int offset, int length)
```
Writes **`length`** bytes into message from **`data`**, starting at **`offset`**. Returns bytes written.

```
void writeString(char* data)
void writeString(char* data, int length)
```
Writes **`length`** bytes into message from **`data`** up to a null terminator. Returns bytes written.

```
void writeByte(int8_t output)
void writeShort(int16_t output)
void writeInt(int32_t output)
```
Writes appropriate data-type to the message buffer.

```
void reset(IO newType = currentType)
```
Clears the message and (optionally) changes the message type. This clears all output in the stream and all input to be cleared (erased.) All data in the buffers will be lost!

```
int bytesAvailable()
```
Returns the number of bytes available (in the message for reading or empty bytes for writing.)

Helper Functions

The lab also includes some helper functions to make UDP networking easier:

```
int sendNetMessage(SOCKET s, NetworkMessage& msg)
int recvNetMessage(SOCKET s, NetworkMessage& msg)
int sendtoNetMessage(SOCKET s, NetworkMessage& msg, sockaddr_in* source)
int recvfromNetMessage(SOCKET s, NetworkMessage& msg, sockaddr_in* source)
```

Functions equivalent to **`send()`**, **`recv()`**, **`sendto()`**, and **`recvfrom()`**; intended only for use with UDP sockets. Return values for these functions are identical to the functions just listed.

Client Class

The Client class is responsible for all network functions of the Meatball Tennis client. It is derived from user.

`Public TODO methods:`

`int init(const char* address, uint16_t port, uint8_t _player)`

This method should connect to the server at **`address:port`** and request connection. NOTE: The client should establish the connection before leaving this method!

-Returns `SUCCESS` upon successful connection to the server.
-If the address is not valid, returns `ADDRESS_ERROR`.
-If a socket cannot be set up, returns `SETUP_ERROR`.
-If there is a network error, returns `DISCONNECT`.
-If the server is full, returns d`,` `SHUTDOWN`.

`int run()`

The main client method. This method is responsible for receiving and parsing all messages but may delegate those tasks to other (private) methods.

Returns `SHUTDOWN` upon graceful shutdown of the client.
-If there is an invalid message, returns `MESSAGE_ERROR`.
-If there is a network error, returns `DISCONNECT`.

`void stop()`

This method will be called by the interface to shut down the client. This should send the close message, tell the class to disconnect and close the socket.

`void sendInput(int8_t keyUp, int8_t keyDown, int8_t keyQuit)`

Should send the current state of the up and down keys to the server.

`void getState(GameState* target)`

Should set the GameState pointed to by **`target`** equal to the current state.

`Private recommended methods:`

`int sendAlive() // Recommended`

This method is recommended but not required. Sends an "alive" message to the server.

`int sendClose() // Recommended`

This method is recommended but not required. Sends a "close" message to the server.

Return Value Notes

Returning "SHUTDOWN" from public methods reflects that stop() has already been called (or is unnecessary due to lack of state initialization.)

Returning "DISCONNECT" from public methods reflects that the class's stop() method should be called to shut down the client/server.

Server Class

The Server class is responsible for all network functions of the Meatball Tennis server.

`Public TODO methods:`

`int init(uint16_t port)`
This method should start a server on **`port`** and prepare to accept connections.

Returns `SUCCESS` upon successful setup of server.
-If the server could not bind to the port, returns `BIND_ERROR`.
-If a socket cannot be set up, returns `SETUP_ERROR`.

`int update()`
This method is called every server tick. It reads & parses messages and updates the state.

Returns `SUCCESS` upon normal completion of a game tick.
-If there is a network error, returns `DISCONNECT`.
-If the server was stopped, returns `SHUTDOWN`.

`void stop()`
This method will be called by the interface to shut down the server. This should send the close message to each client, tell the class to disconnect, and close the socket.

`Private recommended methods:`

`int parseMessage(sockaddr_in src, NetworkMessage& msg) // Recmd.`
This method is recommended but not required. It parses a single message.

`int sendMessage(sockaddr_in dest, NetworkMessage& msg) // Recmd.`
This method is recommended but not required. It sends the message in **`buf`** to **`dest`**.

`int sendOkay(sockaddr_in destination) // Recommended`
This method is recommended but not required. Sends an “okay” message to destination.

`int sendFull(sockaddr_in destination) // Recommended`
This method is recommended but not required. Sends a “full” message to destination.

`int sendState() // Recommended`
This method is recommended but not required. Sends the latest snapshot to each client.

`int sendClose() // Recommended`
This method is recommended but not required. Sends a “close” message to each client.

`Completed private methods:`

`void initState() // Completed`
This method is already written. It initializes the game state within the server.

`void updateState() // Completed`
This method is already written. It updates the game state (called each tick.)

`void disconnectedClient(int player) // Completed`
This method is already written. It marks the player as disconnected.

Submissions & Grading

This lab project is made up of two parts; the first is the client section and the second is the server section. All lab projects are to be submitted via the Course Management System (CMS.) Submit only those files listed in the “Submit.txt” file within the lab package in the **ROOT** of the zip file and follow the naming convention outlined in the course syllabus.

Grading will be according to the lab project rubric available on the studentvfiler with other class materials.

Protocol Specification (Appendix)

In this project, messages begin with a message header made up of a **1 byte `type`**:

There are a total of seven (7) message types:

Type	To/From Server	Code	Description
CL_CONNECT	to	1	Client request to connect to the server (includes player number)
CL_KEYS	to	2	Informs the client of the current state of the player's input
CL_ALIVE	to	3	Notifies the server that the client is still there
SV_OKAY	from	4	Informs the client that it has successfully connected to the server
SV_FULL	from	5	Informs a client that the server is full and that it cannot connect
SV_SNAPSHOT	from	6	Provides client with most recent game state
SV_CL_CLOSE	to/from	7	Informs the receiver of a disconnect (from client or server)

Client-to-Server Messages

The client should issue and the server should be able to handle the following messages:

`CL_CONNECT`	**`Player# (1)ui`**

Type:	**`CL_CONNECT`**
Function:	Issues client request to join game server.
Data:	The client's player number (**`uint8_t`**.)

`CL_KEYS`	**`KeyUp (1)i`**	**`KeyDown (1)i`**

Type:	**`CL_KEYS`**
Function:	Informs server of change in client's key input.
Data:	KeyUp (**`int8_t`**) is **true** if the **up arrow key** is pressed, **false** otherwise. KeyDown (**`int8_t`**) is **true** if the **down arrow key** is pressed, **false** otherwise.

`CL_ALIVE`

Type:	**`CL_ALIVE`**
Function:	Informs the server that the client is still connected.

`SV_CL_CLOSE`

Type:	**`SV_CL_CLOSE`**
Function:	Informs the server that the client is disconnecting.

Server-to-Client Messages

The server should issue and the client should be able to handle the following messages:

Sequencing Number (2)ui	**SV_OKAY**

Type: **SV_OKAY**

Function: Informs a client that it has successfully connected to the server.

Sequencing Number (2)ui	**SV_FULL**

Type: **SV_FULL**

Function: Informs a client that the server is full and that it cannot connect.

Sequencing Number (2)ui	**SV_SNAPSHOT**	**Phase (1)ui**	**BallX (2)ui**	**BallY (2)ui**	**P0-Y (2)ui**	**P0-Score (2)ui**	**P1-Y (2)ui**	**P1-Score (2)ui**

Type: **SV_SNAPSHOT**

Function: Provides the client with the most up-to-date game state.

Data: The game phase (**uint8_t**), ball x/y position (**uint16_t**), and player positions & scores (**uint16_t**.)

Sequencing Number (2)ui	**SV_CL_CLOSE**

Type: **SV_CL_CLOSE**

Function: Informs client that the server is shutting down.

Sequencing Number

The sequencing number is a number that is used to keep messages coming into the system in the correct order. You will need 2 of these numbers in your Server(one for each of the clients connected to the system), and one to keep track of the number on the client. When any message is sent out of the Server attach a sequencing number to the front of the message that corresponds to the client that is going to get the message. If the client gets a message out of order, reject the message and continue with the rest of the code.

Other Notes

Following are a few extra notes regarding Meatball Tennis:

1) The server has four states: DISCONNECTED, WAITING, RUNNING, and GAMEOVER.

2) "Player 1" on-screen corresponds to "Player0" in the game state; it is the client-half of the Listen Server. "Player 2" on-screen is "Player1" in the game state & is the standalone client.

3) Grading will involve testing all returns values and messages, so be sure follow the spec!