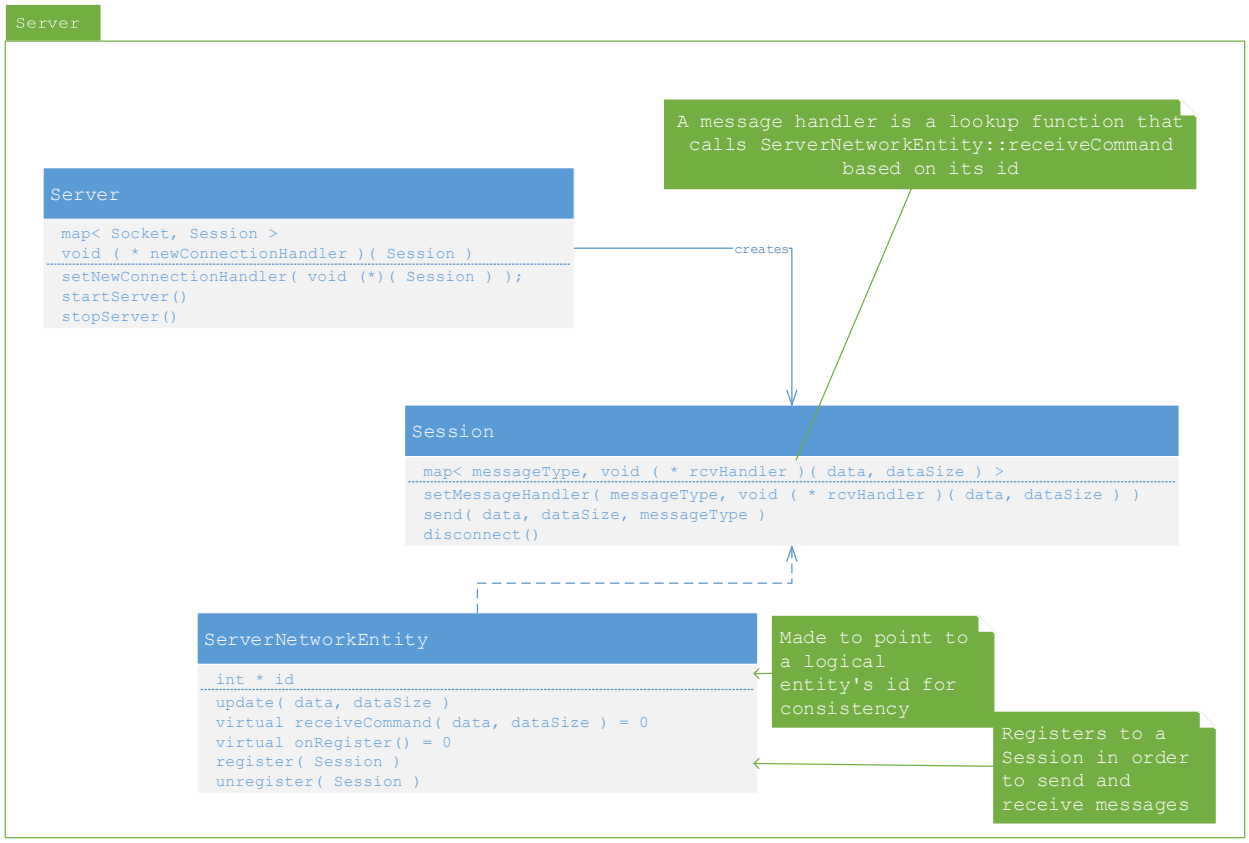
# Networking Team Design

## Server Side

**Jeff, Georgi & Eric**

# Table of Contents

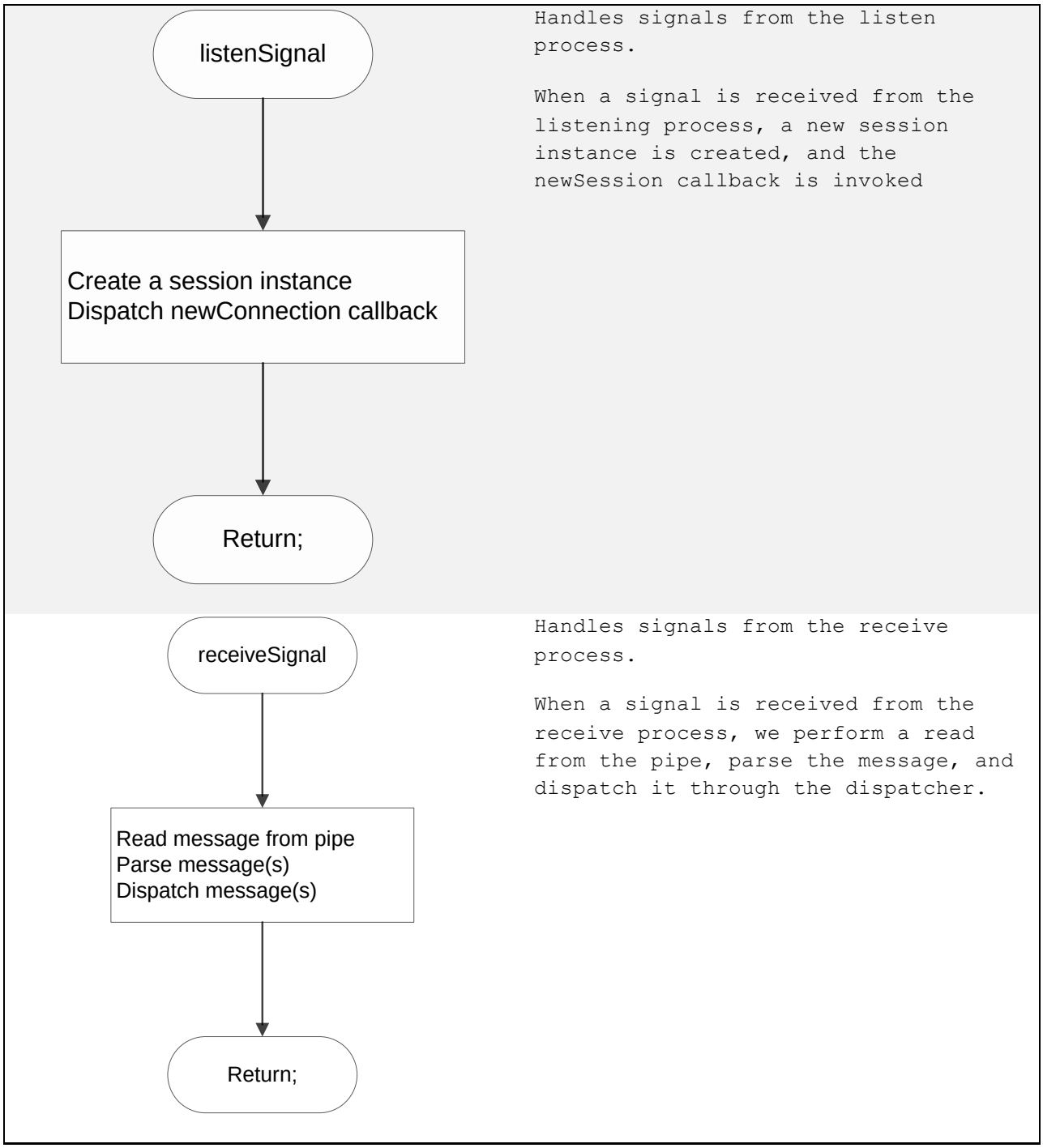
# Class Diagram

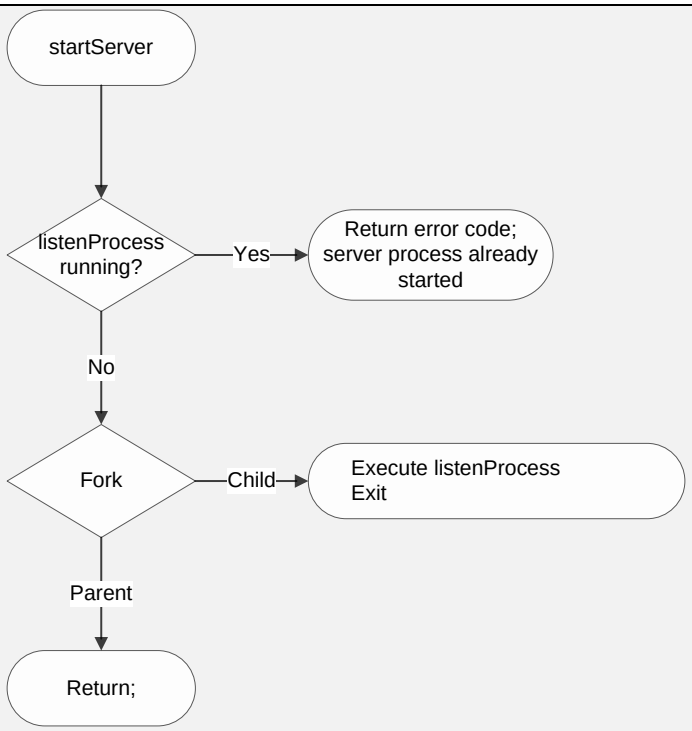


# Flow Diagrams

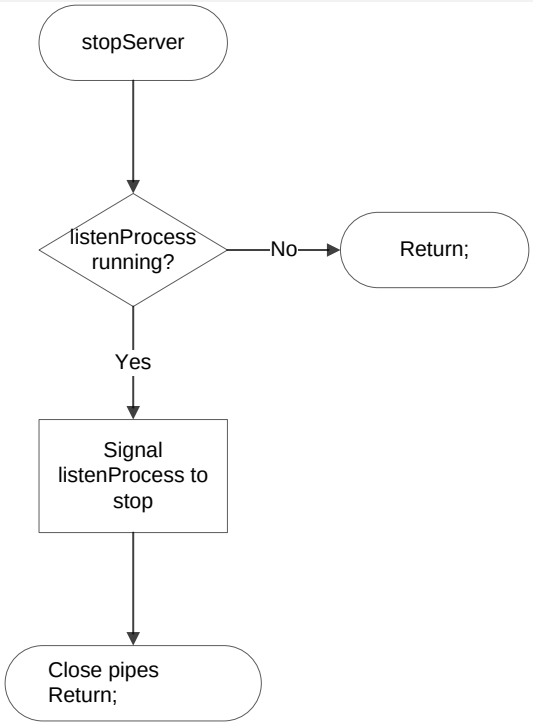
## Game Logic Thread

The networking API exposed to the game logic people are all non-blocking functions.

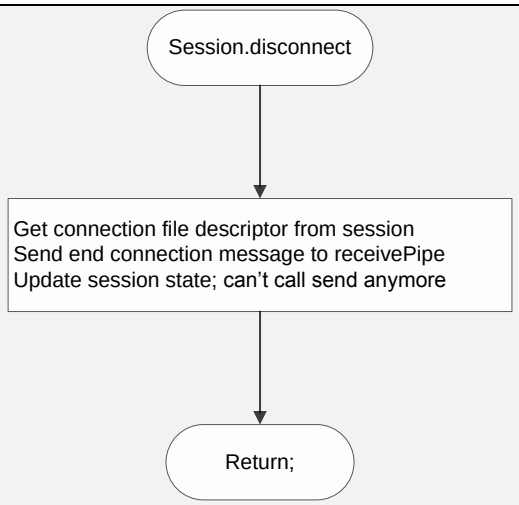




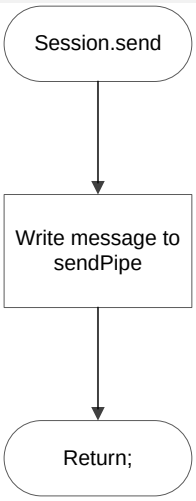

The start server function. It starts the server on a separate process if it isn't already running.




Stops the server process if it is running.




Ends the calling session, and sends an "end connection" message to the receive process.



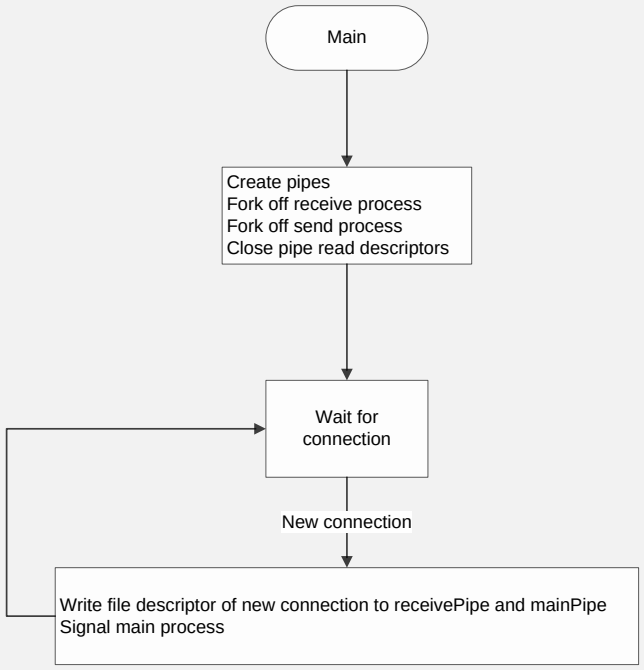

Sends the specified data over the network.

When this function is called, it writes the data to send into a message, and sends the message to the send process.

## Listen Process

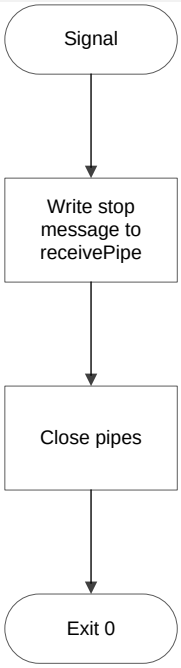
The listen process is responsible for listening for new connection requests, and it informs the main thread (game logic), and receive thread of the new connections whenever a new connection is made.

This process is stopped with a signal.



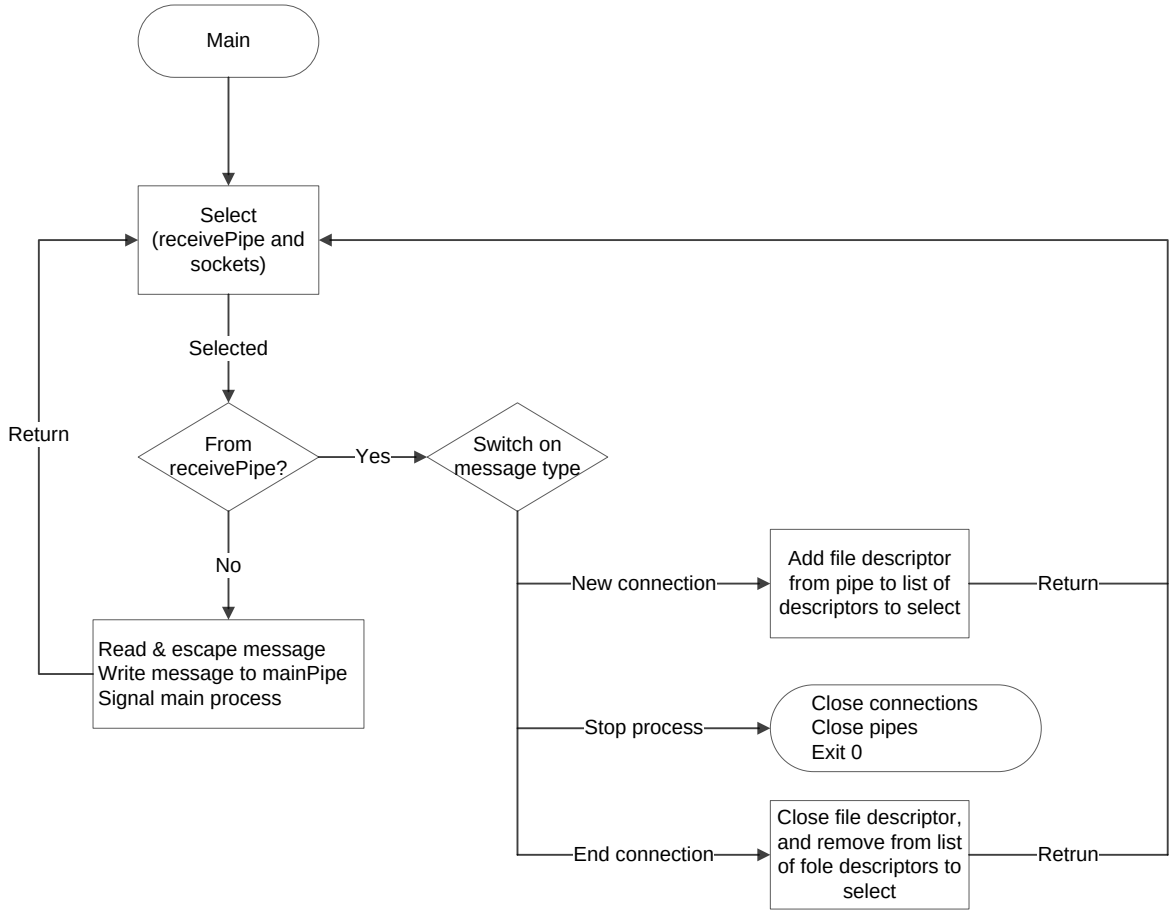

This is the main function of the listening process.

It creates the other processes, then continuously waits for new connections




When the listening process receives a signal, it sends messages and closes pipes as necessary to stop the receive and send processes, then exits itself.

## Receive Process



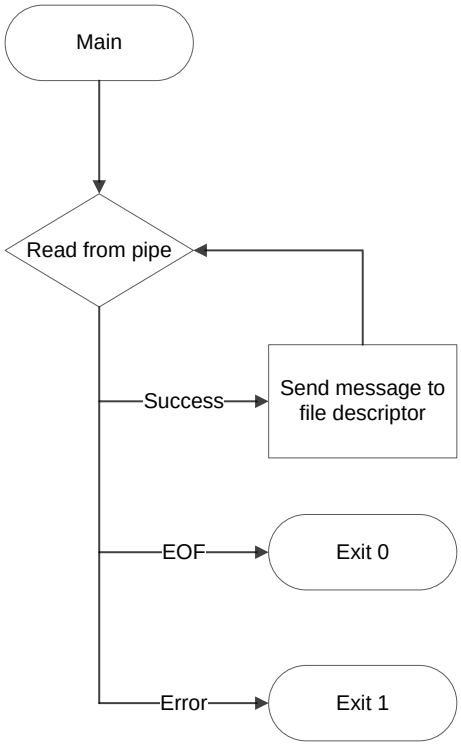

The receive process polls an array of file descriptors. The array contains a pipe for controlling the process, and the rest of the file descriptors are sockets of connected clients.

If the poll function returns a file descriptor of a socket, it adds a header to it, forwards to the main process and finally, signals the main process to notify it of the message in the pipe.

If the poll function returns the file descriptor of the control pipe, it will do one of 3 things depending on the type of control message it is:

- The message is an "add connection" message. This kind of message is sent to this process when a new client has connected, and we want to add the file descriptor of the socket to the array of sockets to poll.
- The message is an "end connection" message. This message is sent to this process when a connection terminates. The corresponding socket file descriptor is removed from the array of file descriptors to poll.
- The message is a "Stop process" message; this process closes all the file descriptors, and ends.

### Send Process




This is the main function of the send process.

This function reads from a pipe in a loop:

- When the pipe closes, and returns EOF, then this process exits.
- When the pipe returns an error, then the process exits in error.
- When the pipe returns data, it parses the header, and sends the message out through the specified socket file descriptor, then it loops back to read more data from the pipe.

# Pseudo-Code

## Functions used on Game Logics main thread

### listenSignal()

```
Create a new session instance, and pass it the socket
Add the socket, and session to our map of sessions and sockets
Server.newConnection(Session)
Return;
```

### receiveSignal()

```
retrieve Session based on socket
extract destination id from message
retrieve ServerNetworkEntity based on id
call ServerNetworkEntity::receiveCommand()
```

### startServer()

```
if LISTENING
    return ERROR // connection process already started
if pid <- fork() fails
    return ERROR
switch pid
    parent:
```

```
        create threads for send thread pool
    child:
        listenProcess()
        exit 0
```

**stopServer()**

```
if !LISTENING
    return
send signal to listen process
close pipes
return
```

**Session.disconnect()**

```
get socket file descriptor
write end connection message to receivePipe with the socket file descriptor
invalidate the session...set state to closed; can't send to this session
anymore
```

**Session.send()**

```
assign thread to sendMessage()
return
```

**ServerNetworkEntity.update()**

```
iterate through session set and invoke their send functions

virtual ServerNetworkEntity.receiveCommand(struct command) = 0

virtual ServerNetworkEntity.getOnRegisterUpdate() = 0
```

**ServerNetworkEntity.register(session)**

```
add session to set of sessions
```

**ServerNetworkEntity.unregister(session)**

```
remove session to set of sessions
```

## Listen Process

**listenProcess()**

```
if create receivePipe or create mainPipe fails
    return ERROR
if (rpid = fork() ) = -1)
    return ERROR
else if rpid > 0
    receiveProcess()
    return
if (spid = fork() ) = -1)
    return ERROR
else if spid > 0
    sendProcess()
```

```
    return
close pipe read descriptors
if !create listenSocket
    kill children
    return ERROR
bind listenSocket to PORT // public server port
listen()
register listenSignalHandler callback
for(;;)
    if ((new_sd = accept()) != -1)
        write new_sd to receivePipe
        write new_sd to mainPipe
        signal main process
    else
        Handle error somehow
```

**listenSignalHandler()**

```
write stop message to receivePipe
close receivePipe and mainPipe
close listenSocket
exit
```

## Receive Process

**receiveProcess()**

```
for(;;)
    if poll(receivePipe[0] and sockets) // look at poll, ppoll. you're right,
i think poll fits our needs better
        if from receivePipe
            switch(message type)
            case newConnection:
                update socket list and number
                break
            case stopProcess
                close all receive sockets
                close pipes
                exit
            case endConnection
                close socket
                remove socket from list
                decrease socket count
                break
        else // from a socket
            read and escape message
            write message to mainPipe
            signal main process
```

## Send Threads

**sendMessage()**

```
    write message to appropriate socket
    if write fails
        return ERROR
```