



Final Project: Internet Radio Application

COMPUTER NETWORKS 2 – LABORATORY



Internet Radio Application

Final project of this course!

More Advanced use of sockets.

Multithreaded programming.

Implement a simple server-client protocol.

Stream audio over UDP.

Runs on your own
GNS3 topology (from lab 6/5/4).

Best programming project.

Internet Radio Application: Radio over Multicast

FUNCTIONALITY AND PROTOCOL

Server

- Multicast stations: each station a different multicast group, all use same port (UDP)
- Sends info to clients (TCP).
- Handles multiple clients connections(TCP)

Client

- Plays music! (acting as a simple UDP receiver).
- Communicates with the server for data(using TCP) .
- Can upload songs to the server (using TCP).
- Interacts with a human (using English).

Client - “Controller”



- Two sockets, one TCP one UDP.
- User input, and the user's interface with the server.
- Handshake with the server to learn the multicast address.
- Can 'ask' about songs.
- Receives audio packets from the server and plays music.
- Changes the multicast stream(station) to change songs.
- Can upload new songs to the server.
- Interacts with a server using a protocol.

Server



Many TCP and UDP sockets

Maintains control connections with clients.

Transmits stations via UDP to multicast

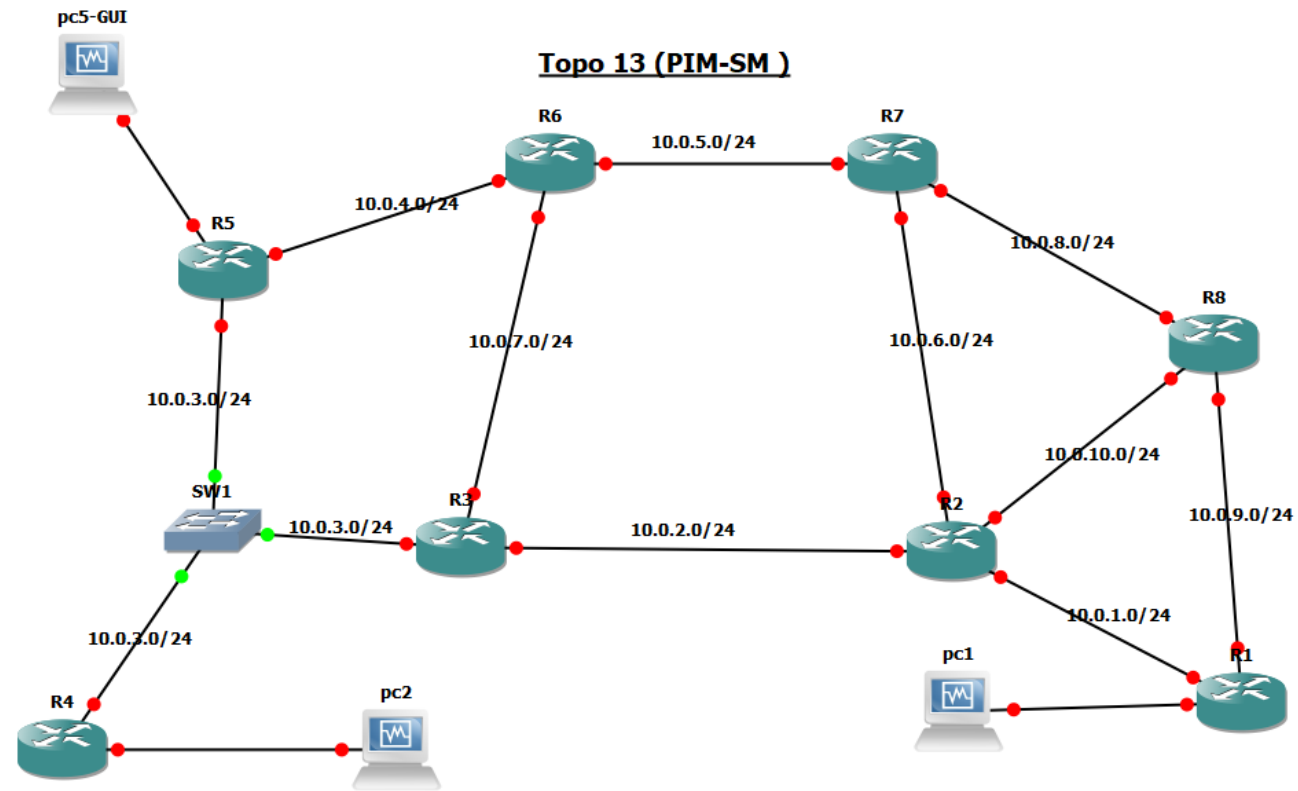
Maintains several stations

Notifies the clients about a new station.

Interacts with clients using a protocol.

Server is stateful.

Demo





Server – Client Protocol

HOW DO THINGS CONNECT

Server – Client Protocol

The server and client interact using a simple protocol.

The protocol is a set of rules, messages, timeouts, etc, that both the server and client agree on.

The protocol is what makes communications possible! Any pair of client server program can successfully connect!

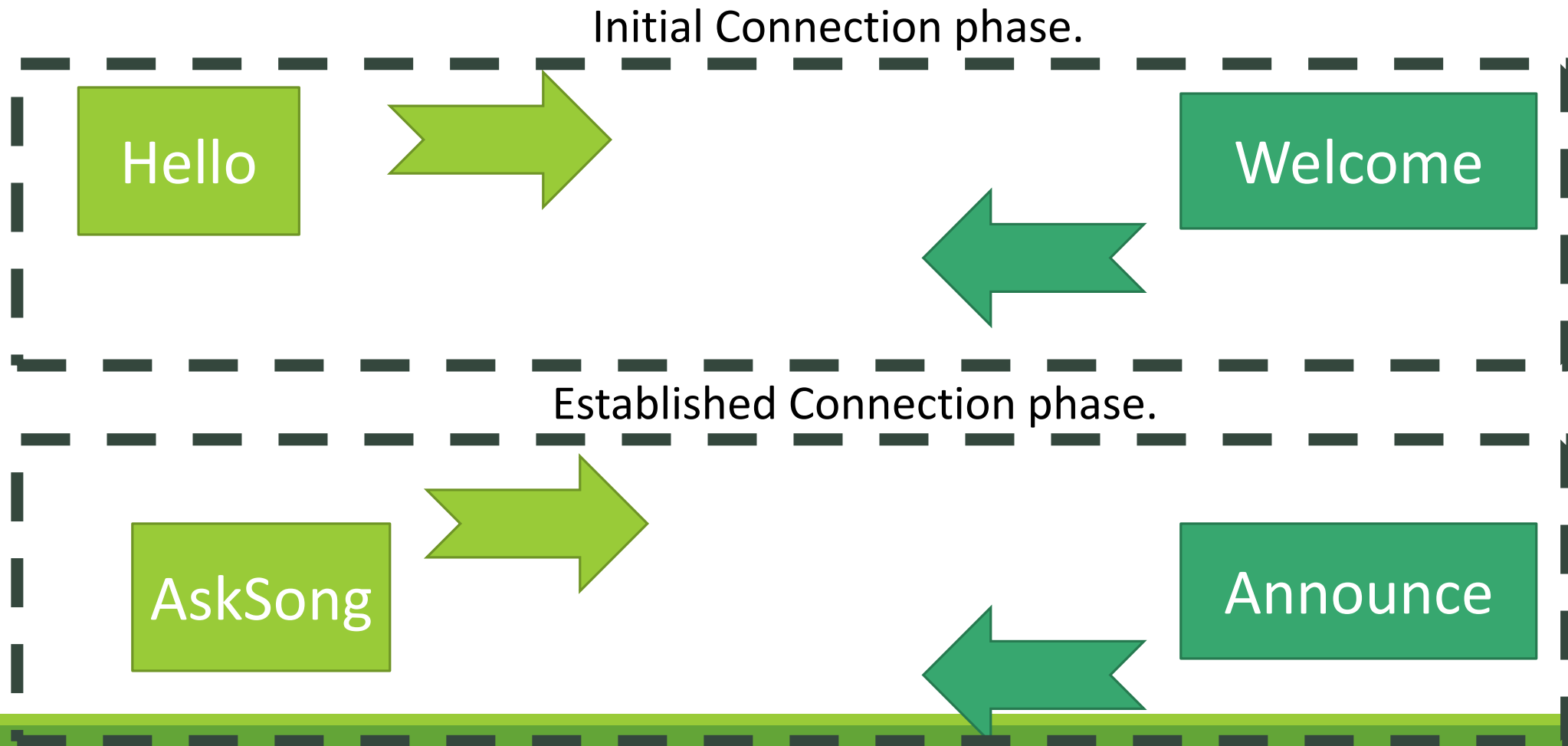
If either the client or the server behave in a way that is contrary to the protocol someone has to quit.

- If the server misbehaves, the client will disconnect itself
- If the client misbehaves the server will disconnect the client .

Server – Client Protocol: basics

Client Controller

Server



Server – Client Protocol: messages

Lets take a closer look at the two most basic messages we saw earlier:

First, “Hello”:

```
uint8_t  commandType = 0;  
uint16_t reserved = 0;
```

Second, “Welcome”:

```
uint8_t    replyType = 0;  
uint16_t   numStations;  
uint32_t   multicastGroup;  
uint16_t   portNumber;
```

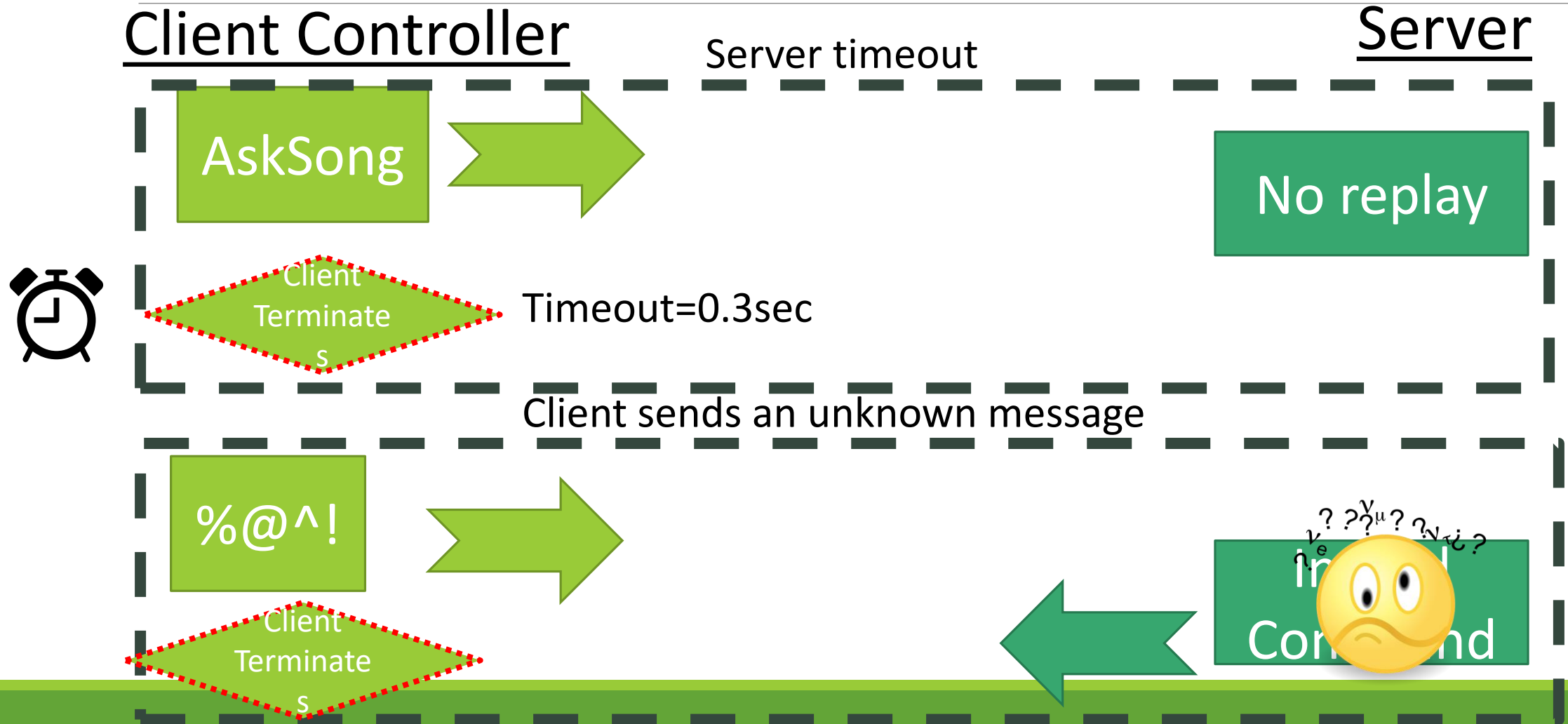
Each message has a known size.

‘Simply’ fill a buffer with the right fields and send!

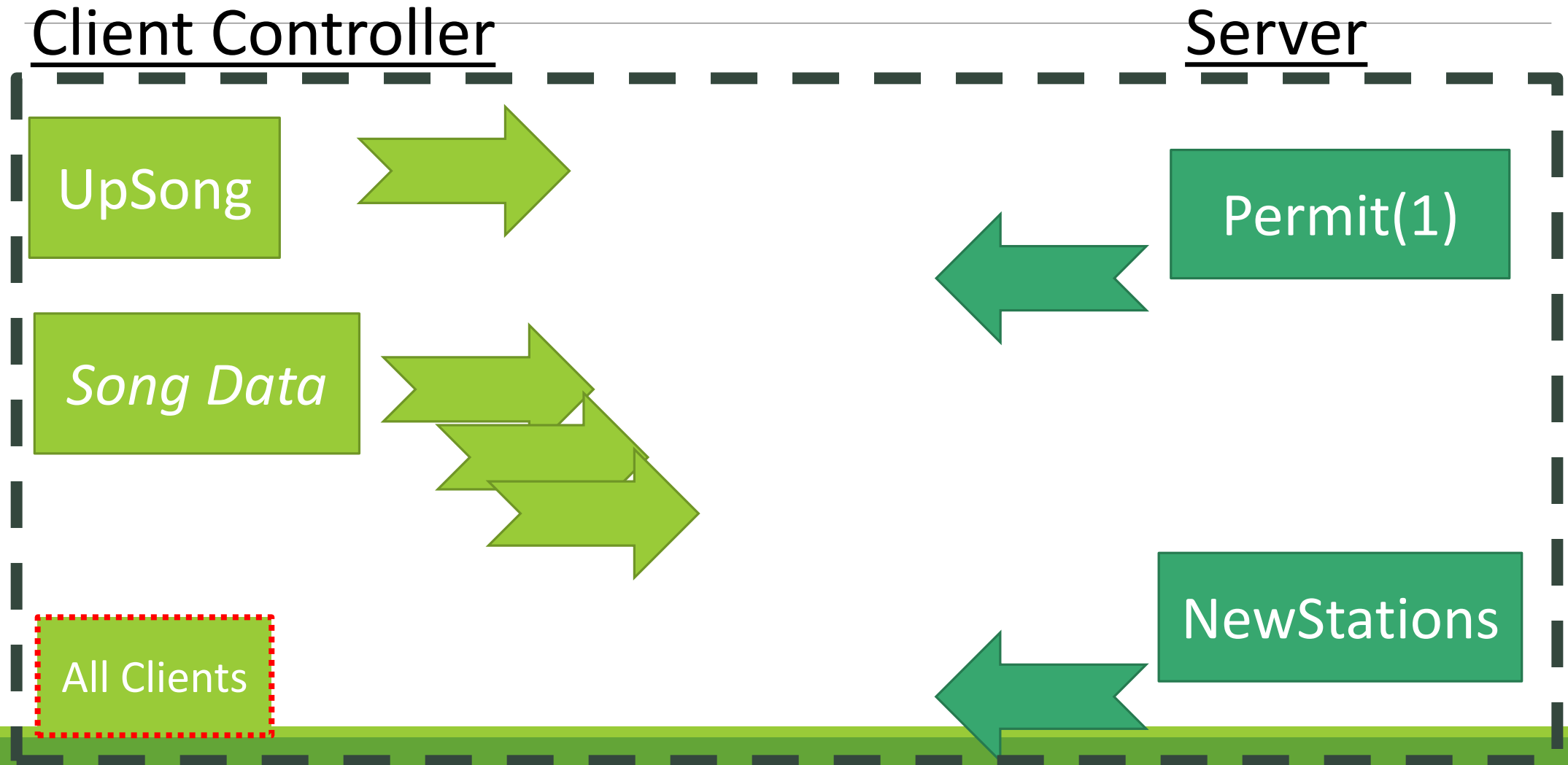
- Remember to use switch from host to network order when needed.

See “Internet Radio Application.pdf” for full details.

Server – Client Protocol: Errors



Server – Client Protocol: upload procedure

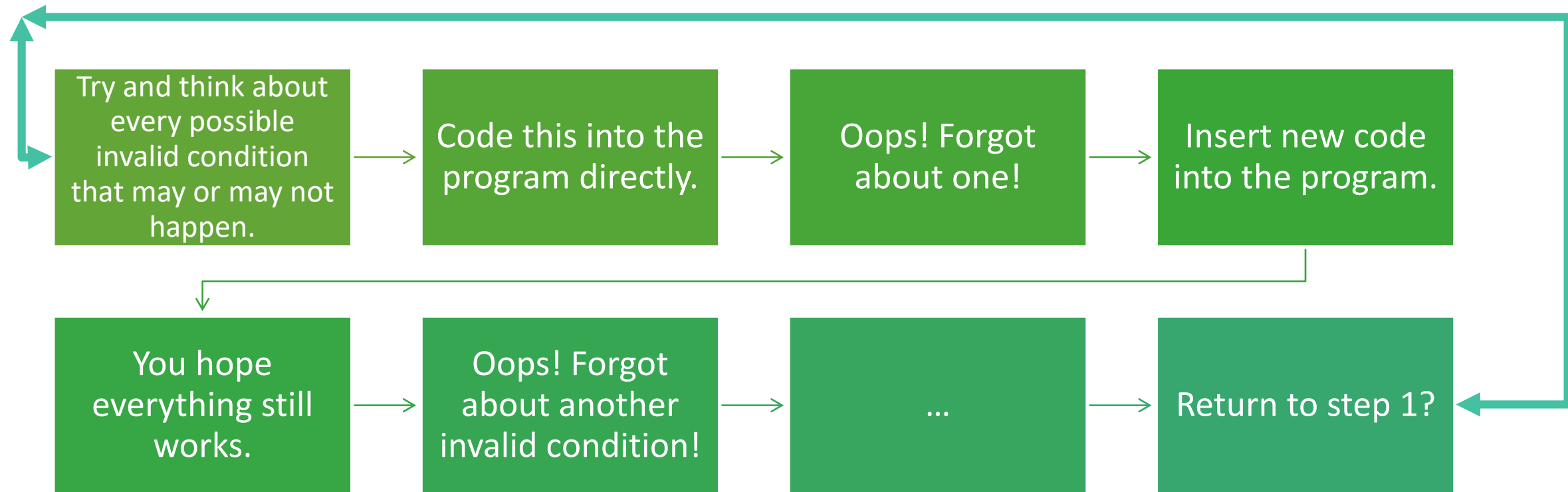




Protocol Implementation

HOW TO MAKE IT WORK

Protocol Implementation: First steps...



Protocol Implementation: Finite State machine

1. Find all possible “states” and “transitions” that are expected to happen.
2. Code this into the program directly.
3. Oops! Something unexpected happened!
4. Just go to the default state.



Protocol Implementation: Finite State machine

This is a very wide topic, we will try to use a very basic and intuitive approach.

We can think of the Finite State machine as a graph where each vertex is a state and each edge is a transition

What is a state?

- A state is any condition where our state machine **waits** for a **specific** input.

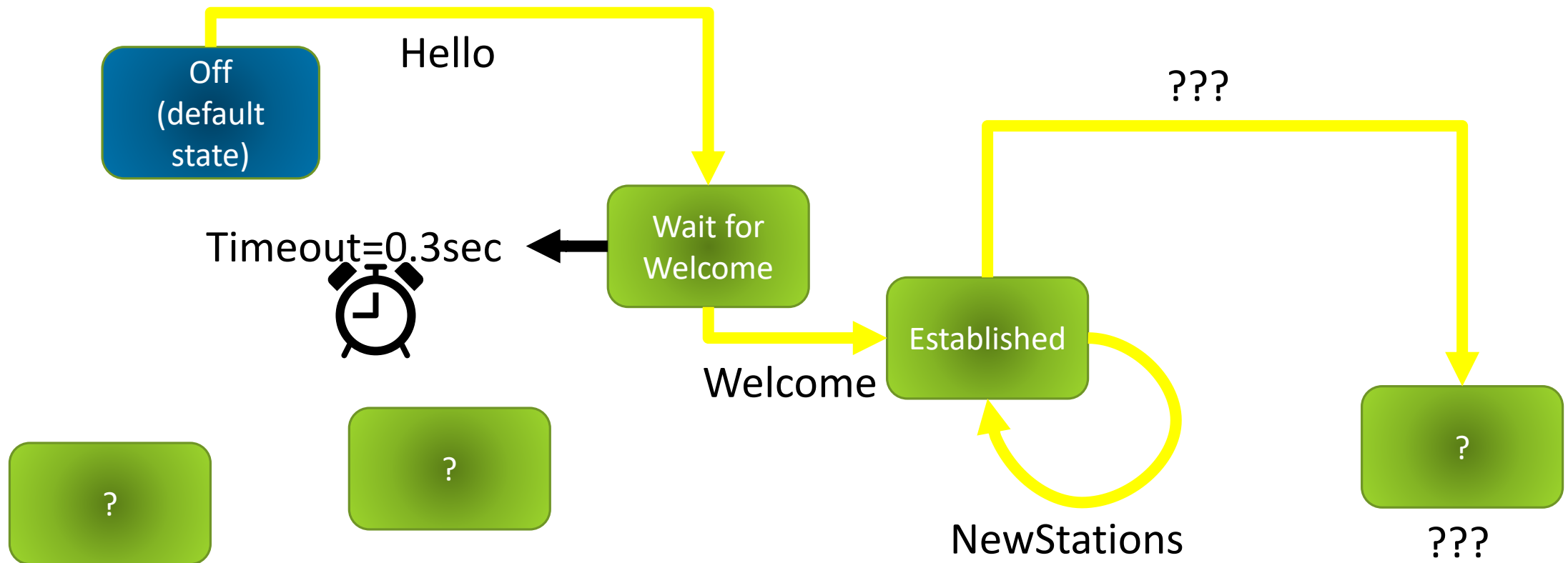
What is a transition

- A transition is any event which causes our state machine to change states.
- Lets assume the transitions are our input.

Client's Finite State machine

To simplify things let's ignore the "invalid command" message type.

Just remember any time we get an "invalid command" we go to the "off" state



Finite State machine

Let see how we can use our state machine in the program.

This is merely an example! you may choose to implement this very differently.

```
//Each time we get a message from the server we use this function//
int handleMsgFromServer( int CurrState , char* msg)
int newState;
switch (CurrState){
    case OFF:{newState =offStateMsgHandler(msg)}
    case WelcomeWait:{newState = WelcomeWaitMsgHandler(msg)}
    case Established:{newState = EstablishedMsgHandler(msg)}
    ...
    default: {terminate() /**/basically go to the "off state" /**/ }
```

Finite State machine cont.

```
//each time we get a welcome from the server we go to this function//
int WelcomeWaitMsgHandler( char* msg)
int newState, msgType=msg[0];
switch (msgType){
    case WELCOME:{
        handleWelcom(msg);
        newState =ESTABLISHED;
    }
    case OTHER???:{newState = ?}
...
    default: {newState =OFF/**/basically go to the "off state" /**/ }
Return newstate;
```