DEVELOPING A REMOTE LAB STATION FOR SOFTWARE ENGINEERING COURSES IN CYBERPHYSICAL SYSTEMS

James Carroll and Janusz Zalewski
Dept. of Software Engineering
Florida Gulf Coast University
Ft. Myers, FL 33965, USA
zalewski@fgcu.edu

ABSTRACT

This paper discusses the process of developing a remote lab station for use in software engineering courses for cyberphysical systems. A cyberphysical system is defined as an embedded system with Internet accessibility. A remote lab has been built at the authors’ institution to accommodate teaching online software engineering courses with remote access to experiments. One specific lab station is described here, which allows remote control of a device access and remote software development, uploading, testing, debugging and operation. In the implementation of the lab station, Raspberry Pi hardware and QP software development environments have been used to provide control of a remote car.

KEYWORDS

Software Engineering, Embedded Systems, Cyberphysical Systems, Remote Labs, Online Labs.

1 INTRODUCTION

Due to the ubiquitous nature and pervasiveness of the Internet, engineering college education faces numerous challenges. Software Engineering discipline is no exception. One specific issue, which needs attention, is offering courses online. In engineering, this is particularly difficult, because one cannot graduate a good engineer without having them gain practical experience in the lab. Therefore, multiple universities around the world have implemented online laboratories, also called remote labs or web-based labs.

The essential component of these labs is a remote computerized lab station, with some sort of a user interface, performing usual functions as in a hands-on local lab, with a capability to offer the same experience to a student who logs on from a remote location. These are not just virtual labs, where remotely accessible software offers simulation experiments to students who log in according to certain procedures. In online labs, real instrumentation is used remotely, with real-time access to experiments.

Multiple instances of such labs have been created for the last twenty years, or so, and are in operation at several institutions of higher education, in the U.S., Europe, Australia, and elsewhere [1-3]. In science disciplines such as physics, chemistry, biology, where remotely controlled instruments can provide the same level of experience as hands-on labs, online labs have been reportedly very successful and are definitely expanding [4-5]. Several engineering disciplines also adopted the concept of online labs, for example, control engineering [6], chemical engineering [7], electrical and computer engineering [8] and so on.
With software engineering, however, the situation is more complicated, because it is not enough for a student to access a remote instrument for just experimentation, as in other science or engineering disciplines. In those disciplines, experiments are pre-set, with experiment control software installed on the instruments and not subject to change. In software engineering, this is not the case. The very essence of the discipline requires students to learn how to develop software for these instruments, then upload it to a remote instrument or device, and ultimately test various software modules and debug them while they are running on a remote device.

This is a critical distinction between remote software engineering labs and remote labs in all other disciplines, whether in sciences or engineering. In previous publications we presented and discussed various aspects of software engineering labs at the authors’ institution, including operation and use of a sample individual lab station [9], as well as an overview of the entire lab [10] and experiences with using the lab in embedded systems courses [11]. This paper focuses on outlining the development of one such remote lab station, to let students learn software engineering concepts with online access to a remotely accessible device. The objective is to show the entire development process, to let the educators know what exactly is involved in building such labs.

The rest of the paper is structured as follows. Section 2 presents briefly requirements for such station, from the educational perspective, Section 3 discusses development tools needed for building and using such station: host and target platforms, and a remote device. Section 4 outlines details of software development and testing, and Section 5 ends the paper with some conclusions.

2 EDUCATIONAL CONSIDERATIONS

Pedagogy is the essential driving factor in every course or lab design. In case of this project, there are three major educational determinants of the lab station project’s contents:

- knowledge of software design issues for cyberphysical systems, including an appropriate design methodology and design notation;
- principles of remote implementation and remote debugging and testing of an application;
- remote execution of an application to control operation of a remote device.

These three stages translate directly into organization of the lab, with three different components in mind:

- Host tools to support and enforce good design and development practices.
- Remote target protocols to support remote development, uploading, debugging and testing.
- Internet application layer protocols to allow for remote operation of the device.

All three architectural components of the lab are presented in Figure 1 and each is discussed separately in the following section.

![Fig. 1. Architectural components of the online lab.](image)
3 DEVELOPMENT TOOLS

3.1 Host Development Tools

To follow appropriate software design methodologies, the primary requirements for a design tool are: compatibility with the UML notation and the ability to design concurrent, event-driven software. The selected Quantum Platform (QP) framework and QP Modeler (QM), a graphical programming tool for generating C/C++ code from state machines [12], are compatible with the UML.

The QP framework allows building concurrent, event-driven software for embedded systems, with object-oriented notation. Objects are represented as state machines that communicate asynchronously via the exchange of events. State machines can be implemented as active objects, which are strictly encapsulated objects that reside on their own individual threads of execution. For example, an LED object could be represented as a state machine with two states, on and off. To toggle a light, an event associated with the TOGGLE signal is posted to it, causing the LED to transition either to the on or off state (Figure 2). In each state, the logic for manipulating the physical device to turn on or off would be implemented, so that changes in software are reflected by the corresponding hardware.

QP Modeler is a graphical programming tool that can be used generate C/C++ code from UML state machines. This means that complex, event-driven systems can be designed in UML, and code is directly generated from the model. The code generated relies on the QP Framework (QF) for handling events and maintaining state machines.

3.2 Target Platform and Remote Device

To prepare the remote system for proper use by students, two components have to be devised accordingly: the target platform and an external device connected to it. This section describes respective developments.

Target Platform. The target platform selected for this project is Raspberry Pi, a very inexpensive credit-card size computer from the non-profit Raspberry Pi Foundation [13]. It can be purchased for $35 and has a number of useful features for an embedded system, including eight general-purpose input/output (GPIO) pins, with support for I²C, SPI, and UART protocols, and two USB ports. These can be used to communicate with and control a number of external devices (Figure 3).

![Fig. 3. Details of the Raspberry Pi board [14].](image)

![Fig. 2. A state machine for an LED controller.](image)
The GPIO pins can be easily manipulated in C code in a simple way using a community library called WiringPi [15]. This library includes functions for the manipulation of digital logic pins, including the single pulse width modulation (PWM) pin available on the Raspberry Pi. The library can be built to support SPI and I²C communication. The library uses a direct memory mapping system call, `mmap()`, to manipulate pin registers, so programs that use WiringPi must be run as root.

The Raspberry Pi has support for a few different operating systems: GNU/Linux, RISC OS, Plan 9, and even Android. The official distribution is a version of Debian Linux adapted for the Pi, called Raspbian. It provides a user-friendly environment with the base installation.

Arch Linux, on the other hand, provides a streamlined, bare-bones Linux installation meant to be fully customizable to users. This seemed like a good choice for a basic embedded system. One major limitation of the Raspberry Pi is the lack of a real-time clock, meaning it is not immediately suitable for hard real-time applications; however, a real-time clock can be added on fairly easily [16].

**Remote Device.** The device controlled by the Raspberry Pi is a very simplistic remote controlled car. It can move forward and backward with variable speed. The components for the car include a USB Wi-Fi dongle for Internet connectivity, a USB webcam for vision, a 12 V battery with a 5 V regulator, a motor controller for controlling the speed and direction of the car, motors, wheels, mounting hardware and a chassis.

The car shown in Figure 4 uses the Gertboard [17], which has an embedded motor controller. The motor controller is being used to control the direction using a logic pin and speed of the car using pulse width modulation (PWM). Pulse width modulation is a way to control the amount of power going to a device by rapidly alternating a signal between the high and low states. The percentage of time a signal is high corresponds to the amount of power sent to a device, which is called the duty cycle.

![Fig. 4. Pi Rover Prototype.](image)

Figure 4 show device connections, that is, onboard devices, such as batteries, motors and LEDs, and USB-connected Wi-Fi adapter and webcam (right).

### 3.3 Internet Connectivity

The remote device has been designed to receive commands from a single operator over a TCP socket. These commands are interpreted by the software running on the target (Raspberry Pi) and translated into control messages by branching and manipulating a hardware device (i.e., a motor).

The target allows for control of external devices over the Internet from a website control panel. A new feature in HTML5 called WebSockets
allows for connection-based sockets from within a webpage. The implications of this feature in the context of this project are that control panels can be easily built using a combination of HTML and JavaScript. These control panels are platform independent, that is, as long as the operator is using a compatible browser, they could be on a Mac, Windows, Linux, etc., computer and have the same experience.

The remote target has to understand not only client requests to control its device, for which it has to run an HTTP server, but also requests from the host for software upload and debugging. The Raspberry Pi is running on Arch Linux, and has OpenSSH installed, allowing for the remote access of the Raspberry Pi via a secure shell (SSH). This section described briefly how these facilities are built into the target.

**Configuring Remote Access.** In order to SSH into the Raspberry Pi, the Pi’s IP address must be known. Most computers reside on a local area network (LAN) and are connected to the Internet through a router. For large networks, like those at institutions, a network administrator must configure the network to provide outside access to computers within the LAN. The following section will explain how to configure a basic home network to provide outside access to a computer within the LAN.

Configuring the Pi with a static IP address is recommended so the address does not change. This is done on the Raspberry Pi within Linux. Once the device has been configured with a static IP address, the network hardware needs to forward SSH requests to the Raspberry Pi on the local network. SSH requests are on port 22 by default, although this can be changed on the Raspberry Pi to be any unused port. Figure 5 shows the port forwarding page on a DD-WRT based router control panel.

![Port Forward](image)

Once these configurations are completed, it is possible to connect to the Raspberry Pi remotely using the router’s wide area network (WAN) IP address. On the host computer, a terminal is opened and the following command is entered, substituting the values depending on the devices configuration.

```bash
ssh <username>@<IP address>:<port>
```

`<username>` should be replaced with user being logged in on the Raspberry Pi. If the port value is left out, port 22 is used by default.

Connecting to the Raspberry Pi via SSH is the equivalent of connecting to the device directly with a keyboard and opening a terminal.

Another useful program is `scp` (secure copy), which uses an SSH connection to transfer a file. This will be used for uploading executables to the device. The syntax is as follows:

```bash
scp /path/to/local/file <username>@<IP address>:/path/to/destination/file
```

The password for the user must then be entered. The user must have write permissions for the destination of the file. If the port for SSH has been changed, it must be specified for `scp` using the `-P` option.

**Building the Target Server.** Before creating a complex program with many features and extensibility, a prototype is often created to gain a better understanding of components for a
larger system. Listing 1 in the Appendix is a basic WebSocket server prototype that establishes a TCP WebSocket with a remote client and reads strings of text from that client, and performs different actions based on the content of the string. This is the basis of the upcoming, more complex software, but narrows down the scope to gain a better understanding of sockets, and more specifically WebSockets.

WebSocketServer.cpp shows how a WebSocket handshake is done, and how simple control messages can be sent over the Internet to the Raspberry Pi and used to manipulate hardware devices. In this case, sending the string “ON” turns an LED connected to pin 0 on. Sending the string “OFF” turns the LED off. The configuration of the LED can be seen in Figure XXX.

To connect to this WebSocket server, start the server, then start a modern web browser with WebSocket support (e.g., Chrome) and open the developer console. Create a new WebSocket object and send the string “ON” (Listing 2).

Listing 2:

```javascript
var ws = new WebSocket("ws://<IP address>");
ws.send("ON");
```

The light should turn on. Sending “OFF” should turn the LED back off. This prototype shows a simple, but effective solution to controlling hardware over the Internet from a web browser.

The socket server created above is blocking, meaning that it blocks the thread from doing anything until it returns from the either the accept() or recv() function. This does not work with QP’s event-driven model because everything must run to completion. What this means is if an active object is doing some processing, say, waiting to read data from a client, it cannot process other events.

The accept() function blocks while waiting to accept a new client. In QP, this will eventually raise an exception. This can be remedied by using signals from the Linux kernel to accept a new client. On the listening socket, a SIGIO signal is tied to an action (function) that posts an AcceptEvt to the server, telling it to try to accept a new client. Listing 3 shows how this is done in C code.

Listing 3:

```c
void connect_handler(int signum) {
    std::cout << "A client is trying to connect...
    AcceptEvt *ae = Q_NEW(AcceptEvt,
    ACCEPT_SIG);
    AO_WebSocketServer->postFIFO(ae);
}
... 
if (bind(socket_fd_, (sockaddr*)
&server_addr_,
    sizeof(server_addr_)) < 0)
{
    perror("bind()");
    exit(EXIT_FAILURE);
}

// process must own file to receive
// signals from it
if (fcntl(socket_fd_, F_SETOWN,
    getpid()) < 0)
{
    perror("F_SETOWN");
    exit(EXIT_FAILURE);
}

// set socket as non-blocking/async
if (fcntl(socket_fd_, F_SETFL,
    O_NONBLOCK | FASYNC) < 0)
{
    perror("F_SETFL");
    exit(EXIT_FAILURE);
}
```
struct sigaction connect_action;
std::memset(&connect_action, 0,
    sizeof(connect_action));
connect_action.sa_handler =
    connect_handler;
sigemptyset(&connect_action.sa_mask);
connect_action.sa_flags = 0;
if (sigaction(SIGIO, &connect_action,
    NULL) < 0)
{
    perror("sigaction()");
    exit(EXIT_FAILURE);
}

Fortunately, the Linux kernel also raises a signal whenever there are data available to be read on a socket (among other things), called SIGIO. This signal can be used like a software interrupt with an associated action (like an interrupt service routine). The Linux system call for assigning an action to a signal is called sigaction(). Listing 4 shows an action to be called whenever there is a SIGIO on the client socket (meaning data is ready to be read). The function posts a RecvEvt to the WebSocketServer object for asynchronous receiving of data from a socket.

Listing 4:
std::cout << "SIGIO on client socket"
<< std::endl;
// assume data is on client socket
RecvEvt *re = Q_NEW(RecvEvt,
    RECV_SIG);
AO_WebSocketServer->postFIFO(re);

Now, the server does not need to block. It only calls accept() and recv() when a client is trying to connect, or there is data on the socket.

Data sent over a WebSocket are character strings by default. Actually, at this time Chrome and Firefox do not support binary large objects (blobs) over WebSockets. JavaScript objects can be easily turned into strings in a data format called JavaScript Object Notation (JSON). Using JSON for control messages makes it easy to form control messages on the client side. There also exists several C/C++ libraries for parsing JSON strings. JSON supports the use of strings, characters, integers, floating-point number, and Booleans.

4 APPLICATION DEVELOPMENT FOR REMOTE DEVICE

After the station has been built, appropriately programmed and prepared for operation, it is ready for use in a remote lab for cyberphysical systems, so applications can be built and uploaded by students. Such applications have to allow for control of an external device (a remote controlled car) over the Internet from a web browser. In respective projects, students are expected to acquire three essential skills for cyberphysical systems development: software design, software implementation and testing on a remote target. This section addresses respective issues.

4.1 Design with QP and QM on the Host

The previous section provided a simple solution for controlling devices, but not all devices are for control. Some, like sensors and probes, must be read from periodically. The previous solution would not be suitable for this without creating a new thread of execution because the socket.read() function is blocking. This means that the program is hung up until it read something from the socket. Anything that needs to be done must be done in a separate thread, or the socket must be set up to be non-blocking.
Using QP, a multi-threaded solution is implemented. However, there is no need to deal with threads directly. Instead, active objects modeled as state machines are created and each resides on their own thread of execution. Active objects communicate asynchronously, so there are no “blocking” methods. To invoke a method on another object, one object simply posts an event to another. Data can be passed along with an event, but usually pointers to data are passed instead, following the “zero-copy” rule.

The most interesting feature that QP has to offer is the QP Modeled. QM allows for the simultaneous design and implementation by create UML statecharts and generating C/C++ code from them. Often, project designs will not be strictly adhered to during the implementation, but with QP modeler, the design is the implementation, so they will always be in sync.

Object-oriented programming is particularly useful with physical computing because real world devices can be represented as software objects, where the manipulation of the software object results in the same manipulation of the hardware.

Another benefit of using QP for real-world programming is the event-driven design. The real world is chaotic and unpredictable. Conventional programs are linear and deterministic. With QP, real-world events, such as the pressing of a button or the sound of an alarm, can be directly translated into software events in the QP framework, (i.e., QEvent). This makes the design and implementation of reactive systems more concise with the aid of the QM graphical programming tool.

Software is developed and debugged remotely on a host computer running Debian Linux using a GNU cross-compilation toolchain.

The remote development of software for controlling physical devices connected to the Raspberry Pi utilizes a cross compiler, which allows for the development and compilation of software on a machine with a different architecture from the Raspberry Pi. While it is possible to program and compile code using a Raspberry Pi, using a cross compiler allows for much faster compile times, as well as keeping extra software off of the Raspberry Pi. The cross compiler is created using Crosstool-NG [18], a free program that customizes and builds cross compilation toolchains.

Crosstool-NG is a tool for building cross compilation toolchains. It is possible to build a toolchain from scratch, but there are many components and dependencies, so things can easily go wrong. A toolchain usually includes a compiler, linker, assembler, standard libraries (e.g., GNU C library, glibc), as well as a cross debugger. The following paragraphs detail how to set up Crosstool-NG and build a toolchain for remote Raspberry Pi development.

Since cross-compilation is an essential part of developing remote application, the process of installing Crosstool-NG is described in the Appendix.

4.3 Remote Debugging

With a cross compilation toolchain, it is possible to compile and debug native software to run on a remote device. A simple “hello world” program can be used to ensure everything is working. Create a new directory and make two new files called hello.c and Makefile, as
shown below.

```c
// hello.c
#include <stdio.h>
int main()
{
    printf("Hello, world.\n");
}
```

```text
# hello Makefile
BIN=hello
CC=rpi-gcc
CFLAG= -g –Wall -03
OBJ=
$(BIN) : $(OBJ)
```

Now, type make. This will read the Makefile and the code will compile to a single executable to run on the Raspberry Pi. The executable has debugging symbols for the debugger. Upload the executable using scp:

```bash
scp ./hello <username>@<IP address>:/path/to/destination/file/hello
```

The executable will be uploaded to the Raspberry Pi so it can be debugged. SSH into the Pi and run the executable. If it runs, the cross compiler has been successfully installed.

Start the GDB server on port 1234 with the uploaded hello executable.

```bash
gdbserver :1234 /path/to/hello
```

If gdbserver is not installed, it can be obtained using the Arch Linux package manager, pacman.

```bash
pacman -S gdb
```

A debugging session should now be active with the hello executable. It should look something like Figure XXX. Create a breakpoint at main, and then run the program.

```gdb
(gdb) b main
(gdb) r
```

5 CONCLUSION

The objective of this paper was to show what is involved in building a remote lab station for software development in cyberphysical systems courses. The essential work is to develop a remote target computer running a server, with an external device connected to it, so students could use Internet protocols to connect to this target for uploading and debugging the control software. After the target has been properly configured and started, tools on the host machine can be used to develop software that would be uploaded to control the remote device from any Internet client.

6 ACKNOWLEDGMENTS

Student of the FGCU software engineering program, Rabeet Fatmi, if gratefully acknowledged for his contribution to programming of the remote car station.

7 REFERENCES


17. Gertboard – IO Expansion Board for RPi. URL: https://www.sparkfun.com/products/11773


APPENDIX. Crosstool-NG Installation

Once Crosstool-NG has been downloaded on the host computer, it must be built. Before building, obtain all dependencies using aptitude by typing the following command into a terminal.

```
sudo apt-get install flex bison
curses-dev texinfo build-essential
gperf patch libtool automake gawk
subversion python-dev libexpat1-dev
```

After installing all of the dependencies, the usual

```
configure
make
sudo make install
```

will build and install the program.

Now that Crosstool-NG is installed, it must be configured to build a toolchain for the ARM architecture. Start the configuration menu by typing

```
ct-ng menuconfig
```

into the terminal. In “Paths and Misc options”, check “Try features marked as EXPERIMENTAL”. The prefix directory tells
Crosstool-NG where to place the toolchain. This can be left at the default value. Changing the number of parallel jobs will result in faster build times for multi-core computers.

In the “Target options” menu, change the target architecture to ARM. There are several options to change the toolchain’s tuple, a list that specifies the target architecture, vendor, operating system, and embedded application binary interface (EABI) for compiled executables. The vendor string can be left out, but setting it to bcm (Broadcom) makes it clear which chip the compiler is for. This menu also has an option for creating an alias tuple, which creates symbolic links to the longer tuple executables. Setting this to rpi (for Raspberry Pi) is advisable and is used in this paper. There is an option to append “hf” to the end of the tuple, which shows support for floating point hardware. This option is purely aesthetic, but the Raspberry Pi does have floating point hardware, so it may be selected.

In the Toolchain options” menu, set the Tuple’s In the “Operating System” menu, set Linux as the target OS.

In the “Binutils” menu, select the latest version of binutils that is not experimental.

Under the “C compiler” menu, select “Show Linaro version” and choose the latest version. Linaro is an organization that creates optimized tools for the ARM architecture [http://www.linaro.org/]. There are options for compilers for other languages. Select the C++ option.

In the “C library” menu, ensure that eglibc is selected. This is a standard C library optimized for embedded systems.

Finally, in order to debug the device remotely, a cross debugger is needed. In the “Debug facilities” menu, select gdb, the GNU Debugger. Exit the configuration menu and save the profile for later use. Build the cross compiler by typing ct-ng build into the terminal. This process takes quite some time, even on fast computers, as there are thousands of files to be downloaded, processed, and compiled.
APPENDIX. Full Code Listings

Listing 1

WebSocketServer.h:

```cpp
#ifndef WEBSOCKETSERVER_H
#define WEBSOCKETSERVER_H

/**
 * An implementation of the HTML5 WebSocket protocol compliant
 * to RFC 6455.
 * Author : Jim Carroll
 * Last modified on 2013-5-27
 */
#endif
```

```
#include <string>
#include <cstring>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

/* DEFINITIONS */
#define HTTP_PORT 80  // WebSockets are created via HTTP
#define BUF_SIZE  1024
#define GUID "258EAFA5-E914-47DA-95CA-C5AB0DC85B11"

class WebSocketServer
{
public:
    WebSocketServer(int port = 80);
    ~WebSocketServer();
    bool accept();
    bool send(std::string data, int length);
    bool connected() const;
    char * receive();

private:
    int         socket_fd;
    int         client_fd;
    sockaddr_in server_addr;
    sockaddr_in client_addr;
    bool        connected_;
    char * base64(const unsigned char * input, int length);
    void        init(int port);
    bool        handshake();
    char * receive_header();
    void        close();
    char * process_frame(const char * frame, int n);
    void        apply_mask(char * data, int length, char * mask);
};
```
#include "WebSocketServer.h"
#include <climits>
#include <ctime>
#include <iostream>
#include <iomanip>
#include <istream>
#include <ostream>
#include <iterator>
#include <sstream>
#include <string>
#include <vector>
#include <cassert>
#include <unistd.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <openssl/sha.h>
#include <openssl/hmac.h>
#include <openssl/evp.h>
#include <openssl/bio.h>
#include <openssl/buffer.h>

/* GPIO STUFF */

// Access from ARM Running Linux
#define BCM2708_PERI_BASE 0x20000000
#define GPIO_BASE (BCM2708_PERI_BASE + 0x200000) /* GPIO controller */
#define PAGE_SIZE (4*1024)
#define BLOCK_SIZE (4*1024)

int mem_fd;
void *gpio_map;

// I/O access
volatile unsigned *gpio;
// GPIO setup macros. Always use INP_GPIO(x) before using OUT_GPIO(x) or
// SET_GPIO_ALT(x,y)
define INP_GPIO(g) *(gpio+((g)/10)) &= ~(7<<(int)((g)%10)*3)
define OUT_GPIO(g) *(gpio+((g)/10)) |= (1<<(int)((g)%10)*3)
define SET_GPIO_ALT(g,a) *(gpio+((g)/10))) |=
(((a)<=3? (a)+4: (a)==4?3:2)<<((int)(g)%10)*3))
define GPIO_SET *(gpio+7)  // sets bits which are 1 ignores bits which are 0
#define GPIO_CLR *(gpio+10) // clears bits which are 1 ignores bits which are 0
#define LED_PIN 7
/***************************************************************************/
void setup_io();
WebSocketServer::WebSocketServer(int port)
{  
  init(port);
}
WebSocketServer::~WebSocketServer()
{  
  // nothing to delete... yet
  // other cleanup?
}
bool WebSocketServer::accept()
{  
  /* wait for a client to connect -- blocks while waiting */
  socklen_t client_len = sizeof(client_addr);
  client_fd = ::accept(socket_fd, (sockaddr *) &client_addr, &client_len);
  if (client_fd < 0)
  {
    std::cerr << "ERROR: Could not accept new client." << std::endl;
    return false;
  }

  std::cout << "Sending handshake...";
  if (handshake())
  {
    connected_ = true;
    return connected_;
  }
  return false;
}
// frames the data per section 5.2
bool WebSocketServer::send(std::string data, int length)
{  
  int packet_size = 2 + length;
  char * packet = new char[packet_size];

/ start setting up frame
std::memset(packet, 0, packet_size);

/* FIN and opcode */
packet[0] = 0x81; // FIN = 1, opcode = 1 (text frame)

/* packet length */
packet[1] = (char) length;
packet[1] &= 0x7f; // turn off mask bit
std::strcpy(packet + 2, data.c_str());

int n = ::send(client_fd, packet, packet_size, 0);
std::cout << "Sent " << n << " bytes to client: " << (char *)(packet+2) <<< std::endl;
delete packet;
return true;

}/**
 */
char * WebSocketServer::receive()
{
    char * buffer = new char[BUF_SIZE];
    std::memset(buffer, 0, BUF_SIZE);

    int n = recv(client_fd, buffer, BUF_SIZE - 1, 0);
    if (n == -1) {
        std::cout << "***ERROR RECEIVING***" << std::endl;
        exit(EXIT_FAILURE);
    } else if (n == 0) {
        std::cout << "recv() returned a zero." << std::endl;
        close();
    } else {
        std::cout << "Read a packet. " << n << " bytes." << std::endl <<
                   "Frame:" << std::endl;
        /* draw line */
        std::cout << std::setw(20) << std::setfill('=') << "" << std::endl;
        /* reset fill */
        std::cout << std::setfill(' ');
    }

    char * data = process_frame(buffer, n);

    return data;
}

bool WebSocketServer::connected() const
{
    return connected_;
}
char * WebSocketServer::base64(const unsigned char * input, int length)
{
    BIO *bmem, *b64;
    BUF_MEM *bptr;
    b64 = BIO_new(BIO_f_base64());
    bmem = BIO_new(BIO_s_mem());
    b64 = BIO_push(b64, bmem);
    BIO_write(b64, input, length);
    BIO_flush(b64);
    BIO_get_mem_ptr(b64, &bptr);
    char * buff = (char *)malloc(bptr->length+1);
    memcpy(buff, bptr->data, bptr->length);
    buff[bptr->length] = 0;
    BIO_free_all(b64);
    return buff;
}

void WebSocketServer::init(int port)
{
    /* create an internet TCP socket with default protocol */
    socket_fd = socket(AF_INET, SOCK_STREAM, 0);
    /* zero out address structures */
    std::memset((char *) &server_addr, 0, sizeof(server_addr));
    /* initialize server_addr structure */
    server_addr.sin_family      = AF_INET;
    server_addr.sin_port        = htons(port);
    server_addr.sin_addr.s_addr = INADDR_ANY;
    /* attempt to bind to a port -- must be super user if port < 1024 */
    std::cout << "Binding...";
    if (bind(socket_fd, (sockaddr*) &server_addr, sizeof(server_addr)) < 0)
    {
        std::cerr << "ERROR: Could not bind to port " << port
                   << ". Try using sudo." << std::endl;
        exit(EXIT_FAILURE);
    }
    std::cout << "Success!" << std::endl;
    /* listen on socket with backlog queue length of 0 */
    listen(socket_fd, 0);
}

bool WebSocketServer::handshake()
{
    /* Now, we need to validate the WebSocket connection with client
     * per the RFC 6455.
     */

std::string header(receive_header());
std::cout << std::endl << "HTTP Header:" << std::endl << header << std::endl;

/* parse the header */

// FIXME All this does is find the Sec-WebSocket-Key value...

// this next part is definitely hacky and error-prone
int key_idx = header.find("Sec-WebSocket-Key:") + 19;
std::string str_key = header.substr(key_idx, 24);
std::cout << "Sec-WebSocket-Key: " << str_key << "END" << std::endl;

// sha-1 key+GUID
unsigned char hash[24];
std::memset(hash, 0, 24);
str_key.append(GUID);
SHA1(reinterpret_cast<const unsigned char*>(str_key.c_str()),
     str_key.length(), hash);
hash[20] = '\0';
char * accept_b64 = base64(hash, 20);
std::cout << "Sec-WebSocket-Accept: " << accept_b64 << std::endl;

// send back an Upgrade/Accept header
std::string accept_hdr(
    "HTTP/1.1 101 Switching Protocols\r\n",
    "Upgrade: websocket\r\n"
    "Connection: Upgrade\r\n"
    "Sec-WebSocket-Accept: " + std::string(accept_b64) + "\r\n");

// send accept header
if (::send(client_fd, accept_hdr.c_str(), accept_hdr.length(), 0))
{
    std::cout << "Sent accept header." << std::endl;
    return true;
}
else
{
    return false;
}
}

char * WebSocketServer::receive_header()
{
    char * buffer = new char[BUF_SIZE];
    std::memset(buffer, 0, BUF_SIZE);
    int n = recv(client_fd, buffer, BUF_SIZE - 1, 0);
    if (n < 0)
std::cerr << "ERROR: Could not obtain header." << std::endl;
// exit? exception?
}
std::cout << "Processing header..." << std::endl;
return buffer;
}

void WebSocketServer::close()
{
  // send status code 1002 (protocol error)
  // then clean up
  shutdown(client_fd, 2);
  shutdown(socket_fd, 2);
  connected_ = false;
}

char * WebSocketServer::process_frame(const char * frame, int n)
{
  bool fin  = frame[0] & 0x80; // bit 7
  bool rsv1 = frame[0] & 0x40; // bit 6
  bool rsv2 = frame[0] & 0x20; // bit 5
  bool rsv3 = frame[0] & 0x10; // bit 4
  int opc  = frame[0] & 0x0F; // bits 3, 2, 1, and 0
  bool is_masked = frame[1] & 1 << 7;
  long payload   = frame[1] & 0x7F; // b01111111

  int payload_offset = 2;
  if (payload == 126) // payload is a 16-bit integer
  {
    payload = *( (uint16_t *)(frame + 2));
    payload_offset += 2; // offset past extended payload length
  }
  else if (payload == 127) // payload is a 64-bit integer
  {
    payload = *( (uint64_t *)(frame + 2) );
    // This next line needs to ensure MSB is 0, and FAIL if it is not
    std::cout << "MSB of payload length is "
      << std::boolalpha << ( payload & (uint64_t)1<<63) << std::endl;
    payload_offset += 8;
  }
  char mask[4]; // mask is 4 octets
  if (is_masked)
  {
    mask[0] = frame[payload_offset];
    mask[1] = frame[payload_offset + 1];
    mask[2] = frame[payload_offset + 2];
    mask[3] = frame[payload_offset + 3];
    payload_offset += 4; // 4 bytes for mask
  }
std::cout << std::setw(10) << std::left << "Name" << std::setw(10) << std::left << "Value" << std::endl;
std::cout << std::setw(10) << std::left << "FIN" << std::setw(10) << fin << std::endl;
std::cout << std::setw(10) << std::left << "RSV1" << std::setw(10) << rsv1 << std::endl;
std::cout << std::setw(10) << std::left << "RSV2" << std::setw(10) << rsv2 << std::endl;
std::cout << std::setw(10) << std::left << "RSV3" << std::setw(10) << rsv3 << std::endl;
std::cout << std::setw(10) << std::left << "OPCODE" << std::setw(10) << std::hex << "0x" << std::hex << opc << std::endl;
std::cout << std::setw(10) << std::left << "Payload" << std::setw(10) << std::dec << payload << std::endl;
std::cout << std::setw(10) << std::left << "Is masked" << std::setw(10) << std::boolalpha << is_masked << std::endl;
if (is_masked) {
    std::cout << std::setw(10) << std::left << "Mask" << std::setw(10) << std::hex << "0x" << *( (uint32_t *) (mask)) << std::endl;
}
std::cout << std::endl << "OPCODE INFO: ";
switch(opc) // TODO #define all of these
{
    case 0x0: // this frame is a continuation
        std::cout << "Continuation frame."
        /* how do we handle continued data streams? */
        break;
    case 0x1: // denotes a text frame
        std::cout << "Text frame."
        break;
    case 0x2: // denotes a binary frame
        std::cout << "Binary frame."
        break;
    case 0x3:
    case 0x4:
    case 0x5:
    case 0x6:
    case 0x7: // reserved for further non-control frames
        std::cout << "Unknown (control frame)."
        break;
    case 0x8: // denotes a connection close
        std::cout << "Close frame."
        close();
        break;
    case 0x9: // denotes a ping
        std::cout << "Ping."
        break;
    case 0xA: // denotes a pong
        std::cout << "Ping."
        break;
    case 0xB:
        //...
case 0xF: // reserved for further control frames
    std::cout << "Unknown (control frame)." << std::endl;
    break;
}

std::cout << std::dec << std::endl << std::setw(20) << std::setfill('-') << "" << std::endl;
std::cout << std::setfill(' ');

char * data = (char *)(frame + payload_offset);
if (is_masked)
    apply_mask(data, payload, mask);

std::cout << std::dec << std::endl;
return data;
}

void WebSocketServer::apply_mask(char * data, int length, char * mask)
{
    // see section 5.3 for description of this algorithm
    for (int i = 0; i < length; ++i)
    {
        int j = i % 4;
        data[i] = data[i] xor mask[j];
    }
}

int main()
{
    setup_io();
    //************************************************************************
    // You are about to change the GPIO settings of your computer.          *
    // Mess this up and it will stop working!                               *
    // It might be a good idea to 'sync' before running this program       *
    // so at least you still have your code changes written to the SD-card! *
    //************************************************************************/
    INP_GPIO(LED_PIN); // must use INP_GPIO before we can use OUT_GPIO
    OUT_GPIO(LED_PIN);
    while(true)
    {
        WebSocketServer client; // default to port 80
        /* block and wait for client */
        client.accept();

        while (client.connected())
        {
            char * data = client.receive();
        }
    }
}
```
std::cout << "Received: " << std::endl << data << std::endl;

std::string str_data(data);
std::string response;

if (str_data == "name") {
    response.assign("Apollo Pi");
} else if (str_data == "time") {
    time_t rawtime;
    struct tm * timeinfo;
    std::time ( &rawtime);
    timeinfo = std::localtime ( &rawtime);
    response.assign( asctime(timeinfo) );
} else if (str_data == "hi") {
    response.assign("Hello!");
} else if (str_data == "ON") {
    GPIO_SET = 1<<7;
    response.assign("Light on.");
} else if (str_data == "OFF") {
    GPIO_CLR = 1<<7;
    response.assign("Light off.");
} else {
    response.assign("Huh?" );
}

std::cout << "Sending back: " << response << std::endl;
client.send(response, response.length());
```

```
if (gpio_map == MAP_FAILED) {
    printf("mmap error %d\n", (int)gpio_map);//errno also set!
    exit(-1);
}

// Always use volatile pointer!
volatile unsigned *)gpio_map;

} // setup_io