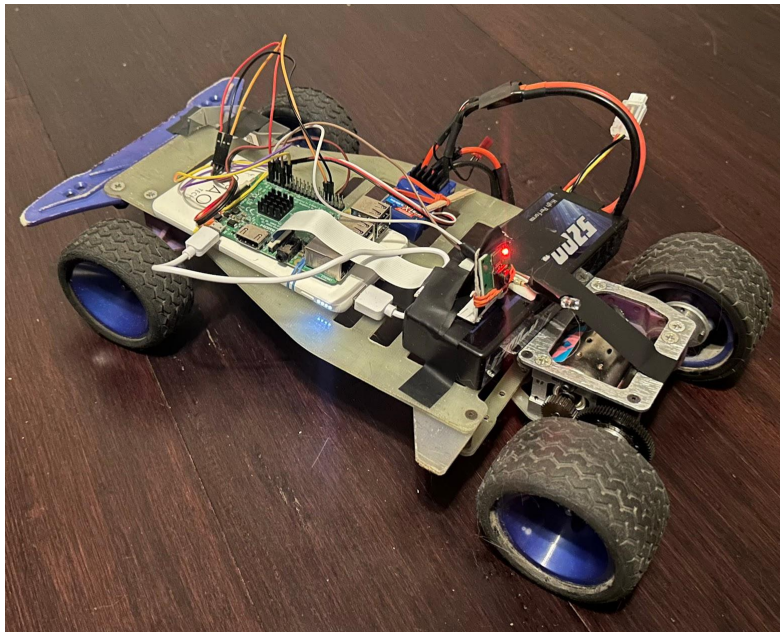


University of Nevada, Reno

CPE 400 Technical Report: Wi-Fi IoT RC Car



By
Michael Kerr
Kyle Knotek
Cole Renfro

Shamik Sengupta
CPE400
12/12/2022

Components and Software

Software

- Apache2 Server Package
- Apache PHP package
- Python 3
- Raspbian OS

Components

- Raspberry Pi 3 Model B
- Router with Port Forwarding Capabilities
- RC Car
- Jumper Wires
- [5V 2A Battery](#)
- [RC Car Battery](#)
- [Pi Camera](#)
- [Motor Controller](#)

Functionality

The method by which this RC Car works is best explained by starting at the web server itself. The web server is running on a Raspberry Pi 3 Model B running the Raspbian OS. The webserver itself is an Apache2 web server, with PHP packages installed in order to allow the webserver to handle requests made client-side. These PHP packages run python scripts, which tell the RC Car components to either turn an LED on or off, or drive the car forward, left, and right. A smaller server running on port 8000 has a video feed associated with it, allowing the HTML page to display a live stream to the user.

This method also requires access to a router in order to set up port forwarding to the web server itself over port 80.

The RC Car itself is directly connected to the board through the GPIO on the Raspberry Pi. The GPIO is connected directly to a servo, motor controller, and an LED. This allows the client to give directions to the web server, which issues directions to a python script, which gives electrical signals to the servo, motor controller, or LED depending on the python script that is run.

Further information regarding the setup process for the Wi-Fi IoT RC Car can be found in the README.md which is within the .zip file that has been sent for grading, or at GitHub.¹

This video of the Car demonstrates the functionality of the UI and car respectively:
<https://www.youtube.com/watch?v=iqjk46UkJUQ>.

¹ <https://github.com/mickeykerr/CPE400-Semester-Project-Server-Car>

User Flow

So, should a user navigate to the public IP that the car is connected to, they will be greeted by the following webpage.

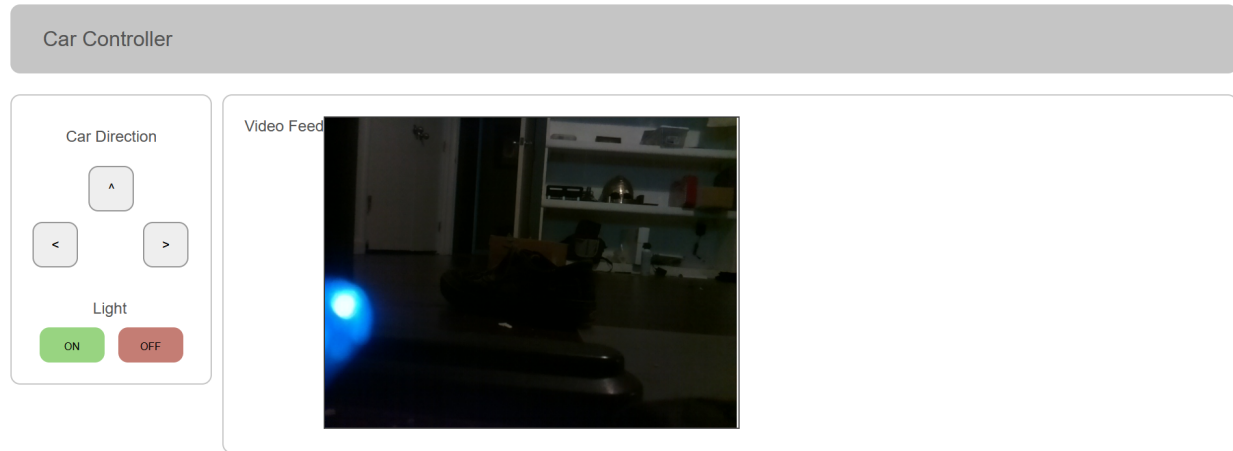


Figure 1: Webpage

The front end was created using HTTP, CSS and Javascript to implement a UI for controlling the RC car. The http code was then translated to PHP to be able to be used by the Apache server. This webpage is accessible through any web browser from the public IP address the server is running on.

This webpage has five buttons, allowing the user to go forward, left, and right, as well as turn an LED on and off. Each of these buttons make HTTP POST requests to the web server. PHP picks up these HTTP POST requests and then runs relevant python scripts held in the "scripts" folder.

Should a directional button be pressed, these python scripts will tell the Raspberry Pi to enable PWM for the GPIO pins. These GPIO pins on the RPi running PWM tell the RC Car to actuate the onboard motor and servos using PWM.

Should the Light On or Off button be pressed, a similar python script will tell the LED GPIO output to turn on or off depending on the argument passed to them. This can be further seen in the /scripts/led.py python script.

Car Setup

Below are two figures of the actual RC Car. Visible Components are listed in both the top and front views of the RC Car.

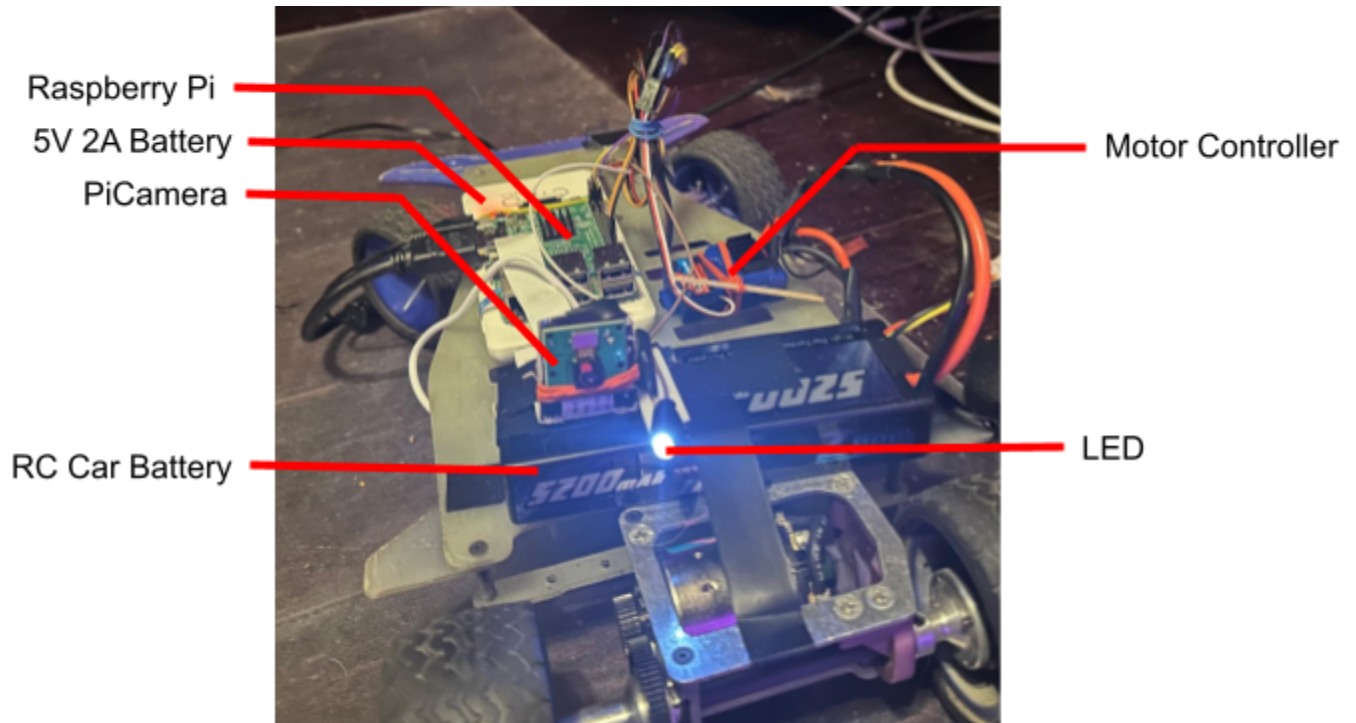


Figure 2: Front View of RC Car

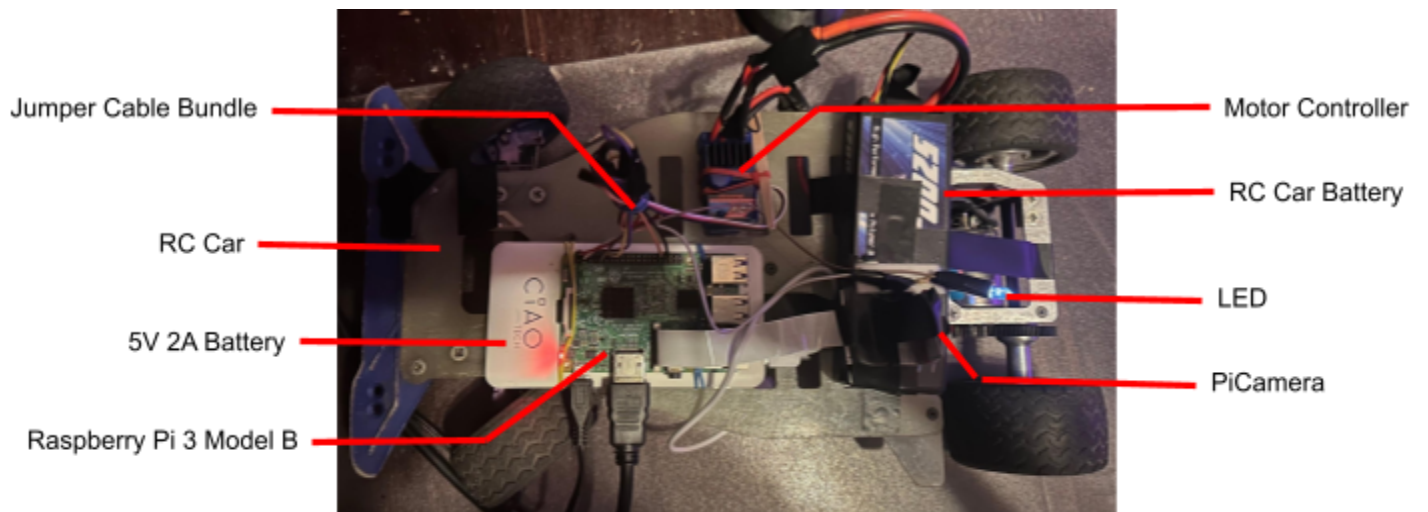


Figure 3: Top View of RC Car

Novelty of the Subject Matter

The creation of this RC car has given us a new perspective on live services using the internet. Since it incorporated a UI, a server and the hardware itself, it truly reveals how much thought has to go into building and maintaining online services. The HTTP, PHP, CSS and Javascript code all contribute to the UI and instructions for the Raspberry Pi to run. All of these components come together to create a larger system that handles human interaction. The Apache server is what handles the port forwarding and networking.

The combination of all of these components is what creates a true IoT-like device. The creation of such a device can lead to other IoT applications. In this case, we have taken networking and applied it to a moving component in the physical world. The same principles can be applied to other vehicular control systems. Delivery drones, self-driving cars, remote-controlled inspection drones, and other robotic applications just to name a few.



Figure 4: A real vehicle being controlled remotely by a team in Virginia. Source: wtop.com

In Fig. 4, we can see this method applied on a larger scale conducted by Phantom Auto. Here, a full-sized car is being controlled remotely and is communicating through a cellular network. This technology is already gaining traction and can lead to massive improvements in traffic flow, efficiency of transportation and even driving safety.

A level of autonomy can be applied as well. For systems that are too complex to be placed locally onto the device itself, communication with the AI through remote means can be a possible solution for that issue. This can also be used as a way to create a database that controls many autonomous devices at once. This can be an efficient way to provide artificial

intelligence to small, simple devices that would not be able to perform those calculations on their own.

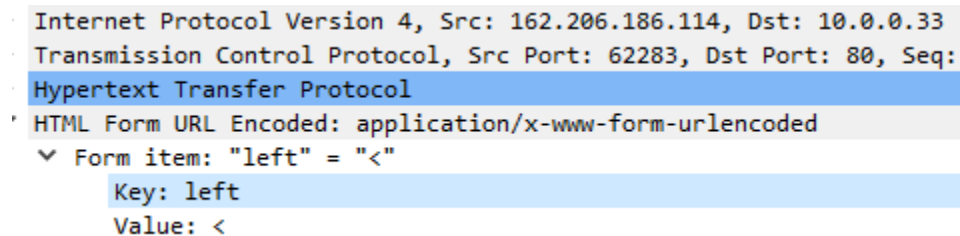
Results

Analysis

External Packet Capture

We recorded 3 separate packet captures using TCP and HTTP to access the car's server from any computer that has access to the internet.

Externalpacketcapture.pcapng is our capture when accessing the car from a computer not on the same LAN as the car. The first http capture represents the user clicking the left arrow and thereby submitting an HTML form with an HTTP POST request of "left" to the router, which is forwarded to the internal ip and port the web server for the car is on. The source is the user computer (client) and the destination being the server/pi attached to the car.



The image shows a screenshot of the Wireshark packet capture details pane. The selected packet is an HTTP POST request. The details are as follows:

- Internet Protocol Version 4, Src: 162.206.186.114, Dst: 10.0.0.33
- Transmission Control Protocol, Src Port: 62283, Dst Port: 80, Seq: [redacted]
- Hypertext Transfer Protocol
- HTML Form URL Encoded: application/x-www-form-urlencoded
 - Form item: "left" = "<"
 - Key: left
 - Value: <

Figure 5: Packet Capture from External IP making an HTTP POST request.

The next HTTP/1.1 capture represents the car's server returning an HTML form. Some of the code in HTML is included in this next figure.


```

> Internet Protocol Version 4, Src: 10.0.0.33, Dst: 162.206.186.114
> Transmission Control Protocol, Src Port: 80, Dst Port: 62283, Seq: 1, Ack: 591, Len: 976
> Hypertext Transfer Protocol
▼ Line-based text data: text/html (92 lines)
  <!DOCTYPE html>\n
  <!-- \t\n
  \tFrontend Done by Kyle Knotek. \n
  \tPHP scripting done by Michael Kerr.\n
  \t2022 \n
  -->\n
  \n
  <html lang="en">\n
  <head>\n
    <meta charset="UTF-8">\n
    <meta http-equiv="X-UA-Compatible" content="IE=edge">\n
    <meta name="viewport" content="width=device-width, initial-scale=1.0">\n
    <title>Controller</title>\n
    \n
    <link rel="stylesheet" href="main.css">\n
    \n
  </head>\n
  \n
  <body>\n

```

Figure 6: HTTP/1.1 data capture with HTML form data.

The third HTTP capture is an HTTP GET which returns the current image stream from the pi's camera capture. This image stream is hosted on port 8000, and is embedded in the HTML form, meaning an HTTP GET request must be made to the server to see the image stream from the remote client computer.

```

> Internet Protocol Version 4, Src: 162.206.186.114, Dst: 10.0.0.33
> Transmission Control Protocol, Src Port: 62284, Dst Port: 8000, Seq: 1, Ack: 1, Len: 474
> Hypertext Transfer Protocol
  > GET /stream.mjpg HTTP/1.1\r\n
  Host: 71.93.65.129:8000\r\n
  Connection: keep-alive\r\n
  Upgrade-Insecure-Requests: 1\r\n
  User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/91.0.4472.164 Safari/537.36\r\n
  Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8\r\n
  Referer: http://71.93.65.129/\r\n
  Accept-Encoding: gzip, deflate\r\n

```

Figure 7: HTTP GET request for the image stream made by remote client

This same set of HTTP sequence can be seen each time the user presses either the forward, left or right keys.

RC Car Movement Isolated Local/Internal Packet Capture

Movepacketcapture.pcapng is our capture of the car's server getting instructions from a server on the same LAN, with the code for the image stream being turned off. This is to isolate the actual instructions being made over the network. Using HTTP, the code to decipher the car's instructions are accessed by the server at 10.0.0.33. These instructions are sent locally from a

client at 10.0.0.16. Then when the client sends another HTTP POST request next command and sends it to the car's server, similar to how it was in the previous capture.

```
> Transmission Control Protocol, Src Port: 60401, Dst Port: 80, Seq: 1, Ack: 1, Len: 475
> Hypertext Transfer Protocol
  HTML Form URL Encoded: application/x-www-form-urlencoded
    > Form item: "right" = ">"
```

Figure 8: Another HTTP POST request to the server

RC Car Livestream Isolated Local/Internal Packet Capture

Recordpacketcapture.pcapng is our capture while recording through the pi's camera over a LAN. So like the first HTTP capture file, there is an HTTP POST followed by a HTTP/1.1 then an HTTP GET though this is done on a local network and the image is sent to the computer the user is on, 71.93.65.129 instead of 10.0.0.16. The HTTP POST request is the actual POST direction from the client telling the car to move, with the HTTP/1.1 being the actual web page being accessed. The GET commands are to GET the stream data going over port 8000. This stream data is embedded in the webpage using HTML, which is why the HTTP GET request must be made.

9865	23.907440348	10.0.0.16	10.0.0.33	HTTP	529 POST / HTTP/1.1 (application/x-www-form-urlencoded)
9885	23.934072081	10.0.0.33	10.0.0.16	HTTP	1030 HTTP/1.1 200 OK (text/html)
9955	24.138022385	10.0.0.16	10.0.0.33	HTTP	346 GET /favicon.ico HTTP/1.1
9959	24.138826838	10.0.0.33	10.0.0.16	HTTP	545 HTTP/1.1 404 Not Found (text/html)
9962	24.171549465	71.93.65.129	10.0.0.33	HTTP	444 GET /stream.mjpg HTTP/1.1

Figure 9: View of the HTTP packets in wireshark

Below is an example of the TCP process that is being utilized for the image stream. From this wireshark data, and utilizing the knowledge we have about which port the image stream is located on, (port 8000), we know that the information from this Wireshark is .mjpg data that is being sent out. Therefore, we know that the image stream is utilizing TCP in order to send out livestream data.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	71.93.65.129	10.0.0.33	TCP	54	57015 → 8000 [ACK] Seq=1 Ack=1 Win=1026 Len=0
2	0.000245351	10.0.0.33	71.93.65.129	TCP	1514	8000 → 57015 [ACK] Seq=36501 Ack=1 Win=501 Len=1460
3	0.000281547	10.0.0.33	71.93.65.129	TCP	1514	8000 → 57015 [PSH, ACK] Seq=37961 Ack=1 Win=501 Len=1460
4	0.000501795	10.0.0.33	71.93.65.129	TCP	1514	8000 → 57015 [ACK] Seq=39421 Ack=1 Win=501 Len=1460
5	0.000512524	10.0.0.33	71.93.65.129	TCP	1514	8000 → 57015 [PSH, ACK] Seq=40881 Ack=1 Win=501 Len=1460
6	0.000612726	71.93.65.129	10.0.0.33	TCP	54	57015 → 8000 [ACK] Seq=1 Ack=5841 Win=1026 Len=0
7	0.000711523	71.93.65.129	10.0.0.33	TCP	54	57015 → 8000 [ACK] Seq=1 Ack=8761 Win=1026 Len=0
8	0.001281492	10.0.0.33	71.93.65.129	TCP	1514	8000 → 57015 [ACK] Seq=42341 Ack=1 Win=501 Len=1460
9	0.001306438	10.0.0.33	71.93.65.129	TCP	1514	8000 → 57015 [PSH, ACK] Seq=43801 Ack=1 Win=501 Len=1460
10	0.001345082	10.0.0.33	71.93.65.129	TCP	1514	8000 → 57015 [ACK] Seq=45261 Ack=1 Win=501 Len=1460
11	0.001366227	10.0.0.33	71.93.65.129	TCP	1514	8000 → 57015 [PSH, ACK] Seq=46721 Ack=1 Win=501 Len=1460
12	0.005559722	71.93.65.129	10.0.0.33	TCP	54	57015 → 8000 [ACK] Seq=1 Ack=11681 Win=1026 Len=0
13	0.005721796	10.0.0.33	71.93.65.129	TCP	1514	8000 → 57015 [ACK] Seq=48181 Ack=1 Win=501 Len=1460
14	0.005744972	10.0.0.33	71.93.65.129	TCP	1514	8000 → 57015 [PSH, ACK] Seq=49641 Ack=1 Win=501 Len=1460
15	0.005855903	71.93.65.129	10.0.0.33	TCP	54	57015 → 8000 [ACK] Seq=1 Ack=14601 Win=1026 Len=0
16	0.005892151	10.0.0.33	71.93.65.129	TCP	1514	8000 → 57015 [ACK] Seq=51101 Ack=1 Win=501 Len=1460
17	0.005899755	10.0.0.33	71.93.65.129	TCP	1514	8000 → 57015 [PSH, ACK] Seq=52561 Ack=1 Win=501 Len=1460
18	0.005987875	71.93.65.129	10.0.0.33	TCP	54	57015 → 8000 [ACK] Seq=1 Ack=17521 Win=1026 Len=0
19	0.006078704	71.93.65.129	10.0.0.33	TCP	54	57015 → 8000 [ACK] Seq=1 Ack=20441 Win=1026 Len=0
20	0.006165574	71.93.65.129	10.0.0.33	TCP	54	57015 → 8000 [ACK] Seq=1 Ack=23361 Win=1026 Len=0
21	0.006760958	10.0.0.33	71.93.65.129	TCP	1514	8000 → 57015 [ACK] Seq=54021 Ack=1 Win=501 Len=1460
22	0.006770176	10.0.0.33	71.93.65.129	TCP	1514	8000 → 57015 [PSH, ACK] Seq=55481 Ack=1 Win=501 Len=1460
23	0.006798768	10.0.0.33	71.93.65.129	TCP	1514	8000 → 57015 [ACK] Seq=56941 Ack=1 Win=501 Len=1460
24	0.006807101	10.0.0.33	71.93.65.129	TCP	1514	8000 → 57015 [PSH, ACK] Seq=58401 Ack=1 Win=501 Len=1460

Frame 4: 1514 bytes on wire (12112 bits), 1514 bytes captured (12112 bits) on interface 0
Ethernet II, Src: Raspberr_fb:4d:6d (b8:27:eb:fb:4d:6d), Dst: Netgear_32:37:e2 (8c:3b:ad:32:37:e2)
Internet Protocol Version 4, Src: 10.0.0.33, Dst: 71.93.65.129
Transmission Control Protocol, Src Port: 8000, Dst Port: 57015, Seq: 39421, Ack: 1, Len: 1460
Data (1460 bytes)
Data: f243243c02ed91bfafbd759aea1bbd4351902871e6bfcc0...
[Length: 1460]

Figure 10: TCP packets in wireshark, demonstrating the livestream image data

Appendix 1: Notes for Graders

Code, setup, and rough documentation is explained in the README.md that is included in the project files. Further documentation can be found within the code itself. Should information become corrupted, project files can be found [here](#), on GitHub. The GitHub was managed by a single individual, so any commit information there will be inaccurate.