Bidirectional Communication Between Two Wallaby Clients
Aaron Pierce
Norman High School

Bidirectional Communication Between Two Wallaby Clients

1. Abstract

Concurrency in Botball has historically been a challenge. In order for two robots to collaborate to complete one task, very precise timing must be employed, or the processes of both robots may be compromised. Be it defensive strategies of other teams, hardware faults, or other unexpected events, this precise timing could be interrupted and the robots would be none the wiser. Bidirectional communication, achieved via Submodule Driven Development (see paper "Reinventing the Botball Programming Process with Submodule Driven Development"), allows for streamlined communication between robots, enabling concurrency in operations, and live reporting of errors to prevent catastrophe. This paper will explore the applications, implementations, and potential problems of connecting two wallaby clients, and concludes that an approach using WebSockets is highly beneficial to a team, and is the superior approach over other protocols.

2. Introduction

The C programming language has its benefits. Being a low level language, and being able to easily interact with hardware, it finds itself a good fit for robotics. However, because C is so low level, it makes interacting with higher level constructs harder, such as networks. Languages like JavaScript, which are specifically built for use on the internet make for a much easier experience developing network attached programs. Libraries like Node.JS remove the need for a HTML document bound scripts, and allow us to create operating system level programs that can interact with our robots. Using Node.JS, as well as Socket.IO, Wallabies can use the WebSocket protocol to communicate, allowing for efficient and reliable transmission of arbitrary files, data, and messages. Using this transmission, robots can communicate via WebSockets, and can execute submodules in response to this data, enabling a wide variety of responsive strategy.

3. Connecting Two Wallabies

In order for Wallabies to communicate via a network, they must be connected to a common network. Creating a server and client paradigm, one Wallaby runs a Socket.IO server, with the client Wallaby connected to the server's WiFi network, and running a Socket.IO client. Despite a client/server relationship, both wallabies can send messages to the other; the Wallaby communication is completely bidirectional. In order to connect one Wallaby to another, we must disable the WiFi server capability and enable WiFi connection services. This can be accomplished with the commands illustrated in Figure 1.

```
$ mkdir /home/root/services

$ mv /lib/systemd/system/hostapd.service /home/root/services

$ mv /lib/systemd/system/wifi.service /home/root/services
```

Fig. 1. Conversion of Server Wallaby to Client Wallaby. These commands can be input by connecting a keyboard to the wallaby, or establishing an SSH connection.

From the Wallaby newly set as a client, the command wpa_cli can be executed to connect the client wallaby to the server's network. This process is documented in Figure 2.

```
$ wpa_cli

    > add_network 0

    > set_network 0 ssid "wallaby1234"

    > set_network 0 psk "wallabyPassKey"
```

Fig. 2. Connection of client Wallaby to server's network.

Now having the Wallabies connected, they can begin communicating via WebSockets.

3. Implementation of Communication Protocol

Implementation of WebSocket protocols are based on two programs, a server, an example of which shown in Figure 3, and a client, an example illustrated in Figure 4.

```
const app = require('http').createServer();
const io = require('socket.io').listen(app);
const url = require('url');

// Create server listening on port 3000 (wallabyIP:3000)
app.listen(3000);

io.sockets.on('connection', function(socket) {
  socket.on('message', function (data) {
        console.log("Recieved message: " + data)
        socket.emit("messageCallback")
  });
});
```

Fig.3. Example Socket.IO server. Run by executing node server.js

```
const serverIP = process.argv[2] //node client.js 192.168.125.1:3000

var socket = require('socket.io-client')("http://" + serverIP);

socket.emit("message", "hello") //send a message named message with data "hello"

socket.on("messageCallback", function(){
console.log("message successfully recieved")
});
```

Fig.4. Example Socket.IO client. Run by executing node client.js wallabyIP:3000. The ip of a Wallaby is usually 192.168.125.1

Inside the socket.on blocks, any arbitrary code can be executed. The best implementation of this being executing a submodule, such as one that would drive a Wallaby forward. This could be implemented such that after the server's submodule finished executing, it would send a message to the client to begin theirs. This would make one wallaby drive forward, and the other to do the same only after the first finished. The real world use cases of Wallaby to Wallaby communication would be far more complex, but this is an excellent first example.

4. Advantages of WebSockets Over Other Protocols

In connecting two wallabies, many different approaches were attempted. Analyzing a simple HTTP request, times of more than a 500ms[1] can be found. WebRTC was also explored, but it is intended for a more media driven experience. SCP file transfer was proposed by the team as

well, but the data collected and illustrated in Figures 5 shows that SCP file transfer was a suboptimal solution

Shown here, WebSockets regularly operated at speeds of less than thirty milliseconds, whereas SCP file transfers took consistently more than six-thousand milliseconds, or six whole seconds (Raw data can be found in appendix 1). SCP was initially favored due to its ability to be executed straight from a C program, or the terminal without the need for a client/server program structure, but because it is transferring files the delay becomes compounded. You have to account for the write speed of one Wallaby, the upload speed of the file, the download speed of the file, and the write speed of the other Wallaby. The WebSocket implementation removes this entirely, and is limited only by the network speed. In a use case of constant sensor data polling, where one robot uses the other's sensor data in realtime, you want as little delay as possible, which is best achieved with WebSockets.

5. Issues

While the WebSocket protocol is quite reliable in its implementations, there are some potential downsides. One is the forced use of Submodule Driven Development. This complicates the workflow of developing Botball programs and thus limits this approach to only advanced teams. Secondly, WebSockets aren't without delay. Because all of the Wallabies are emitting a WiFi signal, the room can get bogged down quickly. This may create some unreliable conditions for network operations, however, due to the nature of WebSockets we know if a connection is not

being established, and using Submodule Driven Development, an entirely different program can be executed that runs without a network requirement.

6. Conclusion

Evidenced by low latency communication, and failsafes in the case of errors such as a missing network connection, a pairing of Submodule Driven Development as well as WebSocket based communication allows for incredibly powerful collaborative robotics approaches. Due to the latency of other protocols and approaches it can be concluded that WebSockets are the vastly superior communication protocol.

6a. Appendix 1

Raw WebSocket latency data, measured as the amount of milliseconds from the server sending the first message to when the server received the success callback message. The server sent a message every second.

47	50	48	34	28	36	36	27	21	37	23	32	21
21	39	34	35	36	23	34	20	37	23	39	21	20
39	38	23	36	24	24	48	21	56	20	69	37	37
18	20	38	84	23	35	22	23	31	21	25	37	24
20	37	36	24	23	35	40	93	25	40	38	36	21
20	41	43	43	35	28	17	20	20	25	22	21	20
23	23	22	116	36	21	23	32	69	40	21	21	36
20	19	60	38	20	22	23	25	22	24	24	67	37
21	61	123	32	98	81	41	19	19	22	46	306	38
22	36	17	21	20	58	20	21	21	85	35	31	38
23	22	21	24	37	27	20	21	42	21	33	21	24
55	37	20	56	17	25	24	22	17	20	23	37	31
34	21	24	30	21	34	36	36	75	37			

Raw SCP latency data, measured as the amount of milliseconds from the initializing of the SCP command to the synchronous process ending. Attempts were one second apart, from completion of one to the initialization of the other

6012	6313	6034	6356	6358	6343	6331	6345	6349	6340	6343	6343	6402
6357	6318	6309	6335	6382	6339	6333	6355	6354	6328	6345	6318	6362
6330	6322	6352	6336	6326	6340	6358	6346	6331	6354	6344	6342	6367
6339	6348	6328	6352	6329	6314	6371	6343	6338	6374	6371	6432	6433

6422	6345	6344	6370	6394	6363	6341	6375	6387	6342	6341	6327	6360
6350	6350	6353	6332	6419	6350	6343	6379	6372	6341	6345	6357	6353
6355	6355	6356	6363	6352	6318	6316	6349	6351	6349	6349	6364	6321
6362	6370	6336	6336	6335	6348	6342	6330	6329	6332	6340	6341	6374
6337	6345	6313	6340	6358	6344	6308	6348	6327	6366	6037	6284	6372
6370	6358	6313	6341	6379	6315	6345	6370	6353	6315	6344	6337	6353
6343	6334	6344	6336	6332	6336	6414	6359	6368	6368	6347	6355	6339
6331	6341	6357	6320	6364	6326	6349	6349	6353	6365	6335	6401	6342
6323	6365	6333	6327	6352	6337	6358	6334	6334	6361			

7. Works Cited

1. "Analysis of HTTP Performance Problems." *Analysis of HTTP Performance Problems*, W3 Foundation, www.w3.org/Protocols/HTTP-NG/http-prob.html.