Botball Robot-to-Robot Communications

Annaleise Kealiher and Tobin Slaven

The Bots Who Say Ni

Botball Robot-to-Robot Communications

In this paper, we will present how to connect your Wallabies, send data between them, and utilize this connection to have communicating robots on the competition field.

When this Botball season started, the new rules were released: there would be colored cubes with randomized placement on the field, AND there would be only one camera in the kit. This meant that only one of our robots could know the positions of these cubes unless we could somehow share information between them. While attempting to get our robots to share data, we found that there was hardly any information out there on how to achieve this. We decided that if we could get our robots to connect, we should document the progress so that other teams could use this knowledge. After some research (Thank you to the Dead Robot Society and the Los Altos Community Team for your advice and encouragement), we concluded that using the Wallaby's built in WiFi to connect our robots would be the simplest way for the robots to share knowledge and had the benefit of not requiring any special modifications to the Wallaby or needing extra parts.

After much trial and error, our team developed the following method to connect our two wallabies.

1: One Wallaby would act as a client and another as a host for the connection. We connected the client to a computer using a USB cable. (During the project, it was easy to get the Wallabies confused, so it helped to label them and change the background menu color in the Wallaby GUI Settings to make it easier to tell them apart).

2: We opened a Linux terminal.

Mac OS machines have Linux built in. Windows computers require separate installation of a Linux terminal. Howtogeek.com has a great tutorial on installing a Linux Bash shell on Windows [1].

3: The Bash shell command "`ssh root@192.168.124.1`" (this is the IP address for the USB cable connection with the wallaby) was used to connect the computer to the client Wallaby through the open terminal. We received a message that the authenticity of our host could not be established and had to type "yes" to continue our connection. This only happened the first time we connected our controllers (see screenshot). The subsequent connections did not require confirmation because our wallaby had been added to the list of accepted addresses.

```
squid@DESKTOP-IKSQO52:~$ ssh root@192.168.124.1
The authenticity of host '192.168.124.1 (192.168.124.1)' can't be established.
ECDSA key fingerprint is SHA256:9ivFkMK0dpFLrgP61SDLvv2alc2Q45q+2hpL26WyB04.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '192.168.124.1' (ECDSA) to the list of known hosts.
```

Once we logged into the client, we were able to execute commands as the user "`root@pepper`" (pepper is the default wallaby username) as though we were inside the Wallaby's terminal.

4: The Bash command "`wpa_passphrase` (host SSID) (host password) `> wpa.conf`" generated a file called `wpa.conf` that contains the host's username and password. Then the client will later compare this file to the host's username and password and if they match, the host will allow a connection.

```
root@pepper:~# wpa_passphrase ####-wallaby ******** > wpa.conf
```

We replaced the "####" and "********" with our host's SSID and password that may be found on the Wallaby's about page accessed on the Wallaby home screen.

5: We created a script using the command "`nano start.sh`".

"Nano" is a command line text editor which allows the editing and saving of text in a file. The screen shot shows our script:

```
  GNU nano 2.2.5                                              File: start.sh

killall hostapd
wpa_supplicant -B -iwlan0 -cwpa.conf
echo Starting WPA in background
```

"`Killall hostapd`" stops the "`hostapd`"program which manages the Wallaby's default hotspot capabilities. By stopping "hostapd" we prevented the Wallaby from broadcasting its own network (this is typically a discoverable network named as the Wallaby default SSID). "`Wpa_supplicant -B -iwlan0 -cwpa.conf`" ran the authentication protocol to ensure the usernames and passwords given to the client matched with the host's. "`Echo Starting WPA in background`" was just like a `printf`() statement for Linux and printed "`Starting WPA in background`" on the console of the Wallaby to provide a message if the script was running correctly.

The Nano text editor was closed by pressing 'Ctrl+X' to save the start.sh file.

We made the start.sh script runnable by using "`chmod a+x start.sh`" to add execute permissions to the file as seen below.

```
To run a command as administrator (user "root"), use "sudo <command>".
See "man sudo_root" for details.

squid@DESKTOP-IKSQO52:~$ ssh root@192.168.124.1
root@pepper:~# wpa_passphrase [redacted]-wallaby [redacted] > wpa.conf
root@pepper:~# nano start.sh
root@pepper:~# chmod a+x start.sh
root@pepper:~#
```

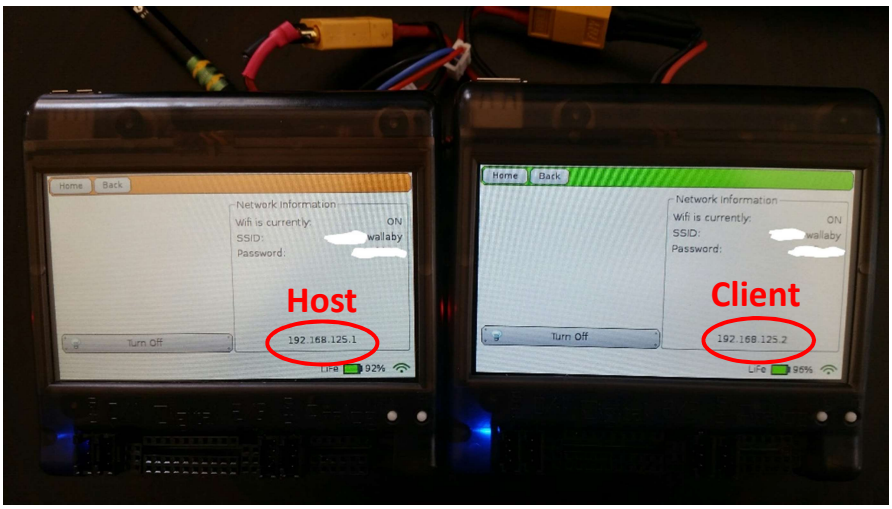
6: Finnaly, we used a C program on the Wallaby that can run the start.sh script, although we could also have used other languages.

The "`system()`" function executes Linux commands to inside of C code. "`Cd;`" ensured that we were in the home directory of root (the directory that contained the start.sh file). Because we modified start.sh to be executable earlier, "`./start.sh`" executed the killall hostpad and wpa_supplicant commands inside of the file. Next, "`dhclient-r`" stripped away the client's old IP address and "`dhclient wlan0`" gave the client a new address. Wallabies default to an address of 192.168.125.1, but because this was the IP address of the host, the client needed to be assigned a new one; we chose usually 192.168.125.2 out of convenience.

File: main.c

```c
#include <kipr/botball.h>

int main()
{

    printf( "Running\r");

    //runs script with killall hostpad and wpa_supplicant
    printf("Starting connection\r");
    system("cd;./start.sh");
    msleep(10);

    //get rid of the old IP address.
    printf(" Flushing ip\r");
    system("dhclient -r");

    //get new IP address from host. This will most likely be 192.168.125.2
    printf("Getting ip\r");
    system("dhclient wlan0");

    return 0;
}

```

To verify that this worked, we went into the client Wallaby's network settings and made sure that the IP address in the bottom right of the page was 192.168.125.2 for the client instead of the default 192.168.125.1.



Throughout this project we experienced many issues in forming the initial connection between the two Wallabies. As we progressed, we identified a cohesive method of verifying the connection between our client and host. The primary instrument we used for connection testing was Ping, a command-line networking tool. Characterized by its simplicity and widespread adoption, Ping perfectly suited our needs for this project. Ping works by bouncing Internet Control Message Protocol (ICMP) packets between devices on a network and tracking the amount of time it takes for the packet to complete its round trip. Furthermore, Ping is pre-installed on nearly every Operating System, including the Linux distribution that runs on the KIPR Wallaby.

Our connection testing procedure was as follows:

a. We turned on both client and host.

b. We connected our computer to our client Wallaby via USB.

c. We used SSH to gain access to the client Wallaby over the USB interface (“`ssh root@192.168.124.1`”).

d. We ran the program that attempts to connect the Wallabies

e. We attempted to ping the host at its IPv4 address (“`ping 192.168.125.1`”)

When it failed, it would give an output like this:

```
root@pepper:~# ping 192.168.125.1
PING 192.168.125.1 (192.168.125.1) 56(84) bytes of data.
From 192.168.125.2 icmp_seq=1 Destination Host Unreachable
From 192.168.125.2 icmp_seq=2 Destination Host Unreachable
From 192.168.125.2 icmp_seq=3 Destination Host Unreachable
```

A success however, gave an output like this, printing the time it took for the packet to complete its round trip.

```
64 bytes from 192.168.125.2: icmp_seq=1 ttl=64 time=0.428 ms
64 bytes from 192.168.125.2: icmp_seq=2 ttl=64 time=0.315 ms
64 bytes from 192.168.125.2: icmp_seq=3 ttl=64 time=0.311 ms
```

To end Ping, we pressed 'Ctrl + C'. In the event of a failure, we found it best to leave host up while rebooting the client before trying again.

Once the Wallabies were connected, many opportunities became available. In one example, screensharing, we opened terminals on both wallabies by going to the wallaby Settings-> Hide UI. We connected keyboards by USB cable to each Wallaby. On the host Wallaby, we typed "`screen`" and hit enter twice. On the client, we typed "`ssh root@192.168.125.1`" and then "`screen -x`" and hit enter twice again. This set up a screen on the host and then the client connected into the host's terminal using ssh. "`Screen -x`" gained access to the host's screen. Once the Wallabies were screensharing, anything typed on either wallaby appeared on both screens.

Another possibility was sending files from one Wallaby to the other. In the client Wallaby's terminal, "`cd`" returned us to the home root directory before we made a file for transmission. Next, we used "`echo "Hello World" > messageSend.txt`" to create our message file called messageSend.txt. "`scp messageSend.txt root@192.168.125.1:messageSend.txt`" sent tour file over.

```
root@pepper:~# cd
root@pepper:~# echo "Hello World" > messageSend.txt
root@pepper:~# scp messageSend.txt root@192.168.125.1:messageSend.txt
```

Then to open the file on the other side, we used "`cd`" to change to the home root directory where the file was sent. Then, we typed "`ls`" to list all of the files in that directory. We viewed the message file contents with the command "`head messageSend.txt`", although there are numerous ways to do this in Linux.

```
root@pepper:~# cd
root@pepper:~# ls
Documents                                       libwallaby-dbg-git-41.0.cortexa8hf_vfp_r
battlehill-dbg-git-51.0.cortexa8hf_vfp_neon.rpm libwallaby-dev-git-41.0.cortexa8hf_vfp_r
battlehill-dev-git-51.0.cortexa8hf_vfp_neon.rpm libwallaby-git-41.0.cortexa8hf_vfp_neon.
battlehill-git-51.0.cortexa8hf_vfp_neon.rpm     may_test
botui_settings                                  messageSend.txt
chrome_user_data                                root@192.168.125.1
harrogate-git-37.0.cortexa8hf_vfp_neon.rpm      root@192.168.125.1
home_folder.tar.gz                              stm32flash
libwallaby                                      test.c
root@pepper:~# head messageSend.txt
Hello World
```

<--The message file is now on the host

<--The message file contents

Although screensharing and sending files could potentially be useful in other projects, in a Botball tournament Wallaby-to-Wallaby interactions will need to take place inside competition code and be beneficial to robots within the short, 2-minute period. There are several applications that could aid competition bots. The communication could be used to allow robots to alert one another of a change in plan so that they avoid collisions. They could find one another to hand off game objects or know when an area of the field is available. Significant to this year's competition, robots could communicate locations of game objects they have found to the other robot.

Here, we will walk through an example of how to get robots to share game object location.

First, the Wallabies connect. Then the Wallaby with the camera determines which of 3 possible zones contains a specific colored cube. The camera bot writes the cube's location into a file and sends it to the blind bot (the bot without the camera). The blind Wallaby opens this file, read the locations, and executes a scripted set of actions corresponding to the locations sent by the camera bot to deliver the blind bot's payload to the proper zone. In this scenario, the proper color of cube the blind robot is looking for is in one of three zones. The code below runs on the wallaby with the camera to check for the proper cube. Earlier in the code, the robot checked the first of the three zones, so now it has now been narrowed down to one of two zones. The code loops through

```
camera_open();

int counter = 0;
int i = 0;
for(i = 0; i < 10; i++){
    camera_update();
    if(get_object_count(2) > 0 && get_object_area(2,0) > 300)
        //if there is at least one object with area greater than 300, increase the counter
    {
        counter++;
    }
    msleep(100);
    printf("objects: %d and area: %d\r", get_object_count(2), get_object_area(2,0));
}
if(counter >= 5)//if there were at least five pictures with objects greater than 300 in area....
{
    system("echo \"1\" > messageSend.txt");
    system("scp messageSend.txt root@192.168.125.1:messageSend.txt");
    printf("YEllOW IN MIDDLE \r");
}
else{

    system("echo \"2\" > messageSend.txt");
    system("scp messageSend.txt root@192.168.125.1:messageSend.txt");
    printf("YELLOW IN LAST \r");
}
camera_close();
```

and takes ten pictures of the cube and evaluates if the pictures contain patches of the correct color (because the cameras aren't accurate, ten pictures are taken and are checked for a majority of the pictures to contain the correct color recognized). Finally, the camera bot sends a message containing the location of the correct cube to the blind robot.

Just like when sending files earlier, the command "`echo`" writes data to a file and "`scp`" sends it to the IP of the blind bot.

Then, on the blind robot, the following code runs to open the message file and extract the data. "`Chmod a+r messageSend.txt`" enables reading permissions. Then, "`fopen`" opens the file and "`fscanf`" searches it for the first integer. That integer data is stored as "`num`" and is used to determine which way the robot should go to deliver its game objects.

```
//modifies messageSend.txt to give you permision to read the contents
system("cd;chmod a+r messageSend.txt");

//will store the value inside of messageSend.txt
int num;

//opens messageSend
FILE * fp;
fp = fopen("/home/root/messageSend.txt", "rb");

//checks if file opened
if(fp == NULL){
    printf("cannot open file\r");
    exit(1);
}

//reads the first integer in the file
fscanf(fp, "%d", &num);

//prints the int data
printf("the message data: %d", num);

//closes the file
fclose(fp);
```

```
//determine where to go
switch(num) {
    case 0:
        goToFirst();
        break;
    case 1:
        goToMiddle();
        break;
    case 2:
        goToLast();
        break;
    default:
        //goSomewhereElse();
        break;
}
```

Because these robots were communicating, it enabled the bot without a camera to benefit from the other bot's sensor, the camera, which was a critical capability to get the blind bot's objective accomplished. This communication method creates numerous new ways for a pair of robots combine their capabilities. By leveraging the equipment and characteristics of both bots, a single bot can be enabled to do things not before possible.

Another application for this communication method during a tournament is that if a robot needs to change its path, it can alert the other bot. If, for instance, a robot notices that its target game object is no longer available and needs to change plans, it can send a message to the other robot to wait. Then, while this other bot freezes, the robot can move to its Plan B without worry of collision and then send another message to the frozen bot when it is safely out of the way. That way, both bots can navigate the field conflict-free, each notifying the other to wait before they cross their path.

Whether trying to get robots to synchronize their paths across the field, convey sensor values, or team up to locate game objects, wallaby-to-wallaby communications can help. Robots who stay in touch on the field allow you to make the most of your limited sensors and equipment. Knowing which path the other robot is taking, lets robots make more informed decisions about their own path to avoid collisions and achieve their goals. Communication allows cooperation, and the more your robots cooperate, the more they can accomplish.

[1] Hoffman, Chris. "How to Install and Use the Linux Bash Shell on Windows 10." *How-To Geek*, How-To Geek, 7 Mar. 2018, www.howtogeek.com/249966/how-to-install-and-use-the-linux-bash-shell-on-windows-10/.