Mirroring a Workspace for Writing Code on Wallabies
Aaron Pierce
Norman High School

Mirroring a Workspace for Writing Code on Wallabies

1. Abstract

Historically, Wallabies have been almost entirely interfaced with through the pre-installed Web IDE native to each Wallaby. While convenient for programs written in C, it becomes infeasible to write in any other language, as the IDE only fully supports this language. This paper will unveil a new method of writing code to a Wallaby, one that allows code to be written in any language, as well as allowing a programmer to use a fully featured desktop IDE, such as Visual Studio Code. The development environment allows for live mirroring of a workspace, such that on save of a file located on an external computer, the changes are immediately transferred to the copy of the program on the Wallaby.

2. Introduction

As outlined in *Bidirectional Communication Between Two Wallaby Clients,* submitted in 2018, Wallabies possess great potential to communicate wirelessly. Wallaby clients have NodeJS pre-installed, and with their capability to create their own WiFi server, they become prime candidates for WebSocket communication, the details of which are explained in *Bidirectional Communication Between Two Wallaby Clients*. While libraries have been created to allow use of WebSocket communication in C, it is much easier to use a higher level language like JavaScript to implement them. Natively on the Wallaby, there are two possible ways to effectively write said JavaScript: write the code on a computer and transfer it via SCP or a similar protocol, or write the code directly to the Wallaby. Neither were optimal solutions, as the first was incredibly inefficient, requiring manual transfer on save, and the second was worse; the Wallaby has no simple package management solution, and as such a programmer is largely limited to the text editing natively packaged with the Wallaby. The program outlined in this paper will provide a clearly superior third option to allow you to write code in any language, greatly easing the process of writing JavaScript and using WebSockets to communicate between two Wallaby clients.

3. Configuring the Development Environment

In order for the program to sync files, it must be connected to the on-board WiFi server native to the Wallaby. In connecting to the Wallaby, however, neither the computer being programmed on, nor the Wallaby, can connect to online sources to download the program's files, so one should download the files before connecting to the Wallaby. To download the server and dependencies, execute the commands in Fig. 1.

```
If your system has cURL installed:
$ curl -O
http://raw.githubusercontent.com/SAXTEN2011/WallabyFileWatcher/master/serverInstallcurl.sh

Then connect to the WiFi server on the Wallaby and execute
$ chmod +x ./serverInstallcurl && ./serverInstallcurl

If your system has Wget installed:
$  wget
http://raw.githubusercontent.com/SAXTEN2011/WallabyFileWatcher/master/serverInstallwget.sh

Then connect to the WiFi server on the Wallaby and execute
$ chmod +x ./serverInstallwget && ./serverInstallwget
```

Fig. 1. One line installation of server, client, and dependency files. Running this command in the highest level directory you want synced will install the server files, transfer the client files, and install server dependencies.

After installing the server files on the computer that is being programmed on, the client files need to be installed on the Wallaby, the installed server files include a script to do so. Running the command in Fig. 2 will transfer the necessary files to the Wallaby. The command in Fig. 2 will be run automatically by the install script, but should an error occur it can manually be executed by following Fig. 2.

```
$ node installToClient.js
```

Fig. 2. Transferring client files. This command will return a series of authentication prompts, resulting in the transfer of the client files to its home directory.

After transferring the files, dependencies must be installed on the Wallaby, and the program needs to be run, Fig. 3 provides the commands to do so.

```
$ cd ~/YOUR_PROJECT_DIRECTORY
$ npm install
$ node wallabyWatched.js &
```

Fig. 3. Installing client dependencies and running. These commands download the script's dependencies, and execute it, opening a channel for file syncing. Calling "node app.js" on the programming computer will open the server and begin syncing files.

3. Usage and Use Cases

Mirroring a workspace only requires executing both scripts (the instructions of which are given with their respective figure). Once both the client and server are running, any changes to files within the directories at or below the level of the server script will be mirrored to the client's copy.

While the program will mirror any arbitrary files, the scripts are authored with the intent of easing the process of writing network-connected JavaScript programs, examples of which can be found in *Bidirectional Communication Between Two Wallaby Clients*. Running a JavaScript program on the Wallaby is tedious, requiring SSH or keyboard access to the Wallaby, and a manual start of the program instead of one based on a light sensor, which is not competition legal or viable. Because of this, a C boilerplate program should be written with the Web IDE, that merely runs a system command to start the main NodeJS script that you have mirrored with the scripts, writing all further code with the development environment presented within this paper. Using a C starting program, the same JavaScript program will always be launched, so any changes to the JavaScript program change how the robot will behave without any need for compiling. This approach allows for instant testing and running of robots that is unparalleled within the Botball ecosystem.

4. Advantages

Writing programs using these scripts provides a vastly superior method of writing code on a Wallaby. Firstly, it enables writing JavaScript. In the ability to write JavaScript, communication between Wallaby clients becomes effortless, further explained in *Bidirectional Communication Between Two Wallaby Clients*. Beyond JavaScript is the ability to write C programs more efficiently. The client script has the possibility of compiling C files as they come in, using the command found in Fig.4.

```
$ gcc -o ./bin/botball_user_program -lwallaby -lm -I ./include ./src/*.c
```

Fig. 4. Compiling a C file using libwallaby. The o flag changes the output directory. Using "./" will place the output file in the current directory. Running this command creates a runnable program to utilise BotBall methods and functions.

This modification creates a powerful tool for writing an entire suite of Botball programs, ranging from JavaScript connected web applications to C programs that operate near the metal.

Transcending language comes the ability to write any of these programs in any IDE or text editor a programmer wishes. Visual Studio Code with Botball snippets installed could provide syntax highlighting, intellisense, debugging, and more functions previously not available in the Web IDE.

5. Disadvantages

While powerful, this method of writing programs adds many levels of complexity to writing Botball programs. First, a team must have an understanding of the terminal, in order to download the scripts. Secondly, a team writing a web connected JavaScript program would firstly have to write a program using the Web IDE to run the JavaScript, and would then have to write the JavaScript that calls further executable C files to use LibWallaby functions, all being managed by a NodeJS program. This is no small task, requiring knowledge of two different languages and a level of terminal prowess adds many barriers to entry. This approach vastly complicates the workflow for composing BotBall programs, but those who are apt to use it can find great utility in it.

6. Conclusion

Evidenced by an ability to effectively write Botball programs in languages other than C, as well as the opportunity to use a third-party IDE, this new approach to authoring Botball code presented herein is wholly beneficial. Mirroring a workspace enables new heights of Botball that could not have previously been achieved with such ease, enabling advanced concepts within robotics to be utilized, such as communication between two robots. The usage of this tool allows for Botball to move towards more sophisticated robots and strategies, without leaving behind the ease and comfort of a fully-featured IDE, improving the experience of programmers writing to Wallaby clients globally. The horizons of this robotics competition lies within the technology that can easily be implemented via the tool presented within this paper, foreshadowing a prosperous future for the competition of Botball.