Go Playground Write-Up

Author: Nick Brown
Email: ncbrown@cs.ucsb.edu
GitHub: @ncbrown1 -- https://github.com/ncbrown1

The Go Playground is much more intriguing and complex than one might first think when considering its function. While it might be easy to create a service that provides the ability to run arbitrary code and get back the output, it is extremely difficult to do so securely and in a way that handles all possible edge cases. I did not expect to find such an intricate runtime systems problem in this project, but I was happy to.

The nature of this class project is to overcome the restrictions placed upon the current playground. These restrictions include:

- Limited File System Access
- Network access only to loopback interface (127.0.0.1)
- Fake/simulated time
- Limited CPU and RAM

However, these restrictions should only be lifted for those who can authenticate to the new application.

There are three essential components to the Go Playground: namely, the JavaScript client, the Server-side Front-End, and the Back-End. The JavaScript client sends the user code to the front-end server over HTTP and receives back program output from the playground system. The Front-end takes arbitrary GoLang code and checks it against a set of cached programs. If there is a hit, the cached outputs are sent back to the client in order to save time. If there is a miss, then the program is sent to the back-end to be compiled and run, returning program output and/or compiler errors.

In order to find the documentation on the design of the original Go Playground, you can visit this link: <https://blog.golang.org/playground>
Also, the implementation of the current version is publicly available on the Go Tools GitHub repository here: <https://github.com/golang/tools/blob/master/playground/>

JavaScript Client

In the original playground, the client is a single HTML file that imports several JavaScript and CSS files. The version currently deployed on <http://play.golang.org> is hosted via Google Cloud Platform (GCP), and because GCP does not support WebSockets (yet), the system is restricted to using the browser's XMLHttpRequest functionality (e.g. via AJAX/JQuery). They loosely define a Transport "interface" in the javascript files in order to provide the same interface for different client-server communication mechanisms. It seems as though there is a WebSocket implementation present, but not used.

In my re-implementation of the playground, I do enable the users to choose between regular HTTP and Websockets when running their programs in the playground. The difference is that for regular HTTP, the user's output screen is blocked until their program finishes, at which time their program output is sent back to them in its entirety, whereas with WebSockets, output is sent to the user as it is generated. Since time is emulated on the server, there are some interesting caveats to that, which I will discuss soon.

This new client library is written in [React.js](https://facebook.github.io/react/) and utilizes several convenient tools, such as: a bundling tool called [Webpack](https://webpack.github.io/) which compiles all of my JavaScript code, CSS styling, and dependencies into a single file to be imported by the browser; Promise-based asynchronous HTTP requests to the server using [Axios](https://github.com/mzabriskie/axios); and global state management with message passing and handling using [Redux](http://redux.js.org/). I believe that in the long run, this framework for dispatching actions through the system should work better and make more sense than the current implementation, which essentially sends itself messages to emulate what it thinks the server is doing.

One feature that I did not have the time to get to was the timed playback of output, which I alluded to above. Because time is emulated on the back-end server (in order to hide details about the underlying system) and it is impossible to get immediate feedback from the back-end server, the output of the programs being run is chunked into pieces over discrete time intervals. The front-end would take that output and play it back as if it were happening in real time. To the user, it seems like the program is running right in front of their eyes.

Here is an example JSON output of ordered events separated by 1 second each:

```json
{
  "Errors": "",
  "Events": [
    {
        "Delay": 1000000000,
        "Message": "2009-11-10 23:00:01 +0000 UTC\n"
      },
      {
        "Delay": 1000000000,
        "Message": "2009-11-10 23:00:02 +0000 UTC\n"
      },
      {
        "Delay": 1000000000,
        "Message": "2009-11-10 23:00:03 +0000 UTC\n"
      }
    ]
}
```

Front-End Server

The front-end portion of the server not only routes run requests from the clients to the backend, but it also uses a sophisticated caching mechanism on top of [memcached](https://memcached.org/). If certain programs are uploaded and processed frequently, the results from the back-end are cached in a distributed memory store so that the next time that same program is requested, the cached version is sent instead of using the computing power to generate the outputs. This saves the architecture a lot of time and effort in the case of very popular programs, such as those that are on the front page of the GoLang website.

As this is a class project, I did not do much to optimize the front-end part of the server as much as the original implementation. Additionally, I did not want to pay to use the resources that Google has on hands. That is not to say that I didn't try a little. In my initial study of the effects of caching the outputs, I took uploaded programs, saved them under filenames named by the SHA-1 hash of their contents, and saved the output as a file in my /tmp directory. For simple "Hello World" programs, the initial request requiring processing would take around 150-400ms to complete. The second time around, by using the cached version, the request then succeeded in about 400-600µs on my localhost. This was very impressive.

One of the other responsibilities of the front-end server is to translate output from the back-end for the client to be able to process and playback the program. Using a simple encoding scheme (`0 0 P B <8-byte time> <4-byte data length> <data>`), the back-end delivers chunked output to the front-end that looks like this:

```
```

```
\x00\x00PB\x11\x74\xef\xed\xe6\xb3\x2a\x00\x00\x00\x00\x1e2009-11-10 23:00:01 +0000 UTC
\x00\x00PB\x11\x74\xef\xee\x22\x4d\xf4\x00\x00\x00\x00\x1e2009-11-10 23:00:02 +0000 UTC
\x00\x00PB\x11\x74\xef\xee\x5d\xe8\xbe\x00\x00\x00\x00\x1e2009-11-10 23:00:03 +0000 UTC
```

The front-end then parses that output, converts it into JSON, and sends it back to the client.

Back-End Processor

The back-end part of the playground is probably the most relevant thing to talk about for the CS 263 runtime systems class. Naïvely, one might assume that the arbitrary code could simply be compiled and executed, directing the output into a file that eventually gets sent back to the client. While that would technically work, whoever implemented this service that way would probably have a broken or otherwise compromised server in no time.

In order to provide certain security guarantees, Google makes use of the Native Client (otherwise known as `NaCl`). NaCl is not very well documented online in terms of using it with golang, but its design is fairly well understood. It was originally designed to permit the safe execution of x86 programs inside web browsers so that web application developers could potentially extract more performance out of their client-side code. There is also a portable version (`PNaCl`) with a bytecode that was heavily inspired by LLVM, though it is not widely talked about in the community.

NaCl essentially implements sandbox environment for 32-bit x86, 64-bit x86, and 32-bit ARM architectures, which provides a an emulated operating system with customized features including system calls, memory allocations, thread handling, signal processing, and more. By arranging the assembly instructions of the compiled binaries in a verifiable way, the NaCl environment can restrict access to certain features, such as pipes, networking, and the file system and ensure that the host running the code is not privy to dangers. Assembling a NaCl executable is typically the job of the linking program (which pulls in the replacements for libraries like `syscall`). However, for the 64-bit x86 architecture, the compiler and runtime must also be modified to compensate for the difference between the 64-bit words in the hardware and the default 32-bit integers in go programs.

Though access to files and the network are restricted, there are sufficient emulations that work “enough” for the kind of small programs that run on the playground. There is a simulated clock for each new user program that starts at Unix time `1257894000`, which also allows for accurate caching of redundant time-sensitive programs. And though the network is not accessible both inbound and outbound from the server running the program, programs are able to communicate locally over the loopback interface (127.0.0.1) through the use of a drop-in library that converts the network system calls into message passing using Go’s built-in channel support. Similarly, file system calls are replaced by routines that reference an in-memory file system, if properly configured.

Summary of Personal Contributions

The interface and initial server setup of this application was completed in just a matter of weeks, and as I context switched to my MS defense, this fell behind and I underestimated the time it would take to complete the project fully. However, I have successfully re-implemented a majority of the Go Playground and was just given a stern reminder that a book should not be judged by its cover.

In summary, the architecture of this re-implementation of the go playground has been decoupled from the Google Cloud Platform and can now be hosted anywhere. The javascript client that I developed used a different state management mechanism than the original playground, which I believe should help data flow transparent in future development. The front-end does some minor file-based caching that could possibly be improved in the future. Additionally, I have added authentication to the application to restrict certain routes to those who have the proper credentials. The backend supports both WebSocket and RPC requests, which will allow for users to be flexible with how they interact with the system.

As it turns out, Go support in NaCl is rather new to the public and as a result, there is not much documentation. I could not figure out how to get my personal development workstation up and running using the NaCl-targeted compilation in the playground process and consequently, my system will not be as “secure” as the existing go playground. I do, however, generate a binary executable for each program, which could potentially be placed into a `chroot`-ed environment in order to segregate it from the hosting file system to mitigate potential harm yet allow for flexible programmability. To satisfy the description of this project in the first place (namely, to replicate the go playground functionality, but remove restrictions to those with a login), we can simply restrict the entire application to require a login.

Had I had an extra week or two on this project, I would have liked to compare the performances of different viable back-end processing mechanisms that are available. The three that I had in mind were: 1) the NaCl executable, as in the original; 2) a `chroot`-ed binary; 3) docker containers (both fresh and recycled). I think it would be an interesting study to compare the security guarantees of each of these approaches and how it measures up to the performance benefits as well. Since the NaCl executables and the `chroot`-ed binaries don’t require extra orchestration to start running, my guess is that they would have similar performance levels, with the `chroot`-ed binary being faster due to the fact that it is running on the built-in OS and not the guest NaCl OS. And of course, a recycled docker container would be faster than a fresh one, but with the caveat that using recycled docker containers reveal certain cross-program security vulnerabilities (e.g. the ability for different programs to communicate with each other).