Group 20

Sean Hinchee

Sam Westerlund

Go^2 -- An exploration of multiplayer and concurrent concepts modeled within a web application.

Table of Contents

Overview

The following is a formal exposition describing the new frontiers and processes covered over the course of the development system for the first portfolio in regards to group twenty.

Initial design for the project was the goal to produce a web application, most likely a multiplayer game, incensed primarily by the example provided in class of the multiplayer connect-four game. Within the early phases of design, the base requirements were set that the end-product was of lesser concern than the concepts expressed therein. Concepts that were laid down as desirable were concurrent programming (multi-threading), inter-thread communication (channels), a web interface (JavaScript/HTML5), web-to-backend communication (HTML templates), and a simple, but effective design stratagem. Upon investigation into potential routes to explore to meet these desirable traits, the language Golang was devised as a strong candidate for both an additional learning hook, being a new language, and enabling all of the above concepts with minimal infrastructure and no external dependencies.

Chief among all frontiers covered within this process was the exploration and introduction to a new language, Golang. Golang, abbreviated to Go for short, is a young language produced by Google as an internal replacement for C++ for Google-specific systems that was (relatively) recently released for public use. The most recent version at the time of writing was the version used, 1.7.3, which was tested on both Windows (10, 64-bit) and GNU/Linux (Red Hat, 64-bit) with success and no deviation upon functionality.

Careful processes were involved with the synthesis of the project. Incremental development was utilized through use of a Git repository (private), contributing greatly to the overall success of the project. Asynchronous programming was leveraged and carefully analyzed through experimentation and meticulous debugging. The first web application the group had composed was explored, with evaluation over what specific features were desired as to not over-feature or over-complicate the task, leading to a design process focused around the core concepts and spirit of the project rather than specific catches. A variety of complications arose, from specific implementation of the Go (board game) rules, but a reasonable resolution that allotted more time to empirical goals over arbitrary specifics was employed. Although it was a difficult call to make, specific refinements of the frontend game experience were cut on principle due to time restrictions and their irrelevance to the learning experience.

New and Complex

Golang was approached as a complex hurdle to jump. Before committing to learning a new language, many comparisons must be made and differentiations established between the target language and viable alternatives. Golang after moderate dealings-with was established as a statically-typed, C-like, simplistic language that accentuated parts of coding styles that were preferable, such as a strict syntax and a consistent format.

Channels were a core feature utilized within Golang for the project. Although the HTTP server library utilized channels and concurrency in its heavy-lifting, a fleet of custom channels had to be set up to properly communicate between the various management and handler functions that were spinning on independent threads. The specific use of channels is treating the channel as a stream, or in the case of all of our custom channels, a buffered stream, with a custom statement to add structure to reading channels called *select* which acts as a switch-case statement upon a case-list of channels provided. In one case within the code, nested select statements are leveraged to create a sense of logical flow where a given signal down one channel sets the governor thread to listen then on another set of channels for more specific information. Calls to this specific structure are made inversely to create a pseudo-stack where the exact information is ensured to be in the child information channel's buffer when the signal is sent to the parent information channel.

Networking was done leveraging the *net/http* library within Golang that allowed the software to listen on a given port and set up handlers for specific patterns of requests to the HTTP server, simplifying hosting matters as a server such as Apache or NGINX is rendered unnecessary, making the application more standalone. Requests to the server are processed in the order they are received, asynchronously, generating a call to each handler upon reception, with the handler performing whatever actions are required, such channel communication, concurrently with the operation of the server, preventing slowdown.

The web application was entirely untrodden ground with the necessity to have the backend (written in Golang) connect with the frontend (written in JavaScript) and be able to respond to changes within the server and update accordingly. With a desire to bring in no further dependencies beyond Golang and network-capability, the limit was set at JavaScript and Golang and a solution was found within the *html/templates* library. The templating system allowed us to translate from a given HTML file to a served HTML file with certain syntax allowing the embedding of Golang variables to add more state to a naturally state-less web interface. Several JavaScript variables were populated by the Golang backend before being served to the user and processed by the JavaScript/HTML5 frontend. With the careful implementation of the API, adding a series of utilitarian requests that could be formulated to modify variables on the Go backend side, JavaScript was able to pass information to the backend.

The frontend (JS) took the injected variables into its code and processed them to be further interpreted and manipulated. The player is able to play and select pieces entirely locally without the page refreshing, with exception to removes, and multiplayer respectfully. The client would periodically query the server (once per second) for an updated string of pieces, and post its moves via http urls.

Bloom’s Taxonomy

Language choice played a strong role in the end result within the project, as Golang was a fresh language to the group, but bore semblances to C and Java that made the choice more approachable. Golang’s multi-threading was almost too simple to pass up, with functions being pushed to their own thread with a simple *go* prefix to the call. Judgements were made as to whether the frontend should exist in JavaScript, or yet another new language. The choice at the end of the day, with many comparisons and selections in the open for a web-programming language were made and found, but JavaScript won the ticket due to the potential value of possessing and, at minimum, primitive understanding of the language (JS). Supported by the choice of JavaScript, HTML5 made a strong joint choice to provide the visual aspects of the web application.

Asynchronous programming is a relatively new front to the group, with threads being a virgin topic first introduced within CS319. The related complications were many and required careful construction and design considerations to formulate a stable system that could operate consistently. Channels were an abstract concept introduced by Golang which has no known equivalent in Java to the group’s knowledge and required a fair bit of experimentation and explanation to properly clarify and appraise as a model.

Web applications were a further open ground topic, with a great deal of unknown territory to cover as neither group member had developed, from the ground up, a web application previously. Judgments had to be made in regards to the proposed scope of the frontend and backend, differentiating between various approaches that were possible, just as a Java backend, a PHP frontend, etc. Examinations were made in regards to the specific portions of the project with a handful of tests being made to see to what degree a given solution was considered viable or productive in regards to time. Golang proved to be an exceedingly productive language to code in, allowing complexly modelled systems to be implemented in a fraction of the time one would expect a similar task to take within Java, for example. JavaScript, once leveraged against the backend API, proved to be less of a time sink than expected and the project proceeded with a rapidity that allowed further exploration into the actual web application development.

Various complications were had on the road to success with the backend being a world of unknowns and the frontend requiring in depth testing and a stable backend to properly begin expansion of scope for the complexity of the project. A backend API was formulated and summarily written as a manageable layer to communicate to the backend from the frontend. Requests to move, for example, would be structured as */move/0513b020* which would then be parsed to place a black piece at X value 5, Y value 13 for game instance/ID 20 (An extensive documentation attempt for the API is provided within API.md in the root source directory). This same system allowed for multiplayer, with board manipulations and ultimately the board state being reduced down to a string or array. Each piece place was processed on the client side. If a removal case was found the client would send a single piece remove case to the server. And if multiple remove cases were found it would it would respectively do the same. After a broad series of examinations, the conclusion was reached to break down specific API requests into more isolated, smaller, HTTP requests to avoid the overloading of a specific request structure, such as adding more potential information to */move/*'s syntax. In retrospect sending a whole encoded string of the pieces would have been best but variable types are interpreted differently between Golang and JS.

Furthermore, issues persisted in ensuring that the requests were processed in the correct order, leading to careful considerations about concurrent systems with judgments and peer-evaluations being had on the exact choice of structure or composition for a given block of code or extension upon the backend. One function being out of sync with the others would lead to the dysfunction of the system as a whole, breaking the backend, bringing the frontend down as well.

JavaScript presented complications of its own as well as the actual rules to the chosen game, Go. Due to the complex rules of Go the game, the rules had to be simplified to retain the scope and essence of this project. For example, in Go the game, the rules do not allow pieces to be placed in the same spot to repeat a board state. That specific rule would call for the storage of game states and ultimately extra bandwidth between client and server especially during a browser exit and rejoin. Removal of pieces was calculated using DFS and looking if any of the connecting pieces has any liberties (open spaces). Another modification was made that did not allow pieces to be placed in spots with 0 liberties. That modification fixed the previous issue and was implemented because it also fixed problems with removal of pieces that next turn would have more than 0 liberties.

In evaluation a more efficient and secure game would have been written almost entirely in Golang on the server side. The program is very “hackable” as it lies, because of its high reliance on client side processing exemplified in piece removal. The decision was ultimately made to design most of the game mechanics client side because writing most of the game in Golang server side would have disabled our exploration of client and server communication and processing and why it matters.