Team Apollo

Packagebird

Elisha Aguilera, Setenay Guner, Denish Oleke

Client and Mentor - Brandon Busby

Instructor - Dr. Bolong Zeng

CPT_S 423

Submitted 02/04/2022

Updates to project functionality, overall system, and system components

As an update to the previous iteration of the project, the server now uses MongoDB for data persistence such as the storage and retrieval of package file-paths and associated metadata. This was changed from the previous version of the server, which instead used the SQLite database with a local database file for data persistence.

The benefits of this change are a more flexible ‘schema’ when interacting with the database; in addition, the MongoDB document object translates into a Go language structure near seamlessly, simplifying the amount of implementation needed to interact with and save data. The detriment of a document-based model is handling related units of data; our solution is to store address strings in arrays embedded within the document object. This solution may not be ideal in some circumstances, but for this project it allows us flexibility and code reuse.

This modification has also affected the setup and configuration process for a fresh server, which now requires the installation and setup of a local MongoDB service before starting the server. As this project will provide deliverables for both Windows and Linux desktop platforms, two different setup processes are required. For the Linux platform, a simple installation script should suffice, whereas Windows will require manual configuration from a user. The server looks for a configuration file in the directory. It is launched from file-system paths where the project binaries are stored, along with connection addresses for the local database.

Packages are now associated with projects, conceptualized as ‘packages in a volatile state’ which incorporate changes and adjustments until a package is created from the project. This package is then registered, the project version is incremented, and the next iteration of development for the project begins. Conceptually, packages are considered snapshots of projects, non-mutable modules of code or binaries to be distributed and utilized.

Project members are now incorporated into the project; originally considered but not fully implemented in the Alpha prototype, members are associated with users in the system, and depending on their assigned privilege levels have access to functionality such as deleting packages and projects, renaming projects, adding and removing team members from a project, and reserved for the highest level of privilege, resetting the server to a clean state.

Test and validation plans for Beta Prototype

To facilitate the testing phase of development towards the end of this semester, testing modules will be implemented. The client uses Pytest for unit testing, the provided test module in the Click framework for testing commands in whole.

Test will be conducted in stages in accordance with convention: unit test will be the initial battery applied towards both the client and the server components. Units are traditionally defined as being constrained to isolated methods, but in the case of our project, certain functions exchange information over gRPC. To account for this, on the client we use Pytest-GRPC to simulate communications over a network.

The companion server component is implemented in the Go language and will use the standard “testing” package for unit-testing, we can directly test certain gRPC services utilizing this, and in addition use the “Testify” framework for mocking objects; in addition, it will use the “grpc/test/buffconn” package to intercept and control for networked remote procedure calls.

Our testing plan for the server is creating listeners on a local network. When an insecure connection is made between the listener and a client, we will keep track of how our server responds. When the server starts and is open to receiving requests, we can keep track of processes in between. This gives us a great breakpoint to test communication between before we even start certain aspects of the server. We can then test out of range, and boundaries.

One of the reasons we chose gRPC was bidirectional streaming, we can utilize streaming messages and server response to test this feature. While doing this we believe we can also test multiple requests, and stream calls. Other than looking at messages we can also look at the payload that is coming as well, which could be a little trickier.

Testing of the 'user-interface' will be a bit trickier; as of now the current plan is to recruit a volunteer to perform some outlined tasks using the command line application interface, such as creating a member account, joining a project, adding and removing packages from a project, modifying the source of a project, and creating a package from the project. The goal isn't testing the functionality directly, but to see if the user can operate the tool with the assistance of the help-command, the manual, and the illustrative command names.

Test Type	*Description*	*Testing*	*Expected Result*	*Status*
Functionality	Downloading packages	Utilizing Clumsy to worsen network environment while installing packages 500 megabytes to a GB	Transferring a GB file should take 14 seconds. Assuming bandwidth is 10mbit/s and packet loss is 0.0001%.	Pass or Fail
Functionality	Add packages	Adding a new package with many dependencies to the DB.	DB structure is unaffected, and memory is still efficient.	Pass or Fail
Functionality	Command Line Interface	Testing commands in Click framework.	Provide a test module for each command and monitor results.	Pass or Fail
Usability	User interaction with command line	Run a functionality and usability test on a client and server hosted on a local Windows desktop with a user unfamiliar with the tool.	User can create, add, remove and alter packages and projects using available guidance material	Percentage of features marked as usable or unusable

Table 1-1: Some test cases that are applied against the client and the server.

Test Data Collection, Generation Plans

For the client, 'Coverage.py' will be used to generate line-coverage reports, in addition to the test output dialog messages to generate coverage and specific output data for review and analysis later.

For the server, the Go language additional toolset has a coverage-test runner, specifically under the 'cover' package. Like other standard line-coverage runners, it will generate reports which will be combined with test output dialog messages for later review and analysis.

The network operations associated with scenarios outlined in the next section will be initiated with a custom Python control script initiating request on the client, the server will log when the request is received and the response is delivered, and the client will log when it received the server's response. These values will be used to compute the latencies associated with each operation, and in addition with the constraints applied in Clumsy or genuine network conditions associated with the VPS remote server, an average latency from 8 test can be associated with each operation. If this average latency is exceptionally large, a particular case being over 10 minutes to transmit a file less than 500 megabytes, this will be considered invalid and marked for review.

For the user interface testing, a volunteer will be asked to perform a series of task without being explicitly told how to use the client. A team member will monitor and take notes on the process. If a particular task is too confusing and the user cannot complete it with assistance from the help-command and documentation, the team member will note the failing user interface feature and adjustments will be incorporated into later design.

Testing Scenarios, Validation

To test operations that are conducted over a network, which are involved in most of the features provided by our project, several scenarios are outlined in the paragraphs below. Each of these scenarios will have a battery of operations initiated: downloading packages of various sizes ranging from less than 500 megabytes to a single gigabyte in size, adding packages with many dependencies, uploading packages of various sizes ranging from 500 megabytes to a single gigabyte, listing the packages and members registered with the server.

To test the application's performance in varying network conditions, both the client and server components will run on a single Windows desktop system and communicate over the localhost connection. To simulate worsening network conditions, the desktop application Clumsy will be running, allowing for incremental increases in latency, packet jitter, and dropped packets. This is a synthetic simulation of volatile environments.

To test the application's performance over distance while mimicking real-world environments, we will run the server component of Packagebird on a remote virtual private server provider Vultr, located in Los Angeles, California with an approximate average roundtrip latency of 38 milliseconds when pinged 64 times from our team liaison's residence in Sultan, Washington using the Windows PowerShell Ping utility. This is most like a long-distance arrangement between a company and its employees. We expect actual geographical distances between the server and the clients to generally be shorter, most commonly on-site arrangements.

Plans for Analysis, Visualization of Test Results

For unit testing the components, the basic metric measured will be line coverage, at a minimum of 90% coverage and targeting 100% coverage. These metrics will be used for verification of functionality in addition to providing assurance to the client that the product has been tested and evaluated in an objective fashion. The data collected will be organized and presented in a table of items covered, to be shipped along with the rest of the project deliverables in a section of the developer documentation.

The data collected from network testing scenarios can be plotted into bar graphs, with columns associated with operations and network conditions; we project that as network conditions worsen, the performance of network operations will also worsen, however operations that don't transmit large units of memory should degrade at a slower rate than operations involving transmitting large files. Should a test reveal a situation wherein the conditions are ideal, but the operation remains slow, that is a particular operation that will need modification and error fixes, and in the worst case the feature may need to be reconsidered and adjusted.

The above data will be collected into Excel sheets, which will also be used to generate any graphical representations of the data. These graphical representations, along with partial tabular snippets of the data, will be included in the later documentation submitted for this course.

Summary of Work Remaining in Semester

Features that need to be fully implemented are the proper handling of packages being removed from a project, server administrative features being connected to and implemented in the client, building and testing select packages on the server, documentation being extended atop throughout each step of development. As indicated in the previous sections, the testing phase of development will begin towards the end of March given the pace of development.

Building and testing packages will be a tricky feature to implement, though it remains as a priority as the client has maintained its importance to the final deliverable. Because the tool is language agnostic, each package may contain code written in virtually any programming language, and therefore custom compilers will need to be uploaded and configured by system administrators. We envision that a few compilers and build tools can be shipped by default with the server in the Beta Prototype, and if enough time remains prior to submission a system of uploading and configuring a custom compiler may be implemented.

Throughout the development of features, concurrently alongside the features will be entries written in the project documentation, with sections for usage and deployment and sections for developers and future modifications. As implied previously, this process will occur throughout the development of the project until the Final Beta Prototype is submitted.

Should there be enough time, beyond the above major features we have concepts in place for a web interface for browsing existing packages, projects and members attached to the projects. This is placed in a secondary status as it does not directly fulfill the client's specified needs but having discussed the concept with all parties would be considered a beneficial feature for onboarding new users and navigating a company's active projects. In addition, we are leaving buffer time in place should more items need be addressed prior to the final deadline. A general outline of the scheduled task can be found in Figure 1-2 in Appendix A, a screenshot of a Jira Gantt chart hosted on a board.

Appendixes

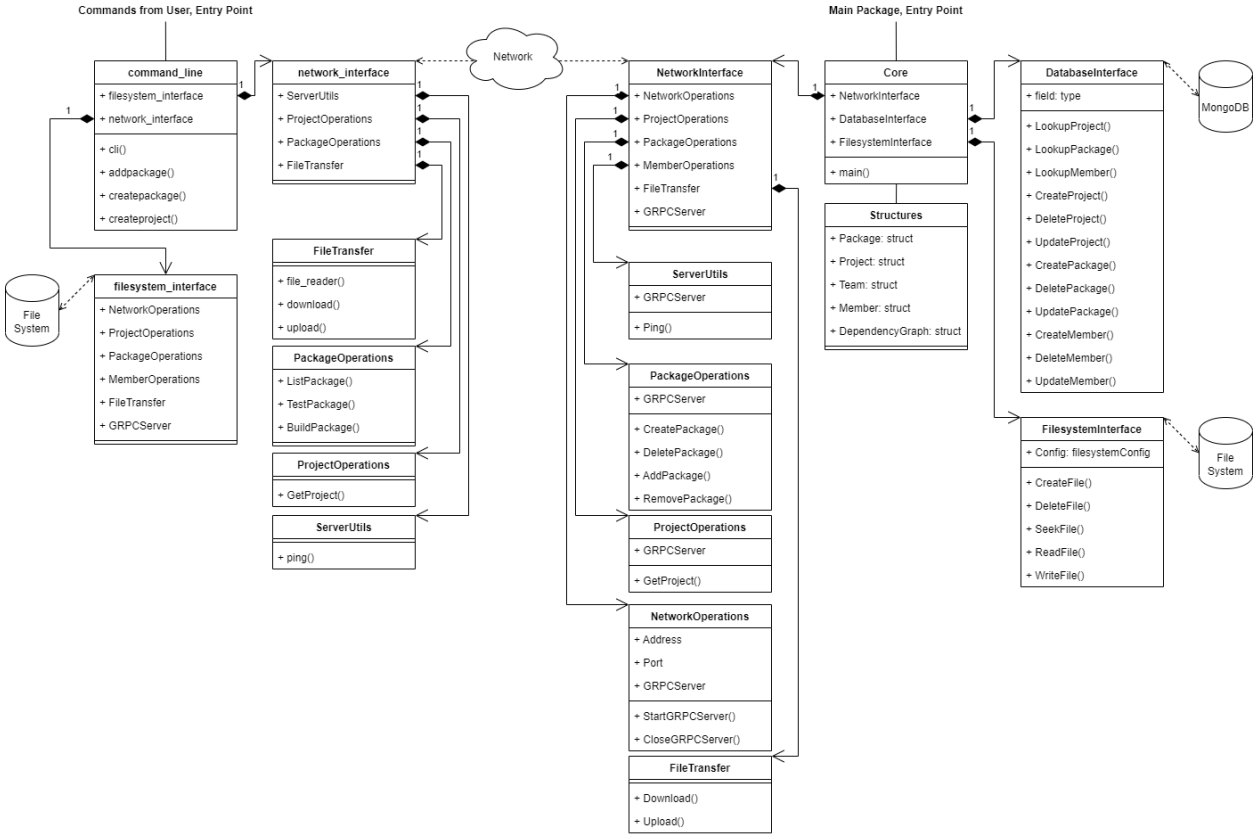

Figure 1-1: Class Diagram with features and network exchanges grouped into 'objects' (Go does not officially support objects, concept is loosely translated into packages).

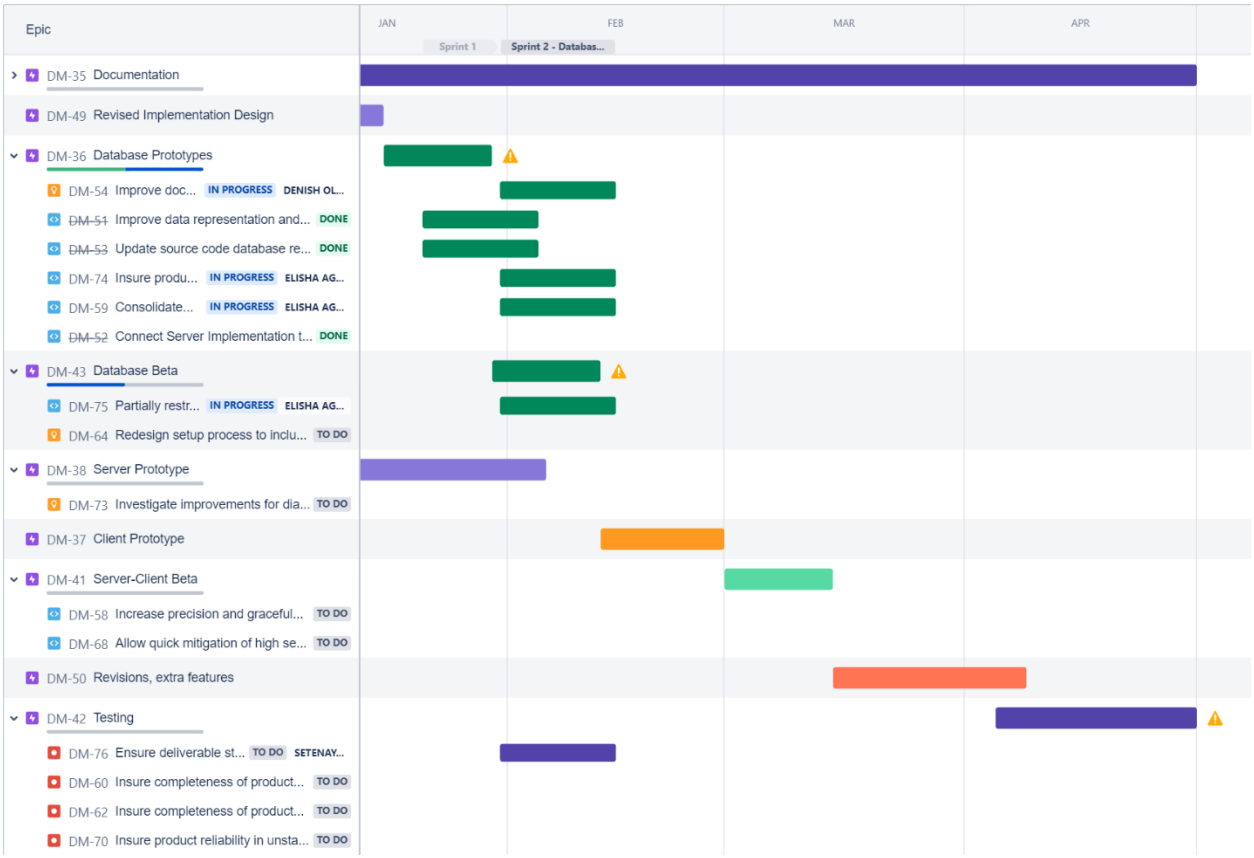

Figure 1-2: Gantt Chart stages of development organized into Epics with Sprints roughly aligned in synch; The change from the existing Gantt chart is incorporating portions of unit test implementation earlier in the project timeline.