Packagebird Solution Approach

Cpt_S 421 – Dr. Bolong Zeng

Mentor Brandon Busby

Elisha Aguilera, Setenay Guner, Denish Oleke

2021-11-19

Product Solution - 1

The product is broken into several components:

> The Client - a front-end executable which serves as the interface between a user, the user's machine, and the remote registry.
> The Server – a back-end server interface handling request from various clients, performing evaluation and lookup functionality, and directing files to and from the system over the network to the remote clients.
> The Registry – a database which acts as a metadata address book for existing packages contained in the system.
> Projects – a continuous ledger of bundled packages considered dependencies for the system.
> Packages – containerized formats for storing and accessing source files, often code language agnostic libraries.

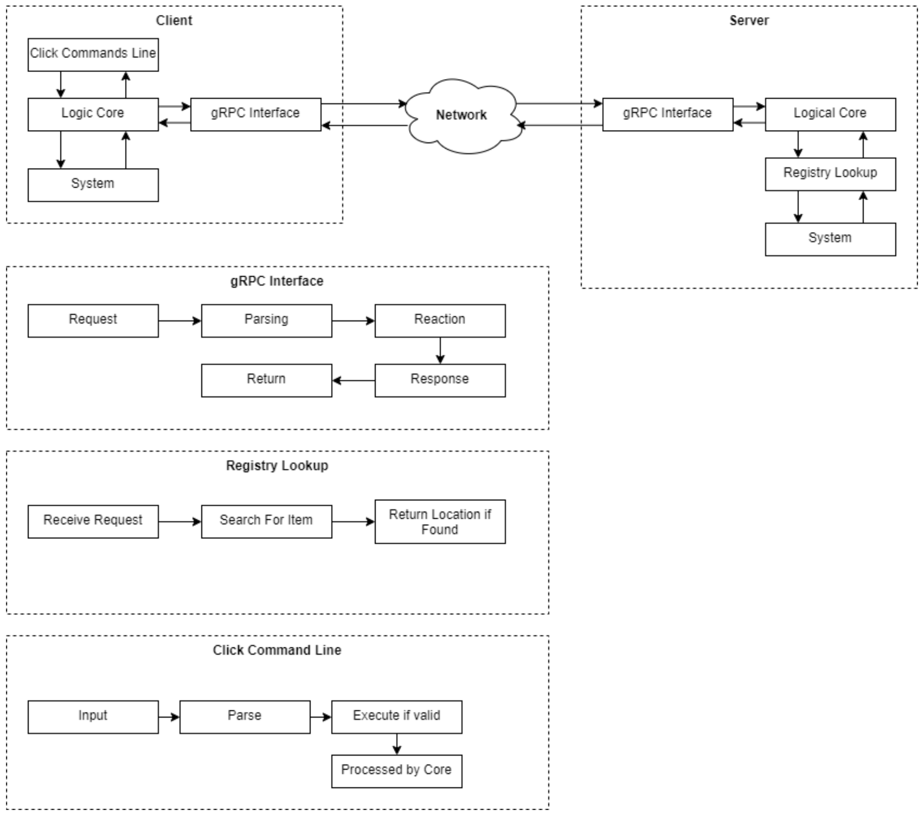

Figure 1.0 – Process Diagram, Components and Subcomponents exchange as illustrated.

Two primary processes are envisioned for this tool: the configuration of a developer's environment, and the creation and registration of packages. These processes are envisioned to execute in linear fashion, as there is no foreseeable circumstance where a user would configure a development environment while packaging it in the same step.

Secondary processes include browsing listed projects and packages registered in the system for research purposes, like other existing package managers.

Examples of package manages similar to our project can be found in the *Package Management Systems* [1], and while the field is at the moment pretty solidified, there are still attempts to improve the concept such as the *OPIUM: Optimal Package Install/Uninstall Manager* [3] project. There are still many challenges involving package managers, including the handling of dependencies, as described in *Dependency Solving Is Still Hard, but We Are Getting Better at It*[2].

For our project, the focus is on simplicity and ease of use, from setup and configuration to ongoing use. Deploying the server should be straightforward and simple, running a configuration executable to point at the appropriate file directory, and initiating. Clients setup on remote machines can quickly dial-in, configure and begin pulling packages from projects defined on the server.

Design Choices – 2

Criteria were chosen based on the customer needs we identified after discussion with the stakeholders and the team and the needs of keeping the minimal risk of liability. Our highest priority is to transfer packages between the server and the client in a short amount of time in the safest way possible.

Selection Criteria	**Concepts**			
	HTTP (ref)	RPC	gRPC	REST
Efficiency of execution	0	+	+	0
Data transfer	0	+	+	0
Automation	0	-	+	-
Latency	0	0	+	-
Request Handling	0	0	+	-
Browser Support	0	0	0	+
Sum of +'s	0	2	5	1
Sum of 0's	6	3	1	2
Sum of -'s	0	1	0	3
Net Score	0	1	5	-2
Rank	0	2	1	3
Continue?	Combine	Combine	Yes	No

Table 2.1 concept-screening matrix

This concept screening shows the different options we considered while building the server. HTTP represents a primary functioning HTTP server. Others use HTTP API calls to communicate with the client while utilizing other aspects that we later found crucial to our system.

REST API uses resources from the server to provide clients. It utilizes many HTTP methods to perform CRUD operations.

A simple HTTP server, which we use as a reference concept, could use all the methods it offers, but it lacks certain elements.
RPC is a distributed system; when the client calls a stub with the same signature as the function in the remote server, the message is encoded and passed to pass back a response within the server as quickly as possible.

Transferring large files over HTTP can be an issue. While all the concepts we defined above could make use of different file transfer protocols when transferring large files, such as dependency packages, REST falls behind. While REST API supports 2GB chunks of files, the browser memory cannot and the connection between the client and server is not as efficient as RPC.

		HTTP (ref)		**RPC**		**gRPC**	
	Weight	Rate	Weighted Score	Rate	Weighted Score	Rate	Weighted Score
Efficiency of execution	20%	3	0.6	4	0.8	4	0.8
Data transfer	25%	3	0.75	3	0.75	4	1
Automation	15%	2	0.3	2	0.3	5	0.75
Latency	10%	3	0.3	3	0.3	3	0.3
Request Handling	20%	3	0.6	3	0.3	4	0.8
Browser Support	10%	3	0.3	3	0.3	3	0.3
Net Score		17	2.85	18	2.75	23	3.95
Rank		3		2		1	
Continue?		No		No		Yes	

Table 2.2 concept-screening matrix

RPC gives us the flexibility of implementing the server in many ways while still using HTTP transport protocol. However, we do not have to specify the details for the remote interaction. We write the same code as if the subroutine is local. While we are receiving data on the server it needs to have the name of the function and the list of parameters being passed to that function from the client.

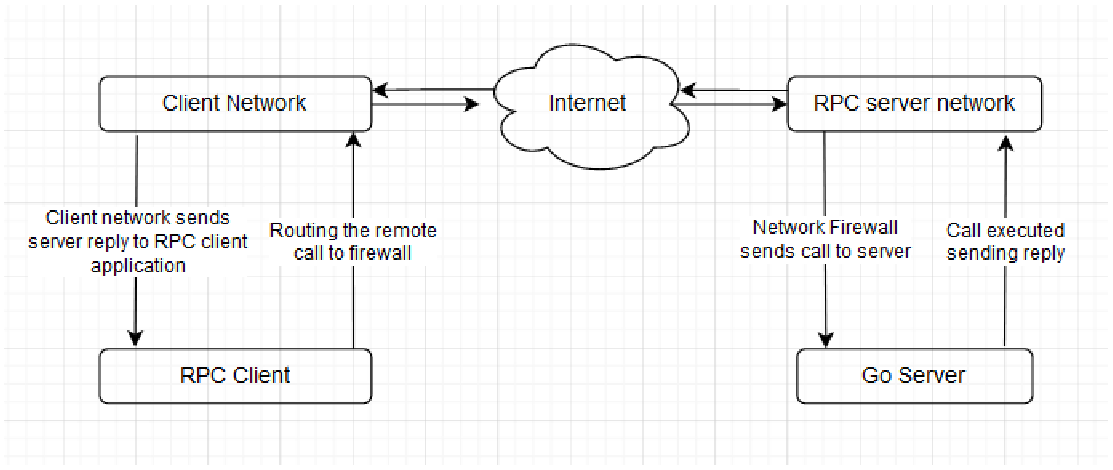
Client Network
Internet
RPC server network
Client network sends server reply to RPC client application
Routing the remote call to firewall
Network Firewall sends call to server
Call executed sending reply
RPC Client
Go Server

Figure 2.1 showing RPC sequence of events

Since we are building a Command Line Interface, we decided to implement the client using Python since scripting and creating executables are easier and the team has previous experience with it.

However, implementing RPC within the Python client was challenging and after discussions with the team, gRPC seemed like the better option. It is language agnostic and offers more along with the base procedure calls. It follows a client-response model of communication which allows bidirectional communication. It separates the client and the server streams apart and they both can transmit messages in any order. This means multiple clients can request multiple calls from the server. Since we are building an app for developer projects this was a decision factor.

gRPC uses Protocol Buffer to serialize the payload data instead of relying on JSON or XML files. The proto is lighter in size and it is binary. This is useful when server and client languages are different from each other (Figure 2.4). It allows us to define our own data format within a *.proto* file and it follows a simple Java-like syntax.

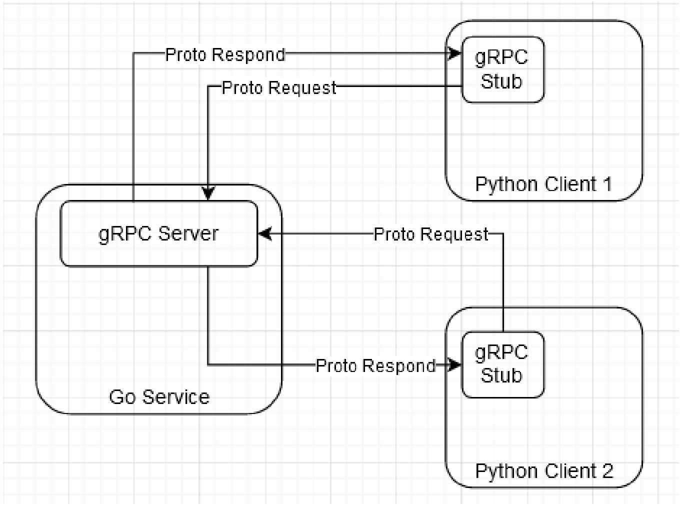

Figure 2.2 Go service communicating with Python clients.

As shown above, the Go server and our Python client can easily communicate using Protobuf call methods. This communication allows us to stream files between two endpoints as well.

Since packages can be big in sizes Protobuf allows us to split the file into small multiple chunks to get across (Figure 2.5).

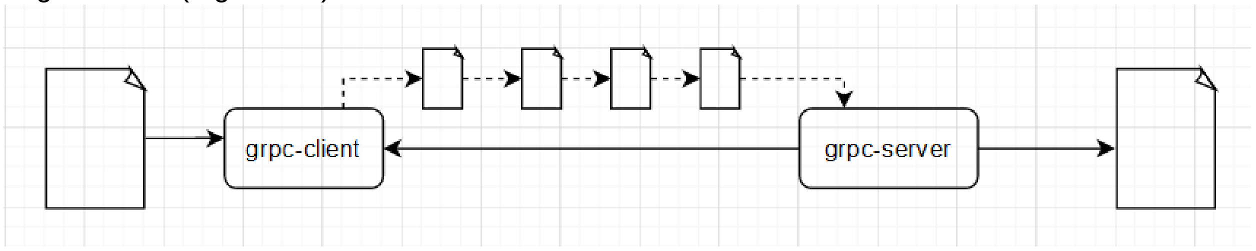

Figure 2.3 file transfer between client and server

The coordination of file transformation is complicated; however, Protobuf's auto code generation is a helpful tool. Boilerplate code creates objects that represent the gRPC API defined in the *.proto* file. These objects contain serialization and deserialization logic to decode data on client's machine from a Proto Binary. Overall, it gives us a fast and reliable file transferring between server and multiple clients.

Implementation Framework – 3

Considering the design choice and the net weighted scores, the team decided to set out the server implementation framework to rely on the gRPC framework which we hope its library will offer us a more easy and highly scalable high performance server client system. Much as we have not yet laid out the gadget/device specifications, we do believe that once the system is fully running it should be able to run on most of the machines that may support the legacy operating system. However, as of now, our client – server system has no restrictions since it is only functional on the local machines but once we scale it to run remotely, we will decide which operating system and versions the project will require at most or bare minimum.

Component	**Subcomponent**	**Source**
Server	Network Request Handler	gRPC Go [6]
	Core Logic	Go Standard Library [7]
	File System Formatter	Go Standard Library [7]
Client	Command-line Handler	Click Framework [4]
	Network Handler	gRPC Python [5]
	File System Formatter	Python Standard Library [8]

Table 3.1 – Components, subcomponents and sources

Preliminary Testing – 4
With all the functionalities in place, Packagebird will be tested at least in two distinct stages. By utilizing the built-in unit test in the GoLang, we will test for the server performance “any specific components if required” of which will be followed by testing for bugs.

Feature	Testing Plan	Target Date
Server	Input: • Service protocols	February -16 - 2022
Client	Input: • Request for a package	February -23 - 2022

Table 4.1- Preliminary Test Table

Project Status – 5
Like all projects, forecasts of the original timeline were unfortunately overly optimistic on progress; as such, the timelines have been pushed back and items have been rearranged, primarily due to unforeseen research being required.

EPIC	SEPTEMBER				OCTOBER				NOVEMBER				DECEMBER		
	WEEK 1	WEEK 2	WEEK 3	WEEK 4	WEEK 5	WEEK 6	WEEK 7	WEEK 8	WEEK 9	WEEK 10	WEEK 11	WEEK 12	WEEK 13	WEEK 14	WEEK 15
Documentation	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧
Conceptual Specification		🟧	🟧	🟧	🟧										
Technical Specification			🟧	🟧	🟧	🟧	🟧								
Client Prototype Implementation								🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧
Server Prototype Implementation											🟧	🟧	🟧	🟧	🟧
Dependency Graph Visualization Prototype													🟧	🟧	🟧
Dependency Graph Visualization Alpha Demo															🟧

Table 5.1 – Updated Project Schedule for Semester Reminder

The general project steps can be split into several stages, the first of which is a prototype implementation of packages being exchanged between the client and server, referred to as the proof-of-concept stage. Progress past that, additional functionality such as user authentication with corresponding privileges, a web interface for navigating listed projects and packages, along with other potential features can be added. Finally, the project will enter a testing and solidification stage, where various test cases will be applied to different units and features throughout the project.

References – 6

[1] D. Spinellis, "Package Management Systems," in IEEE Software, vol. 29, no. 2, pp. 84-86, March-April 2012, doi: 10.1109/MS.2012.38.

[2] P. Abate, R. Di Cosmo, G. Gousios and S. Zacchiroli, "Dependency Solving Is Still Hard, but We Are Getting Better at It," 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER), 2020, pp. 547-551, doi: 10.1109/SANER48275.2020.9054837.

[3] C. Tucker, D. Shuffelton, R. Jhala and S. Lerner, "OPIUM: Optimal Package Install/Uninstall Manager," 29th International Conference on Software Engineering (ICSE'07), 2007, pp. 178-188, doi: 10.1109/ICSE.2007.59.

[4] Welcome to Click — Click Documentation (8.0.x). (n.d.). Click Project. Retrieved November 19, 2021, from https://click.palletsprojects.com/en/8.0.x/

[5] Python. (n.d.). GRPC Python Language Documentation. Retrieved November 19, 2021, from https://www.grpc.io/docs/languages/python/

[6] Go. (n.d.). GRPC Go Language Documentation. Retrieved November 19, 2021, from https://grpc.io/docs/languages/go/

[7] Standard library - pkg.go.dev. (n.d.). Go Standard Library Documentation. Retrieved November 19, 2021, from https://pkg.go.dev/std

[8] The Python Standard Library — Python 3.10.0 documentation. (n.d.). Python Standard Library Documentation. Retrieved November 19, 2021, from https://docs.python.org/3/library/index.html