

Package-Bird Technical Specifications and Requirements

CPTS 421 – Dr. Bolong Zeng

Team Apollo Members: Elisha Aguilera, Setenay Guner, Denish Oleke

Mentor: Brandon Busby

Date Submitted: 2021-10-29

1.0 Client and Stakeholder Requirements and Needs

The client's specific needs, extending to potential stakeholders, is outlined in the Table 1 and Table 2 within the subsections below. It should be noted that due to the nature of the client's needs and resources available to this team, these requirements are subject to change as the project progresses.

1.1 Functional Requirements

Item Identifier	Item Description
FR01	Transmit Packages (binaries, source code) from selected central server to client machine.
FR02	Configure client development directory such that packages can be accessed within the directory.
FR03	Create packages on client machine and register them with the central server.
FR04	Authenticate and manage user privileges when accessing the central server.
FR05	Maintain version history of packages to insure package consistency.

Table 1: Functional Requirements

Note that items are ordered by importance to client's needs,

1.2 Non-functional Requirements

Item Identifier	Item Description
NR01	Ensure all functional requirements can be completed within a reasonable time.
NR02	The client facing command-line-interface is simple to understand and operate.
NR03	Deployment of client and server on separate machines requires minimal configuration.
NR04	Integrates alongside existing workflows potential users are familiar with.
NR05	Is not resource intensive.

Table 2: Non-functional Requirements

2.0 Storyboard

2.1 Primary Scenario

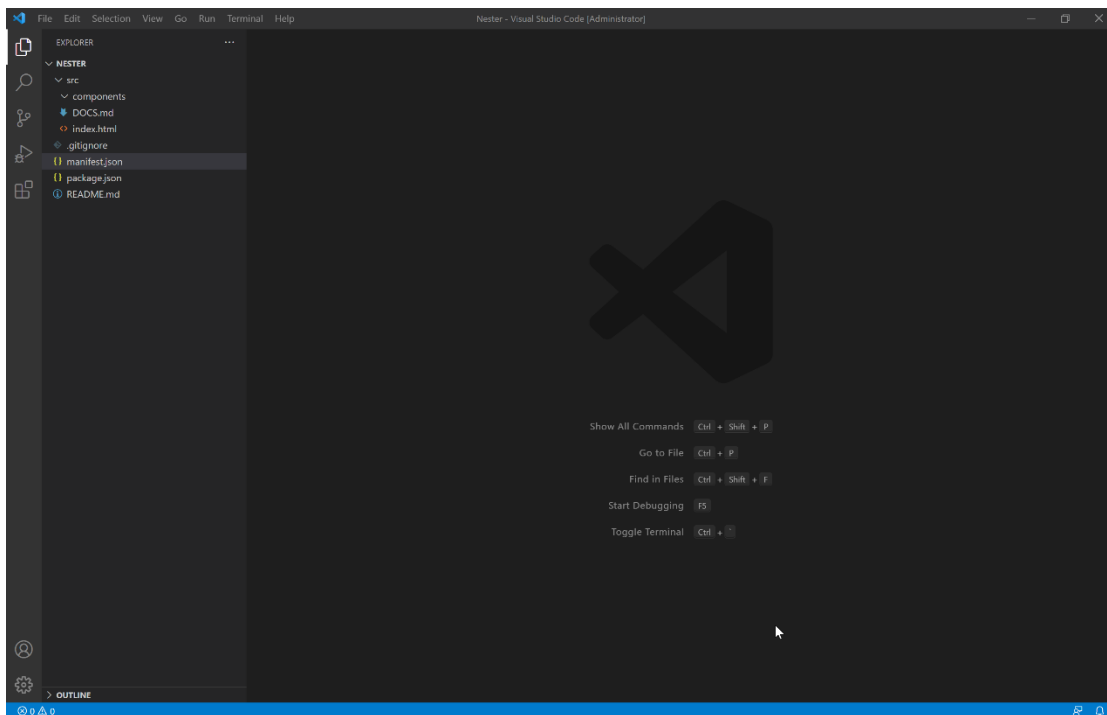
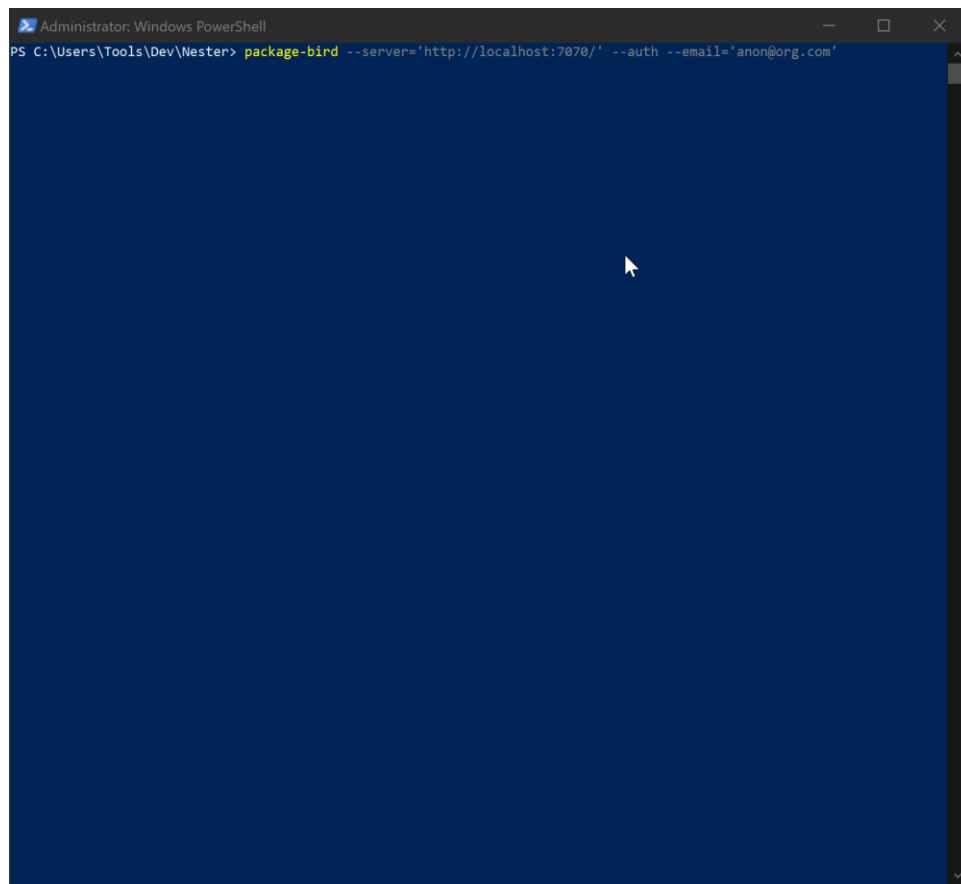


Figure 2-1: Overview of starting developer environment

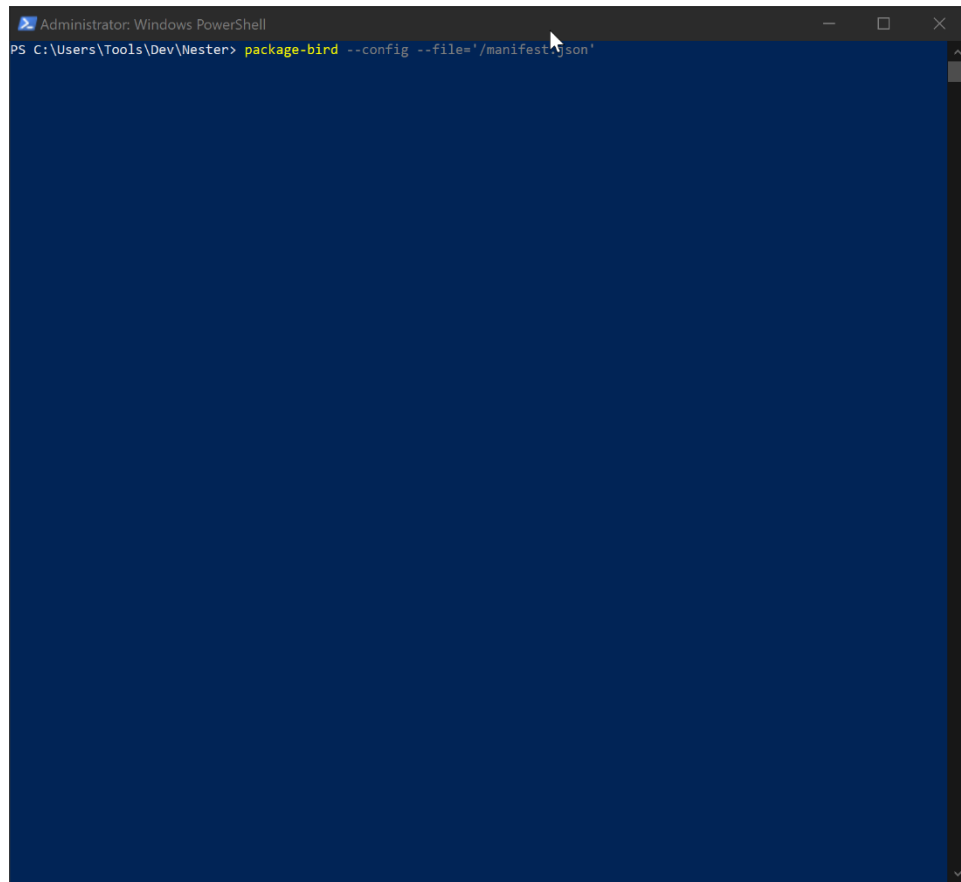
Anon is a software developer working for a firm, assigned to a new project. Like many projects, many external dependencies are involved, and the original project source is distributed without these external dependencies to save on bandwidth during push and pull operations.

A screenshot of a Windows PowerShell terminal window. The title bar at the top reads "Administrator: Windows PowerShell". The command prompt shows the user is in the directory "C:\Users\Tools\Dev\Nester". The command entered is "package-bird --server='http://localhost:7070/' --auth --email='anon@org.com'". The terminal background is dark blue, and the text is white. A mouse cursor is visible over the command.

```
Administrator: Windows PowerShell
PS C:\Users\Tools\Dev\Nester> package-bird --server='http://localhost:7070/' --auth --email='anon@org.com'
```

Figure 2-2: Command line interface authentication command

To initiate the process, Anon first authenticates with the centralized package registry server. Depending on their position within the company, certain privileges may be granted to their account.

A screenshot of a Windows PowerShell terminal window. The title bar at the top reads "Administrator: Windows PowerShell". The command prompt shows the path "C:\Users\Tools\Dev\Nester>" followed by the command "package-bird --config --file='/manifest.json'". The terminal background is dark blue, and the text is white. A mouse cursor is visible over the command.

```
Administrator: Windows PowerShell
PS C:\Users\Tools\Dev\Nester> package-bird --config --file='/manifest.json'
```

Figure 2-3: Command line interface configuration command

Once the authentication is complete, Anon calls the 'package-bird --config' command and points to the manifest file passed with the original source. This initiates a request to the server; the passed manifest only contains references to necessary packages, which themselves may have further dependencies. The process of linking all needed dependencies is conducted within the server.

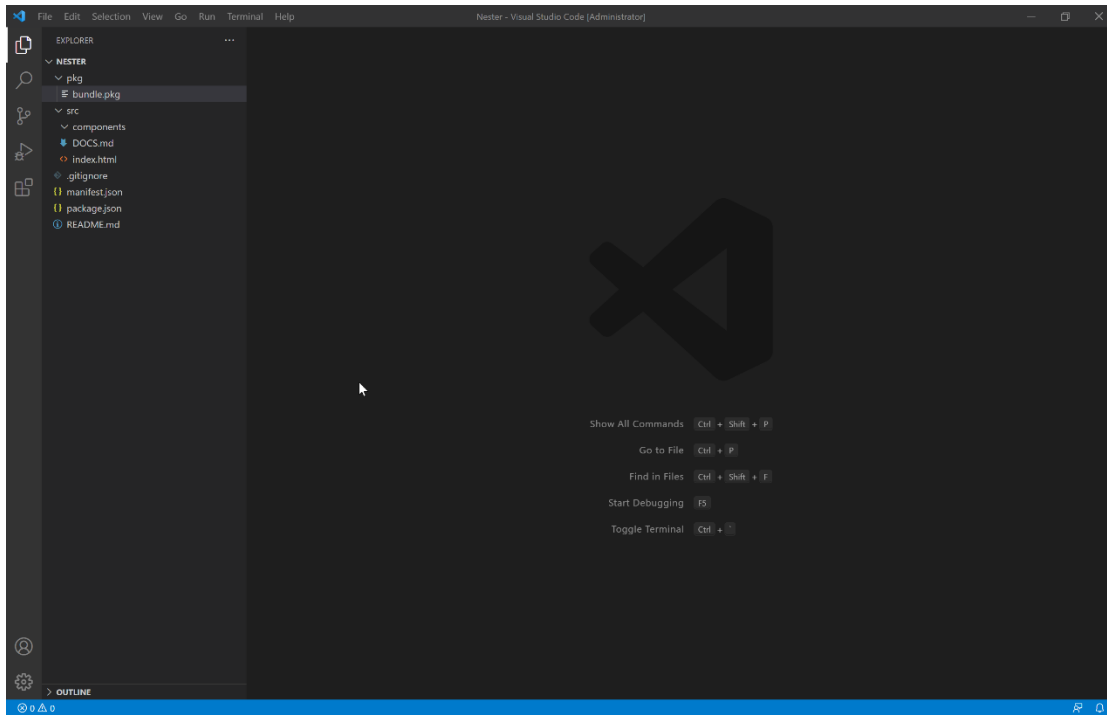


Figure 2-4: Developer environment after configuration

Once complete, the server transmits a compressed bundle of packages back to the client; the client then unbundles and configures the packages in the `./pkg` directory. Whatever dependencies are needed can now be referenced in the `./pkg`.

2.2 Secondary Scenario

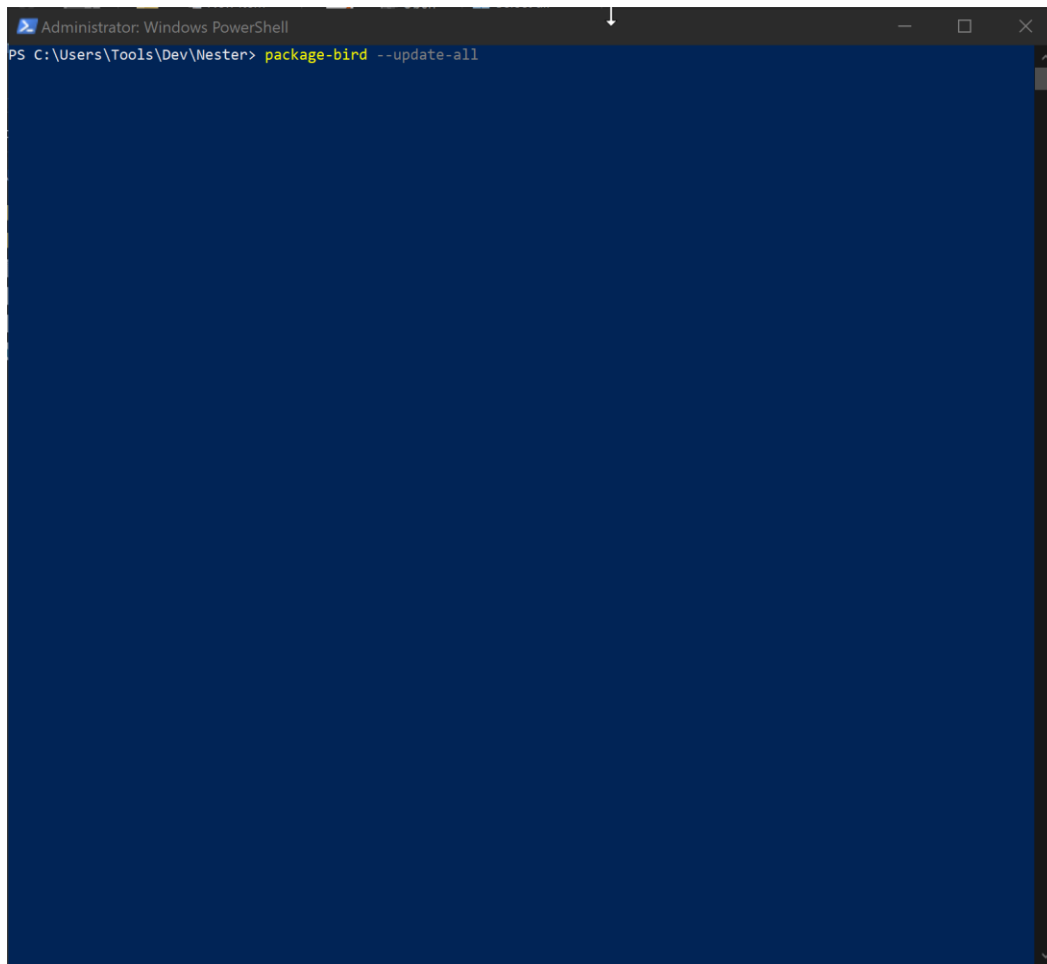


Figure 2-5: Command line interface initiating the update process

During the development process, Anon is notified that a referenced external package has been updated and the new version should seamlessly integrate with the current project. They initiate the ‘—update’ command.

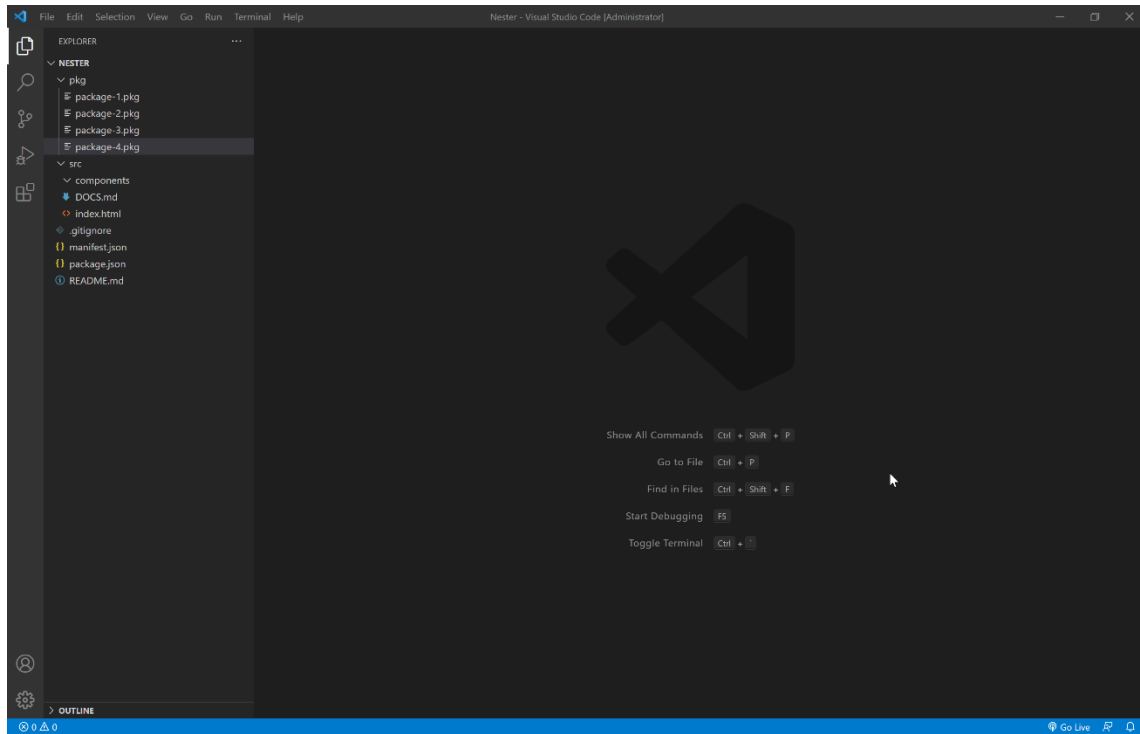


Figure 2-6: Package directory altered

Once the new packages have been transmitted and configured, Anon notices that the newly updated package breaks certain functionality, and to fix the compatibility issues would require significant rewrites of the source code. Anon's team determines that rolling back to the previous version of the dependency is the better option.

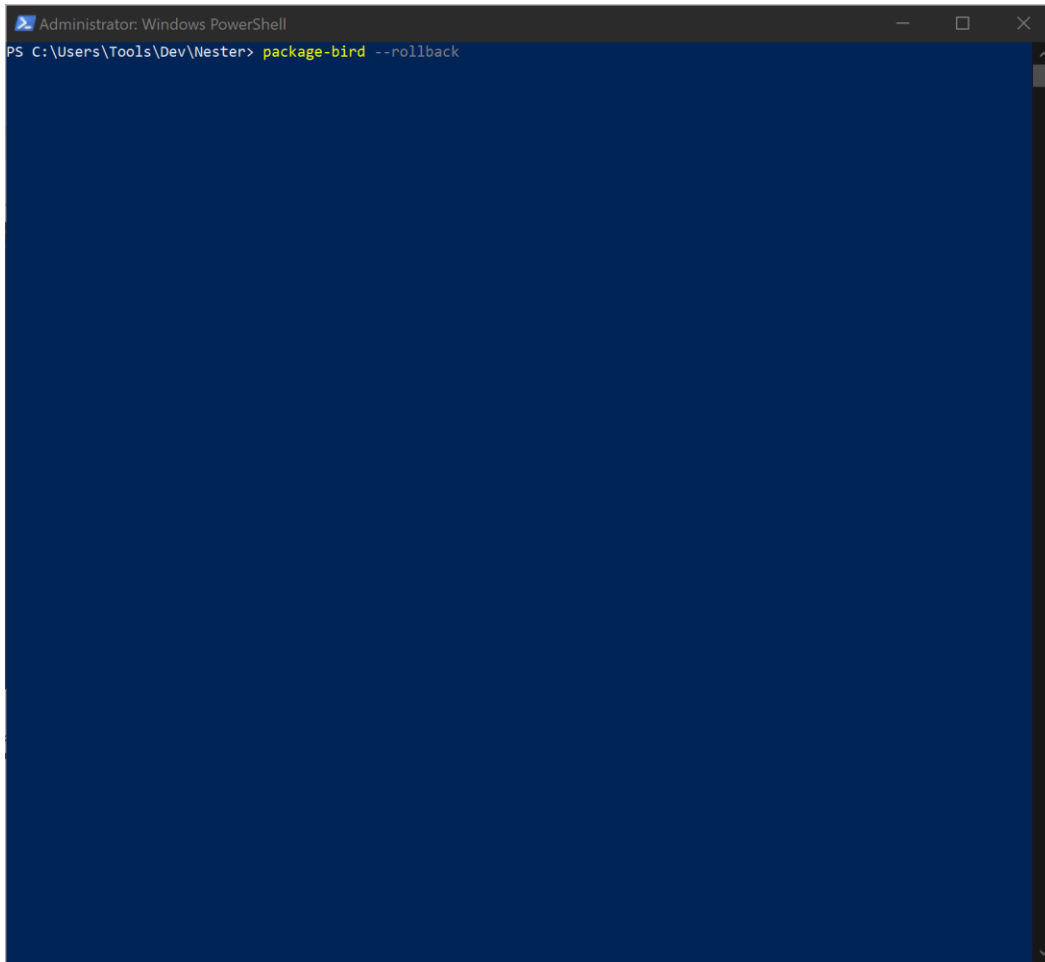


Figure 2-7: Command line interface interface initiating the rollback process

After the decision is reached, Anon initiates the '—rollback' command, which returns to the immediately previous configuration of packages. After the process is complete, Anon resumes development.

3.0 Mapping Requirements to Technical Specifications

3.1 Stakeholder Needs

	Needs	Metrics
1	System hardware requirements	Packages can run on user's current version of Windows, Linux or MacOS.
2	Server availability	Central server responds less than 3 seconds.
3	File server availability	File server and central server are accessible by client.
4	Space requirements	User has enough space on their drive for the dependent package files.
5	Easy CLI commands	It takes 10 minutes to read and understand CLI commands.
6	Maintain package version history	Manifest file updates and keeps track of the dependencies.
7	Add dependencies	User is able to add and edit with minimal configuration.
8	Delete dependencies	Authenticated developers are able to delete dependencies with minimal commands.
9	Server configuration	Server is easily configured and adjusted.

Table 3-1: Stakeholder Needs table

3.2 Metric Targets

ID	Metrics	Target
MT01	Package is transmitted within a reasonable time	10 seconds or less per Megabyte of Package
MT02	Command interface is simplistic and easily understood within a reasonable time	A user can use the command interface after approximately 30 minutes of training
MT03	Server is easily setup and configured within a reasonable time	A user can setup and configure the server within 20 minutes
MT04	Package rollbacks can be completed within a reasonable time	Packages can be reset to a previous state within 2 minutes
MT05	Uncompressed client application consumes only a reasonable portion of memory	Uncompressed client application less than half a Gigabyte
MT06	Uncompressed server application consumes only a reasonable portion of memory	Uncompressed server application less than 3 Gigabytes

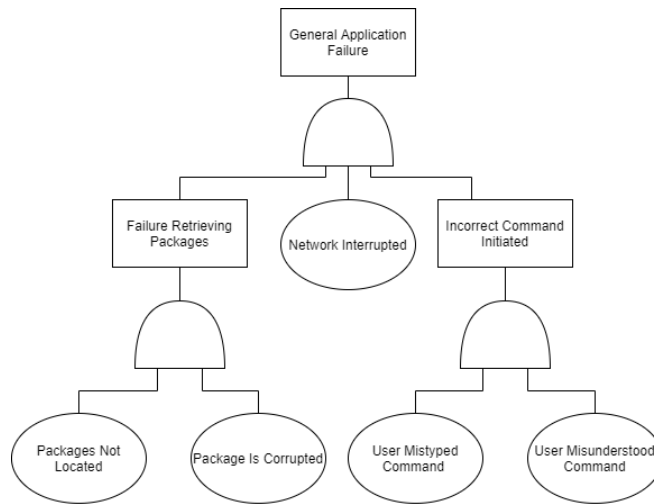


Figure 3-1: Limited Fault Tree Diagram

Impact analysis can be extracted from scenarios described in sections 2.1 and 2.2, along with stakeholder needs listed in *Table 3-1*. In particular, two domains of issues may arise, those originating from the user of the client, and those originating from server processes. The user may misunderstand commands passed to package-bird, or the passed manifest file may be misconfigured for their project, in turn misconfiguring their environment when package-bird executes the configuration command.

Other issues arise when packages are requested which have not been registered with the server, or the user attempts to register a package with metadata that somehow collides with another already registered package.

Impacts will in virtually all scenarios extend only to the firm utilizing package-bird, however the secondary effects of delayed software and bug fixing may extend to affecting their customers and other secondary stakeholders.

4.0 Implementation

The application involves many clients interacting with a centralized server, itself interacting with a database and filesystem referred to as the registry. At this stage in development, the filesystem will be a specified location on disk, and the server will be a process on the same machine.

The database will act as an address book for the packages, storing metadata including the version number, package name, authoring team, unique identifier string, additional miscellaneous information, and most importantly other dependencies. The server will likely be MongoDB, as the document-based schema effectively translates the JSON based package metadata.

The client will be implemented using Python 3 along with the request and click frameworks for server interaction and command-line interface respectively. The server will be written in Go language and interact with the previously stated MongoDB database.

The data components will revolve primarily around packages, however there are also authenticated clients with certain privileges. Packages will possess the schema described previously, however beyond

their metadata they possess a source directory wherein is contained the package 'load', which is likely source code but could also be binaries and other relevant source information.

Clients formulate request by inspecting a manifest file stored at the root level of the development directory. Exchanges of information between the client and the server will utilize remote-procedure-calls over hypertext-transfer-protocol, received by the server in Go and processing the information therein. The server interacts with the filesystem using the standard library and interacts with the database using language bindings for MongoDB.

When a request for packages is received from a client, the server will automatically attempt to find the described packages within the registry. If they are located, the server will recursively build dependency graphs for each of the packages. Once completed, the source files are bundles and compressed, then transmitted back to the client over the SSH protocol.

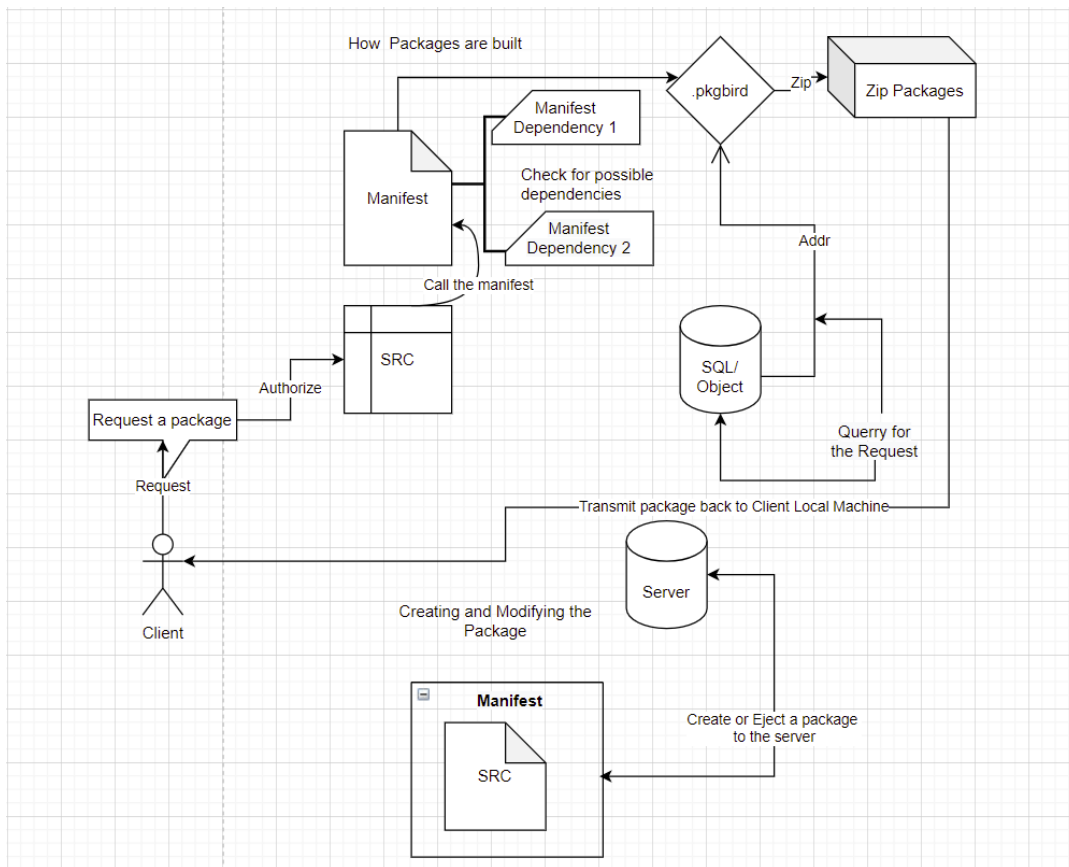


Figure 3-2: Information flow graph