xpac

A Cross-platform Package Manager

Design Document - Team 16

Anunai Ishan, Amol Moses Jha, Adit Kumar, Parth Shelgaonkar

Index

Purpose

A package manager is a standard program included on all unix-based operating systems. However, there is a lot of fragmentation among the different programs used and the different protocols and methodologies employed by each one of them; for example, within Linux itself, one can find multiple package managers being employed based on the distribution in use; aptitude, dnf, yum, YAST, pacman, portage, Homebrew and many others are well known programs used by millions of people everyday. However, the existence of multiple programs to achieve the same goal leads to repeated effort and inability to provide a standardized support. We aim to solve this problem by developing a single do-it-all solution for package management across all standard unix-based operating systems, by leveraging the standardization of POSIX standards and developing an easy-to-use solution, which hopefully will someday become the standard for package managers.

Functional Requirements

1. Users can view packages available on the xpac repository:

   As a user,

   a) I would like to have the ability to the server which hosts repositories.
   b) I would like to have the ability to know the package hosted by the server.
   c) I would like to have the ability to know the versions available of a particular package.

2. Users can download and install packages from the xpac repository:

   As a user,

   a) I would like to have the ability to install packages.
   b) I would like to have the ability to list the packages already installed using xpac.
   c) I would like to have the ability to compile from source from packages if needed.
   d) I would like to have the ability to choose between a stable version, or a current bleeding-edge version package for installation.

3. Users can update installed packages:

   As a user,

   a) I would like to have the ability to update the software database.
   b) I would like to view if there are any updates available for the packages installed.
   c) I would like to have the ability to update packages.
   d) I would like to have the ability to do a full software update.

4. Users can remove installed packages:

   As a user,

   a) I would like to have the ability to delete packages.
   b) I would like to have the ability to delete orphans and unneeded packages.

5. Users can use xpac across multiple distributions:

   As a user,

   a) I would like to have the ability to use xpac to manage packages on different Unix based operating systems.
   b) I would like to be able to automate package management using xpac.

Non Functional Requirements

1. Users should be able to install xpac easily and modify the software depending on their needs:

   As a user,

   a) I would like xpac to be free and open source in nature so that it is transparent and easily accessible for everyone to view and contribute.
   b) I would like to have xpac to be easy to use.
   c) I would like to have the ability to access documentation to get help.
   d) I would like to have the ability to gain help on the software through an IRC channel.
   e) I would like to have the ability to connect with the community of contributors and other users.

2. Users should be able to trust xpac as a package manager for their system:

As a user,

a) I would like to have the ability to install packages from a stable repository.
b) I would like to have the ability to gain access to helpful man pages to be used offline.
c) I would like to have xpac be a stable and reliable piece of software.
d) I would like to have xpac be hosted on a reliable server, and should be deployed in a way that it is easily accessible to everyone.
e) I would like to have a measure of security while connecting to the server.

Design Outline

Our project aims to provide an optimal solution for software management. To achieve this purpose, we run customized servers to provide packages. The client communicates with the server through custom-defined protocols, facilitating a faster and optimized implementation. In conclusion, we utilize a simple server-client model with homebrewed protocols in order to achieve efficiency.

1. xpac Client

The client is written in C/C++ and handles the management of packages required from the server. In order to achieve this, the client constructs the necessary dependencies and requests the necessary package(s) from the server.

2. xpac Server

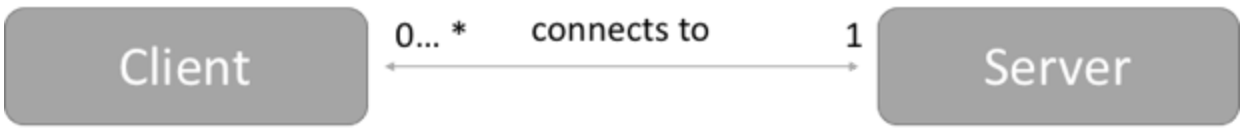
The server is also written in C/C++ for optimal efficiency. The server is multi-threaded and multi-forked. It optimizes search and push operations for the packages required by the client, using custom-defined protocol.

3. Software Repository

Binary files and source code for the packages are maintained in a file-based software repository, implemented using symbol table in order to make accessing particular files easier. The server also runs auxiliary scripts to manage the repository and generate metadata for the packages.

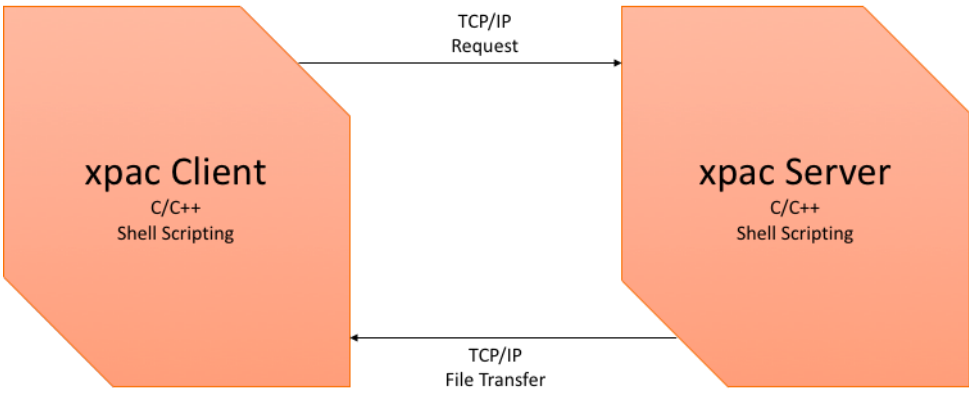
High-Level Overview of the System

Our client and server form a many-to-one client-server architecture. The client requests a package, and in the process also sends to the server all the required packages which serve as dependencies to the package, if required. The server processes the request by fetching the packages as required, and pushes the retrieved packages to the client.



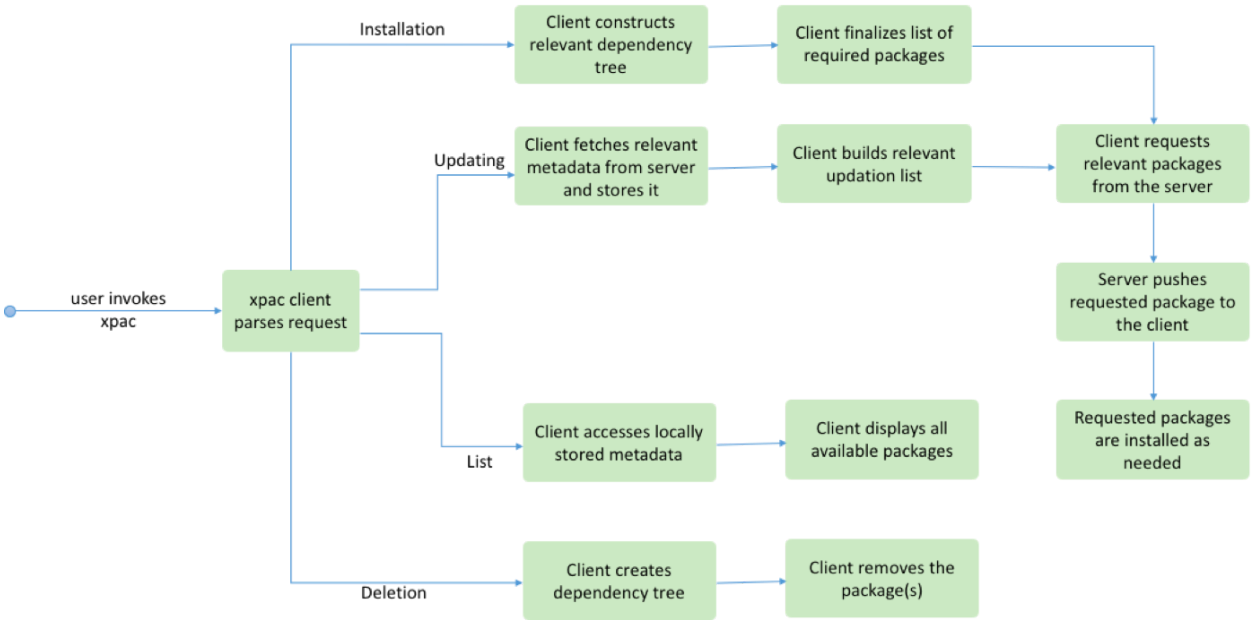
Protocol, Hosting Overview

The C/C++ client and server both communicate over standard internet packets through TCP/IP packets, sending requests and receiving files as necessary through our own homebrewn protocol. File verification is done by comparing the hash values of the files requested vis-à-vis the files compared. This ensures optimal integrity of the files hosted by the server.

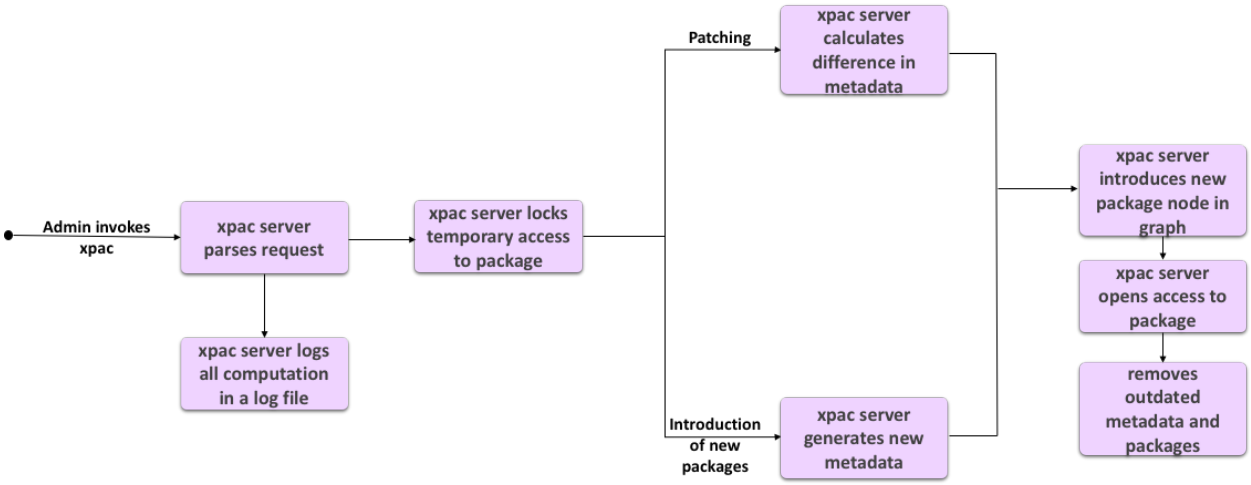


Activity/State Diagrams

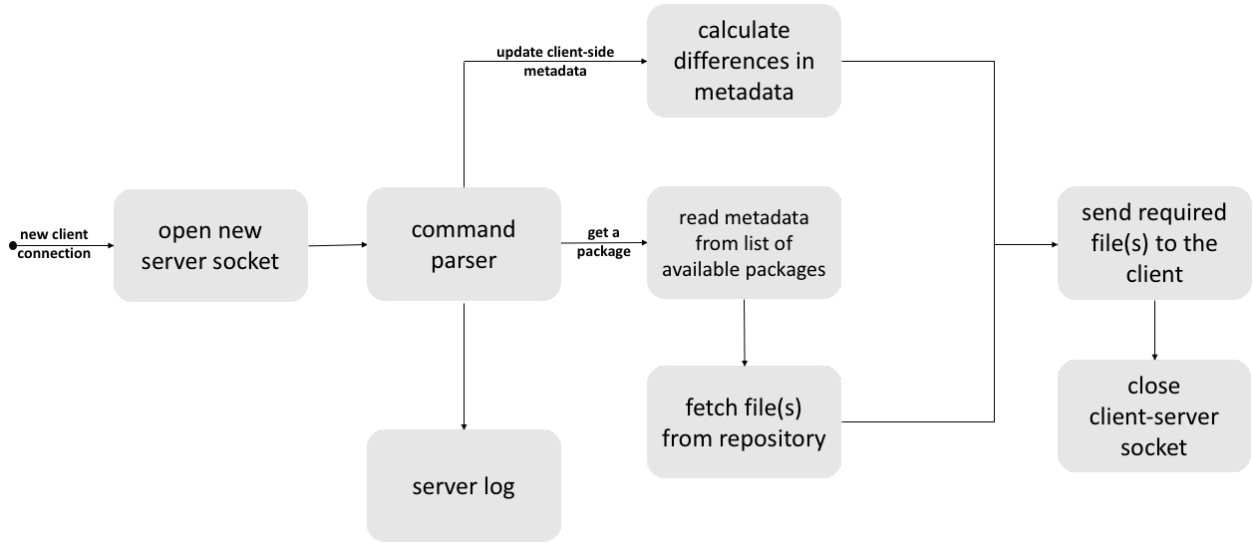
State Diagram for xpac Client



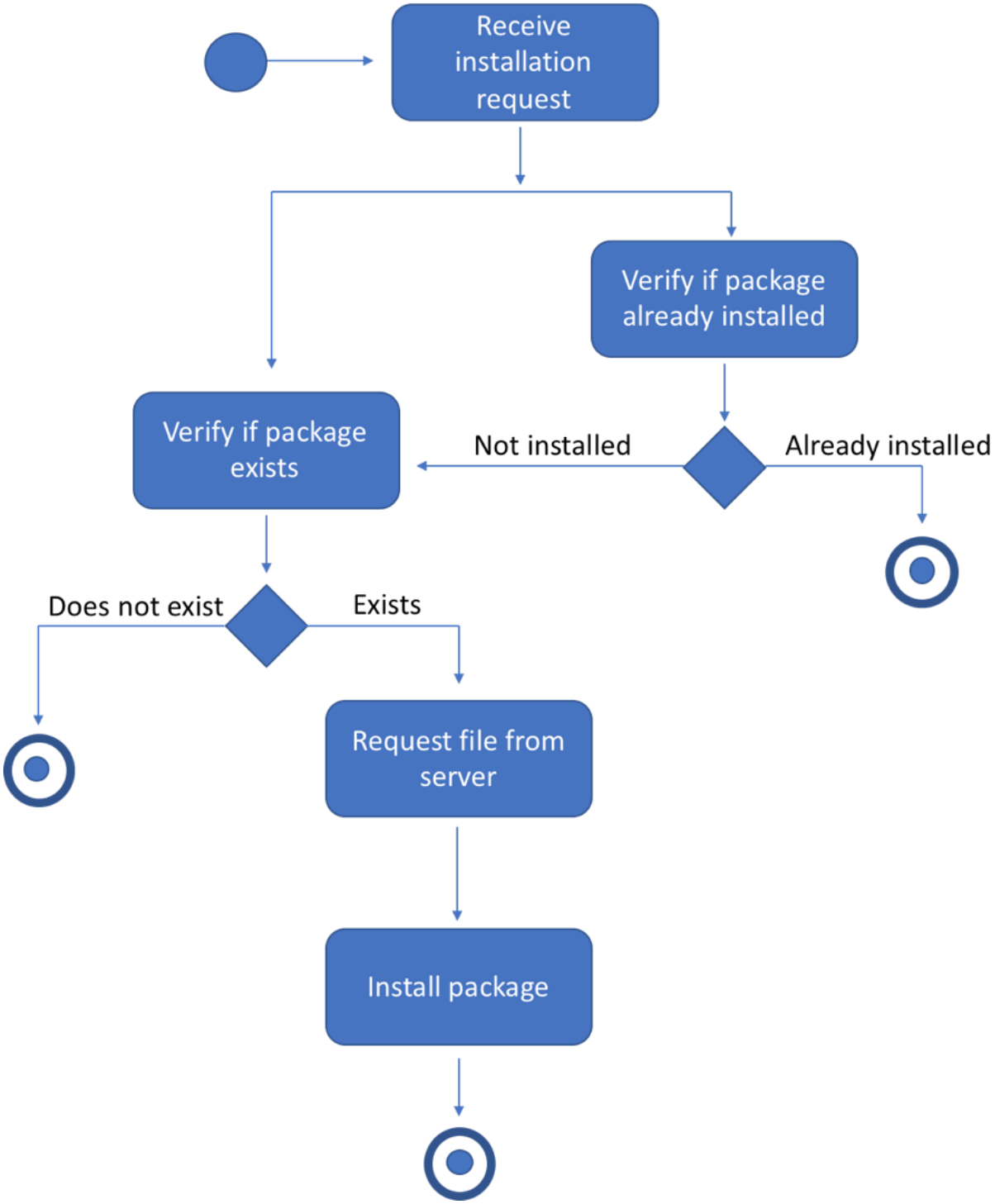
State Diagram for xpac Server



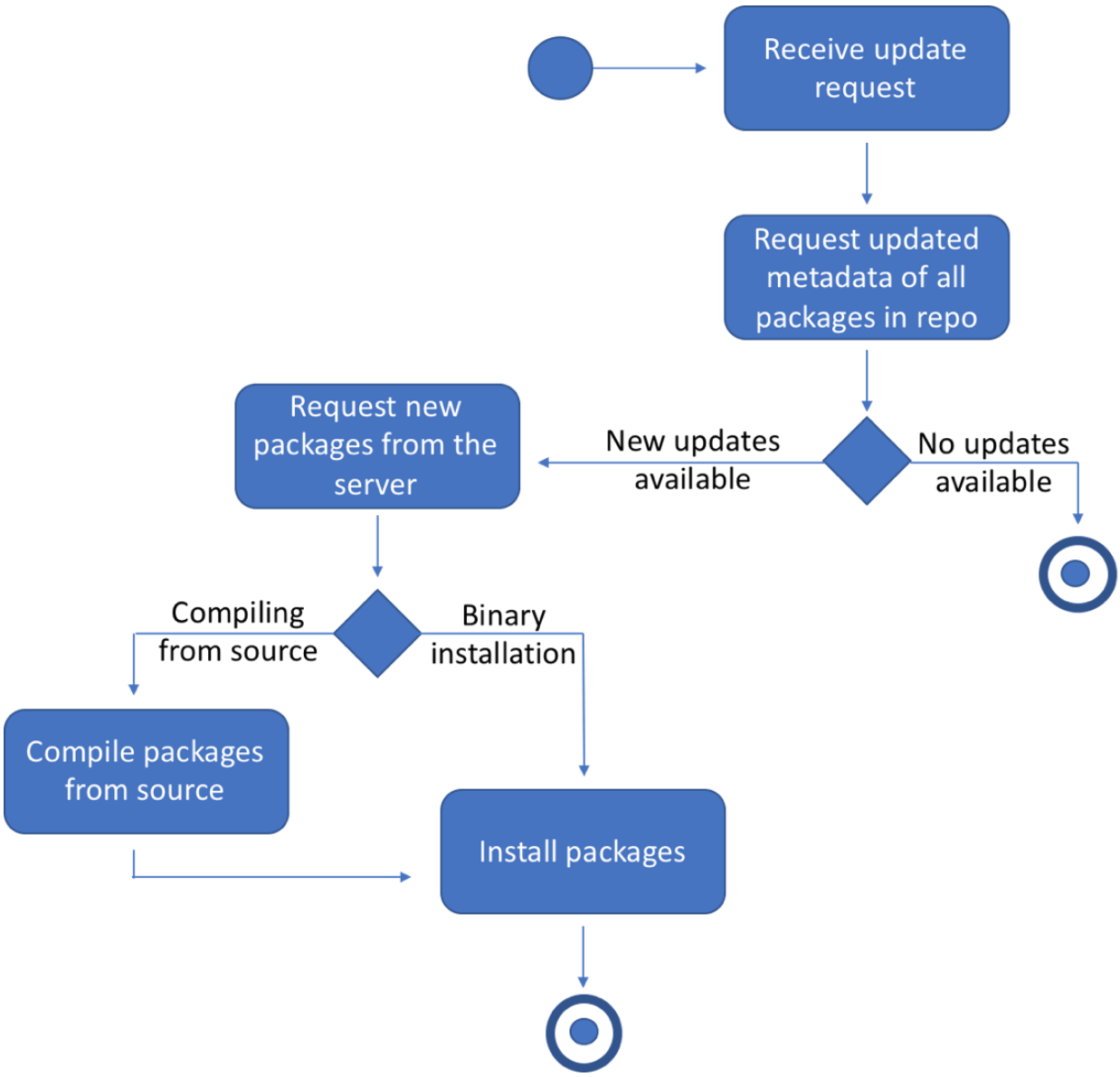
State Diagram for Server Administration



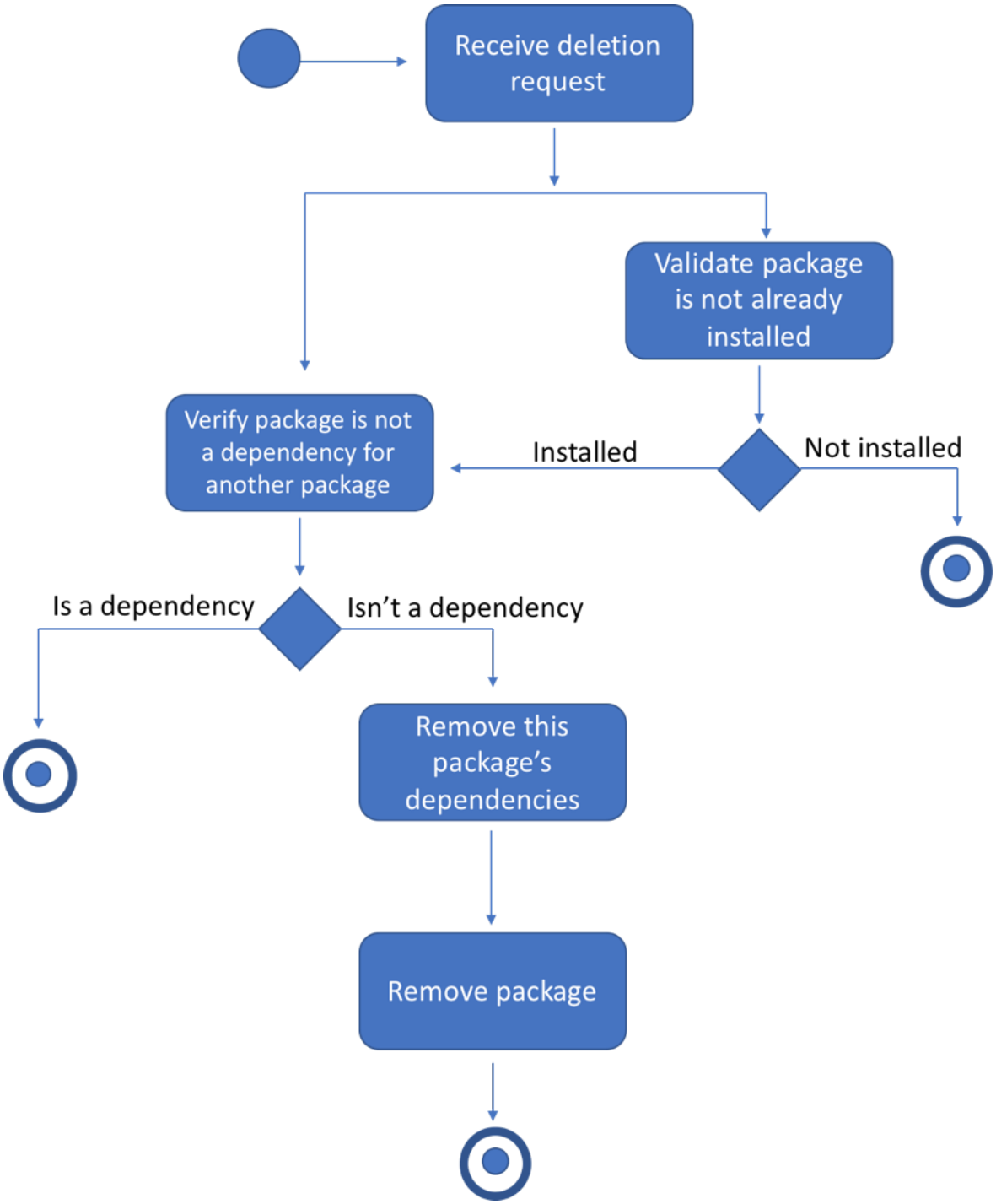
Activity Diagram for Installation



Activity Diagram for Updation



Activity Diagram for Deletion



Design Issues

Functional Issues

1. In what format should the packages be served to the user?

   Option 1: Serve pre-compiled binaries from the repository.

   Option 2: Serve source code of open-source projects fetched from their respective repositories.

   Option 3: **BOTH**

   Decision: We plan on allowing the user to choose between compiling from source (if they want to build a lean system according to their ricing needs) or conveniently fetching pre-compiled binaries from servers and placing them in the required directory.

2. What packages should the server host?

   Option 1: Serve only the bleeding edge versions of the packages.

   Option 2: **Serve previous versions of the packages as well.**

   Decision: We plan on having backwards compatibility with older machines and architectures so that more users prefer xpac. Some users are likely to use stable version of packages catering to their system needs. Developers and testers are also likely to choose the bleeding edge version of packages.

3. What protocol should be used for client-server communication?

   Option 1: File Transfer Protocol (FTP)

   Option 2: TCP/IP

   Option 3: **Develop our in-house protocol.**

Decision: We plan to develop our own protocol, enhancing efficiency while fetching binary file(s) from the server and transmitting metadata to the client.

4. What kind of a client should we have?

   Option 1: Thin client

   Option 2: **Fat client**

   Decision: We plan on designing a fat client since it performs resource-heavy tasks, which include traversing the graph of the packages and building dependency tree for a package being installed or removed. The server only processes retrieval of files and requests to update metadata from the client.

5. How do we make the repository fault tolerant?

   Option 1: **Use reference counts in metadata for files being used, in order to avoid file errors while updating or removing a binary file in the repository.**

   Option 2: Use thread safety to synchronize file access.

   Decision: Given the possibility of the presence of an arbitrary number of file accesses at any point of time, a better design choice would include having reference counts similar to file inodes in the metadata to reduce chances of concurrent accesses, thereby reducing interdependency and chances of errors. This facilitates introduction of a package or removal of an old version in the software repository while the server is running.

Non Functional Issues

1. How should the server be hosted?

   Option 1: Use a hosting service like AWS, Microsoft Azure, Docker or another similar service.

   Option 2: **Build in-house server by building a cluster of Raspberry Pi's.**

   Decision: Since our design involves a fat client and a thin server, we are building a cluster of Raspberry Pi's which involves each individual Raspberry Pi run forked child processes to flush out the requested files to the client.

2. What language should be used to develop xpac?

   Option 1: **C/C++**

   Option 2: Java

   Option 3: Python

   Option 4: Nodejs

   Option 5: Ruby

   Decision: We will primarily use C/C++ for the development in order to maximize possible performance for both the client and the server, because we need efficiency while constructing the sub-graphs and dependency trees.

3. How should the requested file be flushed to the client?

   Option 1: **The server will flush out the entire file when the socket is open.**

   Option 2: The server will flush out the file in chunks that are re-ordered at client end.

   Decision: We plan to flush the entire file over the open socket connection and compute a hash at both ends to verify the received file since it will be more challenging algorithmically and also, consume more hardware resources to dispose of the extraneous metadata and re-order the received packages.

4. What kind of a database do we use for the server?

   Option 1: A conventional relational database.

   Option 2: **A file-based repository indexed using a symbol table.**

   Decision: We will be maintaining a file-based software repository for managing the binaries and source codes for the packages allowing easy facilitation of auxiliary scripts to swap out files and for the server to access the binaries, given the path. We will be using a bookkeeping schema to be able to access the files easier and have a symbol table to map the package name to the path in the repository where compiled binary and/or source code are present.

5. How customizable should xpac be for the user?

   Option 1: Not customizable at all

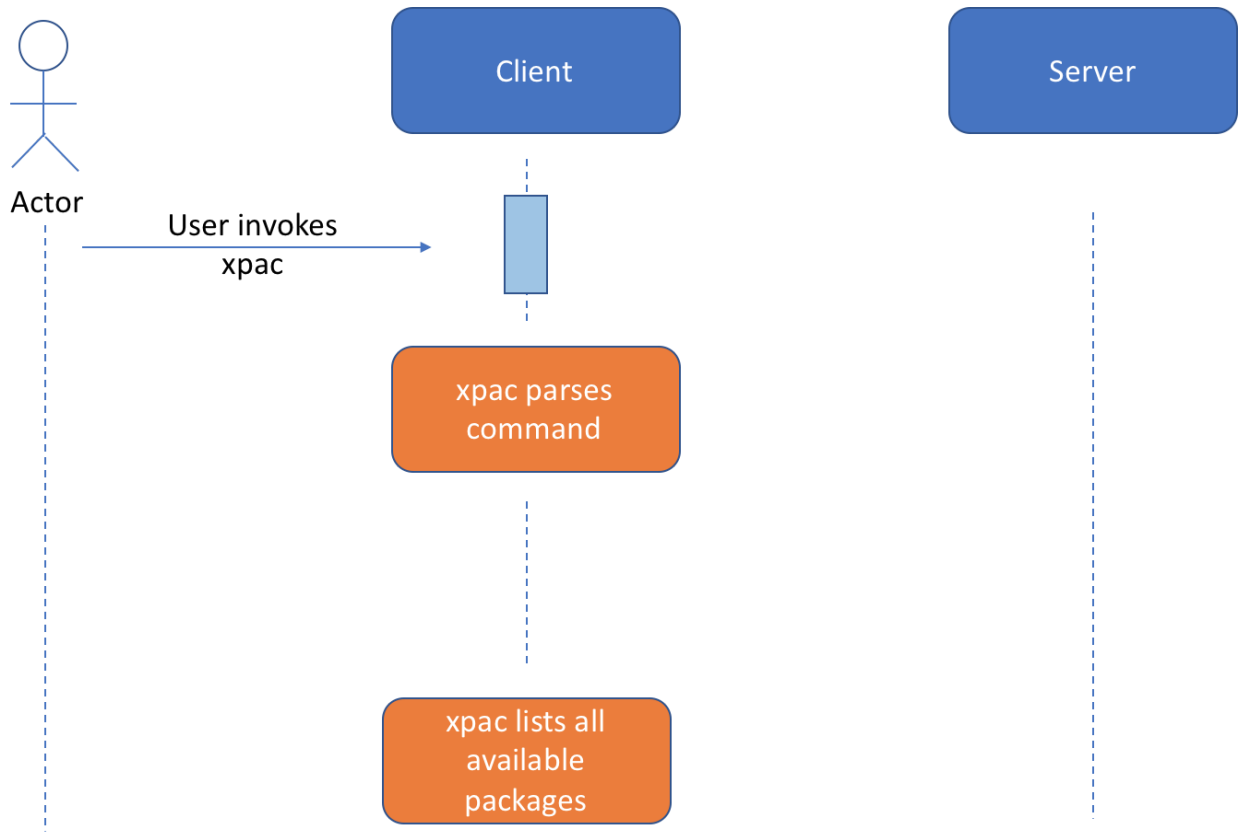
   Option 2: **Some customization available**

   Decision: We decided to make xpac customizable for the user, so that the user can tweak and customize xpac to suit their needs. This will also enable a more personalized experience for the user while using xpac.

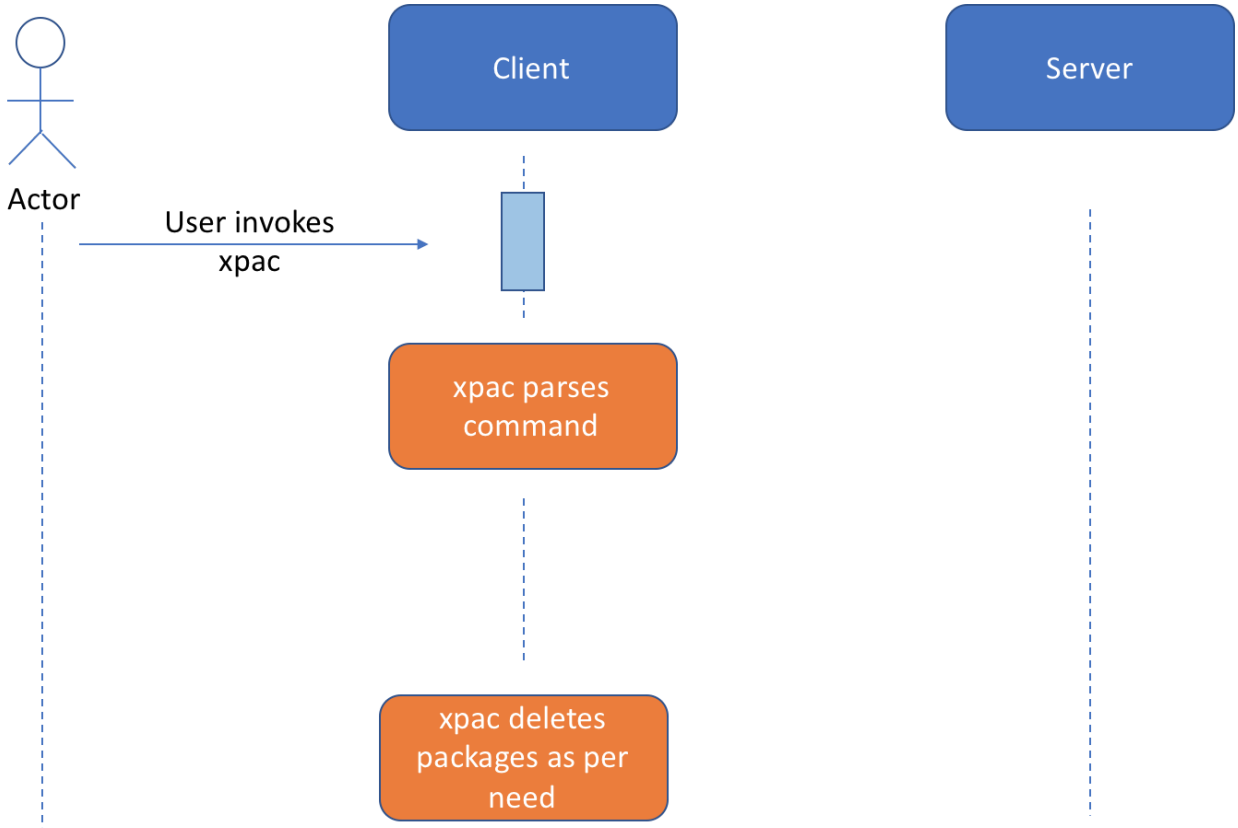
Design Details

Sequence Diagrams

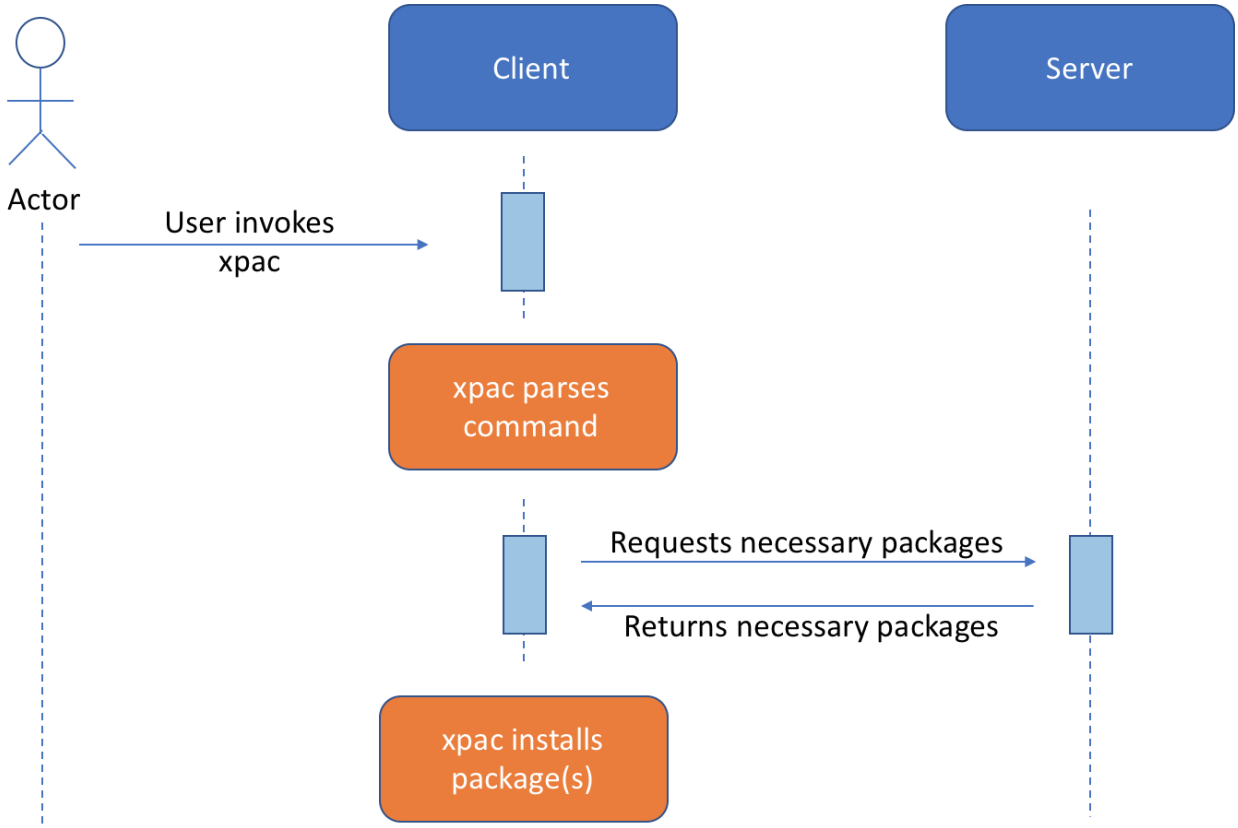
Sequence of events when the user requests to see all available packages



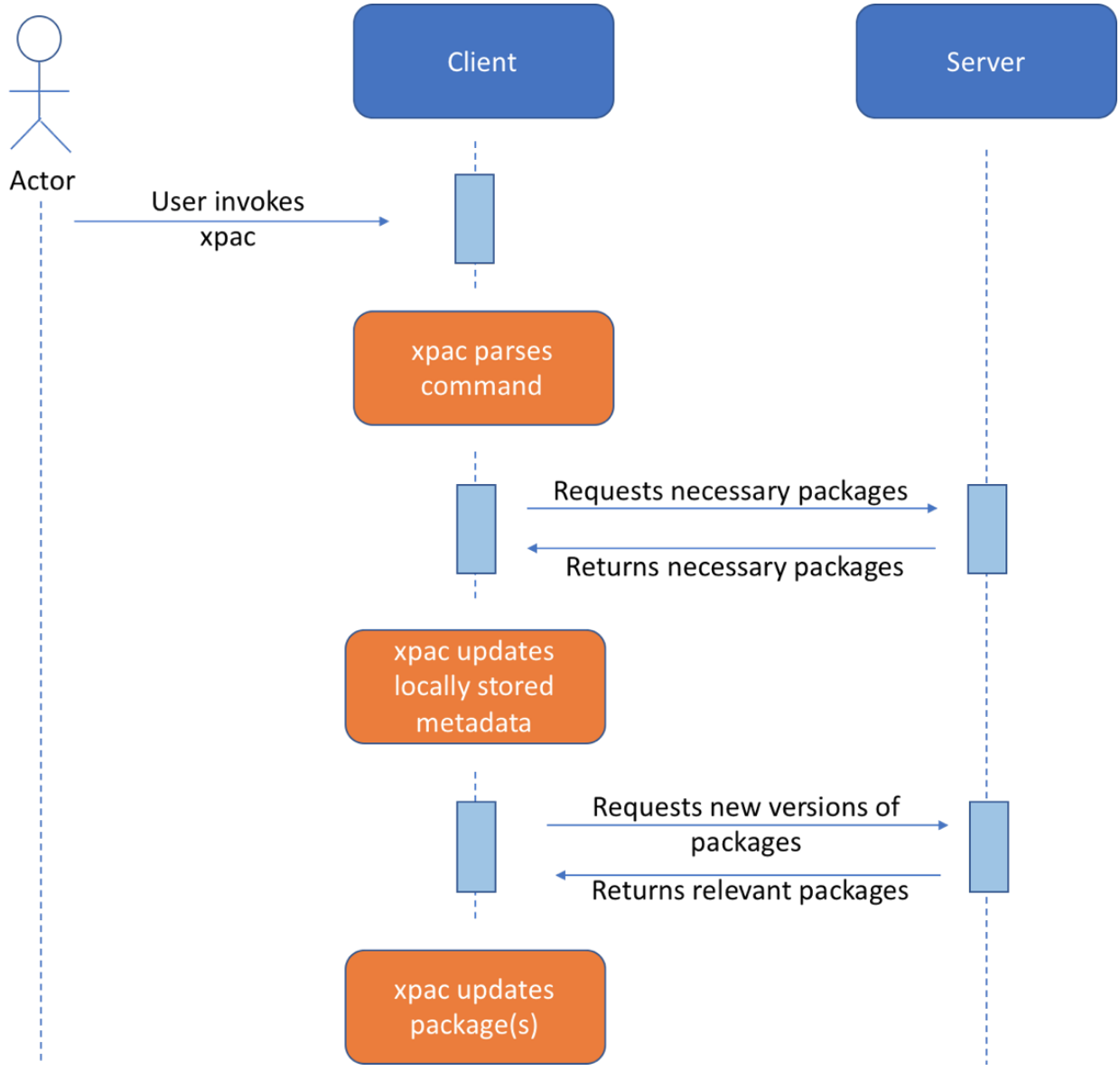
Sequence of events when the user requests to delete a particular package



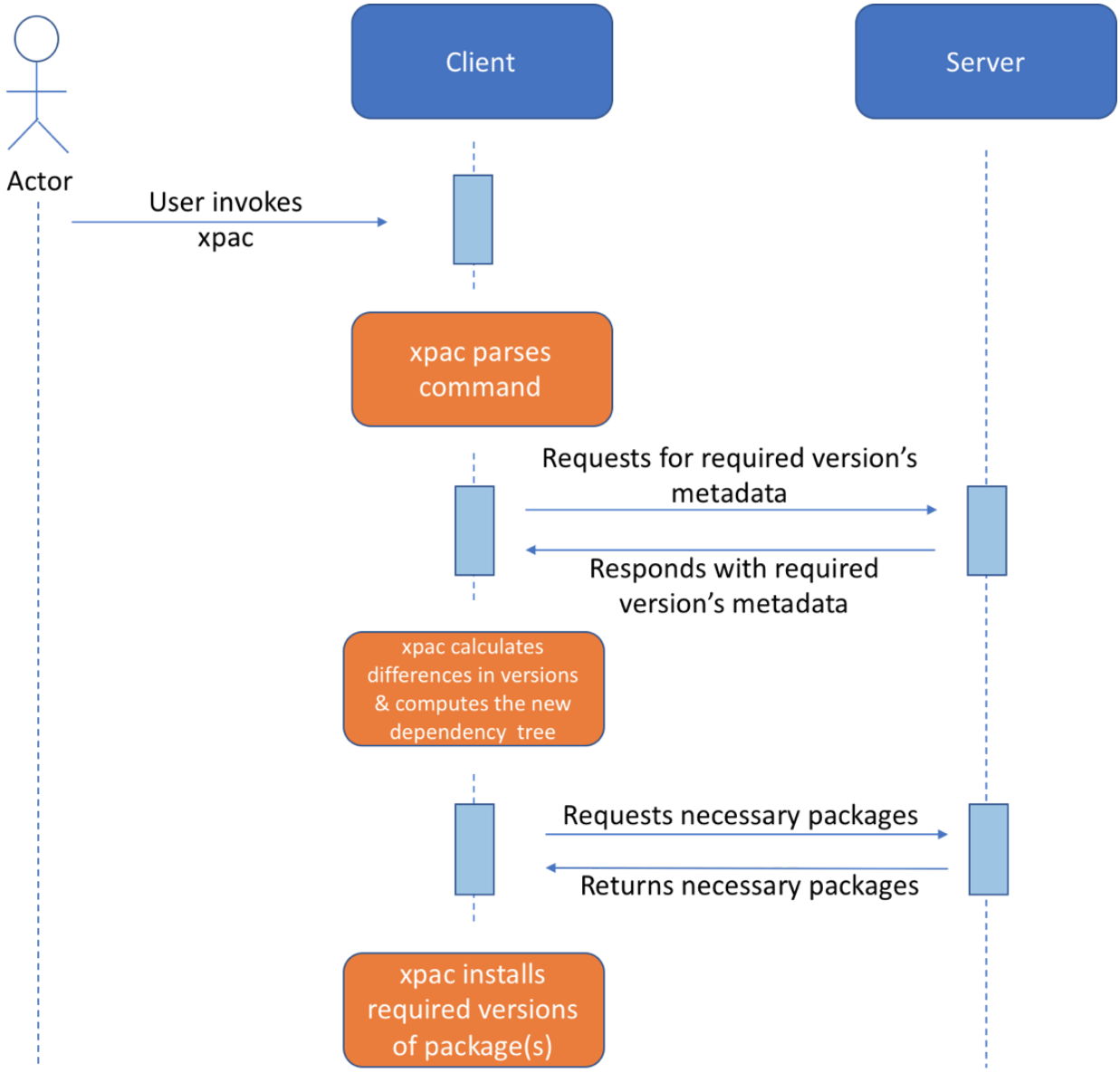
Sequence of events when the user requests to install a particular package



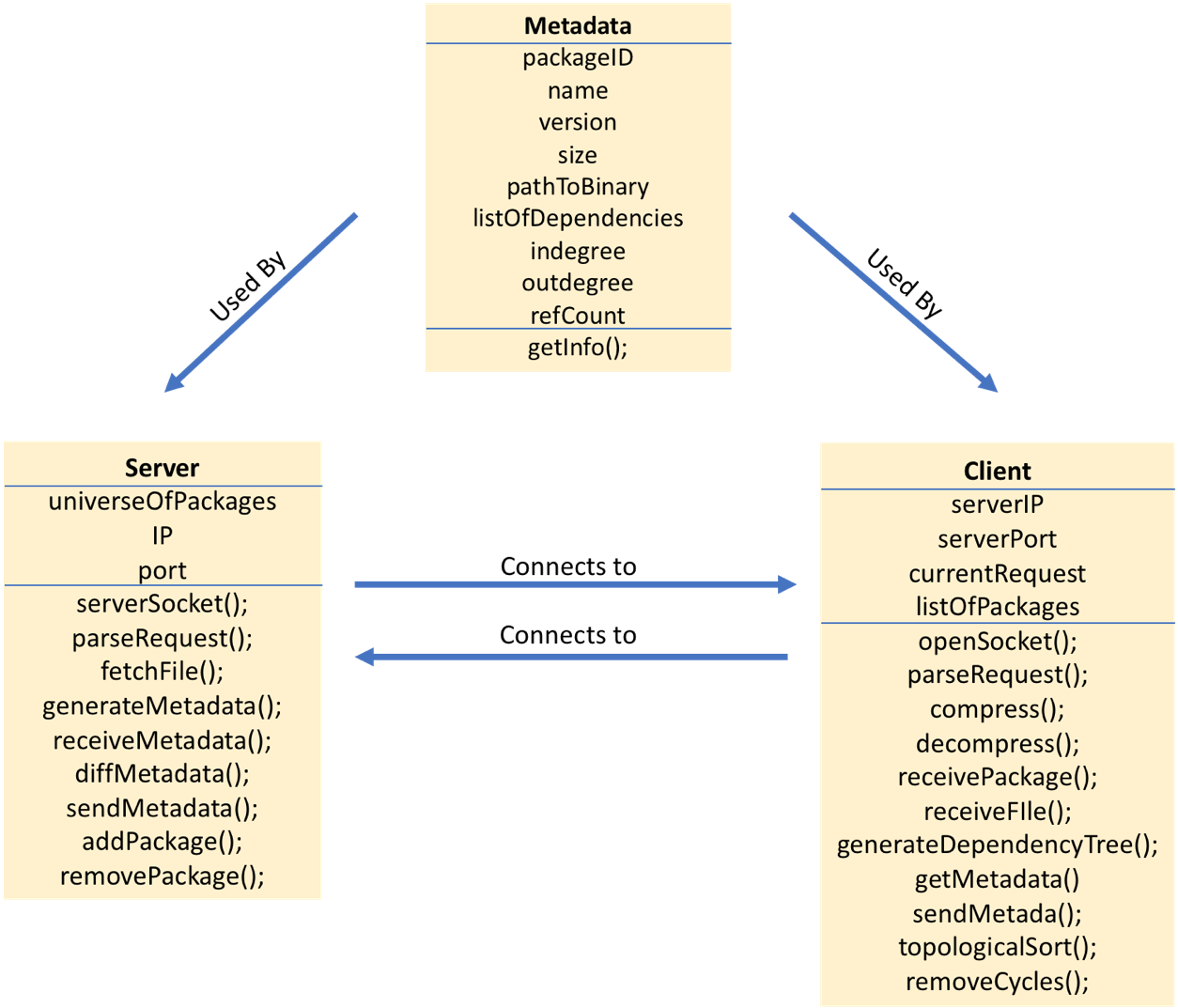
Sequence of events when the user requests to update all locally-present packages



Sequence of events when the user requests a change in the version of package



Class Diagram



Description of Classes and their Interactions

Our classes serve our purpose of modeling packages hence, the metadata abstraction. Furthermore, we have dedicated classes modeling the inherent client-server architecture in this project. We further enumerate the classes used and their interactions:

Metadata

The metadata graph is our abstraction for representing the complex, interwoven structure of the graph of different packages and how they interact with each other. It acts as the layer of abstraction between the state of the packages (binary files) and the abstraction we apply to them: a graph of interconnected packages making up a modern operating system and its applications and tools. This allows us to use standard graph algorithms to make dependency trees and topologically sort packages. This class contains all necessary tools for the graph abstraction, namely, the indegrees and outdegrees (represented by the list of dependencies), and necessary tools to flush the file out to the client.

Server

The server ends up performing the task of connecting to multiple clients and sending them the files required, which includes updated metadata of the whole universe, or the required file to install. It contains methods to open a server socket and flush the requested files to the client. Being multi-forked, it is highly parallelized and optimized to handle multiple clients. It also contains methods to efficiently manage metadata and make changes to the underlying software repository.

Client

Due to our design decision to make the server as thin as possible, owing to hardware limitations of the homebrewed cluster of Raspberry Pi's, the client does most of the work of the heavy computation lifting. The client perform tasks like computing the dependency trees, removing cycle(s) in the resulting subgraph, topologically sorting all the packages, calculating any difference between the universe and the local graph of packages, and other graph functions. Therefore, the client contains several data members and functions dealing with graphs. In addition to these, it contains members for optimal connectivity and exchange of information and files with the server.

Command Line Arguments

Owing to the nature of our project, it does not use a graphical user interface. It is a command line utility and therefore, works accordingly. In order to use xpac, one needs to specify different command line arguments which invoke different behaviors.

The following arguments cater to xpac and its various functionalities:

`$ xpac install <package_name>`

Allows the user to install a package, in addition to all of its required dependencies in the user's system

`$ xpac <package_name>`

Displays the information for the specified package, including its installation status and version, and prompts the user for installation

`$ xpac install --version <package-name>`

Allows the user to install the specified version of a package, in addition to its required dependencies; extremely useful for backwards compatibility and multiple versioning of software

`$ xpac install --compile <package-name>`

Allows the user to install a package by fetching its source code and compiling it according to their needs in order to optimize the package

```
$ xpac update
```

Fetches the most current and up-to-date listing of all the packages that reside in the universe

```
$ xpac upgrade
```

Upgrades all packages and their dependencies, to the most recent version present in the software repository.

```
$ xpac list
```

Lists all packages available to be installed from the current universe

```
$ xpac remove <package-name>
```

Deletes the specified package and its dependencies which do not serve as dependencies to any other package (referred to as orphans or orphaned packages).

```
$ xpac remove --cleanup
```

Deletes the remaining orphaned packages or old dependencies no longer needed by any package; useful for saving space on disk