Pandit

A Distributed Service-Level Proxy

Eric Gerard Moynihan

B.Sc. Computer Science - Final Year Project

Prof. Dirk Pesch

University College Cork

April 2022

Abstract

Pandit transfers abstract data between applications on different servers. It lives on each server and listens for requests by applications on that server.

There are two types of applications that run on the servers: clients and 'servers'. Clients query data and 'servers' provide said data. Pandit acts as a middle-man for these queries.

Services

Pandit provides 'services' to clients. A service is an abstraction on top of a 'server' that just focuses on the data provided by the server.

Services are defined in a 'gRPC [19] Proto file [30]' file that both the client and Pandit have access to; so both know the structure of a particular service.

Pandit handles the querying and conversion of data from the servers; this allows the clients to receive data in a uniform format.

Caching

Since Pandit knows how to translate the data; Pandit knows when it gets the same request for data twice. This allows it to store previously seen responses. If it gets the same request twice, it will return the cached data without adding extra load to the server.

Declaration of Originality

In signing this declaration, you are conforming, in writing, that the submitted work is entirely your own original work, except where clearly attributed otherwise, and that it has not been submitted partly or wholly for any other educational award. I hereby declare that:

- this is all my own work, unless clearly indicated otherwise, with full and proper accreditation;
- with respect to my own work: none of it has been submitted to any educational institution contributing in any way to an educational award;
- with respect to another's work: all text, diagrams, code, or ideas, whether verbatim, paraphrased or otherwise modified or adapted, have been duly attributed to the source in a scholarly manner, whether from books, papers, lecture notes or any other student's work, whether published or unpublished, electronically or in print.

Signed: [illegible]
Date: 23rd April 2022

Acknowledgements

Firstly, I would like to thank my supervisor, Professor Dirk Pesch, for helping me every step of the way during this process. Every time I had a question regarding the project, Dirk got back to me with a timely response. I am also thankful for Dirk taking on this project, as it was proposed by myself. Without a supervisor that allowed me to pursue this project, I wouldn't have been able to do it. For that, I am very thankful.

I would also like to thank Netsoc for providing Netsoc Cloud instances to different deployments of Pandit. I learned a lot as a member of the Netsoc System Administrator committee for the past few years. I gained many technical skills in the areas of Code Review, Programming and System Administration. I also gained teamwork experience during my time there, and for that I am very thankful.

I would also like to thank Google, the company I did my work placement for. In particular, I would like to thank my host Jose-Luis Izquierdo-Zaragoza for providing me with many hours of learning throughout my work placement. It was at Google that I learned the power of using code generation and protocol buffers, which is what inspired me to pursue this very project. For that, I would like to say a big thank you to everyone who helped me during my time there.

External Libraries Used

I used many external Rust libraries to complete this project. Here is a complete list of them:

```
access-json = "0.1.0"
async-recursion = "1.0.0"
api_proto = { path = "./src/api/proto", package = "api" }
bollard = "0.11"
get_if_addrs = "0.5.3"
clap = { version = "3.1.2", features = ["derive"] }
hostname = "0.3.1"
kube = { version = "0.69.1", features = ["runtime", "derive"] }
k8s-openapi = { version = "0.14.0", features = ["v1_22"] }
log = "0.4.11"
config = "0.11.0"
crossbeam-channel = "0.5.2"
tonic = "0.6.2"
h2 = "0.3"
tower-test = "0.4.0"
num = "0.4"
libc = "0.2"
hyper = { version = "0.14", features = ["full"] }
tokio-postgres = { version = "0.7.5", features =
   ["with-serde_json-1"] }
postgres-types = "0.2.2"
postgres-protocol = "0.6.3"
sea-query = { version = "^0", features = ["postgres-array"] }
grpcio = { version = "0.10.0", default-features = false, features
   = ["protobuf-codec"] }
tempfile = "3.3.0"
bytes = "1.1.0"
http = "0.2.6"
tokio = { version = "^1.0.1", features = ["rt", "signal", "time",
   "io-util", "net", "sync", "tracing"] }
console = "0.15.0"
```

```
tokio-test = "0.4.2"
parameterized = "0.3.1"
futures = "0.3"
tracing = "0.1.26"
async-trait = "0.1.52"
serde_json = "1.0"
tracing-subscriber = "0.3.2"
console-subscriber = "0.1.3"
serde-protobuf = "0.8.2"
serde = { version = "1.0.133", features = ["rc"] }
httparse = "1.5.1"
dashmap = { version = "4.0.2", features = ["serde"] }
protobuf = { git =
    "https://github.com/stepancheg/rust-protobuf.git",
    tag="v3.0.0-alpha.2", features = ["with-serde"] }
protoc-rust = "2.25.2"
protobuf-parse = { git =
    "https://github.com/stepancheg/rust-protobuf.git",
    tag="v3.0.0-alpha.2" }
redis = { version = "0.21.5", features = [ "cluster",
    "tokio-comp"] }
```

These packages can be found by looking up their name on https://crates.io/.

Contents

List of Figures

Chapter 1

Introduction

Pandit at its core is a **distributed service-level proxy**.

A *proxy* in this context means an intermediary server that acts as a middle-man between a client requesting some data and the server that is hosting said data.

The proxy will be *service-level* because it will provide an abstraction of the underlying server called a *service* to the clients. This will be achieved by implementing Pandit in such a way that it is able to parse the data provided by the server and convert it into a uniform machine-readable API. This comes in the form of Pandit exposing a *gRPC [19]* API (see gRPC). Each Pandit *service* will be a gRPC service from the perspective of the clients.

The proxy will be *distributed* across physical servers. This will allow clients on one physical server to query a Pandit *service*. This query will be handled by the Pandit instance on the local machine.

Pandit will utilise caching [49] to speed up the query responses, and since the application with the data could be on any physical server, Pandit will delegate the request to another physical server's Pandit instance, allowing for the possibility of another cache hit before the request is translated and sent to the server with the data.

Since Pandit parses the entire payload, smarter caching strategies can be applied than those provided by traditional proxies.

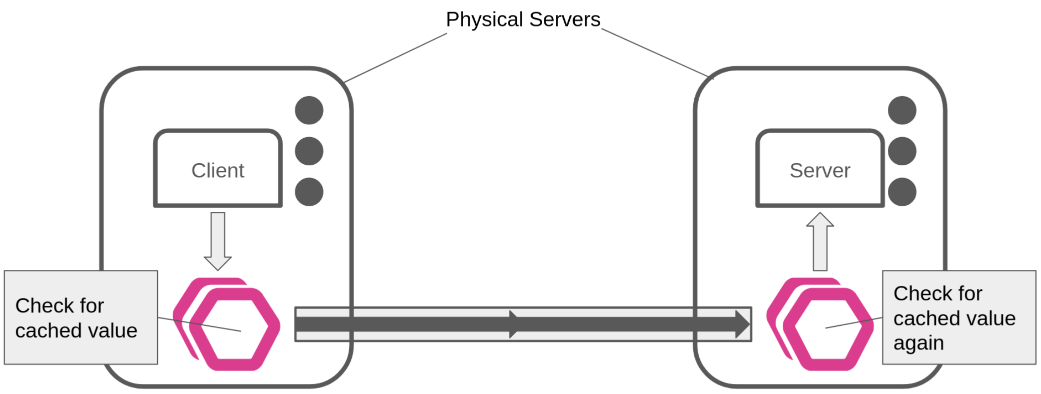


Figure 1.1: Basic Distributed Deployment Overview, showing flow of query between two servers.

Chapter 2

Analysis

2.1 Background

This project utilises many existing technologies and concepts. Provided below is some background information on these technologies.

2.1.1 The Rust Programming Language

The entire project is implemented using the Rust Programming Language. Providing fast execution times, compile-time memory safety and great tooling [46].

Type System

Rust proved to be challenging at times due to its rigid type system. Rust is statically typed; all types must be known on compile-time. This adds to its complexity and development time. Rust reduces the likelihood of runtime errors as a consequence of this. Compared to dynamically typed languages such as Python; Rust provides more efficient, less error-prone code.

Lifetimes

Another benefit of Rust would be its *lifetime system [42]*. Unlike most languages, Rust doesn't have garbage collection. Garbage collection is when a program frees up memory by removing objects from the heap.

Objects in traditional programs are freed from the heap when there is no longer any code that references the object. A garbage collection routine detects this and frees the memory. This is computationally expensive because the program has to realise that an object needs to be freed at **runtime**.

Rust on the other hand does this at *compile-time*. Knowing at what point in the program memory should be freed before it even runs. This removes the need for a garbage collection routine.

It achieves this by introducing the concept of lifetimes. Lifetimes are compiler-enforced rules that ensure that both the programmer and compiler know at what point in the program an object can be freed. This is seen from the programmer's perspective as an object going out of scope.

An object can be **borrowed** from one lifetime to another. This occurs when a **reference** to the original value is used in a different scope. Borrowing an object is allowed when it is **guaranteed on compile time** that the reference will not outlive the underlying value it's borrowing from.

Take a look at this example:

```rust
fn main() {
    let x = "str";
    {
        // new scope
        let y = &x; // borrowing x
        println!("{}", y);
    }
    println!("{}", x);
}
```

The value x is borrowed by y, which is in a separate scope and hence, a separate lifetime. *y* will only live as long as its scope, it will go out of scope before x in this case. Here we can guarantee that x has a longer lifetime than y, and hence the borrow is allowed.

2.1.2 Protocol Buffers

Protocol Buffers [30] is an open standard for the serialising of structured data. It is a binary standard, meaning that it defines the serialising of data into a string of binary.

The structure of the data is defined in what is known as a *Proto* file. This file is used by the applications transferring the data to both serialise and deserialise the data.

The most basic concept in a *Proto* file is a *message*. This defines the basic structure of the serialised data. Here is a basic example of a message:

```
message ExampleRequest {
  int32 id = 1;
  string field1 = 2;
  string field2 = 3;
}
```

The reason why each field has a number is that it defines the numeric *tag* that the field will be defined when serialised. This allows for extremely efficient deserialising; the deserialiser looks up the tag in its local copy of the Proto message in order to understand how to parse this field.

Deserialisation is achieved in a single sweep across the binary string.

Text-based formats such as JSON [36] require to be stored in less space-efficient text-based strings; requiring a more complex stack-based approach to parsing. Protocol Buffers are far more efficient in comparison because:

- The field names are known ahead of time by both the sender and the receiver.
- Binary formats are far more space-efficient than text-based formats due to extra bytes being required. Text-based formats struggle to hold non-text based primitive data types such as booleans, floats and integers.
- Extra bytes are required to delimit objects in JSON, such as curly braces. The JSON parser does not know the data structure being transferred ahead of time, requiring extra bytes to delimit said data structures.

2.1.3 gRPC

A Remote Procedure Call (henceforth **RPC**) is a protocol that allows a program on one computer to call a function and request data from a program on another computer. gRPC is a protocol made by Google that achieves this by utilising HTTP/2 [32] and *Protocol Buffers*.

It utilises HTTP/2 to allow for the multiplexing of multiple queries into a single TCP connection [33].

gRPC extends the *Protocol Buffers* format by providing a *Service*. This service defines RPC methods. Each method takes a single *message* as its parameters and returns a single *message*.

An example of how a gRPC service can be defined in a Proto file would be the following:

```
service ExampleService {
  rpc GetExample(ExampleRequest) returns (ExampleResponse) {
  }
  rpc GetExample2(ExampleRequest2) returns (ExampleResponse2) {
```

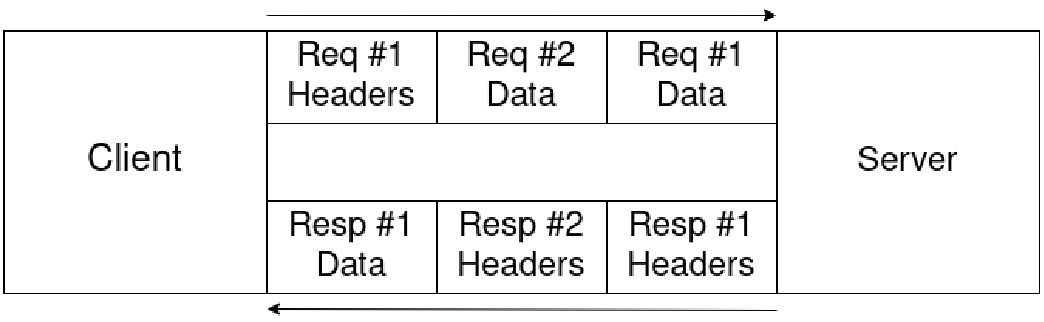


Figure 2.1: HTTP/2 Multiplexing over a single TCP connection

```
  }
}
```

Therefore, gRPC provides an RPC framework that allows for the efficient transfer of data via the use of Protocol Buffers.

2.1.4 Protobuf Options

Options provide a way to embed data inside a Proto specification. They are usually defined inside a package like so:

```
syntax = "proto3";
import "google/protobuf/descriptor.proto";

package pandit;

enum Handler {
  JSON = 0;
  POSTGRES = 1;
};

extend google.protobuf.MethodOptions {
  Handler handler = 50051;
}

extend google.protobuf.ServiceOptions {
  Handler default_handler = 50051;
}
```

This example defines an option called *handler* that can be added to any gRPC method, and an option called *pandit.default_handler* that can be added to any gRPC service. The reason why they are prefixed with *pandit.* is because they are defined inside the package *pandit*.

These options can then be used to add metadata to methods and services, like so:

```
service ExampleService {
  option (pandit.default_handler) = JSON;

  rpc GetExample(ExampleRequest) returns (ExampleResponse) {
    option (pandit.handler) = JSON;
  }

}
```

Fields and messages can also utilise options like so:

```
// pandit.proto
extend google.protobuf.FieldOptions {
  bool key = 50037; // large number.
}

extend google.protobuf.MessageOptions {
  string path = 50030;
}

// message.proto
message ExampleResponse {
  option (pandit.path) = ".obj";
  int32 id = 1 [ (pandit.key) = true ];
  string user = 2;
}
```

The reason why options should be defined with a large number is if another imported package also extends some Proto options. If they use the same number as the previously imported package, it will cause a conflict when parsing the Proto file.

2.2 Objectives

- Pandit should be designed to be able to proxy any data source, such as REST APIs or SQL databases.

  It should read in user-defined Protocol Buffers that are extended to provide information on how to parse, cache, and return data to clients.

- Pandit should provide *Pandit Services*; a gRPC service that contains all the necessary data to convert requests and responses to the format required by the server being proxied.
- Using Protobuf Options, Pandit should provide a way to define the conversion between the application and the Pandit Service.
- It should be able to cache data based on user-defined caching strategies in a distributed manner.
- It should be able to be run on a set of networked servers called a cluster.
- It should support multiple deployment modes, such as Kubernetes [26] and Docker [3].

2.3 Approaches

2.3.1 eBPF Approach

The initial approach to achieving these objectives was to run a daemon on each server. The daemon would load code into a kernel feature called ”eBPF” [15].

eBPF is a technology that allows code to be run inside the Linux kernel. It allows for direct access to features such as network interfaces.

The approach involved using eBPF to hijack packets inside the kernel. Doing some basic parsing inside the kernel, and then sending them to the user-space program.

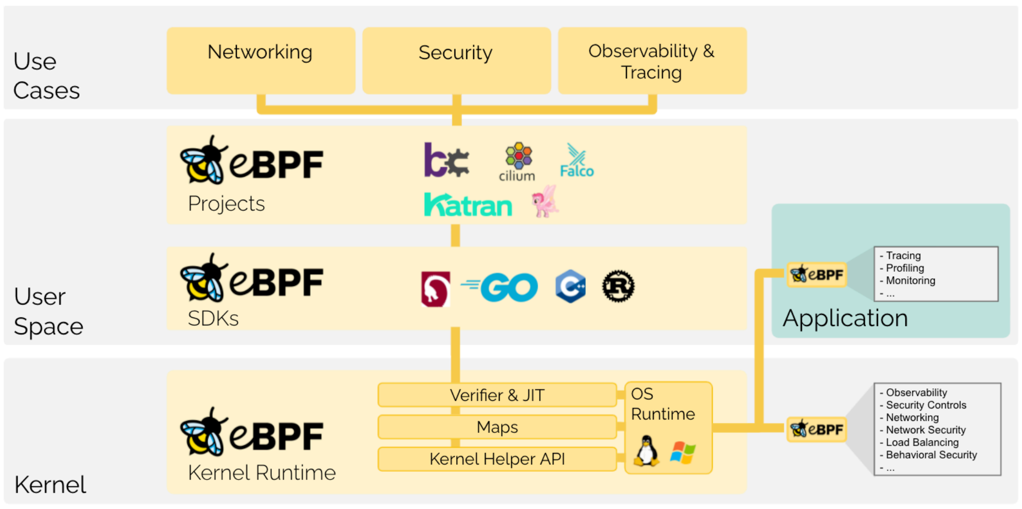


Figure 2.2: Diagram of the eBPF ecosystem. Copyright 2022 eBPF Authors. URL: https://ebpf.io/

The approach was ultimately scrapped due to the complexities involved with writing an eBPF program. For example:

- No shared libraries could be used due to the strict limitations imposed by the eBPF runtime.
- The number of instructions a compiled eBPF program could contain was limited to an arbitrary number hardcoded into the kernel. Reducing the ability to perform complex operations.
- The number of jumps a program could execute during its lifetime was capped. This limited the processing of payloads in the kernel.

2.3.2 Userspace gRPC Server approach

The approach that was decided upon ended up being a lot simpler: Run a daemon on each server that provides a gRPC server that clients will interface with.

The daemon will translate the requests to the necessary format. Forwarding them on to the proxied application, caching the results.

2.4 Existing Solutions

Service level proxies usually fall into two categories:

- A commercial closed-source proxy that focuses on platform-specific use cases.
- An open-source proxy that proxies requests and caches the responses.

2.4.1 Commercial Closed-Source Proxies

An example of the former would be Google Cloud Endpoints with HTTP/JSON to gRPC encoding [25]. This feature exists in Pandit and functions similarly - as in it provides a conversion layer between HTTP/JSON and gRPC. However, Google Cloud Endpoints only support a single data format (HTTP/JSON), Pandit supports many more data formats.

Like Pandit, Google defines their conversions by utilising Protobuf Options. Here is an example taken from their *Book Store example [17]*

```
// Returns a specific bookstore shelf.
rpc GetShelf(GetShelfRequest) returns (Shelf) {
  // Client example - returns the first shelf:
  //   curl http://DOMAIN_NAME/v1/shelves/1
  option (google.api.http) = { get: "/v1/shelves/{shelf}" };
}

// Request message for GetShelf method.
message GetShelfRequest {
  // The ID of the shelf resource to retrieve.
  int64 shelf = 1;
}
```

However, this form of proxy only works on the hosted Google Cloud Platform [24]. Pandit can be deployed in any cluster. This is only designed to operate on JSON sent over HTTP. While being generic enough to parse any data format that has been implemented is core to Pandit's design.

2.4.2 Open-Source Proxies

Envoy Proxy [8] is an example of an open-source proxy that is used to proxy requests and cache the responses. Like Pandit, it supports many data formats and can be deployed in any cluster. It runs alongside each application in the cluster and handles the querying and caching of data. Pandit has a similar approach to this, with one key difference; Pandit provides a gRPC service to the clients.

The benefit of translating data to Protocol Buffers to be transferred over gRPC are numerous:

- Clients can take advantage of code generation based on the Proto specification. [28]
- Protocol Buffers are a binary format. Binary formats are more space-efficient than text-based formats such as JSON. [29]
- gRPC benefits from features of HTTP/2 such as pipelining and compression. [32]

Since the cached data will be stored in the Protocol Buffer format, the cached data will be more space-efficient.

Pandit is aware of all clients that have previously queried an endpoint. Pandit can sync the cache with the local daemon on each host the clients are running on. This means that the data will be widely available close to the clients. When it sees a query for the same data, it will return the cached data. Users define if and how long the data should be cached, as well as how to recognise a query for the same data.

Pandit's features provide a great deal of flexibility for users to define their own caching, and querying strategies.

Chapter 3

Design

3.1 Architectural Diagram

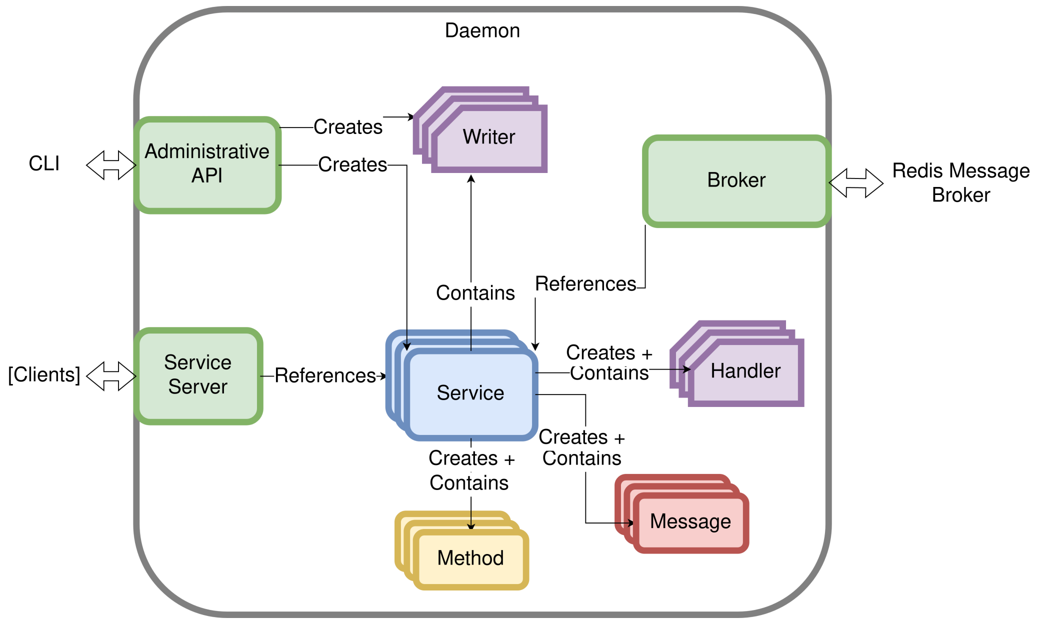


Figure 3.1: Architectural overview of the daemon, describing how its components are created, referenced and stored in memory.

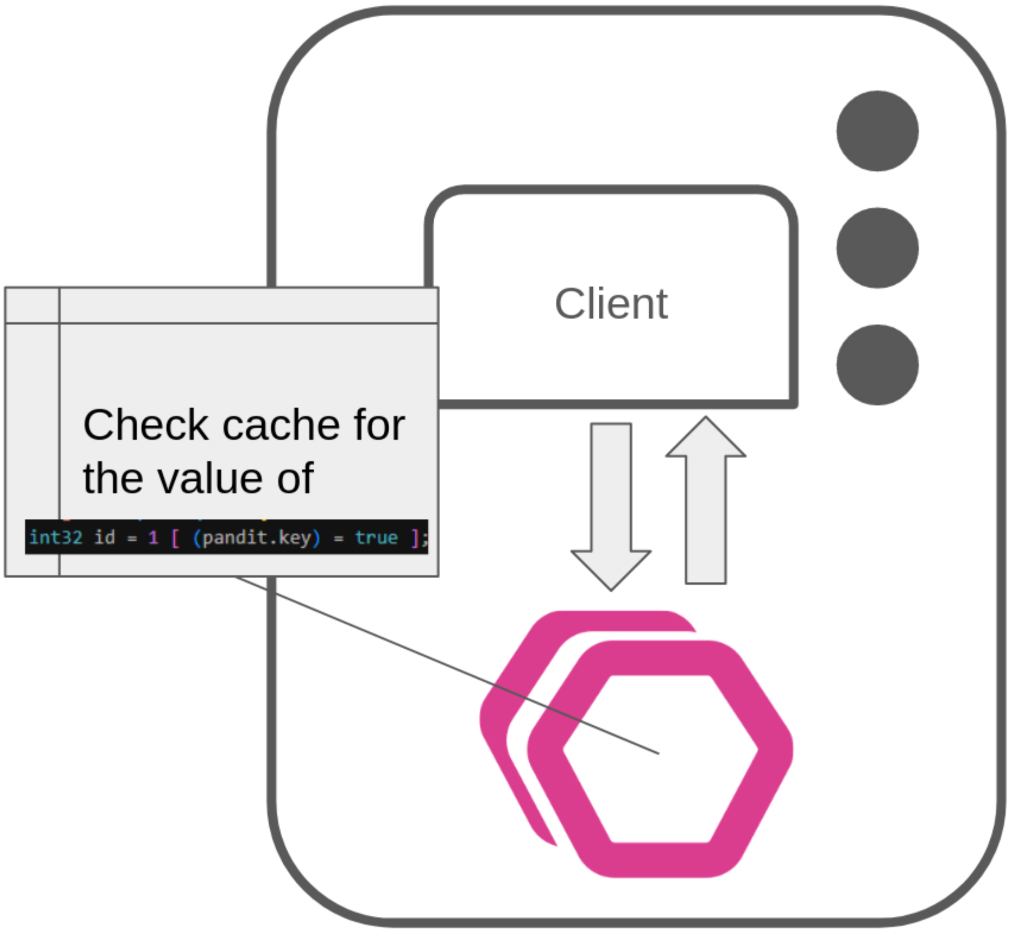
3.2 Overview

A single instance of the Pandit daemon will be run on each host in a cluster of machines. When a client queries a service, the daemon will first check the cache for the data. If it cannot find it, it will delegate the request to an authoritative container in the cluster.

Since the mapping between the data returned by the authoritative container and, the data returned by the gRPC service is user-defined. Pandit provides options to annotate specific fields in return types to allow for the efficient caching of common requests; greatly reducing the amount of queries that need to be sent to the authoritative container.

Each daemon will be responsible for handling all queries sent by clients on their node. As well as handling all queries sent to servers on their node.

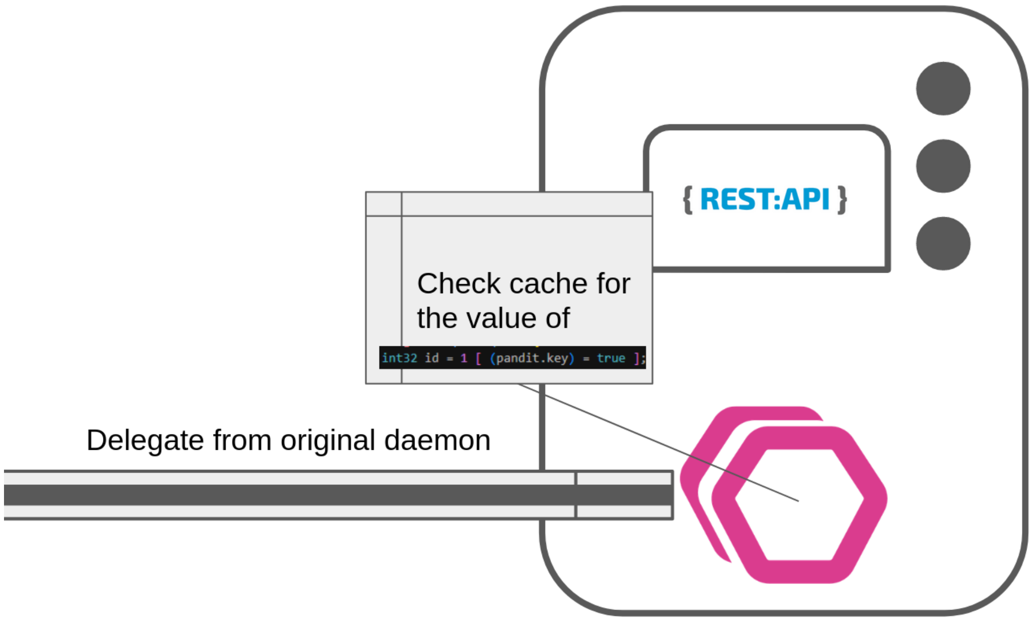
When a query is received by a daemon, it first checks its local cache to ensure that the data has not been previously cached.



```
int32 id = 1 [ (pandit.key) = true ];
```

Figure 3.2: Diagram of cache being looked up after a request is sent to the daemon

If there is no cache hit, and the server that provides this data is on **another** node. The query will be delegated to the node in question.



```
int32 id = 1 [ (pandit.key) = true ];
```

Figure 3.3: Diagram of cache being looked up after a request is delegated from the original daemon to the daemon that is responsible for the application. In this case, a REST API.

Since each daemon maintains its own cache table; a delegated request can have a second chance of a chance of a cache hit.

If no cache hit occurs on the daemon that is responsible, the request will be translated and sent to the application.

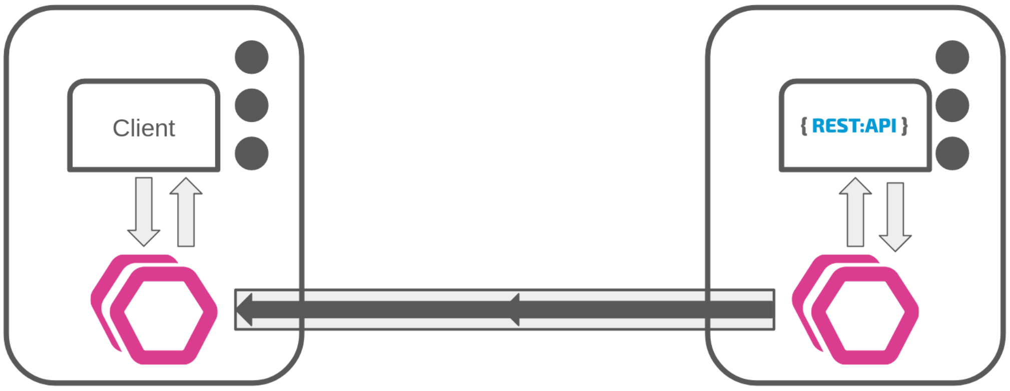


Figure 3.4: Diagram of a query being delegated from one daemon to another. Translating the request and sending it to the application. In this case, a REST API.

The daemon was implemented with the following components:

3.3 Administration API

This is the API that is used to configure the daemon. It is a gRPC service that is exposed via a port on the loopback interface of each host. A CLI client is used to interface with the API. Currently, the only supported method is to add a new *Service* to the daemon. This entails the following:

- Creating a new *Service* based on the parameters provided by the user. This includes providing the Protobuf that was sent from the client as a byte array.
- Adding the service to the *Broker*. The broker tells the other daemons about the service that this node is hosting the service in question.
- Adding the service to the *Service Server*. The server provides a generic gRPC Service for requests from clients.

The Proto file for the Administration API is the following:

```
syntax = "proto3";

package api;

service API {
  rpc StartService(StartServiceRequest) returns (StartServiceReply)
      {}
}

message StartServiceRequest {
  string name = 1;
  bytes proto = 2;
  int32 port = 3;
  oneof container {
    string docker_id = 4;
    string k8s_pod = 5;
    string k8s_service = 6;
    string k8s_replica_set = 7;
    string k8s_stateful_set = 8;
  }
  bool delegated = 9;
}

message StartServiceReply {}
```

This is what the following fields in *StartServiceRequest* represent:

- **Name** - The unique identifier for the service. The identifier is used as the gRPC Service Name by clients querying the service. This is because multiple services can be run from the same Proto file; meaning there must be a unique identifier separate from the Proto file for each service.

- **Proto** - This is the **raw file bytes** of the Proto file. The reason why the entire Proto file is sent in the request, is that the CLI is responsible for managing the Proto files. While the daemon is responsible for parsing the Proto files.

  Storing the Proto files inside the requests allows for all the information necessary to start a service to be directly available inside the request. Making persistent storage of services much simpler to implement and maintain.

- **Port** - This is the Port that the application being proxied is exposed on. It is used to construct the address that the daemon will send traffic to for this particular service.

- **Container** - This is a set of optional fields that are used to deploy services in different Deployment Modes.

  These will be covered in the *Deployment Modes* section.

After the service is created and added to both the Broker and Server. The service must be stored in non-volatile storage for posterity. Pandit daemons can be restarted and redeployed from time to time; Services added to the daemons must persist through this.

After adding a service, Pandit will serialise the raw *StartServiceRequest* and save it to a uniquely named file in its current working directory.

Subsequently, when Pandit starts up, it will look for services in the Current Working Directory. If some are found, they will be sent directly to the *StartService* gRPC method, being added to the local daemon.

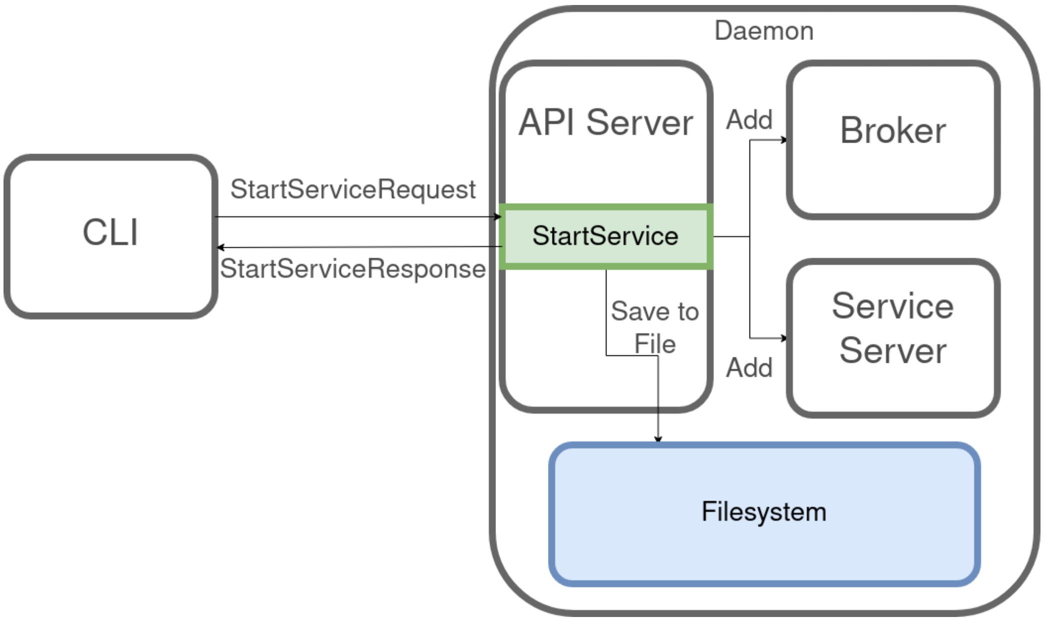


Figure 3.5: Diagram of AddService being called by the CLI and sent to the Administration API

3.4 Service

This is a *struct* that represents a service that is hosted by the daemon. It is created by the *Administration API* and takes in a Protobuf specification.

The constructor parses the provided Proto specification, and builds an in-memory mapping of all methods, messages, and their fields. This includes parsing user defined options imported from the *pandit package*.

Here are the options available to the user in *pandit.proto*:

```
syntax = "proto3";
import "google/protobuf/descriptor.proto";

package pandit;

message CacheOptions {
  bool disable = 60031;
  uint64 cache_time = 60032;
}

extend google.protobuf.FieldOptions {
  string absolute_path = 50020;
  string relative_path = 50021;
  CacheOptions field_cache = 50036;
  bool key = 50037;
}

extend google.protobuf.MessageOptions { string path = 50030; }

extend google.protobuf.MethodOptions { CacheOptions cache = 50034;
    }

extend google.protobuf.ServiceOptions {
  string name = 50010;
  CacheOptions default_cache = 50035;
}
```

This is what the options are used for:

- **Cache Options** - This defines the *caching strategy* for particular Messages, Fields and Services.

  It contains the field **disable**, which when set to true disables the caching of the resource. The field **cache_time** defines the amount of time this revision of the cache should be stored for before being invalidated. If neither field is set, the cache will persist permanently.

- **Key** - The primary key of a message - The key is required to be put on *exactly one* field in each input message. The reason for this is twofold:
    1. It is required to build out the cache table on each daemon. The value of the primary key is serialised and used to index the cache; per particular method on a daemon.
    2. It may be used in the translation between the Protocol Buffer and the desired format. For example, SQL requires a primary key for many commands.
- **Name** - The name of the service - In lieu of the name of the Service defined in the Proto file. A custom name can be used to identity the gRPC service by using this custom option.
- **Path** - This defines the path of a message in an abstract representation parsed from the response from the proxied application.

  This is used in situations where the required data in a JSON field is actually a sub-object of the root object provided.

  For example, take this JSON response:

```
{
    "object": {
        ...
    }
}
```

  If you wanted your output message to be parsed from the *object* field instead of the root object. You would define your message as the following:

```
message ExampleResponse {
    option (pandit.path) = ".object";
    int32 id = 1;
    string user = 2;
}
```

  This is parsed via an abstract representation of the structured data, and so is not format-specific. The format of the path string is based on *jq's parsing language* [10]. It is also designed to be able to parse any structured data format, that has been implemented inside Pandit. Not just limited to only JSON.

- **Absolute Path** - This defines the path of a field in a message; like in the message option **Path**. It uses a parsing language to define the path. In this case, however, it defines the path from the base object to the specific field that should be mapped.
- **Relative Path** - Like the **Absolute Path**. Defines a mapping between a field in the abstract representation of the structured data, and the output message. In this case, the root object is assumed to be the message that the field is defined in.

  Take the following example:

  ```
  message ExampleResponse {
      option (pandit.path) = ".object";
      int32 id = 1; [(pandit.relative_path) = ".user.id"]
      string user = 2;
  }
  ```

  In this case, the field user would be mapped to the path *.object.user.id* in the abstract representation of the structured data.

3.4.1 Sending Data to the Host

The primary use case of the service is to proxy an application. Thus, the service provides a method used to provide data to the application.

Firstly, it translates the request using a method on the Handler called *ToPayload*, to the required format for the application being proxied.

It then passes the translated request payload, as well as the *Handler* for the specific method, to a *Writer*. The writer then sends the request to the proxied application. With the *FromPayload* method on the handler, it then parses and returns the response.

The service struct handles the calling of the writer and handler. This is due to it containing all the necessary data to call the methods on these traits.

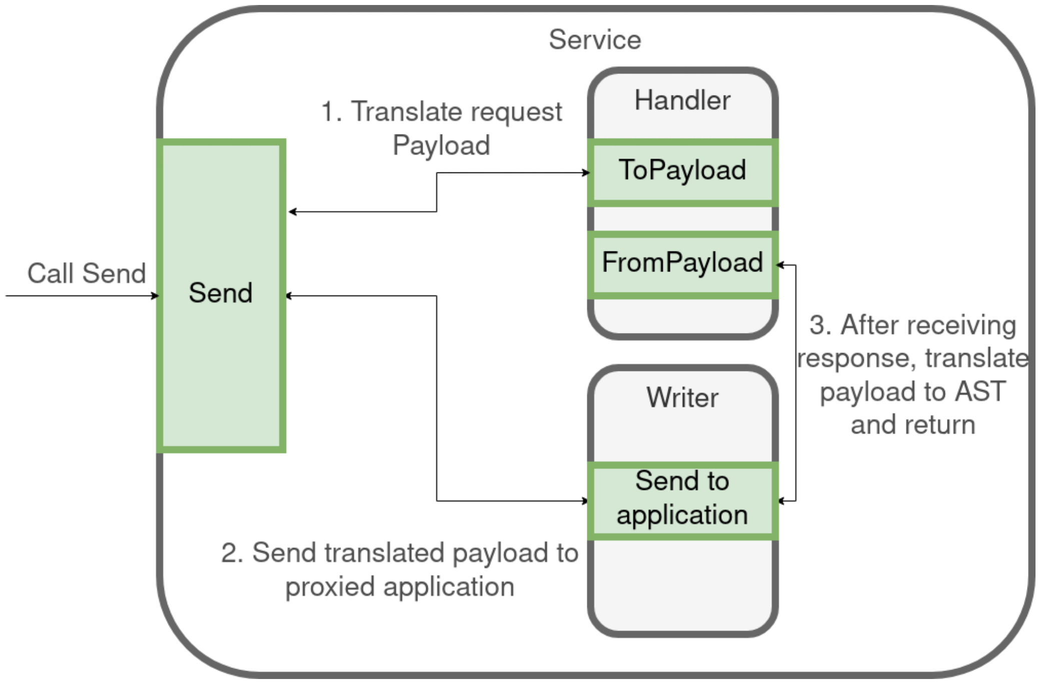

Service
Handler
1. Translate request Payload
ToPayload
FromPayload
Call Send
Send
3. After receiving response, translate payload to AST and return
Writer
Send to application
2. Send translated payload to proxied application

Figure 3.6: A Diagram explaining how the Service sends a payload to the proxied application.

3.5 Message

Pandit stores an abstract representation of the gRPC messages used by Service's methods. It contains the following fields:

- The message name.
- The path of the message - Used during the aforementioned translation stage. Used to parse a returned payload that has the necessary data inside a field.
- Fields - This contains a map of all the fields defined in the Proto message.
- Message - This contains metadata such as options defined inside the Proto message.

The primary purpose of the Pandit Message is to provide the methods used to convert a Protocol Buffer payload to an Abstract Representation called *Fields* and vice-versa.

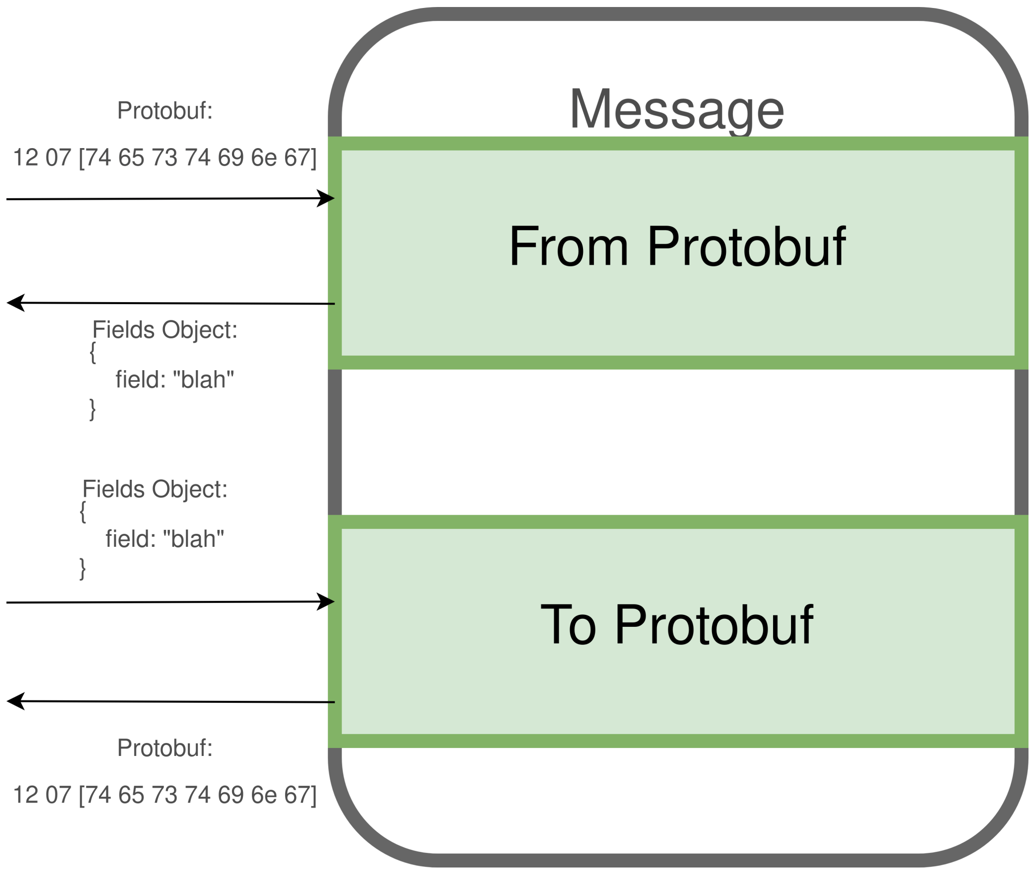


Figure 3.7: A Diagram of the public methods available on the Message struct

3.6 Method

This represents a Protocol Buffer Method. A Method is defined by an input Message and an Output Message. It will only contain two instances of this struct.

The Pandit method contains the following:

- All the Proto options users have set on the particular gRPC method.
- The Handler for this particular method.
- The input (parameters) and output (returned) messages.
- The primary key parsed from the input message. This will be used for building a cache table on the local daemon.

3.7 Writer

This is a trait [40] that represents a way of sending a query to the authoritative container. It takes in abstract representations of the Protocol Buffer Message called *Pandit Messages* and a *Handler*. The writer uses the *Handler* to convert the data to a bytes array. It then constructs a full request in the format required with the payload from the handler. It does this by encapsulating the payload with the required headers.

For example, the Handler could return a JSON payload. The writer would then encapsulate the payload in a series of HTTP headers. This approach is designed to be implemented for any format, for example, SQL. Finally, it then sends the request to the authoritative container.

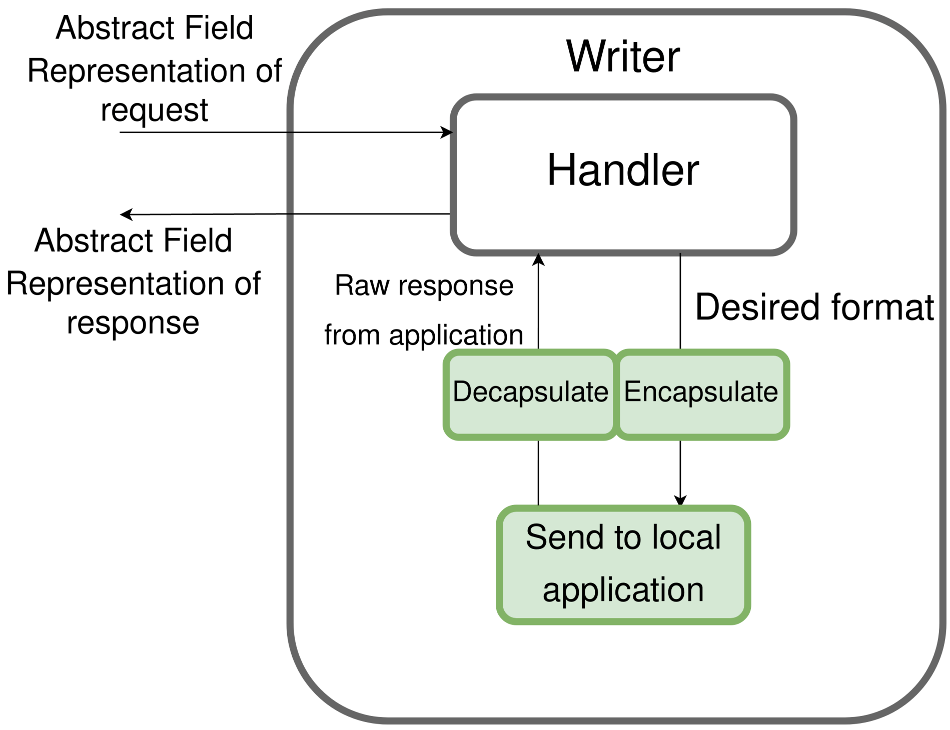

Abstract Field Representation of request
Writer
Handler
Abstract Field Representation of response
Raw response from application
Desired format
Decapsulate
Encapsulate
Send to local application

Figure 3.8: A Diagram of the writer communicating with the local application

3.8 Handler

The *Handler* is used to convert to, and from, the abstract fields representation of the data provided to the writer.

The Handler is a trait (an interface) that provides the following methods:

- **To Payload** - This takes in an abstract field representation parsed from the gRPC request. Through the relevant Pandit Message, converts it into the desired payload that should be sent to the server.
- **From Payload** - This takes in a buffer of bytes containing the decapsulated payload received from the proxied application. Then converts it into the abstract field representation.

3.9 Broker

The Broker is the glue that allows the daemon to be aware of, and communicate with, other daemons in the cluster. The broker interfaces with a Redis [4] instance. The instance is connected to all the daemons to facilitate this.

The reason behind Redis being chosen is as follows. Other message brokers [13] such as Kafka had Rust libraries that were less documented and maintained. In comparison, Redis has an easy to use, well documented Rust library. It also provided all the features Pandit required of a message broker.

The Redis instance can be deployed in any manner. Just as long as it exposes its API to each daemon.

When constructing the broker, the user provides the address of the Redis instance. It will maintain a connection to the Redis instance for the entirety of the run time.

The broker will maintain a connection to Redis's PUB/SUB [5] throughout the lifetime of the program. It will listen for updates to subscribed topics, such as cache updates to subscribed services.

It provides many methods to interact with the Redis instance:

- To add a new *Service* to the broker.

  This will send information to Redis such as:

  - The cache options that were set.
  - The address of the current daemon. This will allow other daemons to look up the name of the service, and find the address of the daemon that's responsible for it.

  - The methods this service has. This includes a serialised version of the entire Method struct.
  - The input and output messages for each method. This includes a serialised version of the entire Message struct.

- To query the broker for the address of the daemon responsible for a *Service*. This will look up the value that was set by the "add service" method.

- To subscribe to cache updates for a particular method. This will be updated in a listener thread that subscribes to Redis's PUB/SUB feature.

- To publish new cache to all interested daemons in the cluster. This will send a serialised version of the necessary data to said daemons.

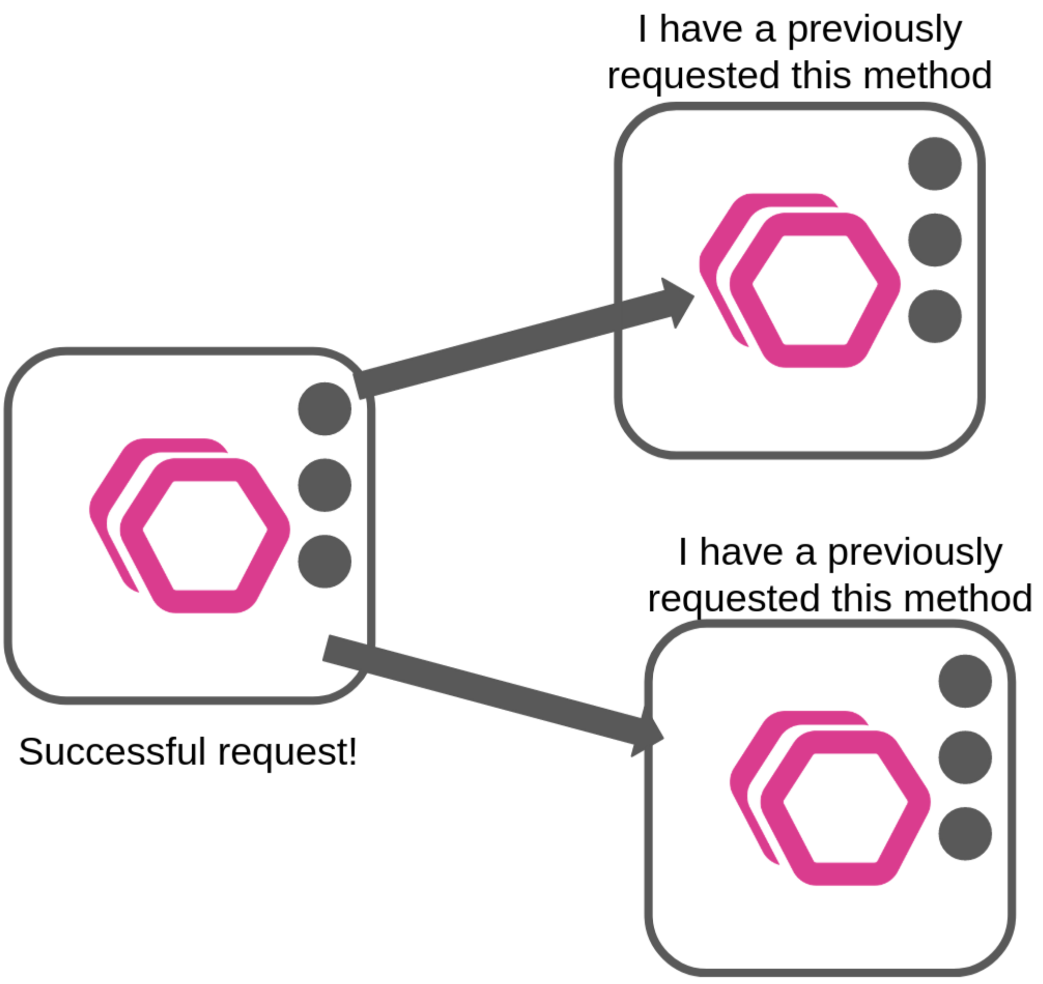

I have a previously requested this method
I have a previously requested this method
Successful request!

Figure 3.9: A Diagram of cache subscription from the perspective of the daemons that are subscribed

- To check if there is a cache hit locally for a particular query. This will read from a local table. The table is updated automatically via the Redis PUB/SUB functionality.

3.10 Service Server

This provides a custom gRPC service handler. It is responsible for handling all requests from clients on the host it's running on. When a gRPC request is received, it will parse the service and method name from the request. It will first check the cache for the data based on this information. If it cannot find it, it will delegate the request to the *Service*.

However, the *Service* may not be available on the current host. In this case, the request will be delegated to the host that the service is available on. The host is found by querying the *Broker* for the address of the host the service is on. After a response is received, it will publish the data to the cache using the *Broker*.

It will also check with the broker to ensure that if the daemon hasn't subscribed to the cache for this method, it must do so.

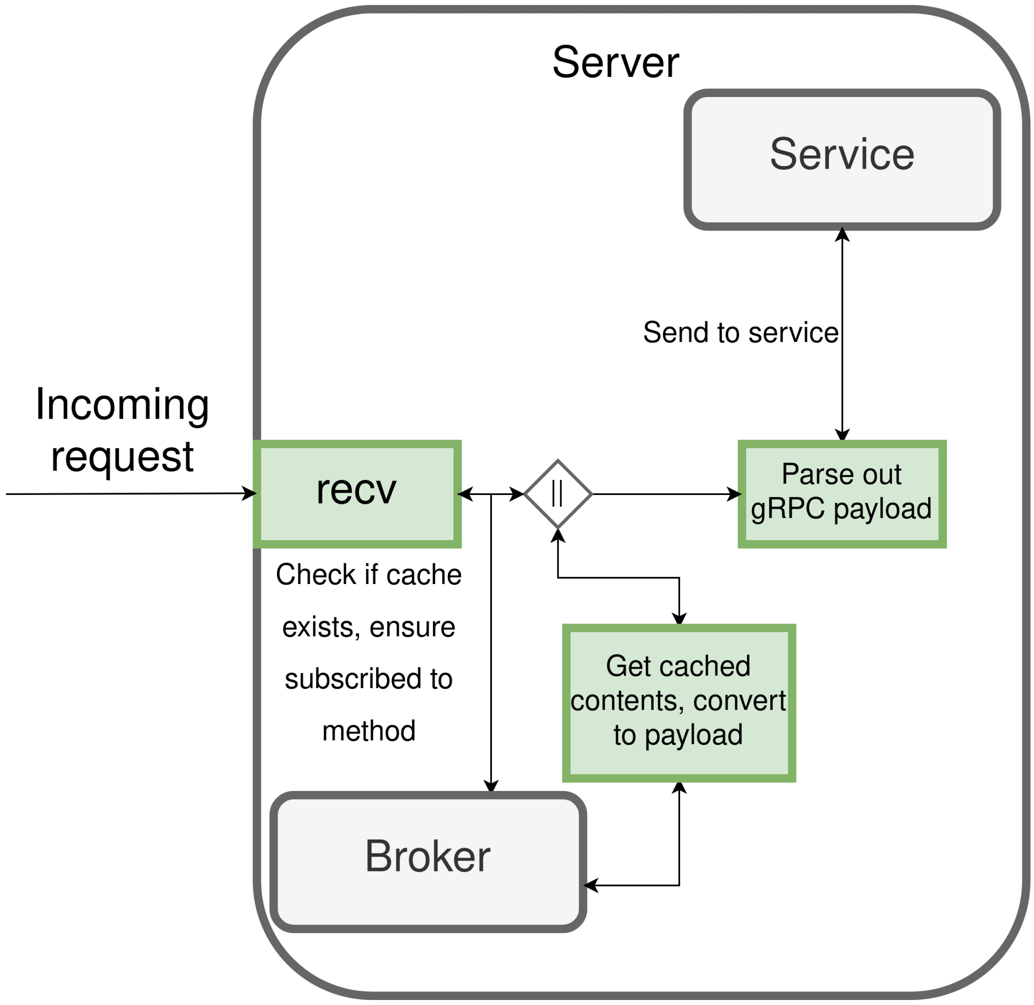
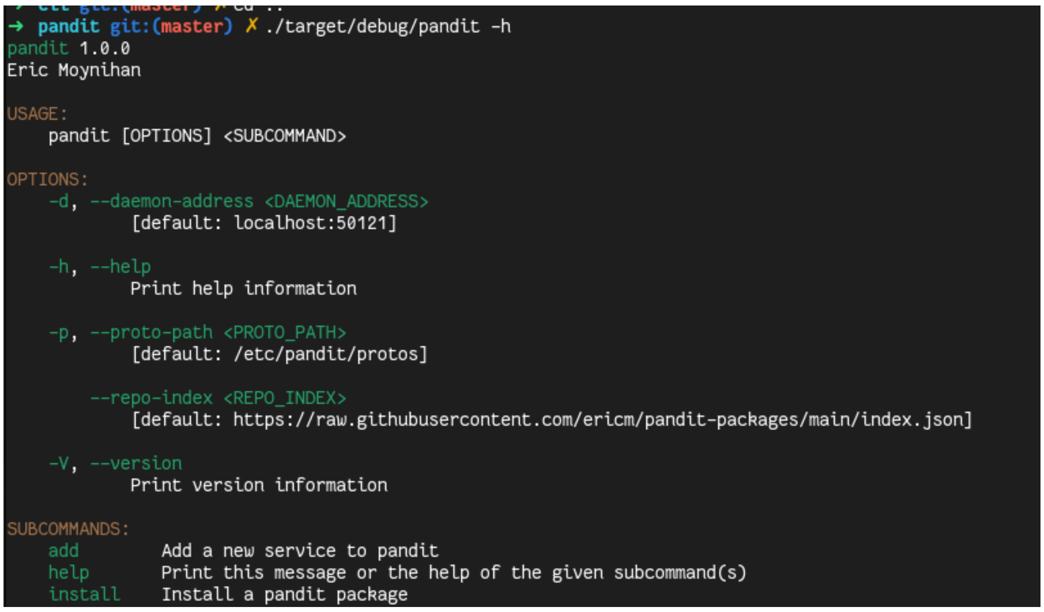

Server
Service
Send to service
Incoming request
recv
||
Parse out gRPC payload
Check if cache exists, ensure subscribed to method
Get cached contents, convert to payload
Broker

Figure 3.10: A Diagram of the operation service server.

3.11 The CLI

The CLI provides a command line interface to configure the daemon. It imports the gRPC client library that was generated by the Protobuf compiler. The CLI uses the client to interface with the daemon's *Administration API*.

It runs as a standalone binary. By default, it connects to the daemon via a known port on *localhost*. However, a flag called *–daemon-address* can be passed to modify the address it will connect to.

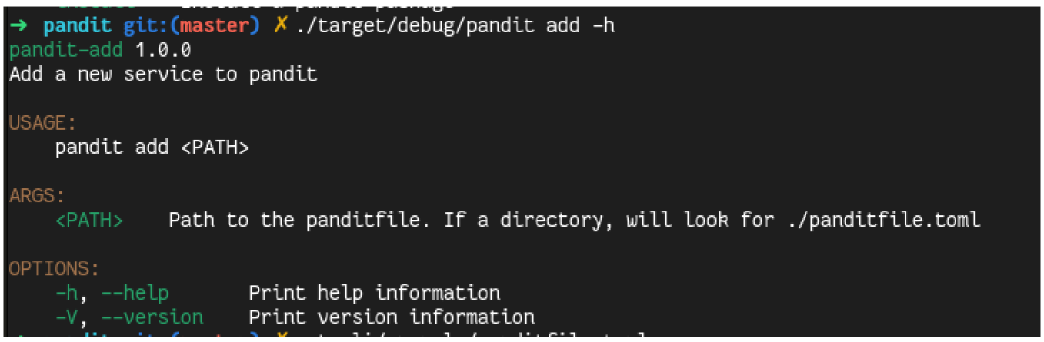


```
→ pandit git:(master) ✗ ./target/debug/pandit -h
pandit 1.0.0
Eric Moynihan

USAGE:
    pandit [OPTIONS] <SUBCOMMAND>

OPTIONS:
    -d, --daemon-address <DAEMON_ADDRESS>
            [default: localhost:50121]

    -h, --help
            Print help information

    -p, --proto-path <PROTO_PATH>
            [default: /etc/pandit/protos]

        --repo-index <REPO_INDEX>
            [default: https://raw.githubusercontent.com/ericm/pandit-packages/main/index.json]

    -V, --version
            Print version information

SUBCOMMANDS:
    add        Add a new service to pandit
    help       Print this message or the help of the given subcommand(s)
    install    Install a pandit package
```

Figure 3.11: A screenshot of the help menu from the CLI.

It supports the following commands:

- **Add** a new service to pandit - This will take in a path to a *Pandit file.* This is a TOML file that defines how the service should be run. It includes fields such as the name of the service, the proto file that should be used, (which is found in the path pointed to by the *–proto-path* flag) and the port that the application is exposed on.



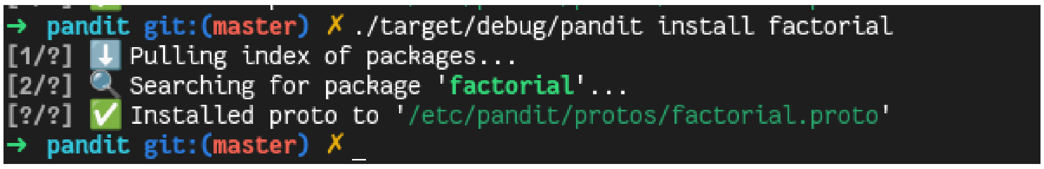
```
→ pandit git:(master) ✗ ./target/debug/pandit add -h
pandit-add 1.0.0
Add a new service to pandit

USAGE:
    pandit add <PATH>

ARGS:
    <PATH>    Path to the panditfile. If a directory, will look for ./panditfile.toml

OPTIONS:
    -h, --help       Print help information
    -V, --version    Print version information
```

Figure 3.12: A screenshot of the help menu for the *add* command from the CLI.

- **Install** a Proto file. In this context, we'll call it a *package* due to it containing extra metadata.

  This is achieved by:

  1. Check a centralised repository for a list of available services.
  2. Based on the name provided to the command, get information from the repository on the Proto file. This includes:
     - The version of the package.
     - The URL to the Proto file.
     - A URL to the README (for display purposes, unused during installation).
     - A URL an image/logo (for display purposes, unused during installation).
     - An optional "image". This is either a link to a Docker Image [34] for Docker or a Helm Chart [16] for Kubernetes.
  3. Install the "image" to the system if present.
  4. Download the Proto file to the path set by *–proto-path* (by default it will be */etc/pandit/protos*).



```
→ pandit git:(master) ✗ ./target/debug/pandit install factorial
[1/?] ⬇️ Pulling index of packages...
[2/?] 🔍 Searching for package 'factorial'...
[?/?] ✅ Installed proto to '/etc/pandit/protos/factorial.proto'
→ pandit git:(master) ✗ _
```

Figure 3.13: A screenshot of the installation succeeding for a package via the CLI.

3.12 Deployment Modes

Pandit is designed to be deployed within pre-existing infrastructure. Meaning, it needs to be able to be easily deployed in different types of distributed environments.

3.12.1 Basic Deployment Mode

This mode allows Pandit to be deployed on a set of servers that have network connectivity. A single Pandit daemon is deployed by the System Administrators on each server. They are then all configured to connect to a Redis endpoint. This allows the servers to communicate information on the services running on each server.

Services are added by running a command on the CLI. The service options available in this mode will be:

- **Name** - Used to identify the service.
- **Proto** - The name of the Proto file installed on the system. Used to define the interface and translation method for the service.
- **Port** - Used to identify the network socket the daemon should connect to. In order to communicate with the application.

3.12.2 Docker Deployment Mode

Background

Docker is a platform used to deploy containerised applications. It runs as a daemon on a machine. It provides an API to manage containers in runtime. Docker is used to isolate single applications using Linux kernel features such as cgroups. Cgroups allow for CPU, memory, disk and network isolation for a collection of processes.

This process is known as containerisation. Docker containers are deployed by providing Docker an Image. A snapshot of the filesystem is passed in to the running process; it contains all the dependencies it requires.

Docker gives each container its own IP address by default. This allows applications on the host machine to interface with the processes in containers via a bridge network. This also provides inter-container network connectivity.

Docker provides resources called ’volumes’ to running containers. These come in three types:

- **Bind Mounts** - Maps between paths on the container's filesystem and the host's filesystem.
- **Docker Volumes** - Maps between a persistent storage resource managed by docker and a path on the container's filesystem
- ***tmpfs* Mounts** - Provides a volatile in-memory filesystem to a path on the container's filesystem.

Docker also provides a 'network' resource that allows containers' networks to be bridged together. By default, containers are all added to the 'default' network. However, they can be configured to be connected to custom docker networks. Custom networks allow for selective isolation of network connectivity between containers.

Docker's API allows users and applications to query information about running containers. Including its IP address, the ports it has exposed and the image it's based on. It also allows resources such as containers and networks to be programmatically created. Docker's API is accessible on the host system via a Unix Socket, which provides an HTTP API that can be queried via docker's provided CLI or programmatically.

Design

Pandit in this deployment mode runs is designed to be run in a Docker container. It's aware that it's in a docker environment because it has been started with the command line argument *–docker*. The container will be given a *Bind Mount* to the Docker API's *Unix Socket*. The bind mount allows access to the Docker API inside the container.

Pandit's Service deployment configuration specification includes the following additional option when Docker Deployment Mode is enabled:

- **Container ID** - A unique ID used by Docker to identify a container on the system.

When a service is added, Pandit queries the Docker API for a container matching the provided container ID. If found, Pandit will do the following:

- It will create a Docker network specifically to facilitate communication between Pandit, and the application.
- It will add both Pandit's container, and the application's container, to the network. Both of them will be assigned an IP address.

Pandit will then construct an address to be associated with the *Service.* It uses the IP address and port received during construction to achieve this.

When Pandit must query the application in the future, it will use this address to do so.

Clients can query Pandit in a number of ways:

- They can be deployed in Docker, and interface with Pandit via a bridge network.
- They can be running outside of Docker. Pandit's gRPC API port can be exposed on the host system through Docker configuration. The client can then interface with Pandit via this port.

Multiple Hosts

When dealing with multiple hosts running separate Docker systems. A Redis instance will be deployed on one of the hosts. It will be exposed via a port forwarding. Pandit's APIs on all hosts will also be port forwarded.

During deployment, each instance will be provided the IP address of the host it's running on. Allowing instances to announce that as their IP address to other instances via Redis.

When a query must be delegated from one server to another. Provided the hosts have network connectivity via the provided IP addresses, Pandit will be able to handle it.

3.12.3 Kubernetes Deployment Mode

Background

Kubernetes is a container orchestration system. Like Docker, its main focus is containerising applications.

A basic unit of work in Kubernetes is a *Pod.* A Pod consists of one or more containers running on a single host machine. A host machine in Kubernetes is known as a *Node.* A set of Nodes that are orchestrated by Kubernetes are known as a *Cluster.*

Pods are ephemeral, meaning they are created and destroyed at will by the *Kubernetes Scheduler.* This is a program in Kubernetes responsible for the management of resources on each node in a cluster. Pods can be destroyed and redeployed on different nodes for reasons, such as balancing resource usage. This is possible because any stateful components of an application are stored in *volumes.* Like Docker Volumes, they maintain persistent storage for an application. Kubernetes usually shares a single volume with multiple Pods. A shared volume, allows for a shared state between the pods. Kubernetes handles the consistency and storage of volumes across Nodes automatically. Allowing for easy deployment of stateful applications across servers.

Kubernetes also define a set of resources that are an abstraction on top of the concept of Pods. These include:

- **Services** - Since a single Pod can only be deployed on a single node. If a user wishes to deploy an application across multiple nodes; there is no way to address that set of pods with a single IP address. A *Kubernetes Service* is defined as being a set of Pods. It provides an IP address for this set of Pods. When a request is sent to this address, Kubernetes will Load Balance the requests between the Pods.

- **ReplicaSets** - A deployment of a set of Pods that are designed to be stateless. A *ReplicaSet* ensures a set number of replicas of Pods are running somewhere in the cluster. *ReplicaSets* don't provide stateful storage to Pods. Hence, should only be used to deploy stateless applications that require to be run in a distributed manner. Unlike *Services*, they don't provide an IP address or Load Balancer. Requiring the Pods to be communicated with directly, or via a separately deployed *Service*.

- **StatefulSets** - Like *ReplicaSets*, *StatefulSets* run a set of Pods that combined make up a stateful application. *StatefulSets* however, labels the Pods. The identity of the Pods are relevant, because it is used to deploy applications that must be deployed in a specific order. The Pods are also able to be given shared volumes, in order to maintain state. Pods are still ephemeral, meaning they can still be deleted and recreated. When they are recreated, they are given the exact same configuration; ensuring a Pod with a specific ID and network identifier always exists.

- **DaemonSets** - A deployment that puts a single Pod on every single Node. It provides shared storage volumes that allow Pods to communicate and share state. It also allows Pods to expose ports on the node it's running on's network. This allows for communication between the pods and daemon on that node.

Kubernetes provides an API that allows for the deployment of said resources. All resources can be deployed via a YAML file. Defining what resource should be deployed, where it should be deployed, etc. The API is available to be queried inside a Pod. The client inside the Pod must be able to authenticate itself, through a user account provided to Kubernetes via a YAML file.

Design

Pandit in this deployment will be deployed inside the Kubernetes cluster via a *DaemonSet*. Allowing it to run on each Node in a cluster. It will be deployed with access to the Kubernetes via a Kubernetes User. The user has access to information on Pods, Nodes, Services, ReplicaSets and StatefulSets.

It knows it's inside a Kubernetes environment due to it being deployed with the *–k8s* flag (*k8s* is shorthand for Kubernetes).

When a *Service* is added to the Pandit, it is defined with extra options. The routine that parses these extra options ultimately returns one or more IP addresses. This to give the node in question a route to the application. If more than one IP address is returned, the IPs are Load Balanced using Pandit's built-in Load Balancer. One of the following options can be added to the configuration file:

- **Kubernetes Pod** - This defines a Pandit service as a singular Pod deployment. The Pod ID is passed to Pandit, which is then used to query for the Pod. If the Pod is on the Node that the service was added to; the IP address of the Pod will be used to construct the address of the Pandit service.

  If the Pod is on a different Node, Pandit will query the address of its Node via the Kubernetes API. It will then delegate the *Add Service* request to the Node in question. That Pandit instance will then announce itself as the sole host of that particular Pandit Service via Redis.

  When the Pandit Service is queried in the future; all queries will be delegated to the Node that the Pod is on.

- **Kubernetes Service** - When a *Pandit* Service is defined to be a *Kubernetes* Service, using the Service ID provided. Pandit will first query the Kubernetes API for details about the service.

  Pandit will parse out the *Label Selectors* of the Kubernetes Service. Label Selectors are a map of Keys and Values that define what Pods should be part of an abstraction, such as a Service. All Pods that have labels that match these Label Selectors, will be proxied by the Kubernetes Service.

Pandit will query the API for the Pods that match these Label Selectors. It will then query each Pod for its Node name, using that Node name to query the API for its Node IP address. Pandit will then construct a set of IP addresses. These will represent all the Nodes that host a Pod proxied by the Kubernetes Service in question.

All Nodes in this set will be sent this *Add Service* request. This is to ensure they are all capable of parsing queries for the Service in question.

The resulting set will be announced as the IP addresses of the Pandit Service to all other daemons. When a request is sent to this Pandit Service, it will be called via the load balancer. This allows Pandit to balance the load of Service parsing among all the nodes, that have a Pod proxied by the Kubernetes Service.

If the Kubernetes Service is using *topology-aware traffic routing [21]*. The Node that the Pandit request was sent to will contain the backend Pod that will process the query. Making this method of Load Balancing quite efficient.

- **ReplicaSets** or **StatefulSets** - These deployments can both be parsed using the same method due to their similar deployment structure. For the usage of either of these deployments as a Pandit Service. It is assumed that any Pod represented by the *labels [20]* defined in the deployment, is capable of responding to queries from Pandit.

  Like the process of deploying a Pandit Service consisting of a *Kubernetes Service*. A set of Node IP addresses will be constructed and announced for every Node that has a Pod belonging to the deployment running on it. This set of Node IP addresses would be Load Balanced the same way.

  Unlike *Kubernetes Services*, these deployments don't announce a Virtual IP address [22] that is handled by the Kubernetes Load Balancer.

  This means that Pandit will connect with the relevant Pods on its own host machine, by storing the relevant IP addresses locally. If multiple Pod replicas exist on a particular Node. Those will also be Load Balanced locally by the daemon in question.

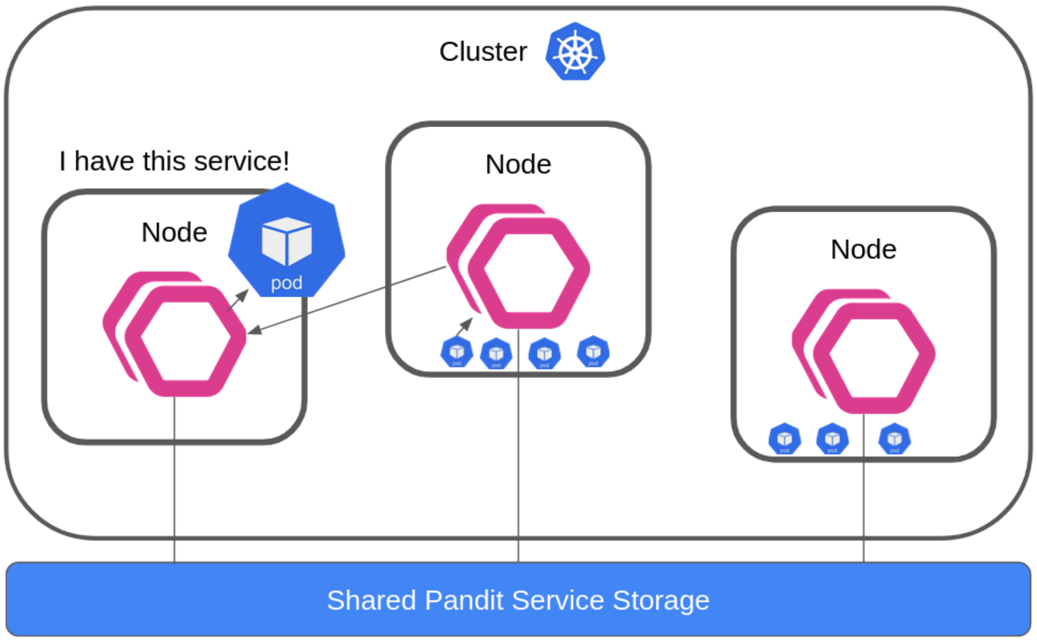


Figure 3.14: A diagram of Pandit deployed in Kubernetes. A service consisting of one Pod is announced on by the leftmost daemon. A client on the centre node subsequently sends a request to this service.

Handling Pod Termination

Regardless of the method of deployment. Pods can be created and destroyed automatically by the Kubernetes Scheduler. This causes a problem for Pandit, because Pods may be deployed on different Nodes. In order for Pandit to be able to handle requests for data on these Pods, it must do the following:

1. Run an event loop that listens for a Pod termination event via the Kubernetes API.
2. When a Pod is terminated, broadcast via Redis that the Pod in question has been terminated.
3. All daemons on all Nodes will start listening for a new Pod that matches the description provided in the Service Specification. All daemons will have access to this specification because they all share a storage volume.

4. When a daemon finds the newly created Pod, run the *Add Service* command locally and announce the Pandit Service.

This deployment mode allows for a lot of flexibility in how deployments should happen. It acts as an extension to Kubernetes itself in a way, due to tightly integrating into the Kubernetes ecosystem.

Chapter 4

Implementation

In the previous chapter, the architecture of the daemon and CLI were outlined. In this chapter, the details on how this was implemented will be outlined.

4.1 Fields & Values

This is the implementation of the **abstract fields representation**.

4.1.1 Fields

The implementation of *Fields* is quite simple:

```
pub type FieldsMap = DashMap<String, Option<Value>>;

#[derive(Debug, Clone)]
pub struct Fields {
    pub map: FieldsMap,
}
```

It is an encapsulated map string keys to options [43] of *Values*. It's a struct encapsulating a map is, because it must implement traits that the underlying map does not implement.

For example, it implements Rust's built-in equality and hash trait:

```
impl std::hash::Hash for Fields {
    fn hash<H: std::hash::Hasher>(&self, state: &mut H) {
        // Build a binary heap of values in a map, ensuring they
            are sorted.
        let vals: BinaryHeap<Value> = self
```

```
            .map
            .iter()
            .filter_map(|entry| entry.value().clone())
            .collect();
        for val in vals.into_iter_sorted() {
            val.hash(state);
        }
    }
}

impl PartialEq for Fields {
    fn eq(&self, other: &Self) -> bool {
        // Equality of the maps is defined by hashing a sorted list
            of thier values and comparing them.
        let mut s_hash = DefaultHasher::new();
        self.hash(&mut s_hash);
        let mut o_hash = DefaultHasher::new();
        other.hash(&mut o_hash);
        s_hash.finish() == o_hash.finish()
    }
}
```

4.1.2 Values

Values are integral to the abstract representation of fields. Implementing them proved challenging due to their unique requirements:

They must be converted to and from Protocol Buffers. As well as to and from any structured data format.

The approach chosen was to implement values as an enumeration of a set of general data types:

```
pub enum Value {
    String(String),
    Bytes(Vec<u8>),
    Int(Arc<dyn Integer>),
    Float(Arc<dyn Floating>),
    Bool(bool),
    Enum(ProtoEnum),
    Message(Fields),
    Array(Vec<Value>),
    None,
}
```

All of these data types represent one or more Protocol Buffer data types. The purpose of this is to create a general set of data types, that the Protocol Buffer interpreter can pull from.

The Protocol Buffer interpreter will eventually need more concrete data types. This is particularly apparent with the integer and float data types.

These may need to be used as anything from an unsigned 8-bit integer or a signed 64-bit integer. These were all abstracted behind the *Integer* and *Floating* traits; in order to simplify the conversion process on between a custom format such as JSON and a *Value*.

The *Integer* and *Floating* traits are defined as follows:

```
pub trait Integer: Sync + Send + std::fmt::Debug {
    fn to_i32(&self) -> i32;
    fn to_i64(&self) -> i64;
    fn to_u32(&self) -> u32;
    fn to_u64(&self) -> u64;
}

pub trait Floating: Sync + Send + std::fmt::Debug {
    fn to_f32(&self) -> f32;
    fn to_f64(&self) -> f64;
}
```

These methods are designed to be implemented by any concrete data type. For example, the unsigned 64-bit integer is implemented as follows:

```
impl Integer for u64 {
    fn to_i32(&self) -> i32 {
        self.clone().try_into().unwrap_or(0)
    }

    fn to_i64(&self) -> i64 {
        self.clone().try_into().unwrap_or(0)
    }

    fn to_u32(&self) -> u32 {
        self.clone().try_into().unwrap_or(0)
    }

    fn to_u64(&self) -> u64 {
        self.clone()
    }
}
```

These methods are implemented to allow the Protocol Buffer interpreter to choose what concrete type it wants; regardless of the underlying type. The interpreter cannot be sure that, for example, the JSON parser decided to parse a ”number” as a ”u32” as opposed to an ”i64”.

4.2 Serialising

Many objects must be serialised, for example, to be transferred from one daemon to another. The Serde library [6].

This provides interfaces such as *Serialize* and *Deserialize*. They can be derived [41] for structs like so:

```rust
#[derive(Serialize, Deserialize)]
pub struct SQLValue(pub Vec<u8>);
```

They can also be implemented manually for a custom serialisation strategy:

```rust
impl Serialize for Fields {
    fn serialize<S>(
        &self,
        sr: S,
    ) -> Result<<S as serde::Serializer>::Ok, <S as
        serde::Serializer>::Error>
    where
        S: serde::Serializer,
    {
        use serde::ser::SerializeMap;
        let mut map = sr.serialize_map(Some(self.map.len()))?;
        for kv in &self.map {
            match kv.value() {
                Some(value) => {
                    map.serialize_entry(kv.key(), value).unwrap();
                }
                None => continue,
            }
        }
        map.end()
    }
}
```

The benefit to working with Serde is that there is an ecosystem of libraries that implement these traits. For example, the *Fields* struct is interoperable with Serde libraries such as *Serde JSON [7]*.

4.3 Handler & Writer Traits

The Writer and Handler are just traits [40] (Rust's version of an interface).
This is the trait for the Handler:

```
#[async_trait]
pub trait Handler {
    fn from_payload(&self, buf: bytes::Bytes) ->
        ServiceResult<Fields>;
    async fn to_payload(&self, fields: &Fields) ->
        ServiceResult<bytes::Bytes>;
}
```

Each method returns a Result [44]. *bytes::Bytes* is a bytes buffer. It can be used to read a series of bytes.

This is the trait for the Writer:

```
pub type WriterContext = HashMap<String, String>;

#[async_trait]
pub trait Writer: Sync + Send {
    async fn write_request(
        &mut self,
        context: WriterContext,
        fields: &Fields,
        handler: &Arc<dyn Handler + Send + Sync>,
    ) -> ServiceResult<bytes::Bytes>;
}
```

The *Write Request* method takes in:

- A context - This is a hash map of strings that can be used to provide extra context to specific writers. For example, the hostname a HTTP header should be set to. This will all be implementation specific.
- Fields - This is the Abstract fields representation of the gRPC request.
- The Handler.

4.4 Message

The Message struct stores information on the Protocol Buffer messages. It is stored within the Method Struct as part of a Pandit Service.

The message stores the following data:

```rust
#[derive(Serialize, Deserialize)]
pub struct Message {
    pub name: String,
    pub path: String,

    #[serde(skip)]
    pub parent: Arc<DashMap<String, Message>>,

    fields: DashMap<u32, Field>,
    pub fields_by_name: DashMap<String, Field>,
    pub message: protobuf::descriptor::DescriptorProto,
}
```

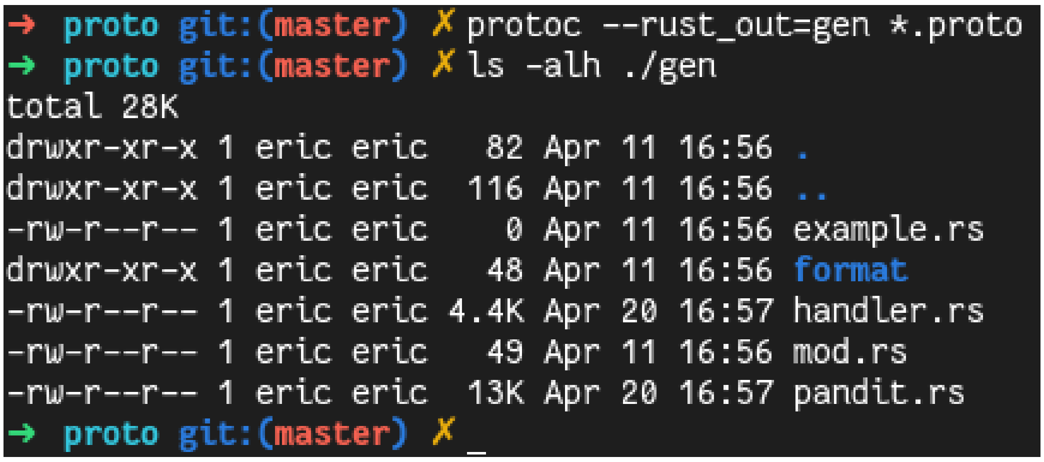
The *parents* field maintains an Atomic Reference Counter [45] (shared reference) to the map of all messages. This is used to aid with parsing gRPC payloads that contain nested messages. It is skipped by Serde due to it not being necessary to be serialised.

4.4.1 gRPC Payload Parsing

The primary functionality of the Message struct is to provide methods to both write a gRPC payload from *Fields*, and create *Fields* from a gRPC payload.

While there is an existing set of libraries to use with Protocol Buffers. The use case this project had meant that said libraries couldn't be used; because the existing methods for using gRPC with Protocol Buffers involve generating a parsing library with code generation.

The binary *protoc [27]* is used to translate the Proto file to the required programming language. The output is then used as a library that provides methods to run both the gRPC server and client.



```
➜ proto git:(master) ✗ protoc --rust_out=gen *.proto
➜ proto git:(master) ✗ ls -alh ./gen
total 28K
drwxr-xr-x 1 eric eric   82 Apr 11 16:56 .
drwxr-xr-x 1 eric eric  116 Apr 11 16:56 ..
-rw-r--r-- 1 eric eric    0 Apr 11 16:56 example.rs
drwxr-xr-x 1 eric eric   48 Apr 11 16:56 format
-rw-r--r-- 1 eric eric 4.4K Apr 20 16:57 handler.rs
-rw-r--r-- 1 eric eric   49 Apr 11 16:56 mod.rs
-rw-r--r-- 1 eric eric  13K Apr 20 16:57 pandit.rs
➜ proto git:(master) ✗ _
```

Figure 4.1: A screenshot of Protoc being used.

This solution will not work for Pandit's use case, due to the Proto files being read on runtime, and running a gRPC server that is able to reflect on multiple gRPC services on runtime.

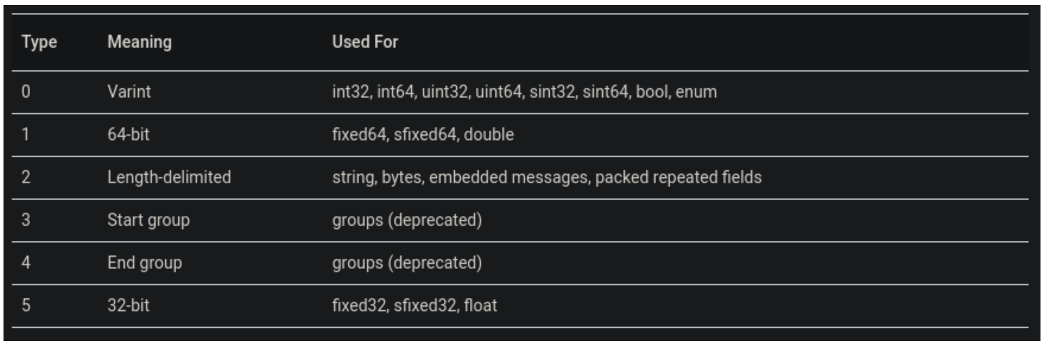
This mean the solution was to implement the majority of the parsing logic for gRPC payload parsing from scratch. It only used a few helper functions imported from existing libraries to help with interfacing with the binary directly.

Implementing the payload parsing from scratch allowed Pandit to benefit from both the flexibility provided by this solution in the form of being able to integrate it with the *Fields* and *Value* structs, as well as efficiency due to it all being written specifically for this use case.

Challenges

Protocol buffers consist of a series of **tags** and **binary data**. The tag is a byte the proceeds the data. It contains a field identifier (the number defined in the Proto file) as well as the **wire type**.

A Wire Type can be best described as follows:



Type	Meaning	Used For
0	Varint	int32, int64, uint32, uint64, sint32, sint64, bool, enum
1	64-bit	fixed64, sfixed64, double
2	Length-delimited	string, bytes, embedded messages, packed repeated fields
3	Start group	groups (deprecated)
4	End group	groups (deprecated)
5	32-bit	fixed32, sfixed32, float

Figure 4.2: A table of the different types of wire types. Copyright Google 2022. URL: https://developers.google.com/protocol-buffers/docs/encoding

The *length-delimited* wire type implies that the next byte will be the length of the coming data. This is used for all variable-length data, including embedded messages.

The main challenges involved with implementing the gRPC payload parsers involved recursion.

When parsing embedded messages, a recursive strategy must be used. This is due to the fact that the specification requires the parser to theoretically parse embedded messages recursively ad infinitum.

The solution was to put the logic for both the reader and writer methods into a private "delimited" version of the methods. The public methods would then call the private method, passing in the number of bytes that the parser should read to. Which in the case of the public method should be the entirety of the buffer.

```rust
pub fn fields_from_bytes(&self, buf: &[u8]) ->
    ServiceResult<Fields> {
    use std::convert::TryInto;
    let mut input = CodedInputStream::from_bytes(buf);
    input.read_raw_bytes(5)?; // Pop gRPC header.
    self.fields_from_bytes_delimited(&mut input,
        buf.len().try_into()?)
}
```

When the parser hits an embedded message, it will read in the length of the message, and then call a method that takes in the Proto for the field that contains the embedded, the byte stream, and the length. It uses the aforementioned *parents* field, to find the specification for the method based on information provided by the field proto. It then handles for the case of multiple embedded messages being repeated in the same field. Then calls the delimited parsing method with the relevant parameters, except this time it's being called on the embedded message.

```
fn parse_another_message(
    &self,
    input: &mut CodedInputStream,
    message_name: &String,
    field: &FieldDescriptorProto,
) -> ServiceResult<Vec<Fields>> {
    use protobuf::descriptor::field_descriptor_proto::Label::*;
    let parent = self.parent.clone();
    let message = parent.get(message_name).ok_or(ServiceError::new(
        format!("no message called: {}", message_name).as_str(),
    ))?;
    let repeated_len = if field.get_label() == LABEL_REPEATED {
        input.pos() + input.read_raw_varint64()?
    } else {
        input.pos() + 1
    };
    let mut output: Vec<Fields> = Vec::with_capacity(2);
    while input.pos() < repeated_len && !input.eof()? {
        let len = input.pos() + input.read_raw_varint64()?;
        output.push(message.fields_from_bytes_delimited(input,
            len)?);
    }
    Ok(output)
}
```

When implementing the Protocol Buffer writer method. The *Fields & Values* proved to make the solution of handling different data types quite simple.

For example, if a method needed a 32-bit integer, this value could be received by simply calling the *to-i32* method on the *Integer* trait.

```
// sint32 required output.
Value::Int(v) => write_value(
    field,
    output,
    protobuf::CodedOutputStream::write_sint32_no_tag,
    v.to_i32(),
    protobuf::wire_format::WireTypeVarint,
)?,
// uint32 required output.
Value::Int(v) => write_value(
    field,
    output,
    protobuf::CodedOutputStream::write_uint32_no_tag,
    v.to_u32(),
    protobuf::wire_format::WireTypeVarint,
)?,
```

4.5 HTTP/JSON Writer

Pandit depends on implementations existing in the codebase of both the Writer and Handler, some example implementations were made. One of these is HTTP/JSON also known as a REST API.

4.5.1 HTTP Writer

The HTTP Writer is an implementation of the Writer trait. It implements the *Write Request* method by:

- Converting the fields to the desired payload with the Handler.
- Using the context provided to the method to intuit the following:
  - The HTTP version of the server.
  - The HTTP method of the request (GET, POST, etc).
  - The full HTTP URI of the request.
- Encapsulate the payload from the handler inside the HTTP headers constructed from the context.
- Send the request to the desired address and return the resulting payload.

4.5.2 JSON Handler

This is an implementation of the Handler trait. It implements the **Payload to Fields** method as follows:

1. Using *Serde JSON [7]*, parse the bytes buffer of the payload into Serde JSON's *Value* object.
2. Traverse the Value tree based on the path provided to it. This will find the relevant object that will be returned.

3. When the wanted Value is found, use Serde JSON to convert it to Pandit's *Fields* struct and return.

It implements the **Fields to Payload** as follows:

1. Convert the fields to a vector of bytes by using Serde JSON.

2. Load the vector's contents to a byte buffer and return.

4.6 Postgres Writer

This is implemented in the form of both a Handler and Writer. For the use case of interfacing with Postgres, the Handler's purpose is converting SQL queries and responses to and from "Fields". The Writer is used to send said queries.

4.6.1 Writer

The implementation of "Write Request" for this writer uses an asynchronous Postgres library to connect to a Postgres instance on the address provided during creation of the writer (this happens during the creation of the service).

When a request is sent, the fields will be converted to a set of SQL commands via the handler.

It will then send the commands to the Postgres instance. The response will come in the form of a series of "Row" structs defined in the Postgres library.

These will be serialised to a bytes buffer and returned. It is the responsibility of the Handler to parse the payload.

4.6.2 Handler

The SQL handler's architecture had to be designed to work with the writer. While still remaining separate components due to the architectural requirements. The "From Payload" method converts to "Fields" by parsing the serialised payload that came from Postgres. It goes through each row and maps its columns to the fields defined in the Pandit "Message".

It handles subsequent rows by mapping them to fields that require embedded messages. This allows Pandit to have embedded messages in SQL. This is achieved by building out a table of "wanted table references", and matching subsequent responses serialised in the payload by the name of their table.

The result is a fully built out ”Fields”. This is then returned.

The ”To Payload” converts ”Fields” to SQL commands. It utilises a query building library called *Sea QL [47]*. The commands are defined in the Proto options for each method. Based on the command type chosen, different things will be added to the query builder.

Some commands such as ”SELECT” require conditions. These are defined on the Proto Message fields. The commands take the format of an additional condition enum on the field.

4.7 Administration API

4.7.1 Building

The implementation of this gRPC server first involved generating the code from the previously defined *api.proto* file. The gRPC library used for this API required a different version of the Protobuf compiler compared to the example files used to test the daemon.

Since multiple different code-gen versions were installed on the system, a Rust build script [39] approach was decided on as opposed to a simple shell script. This allowed rust to manage the dependencies and so multiple Protobuf compilers could be installed.

Here is the build script used:

```rust
fn main() {
    let proto_root = ".";
    println!("cargo:rerun-if-changed={}", proto_root);
    protoc_grpcio::compile_grpc_protos(&["api.proto"],
        &[proto_root], &proto_root, None)
        .expect("Failed to compile gRPC definitions!");
}
```

This will run when the rust project is being built, and if the Proto files are changed; the code will be regenerated.

4.7.2 Adding a Service

The method that the API exposes allows the user to add a service to the daemon. The main process involves:

1. Figure out the Deployment Modes. This will result in a mode-specific method call that will ultimately return the IP addresses of the service.

If it's in Basic deployment mode, the IP address will just be the local host.

2. Create the file structure that Pandit's Service constructor requires in the temporary directory. Since the Service uses a library that parses Proto files from a directory. The service must be set up before the constructor is called.

   All of Pandit's proto library files that provide the options are embedded in the binary. They are then written to the correct file location via a static method call:

   ```
   fn proto_libraries() -> [(&'static str, &'static [u8]); 4]
       {
       [
           ("pandit", include_bytes!("../proto/pandit.proto")),
           ("handler",
               include_bytes!("../proto/handler.proto")),
           (
               "format/http.proto",
               include_bytes!("../proto/format/http.proto"),
           ),
           (
               "format/postgres.proto",
               include_bytes!("../proto/format/postgres.proto"),
           ),
       ]
   }
   ```

   The Proto sent via the StartServiceRequest will then be written to the directory.

3. Construct the Writer based on the provided Proto file.

4. Create the Service.

5. Add the service to both the Broker and Service Server.

4.7.3 Handling Deployment Mode

Docker

Interfacing with the Docker API is done via the library *Bollard [12]*. Since the deployment mode method is run from the gRPC Add Service call; the method is called from the gRPC library's custom asynchronous context.

Rust provides asynchronous features, however, the underlying runtime that provides asynchronicity is decided by the user. The problem is that Bollard is programmed to be used with a specific asynchronous runtime, but it has to be called from the gRPC asynchronous runtime.

The solution was to use a concept called a *channel [9]*. This allows data to be transferred between threads, and hence asynchronous runtimes.

This solution required a dedicated asynchronous listener thread to be run on startup. It would listen for requests via a channel, and return the resulting address or error via other channels.

When a request for a docker container was sent, the commands for the Docker Deployment Mode would be executed and the results returned before being sent back via channels to the gRPC runtime.

The code that listened for requests via channels is as follows:

```
async fn run(&self) {
    loop {
        let container_id = match self.rx.recv() {
            Ok(v) => v,
            Err(err) => {
                log::error!(
                    "an error occurred receiving container id in
                        docker network runtime: {}",
                    err
                );
                continue;
            }
        };
        match self._create_network(container_id).await {
            Ok(host) => {
                self.hosttx.send(host).unwrap(); // Send the IP
                    address back if successful
            }
            Err(err) => {
                self.etx.send(err.to_string()).unwrap(); // Send the
                    error if unsuccessful.
            }
        }
    }
}
```

Kubernetes

For the same reason as Docker, the Kubernetes handler also needs to be run in a separate asynchronous runtime on startup. The main challenge when implementing this deployment mode was that it had to work in a distributed environment.

The handler must figure what node(s) the service should run on (depending on the resource). No matter what the resource being targeted is, it will always result in the querying of Pods. These Pods will be assigned to a specific Node.

Pandit queries these nodes, and constructs a set of IP addresses of the Nodes the Pods are running on.

The solution to getting the service running on each daemon, is to simply interface with each Node's Pandit daemon as a client. The Node IP addresses will be sent a copy of the *StartServiceRequest* to their own administration APIs. Where they will add the Service using the same method.

The one change to the request is that the "delegated" field is set to true. This tells the other nodes not to forward on the request further.

4.8 Service Server

The Service Server is responsible for handling all requests sent to Pandit services on that daemon. The first challenge associated with implementing the server was that it had to parse any gRPC request.

Usually, a gRPC server would be implemented by code generation; due to the aforementioned requirement, the gRPC server had to be implemented from scratch. Since gRPC is based on HTTP/2 [32], the server is implemented as a HTTP/2 server. After researching the abstractions gRPC introduces on top of HTTP/2, a gRPC parser was implemented.

gRPC uses HTTP headers to define the metadata required to parse a request [18]. The gRPC method and service name exist in the "path" of the "URI" [38] HTTP header in the form of *<package name>.<service name>/<method name>*. These are parsed like so:

```
let path = request.uri().to_string();
let mut path = path.rsplit("/");
let method = path.next().unwrap();
let service = {
    let fqdn = path.next().unwrap();
    fqdn.rsplit(".").next().unwrap()
};
```

```
let service_name = service.to_string();
let method_name = method.to_string();
```

Another problem was how to deal with cache hits, as well as sending the query to the proxied application when needed. The solution was to introduce a trait called a "Sender". The *Sender* trait provides a way to both send a request to a method and get a response, as well as probe cache for said method.

Here is the trait:

```
#[async_trait]
pub trait Sender: Send + Sync {
    async fn send(
        &mut self,
        service_name: &String,
        method: &String,
        data: &[u8],
    ) -> ServiceResult<bytes::Bytes>;

    async fn probe_cache(
        &self,
        service_name: &String,
        method: &String,
        data: &[u8],
    ) -> ServiceResult<Option<bytes::Bytes>>;
}
```

The underlying object is provided by the Broker. This allows the Server to exclusively interface with these methods. If a cache probe is successful, the data will be returned to the client, or else the "Send" method will be called, and the response will be returned to the client.

4.9 Services

The Service was implemented to be a *Sender*.

4.9.1 Constructing a Service

The service is constructed by parsing a Proto file using the Rust Protobuf Parse package [37]. This is a package generally used by the Proto to Rust compiler to generate a detailed "description" of the Proto file before compilation.

The Service was engineered to handle these ”descriptors” by converting them into Pandit-specific structures. Such as creating a series of method structs that pointed to an input message and output message.

The service constructs a Handler for each method based on the options set by the user. This has to be reflected on using a method that dynamically returns a Handler, due to the Method struct being Handler-agnostic.

This method demonstrates how the handlers options can be parsed:

```rust
fn handler(
    handler: proto::gen::handler::Handler,
    path: String,
) -> Option<Arc<dyn Handler + Sync + Send + 'static>> {
    use proto::gen::handler::Handler::*;
    match handler {
        JSON => Some(Arc::new(JsonHandler::new(path))),
        _ => None,
    }
}
```

The issue was how to provide format-specific context to the handlers when implementing the ”Send” method. The solution was to write a method that reflects on information available in the Service. Such as options set for a specific format, and construct a ”Writer Context” based on this information.

For example, here is the implementation for the HTTP format:

```rust
fn context_from_api(api: &Option<format::HTTP>) ->
    ServiceResult<WriterContext> {
    let context = WriterContext::new();
    use crate::proto::gen::format::http::http::Pattern;
    match api {
        Some(http) => match http
            .pattern
            .as_ref()
            .ok_or(ServiceError::new("no pattern in api"))?
        {
            Pattern::get(s) => {
                context.insert("method".to_string(),
                    "GET".to_string());
                context.insert("uri".to_string(), s.clone());
            }
            Pattern::put(s) => {
                context.insert("method".to_string(),
                    "PUT".to_string());
                context.insert("uri".to_string(), s.clone());
            }
            Pattern::post(s) => {
                context.insert("method".to_string(),
                    "POST".to_string());
                context.insert("uri".to_string(), s.clone());
            }
            Pattern::delete(s) => {
                context.insert("method".to_string(),
                    "DELETE".to_string());
                context.insert("uri".to_string(), s.clone());
            }
            Pattern::patch(s) => {
                context.insert("method".to_string(),
                    "PATCH".to_string());
                context.insert("uri".to_string(), s.clone());
            }
        },
        None => {}
    }

    Ok(context)
}
```

4.9.2 Handling Services on Startup

Services must be run on startup to maintain the posterity of a service through interruptions such as, restarts or redeployments.

This is achieved firstly be the Administration API serialising the request used to create the service to a file.

When the program starts, it then checks reads all the service files and does the following:

1. Deserialise the contents of the file to a ”StartServiceRequest”.
2. Ensure that the ”delegated” field is set to true to ensure the request does not trigger a delegation.
3. Send the request to the local Administration API endpoint.

4.10 Broker

4.10.1 Announcing a Service

While announcing a Service does mainly entail pointing a key to the host of the daemon it's running on. It also requires the publishing of other information, such as:

- The default cache config for the service. This is used as a fallback cache configuration during a cache probe.
- A serialised map of all methods. This is used to ensue each daemon is aware of all the methods available to it.
- A serialised map of all messages. This was added to publish the caching strategies, as they are stored for the message in these objects.

4.10.2 Publishing Cache

When a new cache is published, the fields are serialised and sent to Redis. The problem is that other metadata is required to enable features such as cache expiry. The solution to this was to add a timestamp to the published data; allowing the daemons to know when a cache should expire.

There is also an edge case where some **fields** may have been disabled from being cached. The solution to this problem was to filter these fields out using a recursive cloning approach:

The fields would be iterated through and cloned, unless the "disabled" option was set to true. These fields would be ignored. The challenge with this approach however was that it had to handle embedded messages recursively. This was achieved by finding the message descriptor for this embedded message, (the descriptor is what stores the user-defined options) and recursively calling the "filter fields" function from there.

Here is a code snippet of the values either being cloned or recursively cloned, depending on the type:

```rust
match value {
    Value::Message(fields) => {
        let message_type =
            field_proto.descriptor.get_type_name().to_string();
        let cached =
            cached.parent.get(&message_type).ok_or(ServiceError::new(
            format!("no message found for type: {}",
                entry.key()).as_str(),
        ))?;
        map.insert(
            entry.key().clone(),
            Some(Value::Message(self.filter_fields(fields,
                cached.value())?)),
        );
    }
    v => {
        map.insert(entry.key().clone(), Some(v.clone()));
    }
};
```

4.10.3 Receiving Cache

The Broker is implemented to run from startup. It will connect to a Redis instance, and start listening for updates to the PUB/SUB database using an event loop.

When a value is received, it will decide what type of update it is based on the name. For example, when the cache is published and the daemon is subscribed; the solution to parsing this cache is to deserialize the payload. The deserialized values include the timestamp that the cache was created at.

Along with the cache expiry time, this can be used to check if a cache is expired. Here is a sample of the code that handles this:

```
let (primary_key, payload): (Value, Vec<u8>) =
    serde_json::from_slice(msg.get_payload_bytes())?;
let fields_map = cached.fields_for_key.clone();
let cached_fields = CachedFields {
    data: bytes::Bytes::copy_from_slice(&payload[..]),
    fields: cached.message.fields_from_bytes(&payload[..])?,
    timestamp: SystemTime::now(),
};
fields_map.insert(primary_key.clone(), cached_fields);
cached.fields_for_key = fields_map;
self.method_fields_map.insert(name.clone(), cached);
log::info!(
    "cache updated for {}: primary key value = {:?}",
    &name,
    &primary_key
);
```

4.10.4 Probing Cache

Cache probing is triggered by the Service Server by calling the *Probe Cache* method on a *Sender*. A *Service* implements this method by parsing the method and service names, then finding the primary key and delegating the request to the Broker.

The Broker does a simple cache lookup, ensuring the cache exists and is in date, or else returns ”None”.

However, when a Service is not on the local daemon, there is no service object to use. In this case, the Service Server queries the broker for cache for a specific Service. The broker will construct return a ”Remote Sender”. An implementation of the Sender that fulfils the following purposes:

- When queried for cache, it will look interface with the broker’s cache table to find the latest cache, ensuring it is in date.
- If the server doesn’t find cache and wants to send the query to the proxied application via the ”send” method. The remote sender will instead send the raw query to the daemon that hosts the Service, delegating the query.

4.11 CLI

The Command Line Interface is implemented using a CLI library called *clap [2]*.

The main challenge experienced when implementing the CLI involved the package installation process. Since the packages were defined in a remote JSON file, a series of structs were made to represent them. These were used to deserialise the JSON stream.

```
#[derive(Deserialize, Serialize)]
pub struct Index {
    pub packages: HashMap<String, Package>,
}

#[derive(Deserialize, Serialize)]
pub struct Package {
    pub version: String,
    pub proto_url: String,
    pub image: Option<Image>,
    pub readme_url: Option<String>,
    pub logo_url: Option<String>,
}

#[derive(Deserialize, Serialize)]
#[serde(tag = "type")]
pub enum Image {
    Helm {
        repo_url: String,
        repo_name: String,
        chart: String,
        container_name: Option<String>,
    },
    Docker {
        compose_file: String,
    },
}
```

When installing a package, the optional ”image” provided a way to install a resource in either Docker or Kubernetes.

If the Docker config was present, it would link to a Docker Compose [35] file. This is a YAML file that provides a method of deployment for Docker containers. The solution here was to create a docker compose file in the local filesystem alongside the Proto file. Docker Compose would then be called to deploy the Docker containers.

Helm [16] is a package manager for Kubernetes resources. It provides deployment files from a remote location. When this option is chosen in the package, the CLI will interface with the local helm instance to both download and install the relevant package. Integrating this into the CLI was challenging, since a progress bar was implemented to show the user the progress of the installation. This required the integration of a package called *Indicatif [48]* to the CLI.

Chapter 5

Evaluation

5.1 Unit Testing

Many unit tests were implemented to test the functionality of the components of the daemon.

5.1.1 Messages

The *Message* component's methods used to parse Protobuf payloads were tested to ensure that they provided the correct data to the rest of the program. This was done because there was no feasible way of reliably testing this functionality manually.

The Protobuf writer test was implemented as a ”table” of tests. This is a design pattern that involves creating a structure that represents a set of test parameters. The structure is then filled with different values that increase coverage of the tests. The structure used was as follows:

```
struct Table {
    name: String,
    field_type: Type,
    number: i32,
    type_name: String,
    label: Label,
    value: Value,
}
```

Since the action of writing a field to the bytes buffer is atomic. The field values in each table are put into a single ”Fields” object, and sent to the method. The buffer is then checked against the desired value after the test is executed to ensure it behaved properly

The reading of a Protobuf to the ”Fields” object was tested by constructing a desired ”Fields” object alongside a hardcoded Protobuf payload:

```
let buf: &[u8] = &[
    0, 0, 0, 0, 0, // gRPC header.
    0x08, 0x96, 0x01, // Field varint
    0x12, 0x07, 0x74, 0x65, 0x73, 0x74, 0x69, 0x6e, 0x67, // Field
        string
    0x1a, 0x03, 0x08, 0x96, 0x01, // Embedded message
    0x22, 0x08, 0x03, 0x08, 0x96, 0x01, 0x03, 0x08, 0x96,
    0x01, // Embedded message repeated x2
];
```

The ”Field” object is provided to the method and the output is checked against the hardcoded payload.

5.1.2 Service

The *Service* was also tested to ensure that it called a *Writer* when a query was made. This was done by creating a fake Writer in order to ensure that a payload was sent to the desired application.

A Service is created with all the necessary values passed to it, including a sample Proto file that was made for testing purposes. The ”Send” method is then called with the necessary parameters.

Finally, the values sent to the fake Writer are checked to ensure the method works correctly.

5.2 System Testing

5.2.1 Basic Deployment

In order to ensure the daemon was working correctly. A fake REST API was implemented in python to serve as an authoritative container. A Protobuf specification was created for the API, and it was passed to the daemon via *The CLI*.

The Protobuf specification was used to generate a Rust client library [28] that was used to interact with the daemon. A standalone testing binary was implemented by importing this library, and calling the service in question. It sent the same query twice. The first time, it was ensured that the data came directly from the authoritative container. The second time, it was ensured that the data came from the cache.

5.2.2 Docker Deployment

The sample application as well as the Pandit daemon were put inside Docker images and deployed to docker containers. They were deployed using a Docker Compose [35] file:

```
version: '3.9'
services:
  redis:
    image: redis
    networks:
      - redis

  example:
    build: src/proto/examples
    entrypoint: "python /example_rest.py"

  pandit:
    build: .
    depends_on:
      - redis
    ports:
      - "50122:50122"
      - "50121:50121"
    entrypoint: "/panditd --docker"
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
    networks:
      - redis

networks:
  redis:
```

The docker socket was passed to the Pandit container to allow it to query and modify the Docker state. Then, the example service used in the basic deployment was deployed and tested.

5.2.3 Kubernetes

A Kubernetes cluster was deployed locally using a tool called *Minikube [23]*. The resources were deployed using a series of deployment scripts, which take the form of YAML files. The Minikube cluster deployed 3 virtual machines as 3 nodes in the cluster.

These files deployed a Pandit daemon on each host in the form of a "DaemonSet". They also deployed a Redis Pod and Service, as well as an example application to be proxied in the form of the aforementioned REST API. A Pod to use as an example client was also deployed.

The Pandit deployment also got access to specific API features by allowing the default user to access said endpoints:

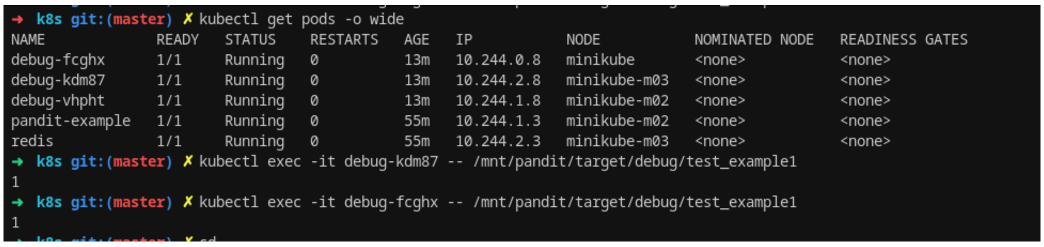
```
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  namespace: default
  name: pod-reader
rules:
  - apiGroups: [""]
    resources: ["pods", "nodes"]
    verbs: ["get", "watch", "list"]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: read-pods
  namespace: default
subjects:
  - kind: "ServiceAccount"
    name: "default"
    namespace: "default"
roleRef:
  kind: ClusterRole
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
```

The CLI was then used to interface with a Pandit daemon on a single host whilst the Pod was on another host. This to validate that a Service could be added to a specific node from another node.

An example client was used from a Pod on another node to test Pandit's query delegation.

Another feature that was tested was Pod eviction handling. Upon first try, the feature was very buggy and difficult to debug. The daemons running on each node required a lot of debug information to be logged in order to understand how they listen for state changes in Pods.

After days of testing, bug fixing and recompiling, the feature began to work as expected, allowing services to move between Nodes as the Pods's states change.



```
➜ k8s git:(master) ✗ kubectl get pods -o wide
NAME            READY   STATUS    RESTARTS   AGE   IP           NODE           NOMINATED NODE   READINESS GATES
debug-fcghx     1/1     Running   0          13m   10.244.0.8   minikube       <none>           <none>
debug-kdm87     1/1     Running   0          13m   10.244.2.8   minikube-m03   <none>           <none>
debug-vhpht     1/1     Running   0          13m   10.244.1.8   minikube-m02   <none>           <none>
pandit-example  1/1     Running   0          55m   10.244.1.3   minikube-m02   <none>           <none>
redis           1/1     Running   0          55m   10.244.2.3   minikube-m03   <none>           <none>
➜ k8s git:(master) ✗ kubectl exec -it debug-kdm87 -- /mnt/pandit/target/debug/test_example1
1
➜ k8s git:(master) ✗ kubectl exec -it debug-fcghx -- /mnt/pandit/target/debug/test_example1
1
```

Figure 5.1: A screenshot of a Pandit service being queried from 2 different nodes.

5.2.4 Handlers and Writers

Due to time constraints, most of the testing was focused on the deployment modes using a HTTP/JSON application. Testing also takes a long time due to the requirement of building a sample application and client. For this reason, the SQL writer and handler are not confirmed to be fully functional. Some functionality was tested, such as SELECT statements on a Postgres database. A sample Proto file was also made to ensure the Service can successfully construct an SQL Handler:

```
message ExampleTable {
  int32 id = 1 [ (pandit.key) = true,
      (pandit.format.postgres_field) = {key : true} ];
  string name = 2;
}

message Empty {
}

service PostgreSQL {
  option (pandit.name) = "postgres";
  option (pandit.format.postgres_service) = {
  };

  rpc SetExample(ExampleTable) returns (Empty) {
    option (pandit.format.postgres) = {
      command : INSERT
    };
    option (pandit.cache) = {
      cache_time : 3000
    };
  }

  rpc GetExample(ExampleTable) returns (ExampleTable) {
    option (pandit.format.postgres) = {
      command : SELECT
    };
    option (pandit.cache) = {
      cache_time : 4000
    };
  }
}
```

5.3 Appraisal

This project went through many iterations, both in concept and implementation.

It was originally conceptualised as a proxy that performed kernel-level packet parsing (using eBPF [15]), ultimately providing a distributed gRPC service. After working on a sample implementation of the eBPF code, the problems associated with eBPF (see *eBPF Approach*) became apparent.

After deciding on doing all packet parsing in user space [11], the project began to take form. C++ was originally the language being used to implement the user space code. However, the implementation was switched to Rust due to it having a more suitable set of libraries available to it when it came to implementing the gRPC server. Some of these libraries were: Protobuf Parse [37], Hyper [31] and Serde [6].

5.3.1 Does Pandit meet the Objectives?

Despite deviations in approaches, the Objectives always stayed the same. This is where the final implementation stands in relation to these objectives:

- The final product's architecture enables developers to easily integrate new data formats into Pandit. All a developer needs to do is provide implementations for the traits Writer and Handler.

  These traits should allow developers to implement support for any data format they require; as demonstrated by the existing implementations of the HTTP/JSON Writer and Postgres Writer. JSON being a text-based object notation and Postgres being a Structured Query Language (henceforth SQL); means that these two implementations differ wildly in how they interface with applications. But since the fact that they are both interfaced with by the rest of the Pandit components behind the same trait; the architecture meets the objective: being able to proxy any data source.

- Another objective was the ability to provide a Pandit Service. This objective was met when the Service and Service Server components were implemented. However, outside unit tests, the product was only able to be verified to work after the Administration API was implemented as this is what allows users to add services.

- Pandit achieves the objective ”Using Protobuf Options, Pandit should provide a way to define the conversion between the application and the Pandit Service” for the most part.

  Users can define how conversions are done by importing the provided Proto files. They are able to get around edge cases such as the required object being a child of the object returned by the application via a ”path” option available on each message.

  Overall, this provides a decent user experience; however, the limitations of Protobuf Options are apparent in some ways when writing a service. For example, the amount of copied and pasted syntax and repetition required to implement a message is more than desirable. This is particularly evident with the primary key, which requires the following option to be attached to a field in every request message:

  ```
  message ExampleRequest {
    // Primary key:
    int32 id = 1 [ (pandit.key) = true ];
    string user = 2;
  }
  ```

- The objective to allow for user-defined caching strategies was met by providing options for users to configure how entire messages as well as individual fields can be cached. However, in terms of feature-set, it is quite limited due to the fact that the caching features are primarily a proof of concept.

  Currently, the only available caching options are the following:

  ```
  message CacheOptions {
  ```

```
    bool disable = 60031; // If set, disables the target
        from being cached.
    uint64 cache_time = 60032; // If set, the cache will
        expire after this amount of seconds.
  }
```

This was deemed sufficient due to the scope of this project; however, given more time, many more options would be added. These could include a way to define more complex associations between fields. Right not the only way to get a cache hit is with a matching primary key. A more flexible (and technically challenging) approach would be to allow users to define cache hits via conditions such as:

```
  if field_a > field_b:
      cache hit
  else if field_c exists:
      cache hit
```

- Pandit is able to run in a cluster of servers whilst coordinating requests between servers due to every server being connected to a message broker. It is confirmed to be working sufficiently during the aforementioned system System Testing.

  The deployment of Pandit in a distributed environment requires prerequisite knowledge of the cluster's network architecture. So isn't designed to be easy to installed out of the box. It is instead designed to be deployed by an engineer familiar with the cluster, and to that end it has met this objective.

5.3.2 Qualitative Evaluation

Caching

Compared to other industry solutions such as those mentioned in Existing Solutions. Pandit in its current form provides a set of features that stand out from the competition.

The caching features have limitations, however. Pandit assumes that when an object is cached, the underlying data will not change, i.e. the underlying methods are deterministic. This is because maintaining coherency between the cache, and the underlying data is a problem outside the scope of this project.

Since the user is responsible for defining how and what should be proxied. Pandit has no way of knowing what is happening on the application outside the snippets of information it gets from the query responses. This means, if information changes due to an action handled outside of Pandit, there is no way to maintain consistency between the application and daemon.

See Future Work for possible solutions to this problem.

Feature Set

The feature set of Pandit in its current form can best be described as a proof of concept. However, its architecture allows it to be both easily extended and deployed in many environments. Therefore, it could be integrated into a production environment if a company wanted to benefit from its current feature set.

The ability to turn any application into a gRPC service without any modification of the application's source code is something many companies would see value in. This is due to the innate benefits of a gRPC service over other data formats, such as: being able to generate a client library in any language [28].

Pandit integrates basic package management functionality into its CLI. This feature could be used to interface with an internal repository of Proto files inside a company. When a developer needs to implement a client for a particular service, they could simply use Pandit to both download the relevant Proto file and deploy a local version of the application for testing. A system administrator can use the same feature to install the relevant Proto files in a production environment. They can subsequently use said Proto files to deploy services to Pandit. Therefore, this feature allows there to be an ecosystem of Pandit packages that simplify both development time and deployment.

As outlined in Deployment Modes, Pandit integrates into industry standard deployment models [14] such as Docker and Kubernetes. Since these technologies are widely adopted within the industry, most companies could integrate Pandit into their existing software stack.

Comparison to Other Solutions

As mentioned in the Existing Solutions section, there are alternative solutions to Pandit. Below is a comparison between said solutions and Pandit, in its current form:

- Pandit provides similar functionality to *Google Cloud Endpoints with HTTP/JSON to gRPC encoding [25]*; but Pandit provides far more flexibility due to its ability to translate any data format. Both Pandit and Google Cloud's solution translate HTTP/JSON to gRPC. Pandit's support for HTTP/JSON is fully tested with an example application. This makes Pandit's current feature set subsume the features provided by Google Cloud's solution. This is because Pandit also includes the ability to define custom caching strategies. Pandit can also be deployed inside any distributed system, while Google Cloud's solution is exclusively available in Google Cloud.

- *Envoy Proxy [8]* has a different approach to proxying than Pandit. It runs an instance of itself alongside each application, inside the same container. It also only performs translation between application layers, such as "HTTP/1.1 to HTTP/2".

  Pandit is implemented to run on each node, reducing the amount of instances that must be deployed compared to Envoy.

  Pandit also translates the application into a gRPC service. This allows for a different approach to caching compared to Envoy. Envoy caches HTTP queries by exclusively parsing the HTTP headers. This means it indexes its in-memory cache exclusively by URI. Pandit on the other hand maintains a cache based on a user-defined field sent in via gRPC. This provides more flexibility in defining caching strategies as it is more configurable.

5.4 Summary

Pandit is a great tool for caching and querying data in a distributed system. It also provides a great deal of flexibility for users to define their own caching and querying strategies.

Whether it be a REST API or an SQL Database, all it takes is a single Protobuf specification [30] to turn it into a **widely available gRPC [19] Service**. Services become widely available due to the caching mechanism in the daemon being on every host. As well as the publish/subscribe mechanism used to distribute the cache to all interested daemons.

The fact that anything can be turned into a gRPC service provides a lot more flexibility to the users of the product compared to any existing solution. This is because you can build a client library for any language or framework using a Protobuf [28].

This also means that Pandit has innate benefits for any application that requires data from other services. For example, instead of using an ORM [1] and manually encoding the data to be read in the application. You can use

Pandit to to turn the database into a gRPC service and then use the gRPC client library to read the data. This will give you the added benefit of being able to cache the data, and query it in a distributed system if it needs to be deployed to multiple hosts later on. You could also implement another client in a different language in one step by generating code from the Protobuf [28].

Pandit also requires no configuration of the application being proxied. As long as pandit has the service added, it will be ready to accept requests. This is because the daemon interfaces with the application as a client.

Overall, Pandit's quality is fitting of a proof of concept. It demonstrates the possibilities of this approach to distributing services, meeting the objectives set for it. It is not ready to be deployed in a production environment. However, as far more testing would be required; especially in the areas of stability and performance.

Pandit in its current form could find success as an open-source project, where it would benefit from having many eyes on it, reducing the likelihood of it experiencing bugs and instability.

All in all, Pandit has the potential to be a very useful product, providing practical solutions to problems companies are currently experiencing when deploying their applications to distributed systems.

Chapter 6

Conclusions

6.1 Reflection

I have learned many things throughout the research, design and implementation of this project. I have learned the Rust Programming Language [46]: an up-and-coming programming language that has many innovative and exciting features.

I also learned the core concepts behind Kubernetes [26] such as Pods, Deployments, Resources, etc. I also learned how to write an application that works inside Kubernetes.

During the early days of my research, I focused primarily on how packets can be parsed in eBPF [15] (see eBPF Approach for more details). I spent a long time trying to get a solution in code. This proved to be a dead end due to eBPF's limitations, and the approach was scrapped. If I could start again, I would weigh the ups and downs of multiple approaches from the start and make a decision based on the objectives of the project.

After modifying my approach to interface with the application in user space (see *Userspace gRPC Server approach* for more details), development went a lot smoother.

Overall, I learned a lot of skills both technical and non-technical that will really help me in the future. I have learned from my mistakes in the planning phase, which has taught me how to approach project planning in the future.

6.2 How the project was conducted

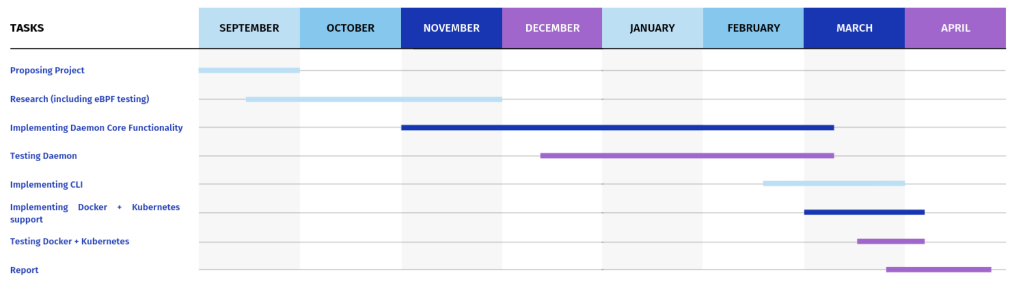


Figure 6.1: A Gantt chart showing the time taken to do each general task.

A lot of thought was put into the initial project proposal. After the project was accepted by Professor Dirk Pesch, research began.

As stated above, a lot of research went into the area of eBPF. After this approach was scrapped, research began in areas such as gRPC and The Rust Programming Language.

During the research phase, implementation of the daemon's core features began. This included implementing the Service, Message, Handler, Broker and Writer components.

After a basic version of a component, such Service was completed, testing began alongside further development. This allowed bugs to be found whilst development continued. This section of development was the largest bottleneck in the development process due to the complexity and scope of the task. If the project was started again, this stage would have been started much sooner to allow for ample time to complete the rest of the tasks.

The CLI began development in mid-February, alongside the Administration API in the daemon. After the "Install" method was completed, the Kubernetes/Docker Deployment modes began development. Development after this switched between Kubernetes and the CLI, to ensure the product could be deployed.

During the end of development, work on the report started. After the open day, all work was focused on the report. Due to the complexities involved in implementing this project, the report necessitated lots of length explanations, increasing the workload in April. If this project was started again, the report would be done alongside the implementation stage, reducing the stress involved in writing the report in April.

6.3 Future Work

6.3.1 Solutions to Caching Problems

In the Qualitative Evaluation, the problems associated with caching were introduced. A possible solution to some of these issues could involve a user-defined refresh time. For example, if it is known that a particular value is mutated on a recurring daily interval, such as every day at 1am. The user could configure Pandit to either invalidate the cache at 1am or query for an updated version of the cache. This solution would likely be implemented given more time.

There are still coherency issues with this solution if there is no set interval for mutations of the data. In its current form, the user would be advised to disable caching on these methods; they would still benefit from the gRPC translation aspects of the project.

A solution that could solve this may involve a callback method the application can avail of that will notify Pandit of changes to the data. This would have many issues and is in conflict with one of the core selling points of the project: "the application should not need to be modified in order to work with Pandit". Since there was no desired solution to this problem, it was deemed outside the scope of this project.

6.3.2 CLI/Ecosystem improvements

The CLI currently supports a very basic install operation. It simply reads from a remote JSON file, looks for the package, and downloads the necessary files.

If more time was granted, the CLI would be designed to pull from a fully implemented API. The API would provide server-side authentication, a way to upload/update packages, as well as many other features.

A Web UI was set as a stretch goal; it would demonstrate how a user could download packages by copying a command from a website. An implementation was started but was set aside and later removed due to time constraints. If more time was granted, a full website would be made. It would allow users to manage and upload packages, as well as discover new packages.

Terminology

- **JSON** - JavaScript Object Notation, an open standard text-based data format.
- **ORM** - Object-Relational Mapping, a design pattern allowing for the manipulation of a database using the Object-Oriented paradigm.
- **gRPC** - gRPC Remote Procedure Calls; a recursive acronym.
- **Protocol Buffer** - A binary format for serialising structured data.
- **eBPF** - Extended Berkley Packet Filter, a kernel feature that runs sandboxed bytecode in kernel mode.
- **Message Broker** - A program responsible for sending/broadcasting a message from one application to other applications.
- **SQL** - Structured Query Language, used to query data in a relational database.
- **CLI** - Command Line Application.

Bibliography

[1] Ehab Ababneh. *Object-Relational Mapping (ORM)*. 2011. URL: https://home.cs.colorado.edu/~kena/classes/5448/f11/presentation-materials/orm_ababneh.pdf.

[2] Clap Authors. *Clap - GitHub*. 2021. URL: https://github.com/clap-rs/clap.

[3] Docker Authors. *Docker Homepage*. 2022. URL: https://www.docker.com/.

[4] Redis Authors. *Redis*. 2022. URL: https://redis.io/.

[5] Redis Authors. *Redis Pub/Sub*. 2022. URL: https://redis.io/docs/manual/pubsub/.

[6] Serde Authors. *Serde Homepage*. 2022. URL: https://serde.rs.

[7] Serde Authors. *Serde JSON*. 2022. URL: https://docs.serde.rs/serde_json/.

[8] Envoy Project Authors. *Envoy Proxy - Architecture Overview*. 2022. URL: https://www.envoyproxy.io/docs/envoy/latest/intro/arch_overview/arch_overview.

[9] Commbox. *Synchronous and Asynchronous Messaging Channels - Essential Guide*. 2021. URL: https://www.commbox.io/synchronous-and-asynchronous-messaging-channels-essential-guide/.

[10] Stephen Dolan. *JQ Introduction*. 2022. URL: https://stedolan.github.io/jq/manual/#Basicfilters.

[11] Rick Donato. *A Brief Explanation of Kernel Space and User Space*. 2017. URL: https://www.fir3net.com/UNIX/Linux/a-brief-explanation-of-kernel-space-and-user-space.html.

[12] Niel Drummond. *Bollard - GitHub*. 2022. URL: https://github.com/fussybeaver/bollard.

[13] IBM Cloud Education. *Message Brokers*. 2020. URL: https://www.ibm.com/cloud/learn/message-brokers.

[14] Enlyft. *Companies using Kubernetes and its marketshare*. 2021. URL: https://enlyft.com/tech/products/kubernetes.

[15] The Linux Foundation. *eBPF Introduction*. 2021. URL: https://ebpf.io/.

[16] The Linux Foundation. *Helm Charts*. 2022. URL: https://helm.sh/docs/topics/charts/.

[17] Google. *Book Store Example*. 2016. URL: https://github.com/GoogleCloudPlatform/python-docs-samples/blob/main/endpoints/bookstore-grpc/http_bookstore.proto.

[18] Google. *gRPC over HTTP2 - GitHub*. 2020. URL: https://github.com/grpc/grpc/blob/master/doc/PROTOCOL-HTTP2.md.

[19] Google. *Introduction to gRPC*. 2021. URL: https://grpc.io/docs/what-is-grpc/introduction/.

[20] Google. *Labels and Selectors in Kubernetes*. 2021. URL: https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/.

[21] Google. *Topology-aware traffic routing in Kubernetes*. 2021. URL: https://kubernetes.io/docs/concepts/services-networking/service-topology/#topology-aware-traffic-routing.

[22] Google. *Virtual IP addresses in Kubernetes*. 2021. URL: https://kubernetes.io/docs/concepts/services-networking/service/#virtual-ips-and-service-proxies.

[23] Google. *Welcome! - Minikube*. 2021. URL: https://minikube.sigs.k8s.io/docs/.

[24] Google. *Google Cloud*. 2022. URL: https://cloud.google.com/.

[25] Google. *Google Cloud Endpoints Transcoding HTTP JSON to gRPC*. 2022. URL: https://cloud.google.com/endpoints/docs/grpc/transcoding.

[26] Google. *Kubernetes Homepage*. 2022. URL: https://kubernetes.io/.

[27] Google. *Protobuf Compiler*. 2022. URL: https://developers.google.com/protocol-buffers/docs/cpptutorial.

[28] Google. *Protocol Buffer Code Generation*. 2022. URL: https://developers.google.com/protocol-buffers/docs/reference/go-generated.

[29] Google. *Protocol Buffer Encoding*. 2022. URL: https://developers.google.com/protocol-buffers/docs/encoding.

[30] Google. *Protocol Buffers - Google Developers*. 2022. URL: https://developers.google.com/protocol-buffers/.

[31] Hyperium. *Hyper - GitHub*. 2022. URL: https://www.fir3net.com/UNIX/Linux/a-brief-explanation-of-kernel-space-and-user-space.html.

[32] IETF. *HTTP/2 RFC 7540*. 2015. URL: https://www.rfc-editor.org/rfc/rfc7540.html.

[33] IETF. *Transmission Control Protocol (TCP) Specification*. 2021. URL: https://www.ietf.org/archive/id/draft-ietf-tcpm-rfc793bis-25.html.

[34] Docker Inc. *Docker Images*. 2021. URL: https://docs.docker.com/engine/reference/commandline/images/.

[35] Docker Inc. *Overview of Docker Compose*. 2021. URL: https://docs.docker.com/compose/.

[36] ECMA International. *The JSON Data Interchange Syntax*. 2017. URL: https://www.ecma-international.org/wp-content/uploads/ECMA-404_2nd_edition_december_2017.pdf.

[37] Stepan Koltsov. *Rust Protobuf Parse - GitHub*. 2022. URL: https://github.com/stepancheg/rust-protobuf/tree/master/protobuf-parse.

[38] Mozilla. *Identifying resources on the Web*. 2021. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/Identifying_resources_on_the_Web.

[39] Mozilla. *Rust Build Scripts*. 2021. URL: https://doc.rust-lang.org/cargo/reference/build-scripts.html.

[40] Mozilla. *Rust By Example - Traits*. 2021. URL: https://doc.rust-lang.org/rust-by-example/trait.html.

[41] Mozilla. *Derive - The Rust Reference*. 2022. URL: https://doc.rust-lang.org/reference/attributes/derive.html.

[42] Mozilla. *Rust Lifetimes - The Rustonomicon*. 2022. URL: https://doc.rust-lang.org/nomicon/lifetimes.html.

[43] Mozilla. *std::result::Options - Rust Documentation*. 2022. URL: https://doc.rust-lang.org/std/option/.

[44] Mozilla. *std::result::Result - Rust Documentation*. 2022. URL: https://doc.rust-lang.org/std/result/enum.Result.html.

[45] Mozilla. *Struct std::sync::Arc - Rust Documentations*. 2022. URL: https://doc.rust-lang.org/std/sync/struct.Arc.html.

[46] Mozilla. *The Rust Programming Language*. 2022. URL: https://www.rust-lang.org/.

[47] Sea QL. *Sea Query*. 2022. URL: https://github.com/SeaQL/sea-query.

[48] Console RS. *Indicatif - GitHub*. 2022. URL: https://github.com/console-rs/indicatif.

[49] Amazon Web Services. *Caching Overview*. 2022. URL: https://aws.amazon.com/caching/.