jamk.fi

Centralized system for managing Netem configurations for VyOS/Vyatta routers

Mikko Neijonen

Bachelor's thesis

June 2015

Degree programme in Software Engineering

The School of Technology

JYVÄSKYLÄN AMMATTIKORKEAKOULU

JAMK UNIVERSITY OF APPLIED SCIENCES

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

Description

Author(s)	Type of publication	Date
Neijonen, Mikko	Bachelor's thesis	5.6.2015
		Language of publication: English
	Number of pages 39	Permission for web publication: yes

Title of publication

Centralized system for managing Netem configurations for VyOS/Vyatta routers

Degree programme

Software engineering

Tutor(s)

Ari Rantala

Assigned by

JAMK University of applied sciences/JYVSECTEC project, Marko Silokunnas

Abstract

This thesis describes a system developed for the RGCE environment for JYVESECTEC project at JAMK University of Applied Sciences. The system is used to introduce delay and other such characteristics of wide area networks into RGCE.

Currently RGCE lacks the ability to introduce wide area network characteristics, such as delay and packet loss, into the environment.

The thesis describes a centralized system for managing Netem configurations on VyOS/Vyatta routers in a virtual Internet. The implemented system includes a browser-based user interface, a central control server, and the VyOS/Vyatta configuration agents. The thesis describes the process and problems encountered during the design of the system. The technical architecture and a text-based application protocol developed for the system are also described. Finally the solution is evaluated and found suitable for the intended purpose. It is also concluded that further work is needed to improve the user interface.

Keywords/tags (subjects)

Netem, network, Linux, configuration management, centralized control, application protocol

Miscellaneous

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

Kuvailulehti

Tekijä(t)	Julkaisun laji	Päivämäärä
Neijonen, Mikko	Opinnäytetyö	5.6.2015
	Sivumäärä 39	Julkaisun kieli English
		Verkkojulkaisulupa myönnetty: kyllä

Työn nimi
Centralized system for managing Netem configurations for VyOS/Vyatta routers

Koulutusohjelma
Ohjelmistotekniikan koulutusohjelma

Työn ohjaaja(t)
Ari Rantala

Toimeksiantaja(t)
Jyväskylän ammattikorkeakoulu/JYVSECTEC projekti, Marko Silokunnas

Tiivistelmä

Opinnäytetyö esittelee Jyväskylän Ammattikorkeakoulun JYVSECTEC-projektin hallitsemaan RGCE-ympäristöön kehitettyä järjestelmää viiveiden ja muiden laajaverkon ominaisuuksien tuottamiseksi.

Tällä hetkellä RGCE-ympäristöstä puuttuu laajaverkolle ominaiset häiriötekijät, kuten viiveet ja pakettihäviö.

Opinnäytetyö kuvaa keskitetyn järjestelmän Netem-konfiguraatioiden hallintaan VyOS/Vyatta virtuaalireitittimillä. Työssä toteutettiin järjestelmä kokonaisuudessaan: selainpohjainen käyttöliittymä, hallintapalvelin ja konfiguraatio-agentit. Opinnäytetyössä kuvataan suunnittelun vaiheet, sekä järjestelmän tekninen arkkitehtuuri ja sitä varten kehitetty tekstipohjainen sovellusprotokolla. Lopuksi tuotos arvioidaan ja sen todetaan soveltuvan suunniteltuun käyttötarkoitukseen, mutta vaativan jakokehitystä käytettävyyden parantamiseksi.

Avainsanat (asiasanat)
Netem, verkko, Linux, konfiguraation hallinta, keskitetty hallinta, sovellusprotokolla

Muut tiedot

Contents

1 Introduction **2**
1.1 Background . . . 2
1.2 Problem . . . 2
1.2.1 Network emulation . . . 3
1.2.2 Centralized configuration . . . 4
1.2.3 Performance monitoring . . . 4
1.2.4 User interface . . . 5
1.2.5 Agent-server communication . . . 5

2 Related work **7**
2.1 Linux traffic control . . . 7
2.1.1 Netem . . . 7
2.2 The Go programming language . . . 8
2.2.1 Syntax . . . 8
2.2.2 Interfaces . . . 9
2.2.3 Concurrency . . . 9
2.2.4 Tooling . . . 12
2.3 Web frameworks . . . 13
2.3.1 AngularJS . . . 13
2.3.2 Bootstrap . . . 13

3 Concept and implementation **14**
3.1 Architecture . . . 14
3.2 Agent . . . 14
3.2.1 Drivers . . . 15
3.3 Server . . . 17
3.3.1 Data storage . . . 17
3.3.2 HTTP server . . . 18
3.4 User interface . . . 20
3.4.1 Requirements . . . 20
3.4.2 Implementation . . . 21
3.5 Agent-server communication . . . 22
3.5.1 Messaging patterns . . . 23
3.5.2 Synchronous and asynchronous messaging . . . 24
3.5.3 Protocol definition . . . 26
3.5.4 Connection . . . 29
3.5.5 Request multiplexing . . . 31

4 Evaluation **34**

5 Conclusions **36**

6 Future work **37**

References **38**

1 Introduction

This document outlines the background and the design behind a solution for managing network traffic control configurations on a set of VyOS/Vyatta routers in a virtual Internet.

The solution described in this thesis was developed to address the client's need to make Netem configuration across various virtual routers easier, as well as to test the viability of the solution.

The rest of this chapter explains the problem in more detail before proposing a set of requirements that a solution needs to address. The rest of the document strongly emphasizes the back-end design and specifically the agent- server communication over the user interface.

1.1 Background

The solution was created for JYVSECTEC project at JAMK University of Applied Sciences. JYVSECTEC has created and maintains RGCE, Realistic Global Cyber Environment. It is an isolated research, development and training environment that provides facilities and structures similar to the real Internet.

1.2 Problem

The target environment is a hybrid environment with physical and virtual appliances. The core routers are virtualized VyOS/Vyatta instances while the edges can be anything from virtual to physical appliances. Due to these limitations, and the fact that physical distances are relatively small, the latency and errors in the environment are minimal.

In fact there is a system in place to generate broken traffic into the network, however it is used mostly to generate background traffic; legitimate traffic is left untouched.

There can be many approaches to the problem, however this document will focus on a solution from software engineering standpoint: it needs to be a software solution.

The problem can be broken down into four individual categories, in order of precedence:

Network emulation. First priority is adding latency and errors into arbitrary links within the environment.

Centralized configuration. Manual configuration of each link is time-consuming and error-prone. A centralized configuration solution reduces setup times and makes it easier to manage different configurations.

User-interface. An approachable user-interface makes it easier to update configurations and observe performance of the system.

Performance monitoring. Any software-based emulation is limited by the available resources; it can be hard to predict how configuration changes affect the overall performance. Monitoring the system performance will help troubleshoot any issues that may rise; during the development and after.

Agent-server communication. Centralized configuration suggests that the system will comprise of multiple clients and a server. The individual parts need to have a way to communicate with each other.

1.2.1 Network emulation

The core infrastructure consists of 26 virtualized VyOS/Vyatta routers. Vyatta configuration interface allows the configuration of traffic shaping parameters, such as rate limiting, but it does not appear to allow the configuration of network emulation. This is strange, since they are managed by the same traffic control subsystem in the Linux kernel.

The current Linux kernel versions include the Netem kernel module which can be used to emulate the behavior of a wide area network, including *variable delay*, *packet loss*, *duplication* and *re-ordering*. Netem is applied to leaving traffic.

The network emulation can be applied at the VyOS/Vyatta routers or at the links between the routers.

If one wishes to configure, for example, 200 millisecond latency between two nodes A and B, there are two approaches on how to achieve the same result:

- Configure latency for the outbound queues on nodes A and B.
- Add a third node, C, with two bridged interfaces and add configurations for the latency to the outbound queues on the bridged interfaces.

It could prove challenging to maintain a consistent configuration if the emulation is applied to the routers. For example, setting latency to the route between `R1/eth1`

and `R7/eth4` requires creating matching outbound rules for both devices.

On the other hand, applying the configurations on the links between the routers allows isolating the configurations to a single node at the cost of introducing new instances into the environment.

1.2.2 Centralized configuration

The purpose of centralized traffic control configuration is to simplify the configuration and monitoring of a changing laboratory network. While it is possible to apply traffic control configuration manually, it will quickly become unwieldy as the number of links increases and the need for changes becomes more frequent.

With a centralized configuration solution the configurations only need to be created once and then applied to the target systems. This also makes it easier to verify that the systems are configured correctly.

Configuration management tools

There are various commercial and open source tools (such as Ansible, Chef, Puppet, etc.) for configuration management that could be used in the target environment. This would probably involve writing custom modules that can handle Netem configuration.

One could argue that using an existing configuration tool (that is already used in the environment) would allow integrating the network emulation configuration into the normal work flow. It would also reduce fragmentation in the environment.

This approach was briefly considered and then abandoned. While it is a natural choice in an environment with an existing configuration management software, it would be an uninteresting exercise in software design and implementation.

1.2.3 Performance monitoring

Software-based network emulation is limited by the resources available to the virtual instances; adding latency increases memory consumption since the packets need to be buffered. Some of the network characteristics that need to be emulated are also interlinked: re-ordering of packets implies that there is latency. Therefore, it is important to be able to monitor the performance of the system in order to verify that the system performs as expected, as well as to troubleshoot potential problems.

1.2.4 User interface

The user interface needs to provide an overview of the state of the system as well as easy access to traffic control configurations.

A straightforward approach is to directly expose the underlying network emulation configurations to the user interface. It would make the solution logic apparent and any new features or changes to the network emulation layer would be straightforward to implement in the user interface.

Many of the configuration options have dependencies that need to be addressed in the user interface to make it easier to use. Also the unknown resource requirements of some configurations can make it difficult to provide immediate feedback for configurations that are technically correct but unfeasible once put into practice.

In summary, the user interface has following broad requirements:

- Reflect the different aspects of the solution: configuration management and performance monitoring.
- Provide access to logs from the agent and from the server.
- Provide an overview to the system.

1.2.5 Agent-server communication

Various approaches were considered before settling on the RPC protocol. This section will describe the different choices.

SSH

A straightforward approach would be to execute commands remotely on the client machine through SSH (Ylonen and Lonvick 2006). This could be achieved by writing some scripts that read configuration descriptions and targets from a file, connect to the clients and applies the configurations. Collecting information from the clients would work similarly.

While the system could be relatively stable most of the time, it would likely break regularly. Error handling in remote commands via SSH can be complicated and error-prone. Also troubleshooting any issues could be problematic.

Message queue

Message queues are good for service integration in larger systems, which would allow building a solution that could be later expanded using the message queue as the common interface.

To get the full benefit from the message queue it would require a dedicated server. In a larger system this is not a problem, in fact separating the message passing from the components could be desirable.

There are Go libraries that can provide client-side APIs for a message queue. For example, bindings to ZeroMQ are available. (Kleiweg 2014)

Unfortunately this approach also incurs overhead, namely the server running the message queue. While it would be easy to overlook the added complexity in any larger system, for the purposes of this solution it looked like an overkill.

Initially, a message queue was considered as the communication channel for the system. While it is a very flexible solution that appears to scale well for larger systems, it also introduces unnecessary complexity that was hard to justify for the solution described in this thesis.

Remote procedure call

A remote procedure call (RPC) allows executing procedures on another computer over a network connection without leaking the details of the remote interaction. (Arpaci-Dusseau and Andrea 2014)

While RPC could be implemented over a message queue, that is not a necessity. In fact it is much easier to implement over HTTP, TCP or even UDP.

RPC over HTTP would be relatively simple to implement. It would require an HTTP server running on the client machines, with an API endpoints for traffic control configurations and logs. The system could not show live status, since the clients would have to be polled[1], but it would be reasonably simple and robust.

Go's `net/rpc/jsonprc` package provides a stateless RPC mechanism that is mapped to Go data types and method calls. The lack of state would make it necessary to include state information in the RPC messages.

1. There are ways around the polling requirement, such as websockets, but they would complicate the implementation. And once websockets are added it is just simpler to use TCP sockets and remove the HTTP server dependency altogether.

2 Related work

This chapter covers some of the technologies and theory relevant to the solution.

2.1 Linux traffic control

Linux traffic control consists of *shaping*, which controls the rate of transmission of packets. *Scheduling*, which controls the delay and re- ordering of packets. *Policing*, which applies to monitoring and classifying arriving packets, And *dropping*, which applies to dropping of packets.

The processing of the traffic is controlled by three kinds of objects:

Queuing discipline Queuing discipline, or *qdisc*, handles packets queued for an interface. When the kernel needs to send a packet to an interface, it pushes it into the corresponding *qdisc*. Afterward, it dequeues all available packets before giving them to the network adapter driver.

Classes A *qdisc* can contain a *class*, which can contain more *qdiscs*. When the kernel attempts to dequeue packets from such a *qdisc*, it can come from any of the internal *qdiscs*. This can be used to queue traffic based on class.

Filters are used by classfull *qdiscs* to determine in which class a packet will be enqueued.

Linux traffic control manual states that *shaping* and *scheduling* take place on egress[1].

The Linux kernel traffic control is configured with the `tc` command. It is used to control the abovementioned shaping, scheduling, policing and dropping parameters. (Linux User's Manual 2001)

2.1.1 Netem

As briefly described in 1.2.1, Netem is the kernel module that provides facilities to emulate the characteristics of a wide area network on the local machine.

Linux kernel 2.6.7 introduced `delay`, which supported constant delay. That code later

1. *Egress* pertains to outbound traffic and *ingress* to inbound traffic.

evolved into *Netem* that was first included in 2.6.8 kernel (Hemminger 2015). Current versions of Netem support emulation of *variable delay*, *packet loss*, *duplication* and *re-ordering* of packets. (Linux User's Manual 2001)

2.2 The Go programming language

Go is a general purpose programming language. It was conceived at Google in 2007 by Robert Griesemer, Rob Pike and Ken Thompson. It became a public open source project in 2009 and version 1.0 was released in 2012. (Frequently Asked Questions (FAQ) - The Go Programming Language 2015)

Go is a statically typed compiled language with a garbage collector and built- in concurrency features.

Due to the strong C background of Go's designers, some of the language features are modest improvements over C. One such small detail is the lack of pointer arithmetic in Go. While the language allows pointers, it does not provide mechanisms that would allow deriving invalid pointers[2].

This section covers some of the identifying features of the Go programming language, as well as the features that are important to the solution described later in section 3.

2.2.1 Syntax

Go borrows its syntax from the C and Pascal family of languages. It uses C-style braces for code blocks, and Pascal-style keywords and declaration order.

For example, a function in C that takes two arguments and returns an integer (Java uses mostly the same syntax, except for the pointer notation):

```
int magic(int a, int b, const char *c) {
        /* ... do something with a, b and c ... */
        return 0
}
```

The same function in Pascal:

2. Go's `unsafe` package allows this, but its use is strongly discouraged

```
Function Magic(A, B: Integer, C: String): Integer
Begin
        (* .. do something with A, B and C ... *)
        Magic := 0
End
```

Note how braces denote the function body and type precedes the identifier in C. Now compare this to Pascal which favors keywords to denote functions and code blocks and the how the type follows the identifier.

```
func magic(a, b int, c string) int {
        /* ... do something with a, b and c ... */
        return 0
}
```

A side effect of Go syntax is that it avoids a pitfall in the pointer notation. Namely a declaration such as `int* a, b` in C declares two variables of types `int` and `int*`. Whereas a similar declaration in Go, `var a, b *int` declares to variables of type `*int`.

2.2.2 Interfaces

Go does not support polymorphism through inheritance. Instead it has implicit interfaces. A type implements an interface if it has a method that matches the interface signature. (Effective Go 2015)

For example, the interface `Reader` that is used throughout the standard library:

```
type Reader interface {
        Read(b []byte) (n int, err error)
}
```

The interface is implicitly implemented by any types that expose a `Read` method that matches the interface signature. For example, `os.File` and `bytes.Buffer` both implement the `Reader` interface even though neither of the types even mention it.

2.2.3 Concurrency

In addition to more traditional methods of protecting shared memory, Go provides goroutines and channels as a way to structure concurrent software. (The Go Blog: Share Memory By Communicating 2015)

These primitives are based on *Communicating Sequential Processes* (Hoare 1985).

Goroutines

A goroutine is a kind of a thread that is scheduled by Go runtime instead of the operating system. They are very lightweight compared to operating system threads. Whereas the default stack size for a POSIX thread on 64-bit Linux is 8 MB[3], goroutine starts with a 2 KB stack in Go 1.4 (Go 1.4 Release Notes 2015).

A goroutine has a simple model when compared to an operating system thread. It is a function executing concurrently with other goroutines in the same address space. (Effective Go 2015)

A downside to goroutines is that a lot of the control associated with operating system threads is lost. The developer must make sure that a goroutine terminates gracefully.

For example, a simple program that prints `Hello, world.` from a goroutine:

```
func main() {
        go func() { fmt.Println("Hello, world.") }()
}
```

The above example shows how the closure is launched as a goroutine with the `go` statement. Note that the example has a flaw: the program may well exit before the goroutine is executed.

Channels

Channels provide a method of sharing data between goroutines. The method is somewhat different from locking. Whereas locking is used to synchronize access to a shared resource, channels can be used to pass a reference to a resource to another goroutine.

Channels have two operations: send and receive. Sending on a channels passes the value to another goroutine. In the case of pointers, it is the responsibility of the sender to ensure that a pointer is not used by the sending goroutine once it has been sent. Consequently, a receiving goroutine should assume that values received from the channel are safe to access.

For example, a small program that returns two values from goroutines and prints an output when they are both available:

3. `pthread_create()` documentation notes that the default stack size for a new thread on Linux/x83-32 is 2 MB. It does not mention the default for a 64-bit system but `ulimit -s` gives 8MB.

```
func main() {
        // create a channel for the response
        out := make(chan string, 2)

        // run closures in goroutines
        go func(){ out <- "Hello" }()
        go func(){ out <- "world" }()

        // block until out receives both values
        fmt.Printf("%v, %v.\n", <-out, <-out)
}
```

Note that the output of this program is not necessarily "Hello, world". Since the closures are executed concurrently there are no guarantees about the order of execution.

The above example also demonstrates how channels can be used to synchronize goroutines. The program will not exit until it has received two values from `out`. This is different from the earlier example that exited as soon as the goroutine was launched.

Select statement

Select statement is a way to choose which of a set of possible send or receive operations will proceed.

Execution of a select statement has some conditions:

1. If one or more cases can proceed, a single case is chosen randomly. Otherwise a default case is chosen.
2. Default case is selected if no other cases can proceed.
3. Operations on a `nil` channel never proceed.
4. Receive on a closed channel always proceeds.
5. Send on a closed channel panics.

For example, a common use of (1), where select is used to choose from two channels[4], whichever receives first:

4. `time.After(d)` is a function that returns a channel that receives after duration `d`. (Package time 2015)

```
select {
case x := <-ch:
        fmt.Printf("received %v from channel\n", x)
case <-time.After(timeout):
        fmt.Printf("select timed out\n")
}
```

An example of (2), where select is used to test if a signal channel is closed:

```
select {
case <-ch:
        fmt.Printf("channel is closed")
default:
}
```

The above example assumes that the channel is not used to send anything. The closed state and the fact that receive on a closed channel always proceeds (4) is simply used as a signal.

It is also possible to differentiate whether a channel receives because it has a value or because it is closed:

```
select {
case x, ok := <-ch:
        if ok {
                fmt.Printf("received %v from channel\n", x)
        } else {
                fmt.Printf("channel is closed")
        }
default:
}
```

In the above example, `ok` is true if channel received a value and false otherwise. If the channel was closed immediately after sending a value to it and before receiving from it, `ok` will be true on the first receive and false on the second receive.

2.2.4 Tooling

Go ships with a set of tools for working with Go source code. The toolchain contains tools to compile packages and dependencies, format package sources to a canonical form, download and install packages from the Internet, run tests, etc.

Go tools and their usage are described in detail in the official documentation (Getting Started 2015; How to Write Go Code 2015).

2.3 Web frameworks

The user interface relies on client-side web frameworks to ease the development.

AngularJS is used for the client-side data-binding while Bootstrap is used for consistent look and feel.

2.3.1 AngularJS

AngularJS provides a usable API for client-side data-binding and allows the separation of application logic from the layout.

2.3.2 Bootstrap

Bootstrap is a framework for creating web applications that have a consistent look. It provides templates for typography and interface controls.

3 Concept and implementation

This chapter attempts to explain the approaches taken in the implementation and the issues that were encountered.

It describes the general architecture of the system before going into the implementations of the specified subsystems implied in section 1.2.

3.1 Architecture

The overall design of the system is based on agent-server architecture:

Agent is a remote program that knows how to apply configurations to the target and collect performance statistics. An agent connects to a *server*.

Server maintains a central repository of traffic control configurations and maintains status of individual agents. It also server the user interface with a built-in HTTP server.

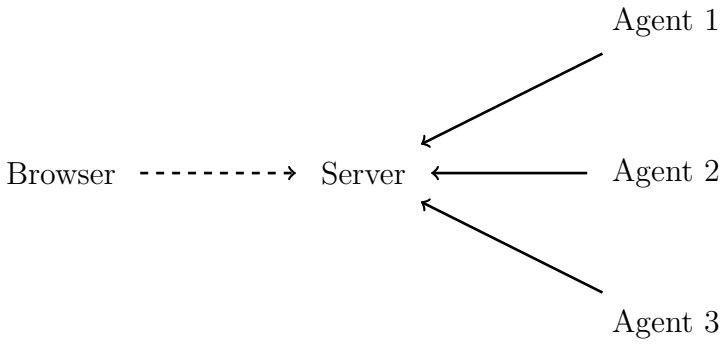

Figure 1: Agent-server architecture

The architecture has a simple goal: a user updates configurations through the user interface and the server makes sure that those configurations are up to date on the agents.

3.2 Agent

An agent is a program that runs on a remote machine and communicates with the server. It receives commands from the server and applies them on the local machine. Additionally it collects system statistics and sends them to the server.

An agent establishes a connection to the server at startup. If it cannot contact the server, it uses a backoff algorithm to increase the time between connection attempts. This is also done when a connection is lost without first receiving `disconnect` message. (see 3.5)

An agent is composed of a number of *drivers* that translate RPC requests into actions on the local machine. An example of such a driver is the `tc` driver, which is responsible for applying traffic control configurations (see 3.2.1). Another example would be the `stat` driver that extracts performance statistics from the system.

3.2.1 Drivers

Each driver is a separate sub-package that is statically linked into the agent executable. A driver is responsible of a group of related actions such as collecting system statistics, or applying traffic control configurations.

Drivers are addressed through a file system-like addressing scheme (see 3.5). For example, the traffic control driver which provides an interface for controlling Linux `tc` command uses `/tc` as the RPC entry point.

The addressing scheme was chosen so that the translation of user interface actions to RPC calls would be straightforward.

Four divers were implemented during the development:

`exec` executes commands on the local system and collects the output. This is was used in development but was later removed.

`info` collects system information such as hostname and network interfaces.

`tc` provides an interface for calling the `tc` command on a Linux machine.

`stat` collects general system statistics from the `/proc` file system.

`tc` driver

The `tc` driver is an important part of the system. It translates configurations sent from the user interface into calls to the `tc` command in the agent system.

To simplify the operation of the driver, it is assumed that the operations are applied to an empty configuration. While `tc` provides operations to add, modify and delete objects, the above rule requires that only `add` operations are allowed for `qdisc`, `class` and `filter` objects.

The user interface constructs a JSON object that describes an ordered set of configurations that should be applied to the agent system (see 2).

Field	Type	Description
object	`string`	Type of object (`qdisc`, `class` or `filter`)
dev	`string`	Target device
parent	`string`	Parent handle
handle	`string`	Object handle
object_kind	`string`	Kind of the object
options	`object`	Options specific to the object

Figure 2: `tc` configuration entry for a traffic control object

The exact content of the *options* field is specific to the object in question. It is generally a JSON object with a variable number of keys associated with arrays of values.

For example, given a JSON object sent from the user interface:

```
[
        {
                "object": "qdisc",
                "dev": "eth0",
                "parent": "root",
                "handle": "1",
                "object_kind": "netem",
                "options": {
                        "delay": ["50ms", "10ms"],
                        "reorder": ["0.5"]
                }
        },
        {
                "object": "qdisc",
                "dev": "eth1",
                "parent": "root",
                "handle": "2",
                "object_kind": "netem",
                "options": {
                        "rate": ["1Mbit"]
                }
        }
]
```

The `tc` driver translates it into the following commands:

```
tc qdisc add dev eth0 root handle 1 netem \
   delay 50ms 10ms \
   reorder 50%

tc qdisc add dev eth1 root handle 2 netem \
   rate 1mbit
```

The above commands set two configurations:

eth0 100 ms delay with 15 ms jitter using a normal distribution. 50 % of the packets are sent immediately while the rest are delayed.

eth1 1 Mbit rate limit.

When a new configuration is applied, the driver must remove all existing rules before applying new rules. Since all objects must be attached to a parent object or `root`, it is simple to remove all configurations by removing the `root` object:

```
tc qdisc del dev eth0 root
```

The driver functions similarly with `class` and `filter` configurations.

3.3 Server

Server is a program that listens for incoming agent connections. It maintains a list of known agents and their connection status and configurations. When a new agent joins, the server cross-references the agent's ID with known agents. If the ID is not currently connected, it accepts the connection.

3.3.1 Data storage

Agent meta-data, configurations and logs are stored in flat files in the local file system and cached in memory. The file system is touched when data is updated or deleted, or when it is first read. Access to the data is synchronized with a shared-exclusive lock[1] to avoid data corruption, but also allowing concurrent read access from multiple processes when the data is not being modified.

1. Go standard library provides `sync.RWMutex` which implements a shared-exclusive lock. A shared-exclusive lock guarantees exclusive access for writing, while allowing concurrent access for reading.

```
/var/frob/data/
    {id}/
        agent.log           # agent logs truncated to last N logs
        agent.config        # agent configuration
        agent.info          # agent meta-data
        agent.status        # agent status
```

A major limitation with this approach is unbounded cache size. There is no general limit to the amount of memory used by the cache. While this is not a problem with agent meta-data and configurations in this particular setup, it could become a problem with logs. The issue with logs is be side-stepped by using a circular buffer for the log cache. This will make the retrieval of old logs slower but it is not a common case.

In addition to the agent specific storage, all agent and server logs are written to `/var/log/frob/`:

```
/var/log/frob/
        server.log
        agent-{id}.log
```

It would be possible to use an existing key-value store or even a relational database for data storage but it is difficult to justify the added dependency at this point.

3.3.2 HTTP server

The server has a built-in HTTP server that is used to serve the user interface. It has two entry points: one for the user interface and another one for the REST API [2]. The server is implemented using Go's `net/http` package.

`net/http` package

Go standard library `net/http` package (Package http 2015) provides a built-in HTTP server implementation.

A simple web server has a server and a request handler:

2. The API is not strictly a REST API as defined in (Fielding 2000, Chapter 5).

```
package main

import (
        "fmt"
        "net/http"
)

func hello(w http.ResponseWriter, r *http.Request) {
        fmt.Fprintf("Hello, %v", r.URL)
}

func main() {
        http.ListenAndServe(":8000", http.HandlerFunc(hello))
}
```

All requests coming to the server are passed to the `handler`, which reads the request and writes response using the `http.ResponseWriter`.

A more complex server can replace the default handler with a router that routes requests to handlers based on path and possibly the method. While the `http.ServeMux` provided by the standard library is useful in simple cases, it does not allow parametrized routing.

`httprouter` package

Frob uses `httprouter` package for the HTTP request routing. It supports variables in the routing pattern instead of a simple prefix matching as in the default mux in Go standard library `net/http` package. (Schmidt 2013)

`httprouter` was chosen over other routers that provide parametrized routing mainly because of a compact implementation and lack of external dependencies. While the package also promises good performance and zero garbage, those were minor considerations.

Named parameters provide a convenient way to define the server API. Specifically, `httprouter` guarantees that any single pattern matches exactly one or no routes. This is an improvement over the standard library that returns the first registered handler with a matching route.

For example, the above example written using `httprouter`:

```
package main

import (
        "fmt"
        "net/http"
        "github.com/julienschmidt/httprouter"
)

func hello(w http.ResponseWriter, r *http.Request, p httprouter.Params) {
        fmt.Fprintf("Hello, %v", p.ByName("name"))
}

func main() {
        router := httprouter.New()
        router.GET("/hello/:name", hello)

        http.ListenAndServe(":8000", router)
}
```

The example creates a new `httprouter.Router` and attaches a handler for a parametrized route. Since the router implements the `http.Handler` interface, it can be supplied to the server as a handler.

3.4 User interface

The solution provides a web user interface served from a built-in web-server in the server application. It provides a dashboard that summarizes all connected agents and recent log events. Individual agents can be accessed through the dashboard or through an index of known agents. Agent interface allows viewing and updating agent configurations as well as accessing statistics and latest log messages for the agent.

This section summarizes the requirements for the user interface before going into some details about the implementation.

3.4.1 Requirements

In addition to the requirements listed in 1.2.4, there are some more specific requirements that need to be addressed make the application more usable to users:

- Show VyOS/Vyatta interface description in the user interface.
- Live view to agent statistics and logs.

3.4.2 Implementation

The user interface uses AngularJS for client-side data binding and templating. This makes it easier to create the dynamic views required by the dashboard and the agents.

AngularJS was chosen mainly because it is already used in an existing system. In hindsight the choice was unfortunate because halfway through the development AngularJS 2.0 was introduced. It makes changes that are not backwards compatible, making it less appealing in the target organization.

Bootstrap 3 is used to give a consistent look and feel to the user interface.

Traffic control configuration

A particularly challenging part of the user interface is the traffic control configuration. Ideally it needs to be able to provide list of possible configuration options as well as checks against invalid configurations.

For example, Netem object has the following options that can be applied per network interface:

limit limits how long the selected options take effect.

delay adds a delay to packets outgoing packets.

loss adds a random loss probability to outgoing packets.

corrupt emulates random errors introduced in some of outgoing packets.

duplication of random outgoing packets.

reordering of random outgoing packets.

rate throttles connection to a specified transmission rate.

Additionally, each option can have several parameters, for example random, 4-state Markov and Gilbert-Elliot for the `loss` option. Also `reordering` requires that the `delay` option is set. An example of the Netem configuration interface can be seen in figure 3.

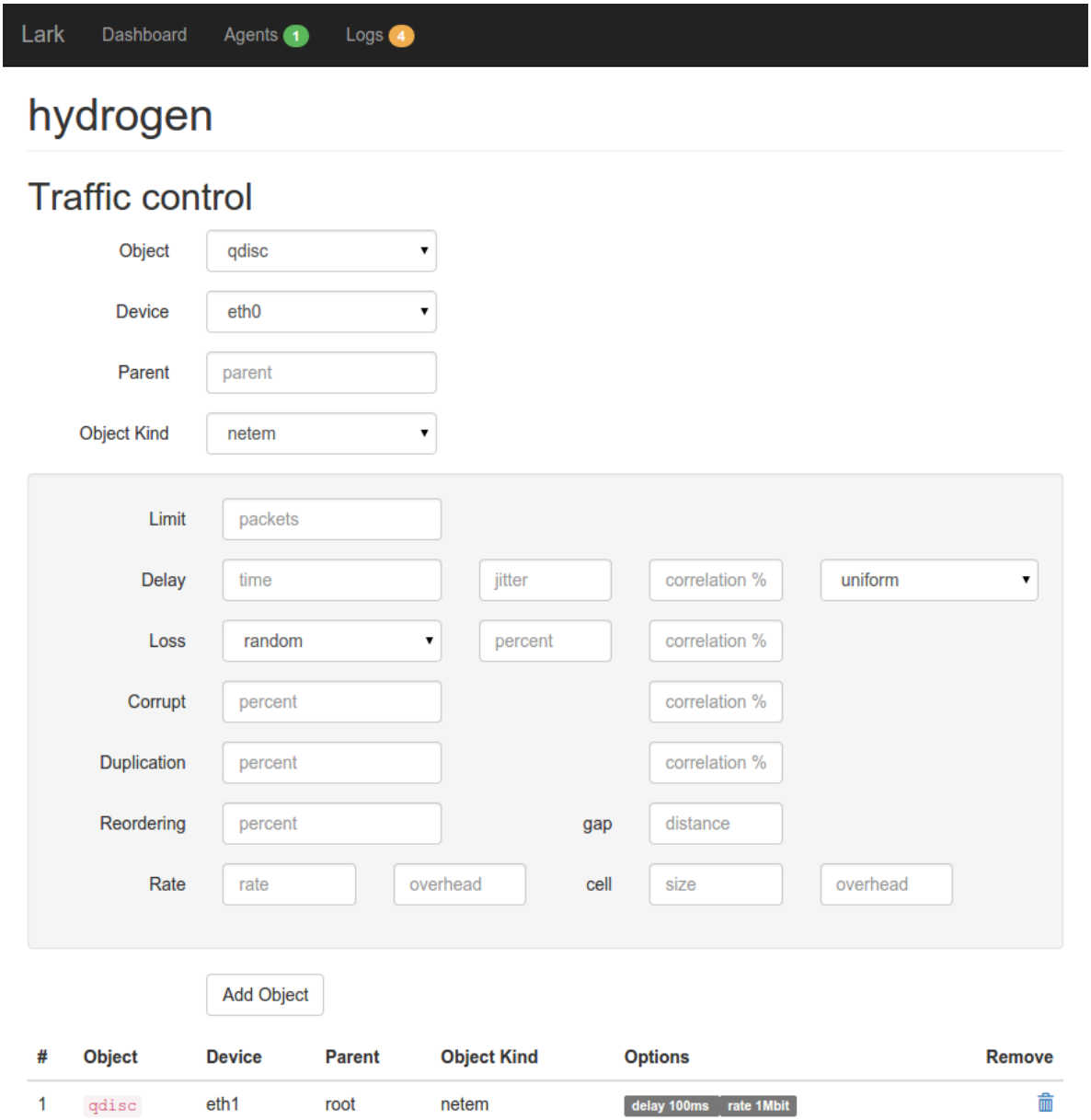

Figure 3: Netem configuration interface

3.5 Agent-server communication

Two different approaches were considered for the agent-server communication: a *message queue* and an *RPC protocol*. It was difficult to decide on the approach. Both seemed like they would fit the need while also presenting some trade-offs (see 1.2.5 and 1.2.5).

Before decision could be made between the two approaches it was necessary to come up with a minimum set of requirements the approach should satisfy:

- *Agents* and *server* should be able to negotiate a connection and read and write

data from one another (`connect`, `disconnect`).

- *Server* should be able to read and write data from *agent* (`read`, `write`).
- Above requests should receive a response if they succeed or fail (`ok`, `error`).
- *Agents* should be able to push events to the server (`event`). Events are out-only. They are used to collect log messages and statistics from remote machines.

3.5.1 Messaging patterns

A messaging pattern is a network-oriented architectural pattern for a system with separate communicating parts.

This section frequently refers to *client* and *server* in the descriptions. These terms should be considered in the context of a single message. A *client* is a component that initiates an exchange with a message and *server* is a component that receives the message.

Request-response

In request-response message exchange every message on the wire is either a request or a response to a request (see 4). Although more verbose than a simple push This allows the *server* to acknowledge *client* that a request was received and processed.

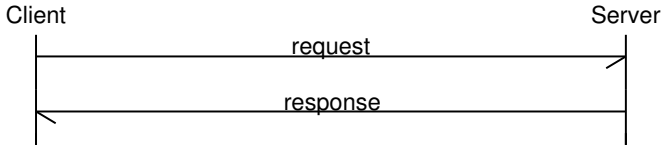

Figure 4: Request-response pattern

An obvious issue with the request-response pattern is that it can slow down the communication considerably. Assuming an ideal network with infinite bandwidth and latency of K, it takes at least $2K$ for the client to send a message of any size and receive a response of any size. The communicating process must block[3] until a response is received.

Request-response pattern is ill-suited for situations where latency or potential of blocking communication is an issue.

3. In computing, a process is *blocking* when it is waiting for an event before continuing.

One-way

One-way pattern is a messaging pattern where the *client* sends a message to the *server* without expecting a response (see 5). This is a good pattern when the client is not overly concerned whether the messages are received or when long response time could be an issue.

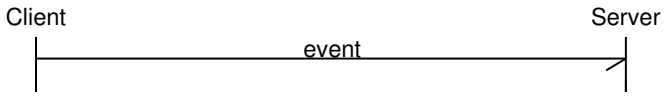

Figure 5: One-way pattern

An example of a situation where one-way communication could be desirable is logging.

Typical IP network is relatively unpredictable when compared to memory or disk I/O. It only guarantees best-effort delivery; each packet may or may not reach the destination intact, and in a timely and predictable manner, if at all. (Postel 1981a)

In such an environment it might not be desirable to expose an application to the latencies of the network. Rather, it is easier to rely on the guarantees provided by the network layer and only deal with situations where the connection cannot be established at all or where it is unexpectedly terminated.

3.5.2 Synchronous and asynchronous messaging

In the context of this thesis, synchronous and asynchronous message exchange refers to request-response type messaging:

Synchronous messaging describes communication between two endpoints where the sending endpoint must wait for a reply before continuing processing.

Asynchronous messaging describes communication between two endpoints where the sending endpoint does not need to wait for a reply before continuing.

This distinction makes most sense on the transport[4] level. A request-response exchange over a transport can be considered asynchronous when it is possible to start a new exchange before the previous request has returned with a reply. Likewise, an exchange can be considered synchronous if the exchanges must be sequential.

It is important to note the difference between asynchronous transport and API. While a non-blocking API might appear superficially asynchronous, it is not truly so if the requests are handled synchronously. For a non-blocking API to behave

4. In this context transports means any mechanism that is used to perform a request-reply exchange.

asynchronously it has to be able to buffer messages sent over it. If buffering is not possible, the API hast to block after each request, thus making it synchronous.

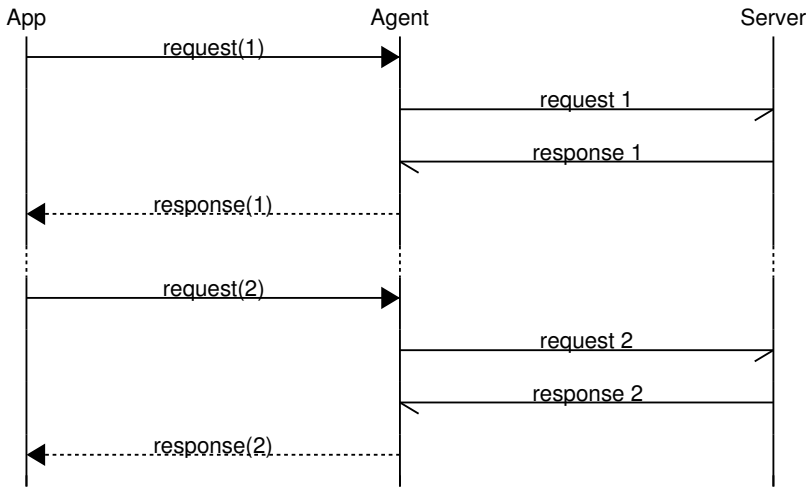

Figure 6: Synchronous transport

Figure 6 describes a blocking API over a synchronous transport. The request-response exchanges are performed sequentially, one after the other.

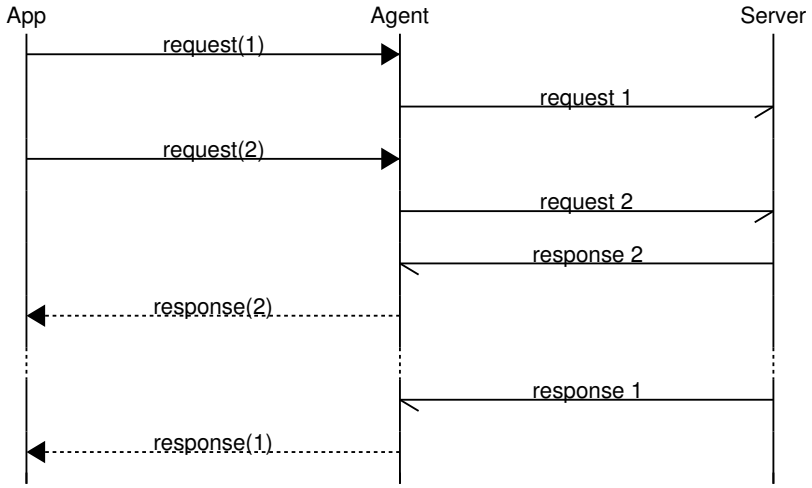

Figure 7: Asynchronous transport

Figure 7 describes a non-blocking API over an asynchronous transport. The request-response exchanges can overlap.

Figure 6 describes a blocking API over a synchronous transport. The request-response exchanges must happen in sequential order. Therefore it is necessary to somehow ensure that overlapping exchanges won't happen. Easiest way to do this is with a mutex or a semaphore.

It is also possible to have a blocking API over asynchronous transport. In such case it is not necessary to guard the API with a lock; the transport can handle overlapping exchanges. It is left to the consumer of the API to decide how to use it.

In conclusion, it is easy to implement seemingly asynchronous API that behaves like a synchronously . The distinction between these two should be taken into account when designing these systems. As noted in section 4 the system described in this thesis could have been implemented with a synchronous protocol.

3.5.3 Protocol definition

Simplified Backus-Naur form grammar (Backus-Naur Form 1997) for the agent-server protocol is described below:

⟨*integer*⟩ ::= 32-bit unsigned integer

⟨*message-stream*⟩ ::= ⟨*message*⟩ '`\n`'

⟨*message*⟩ ::= ⟨*id*⟩ ⟨*command*⟩ ⟨*resource*⟩
| ⟨*id*⟩ ⟨*command*⟩ ⟨*resource*⟩ ⟨*body*⟩

⟨*id*⟩ ::= ⟨*integer*⟩

⟨*command*⟩ ::= ⟨*request*⟩
| ⟨*response*⟩
| ⟨*event*⟩

⟨*request*⟩ ::= '`connect`'
| '`disconnect`'
| '`read`'
| '`write`'

⟨*response*⟩ ::= '`ok`'
| '`error`'

⟨*event*⟩ ::= '`event`'

⟨*resource*⟩ ::= a string containing letters [a-z], numbers [0-9] and characters '`-`', '`_`' or '`/`'.

⟨*body*⟩ ::= any printable UTF-8 character

Message structure

As described in figure **??**, each protocol message has following format:

```
<id> <command> <resource> [body]
```

A message has three mandatory fields and a body that may or may not be present, depending on the type of message:

`<id>`

Message identifier. The protocol uses it to map responses to requests. An agent message identifier is always an odd number and, respectively, server message identifier is an even number.

`<command>`

Message command. Command is a case sensitive lowercase word: `connect`, `disconnect`, `read`, `write`, `ok`, `error` or `event`.

`<resource>`

Message resource. For `connect` and `disconnect` messages, the resource is (`/`). For other messages the resource is used to forward the message to an appropriate handler. In either case, only lowercase alphanumeric characters, dash (`-`), underscore (`_`) and forward slash (`/`) are valid characters.

`[body]`

Optional message body. This is a JSON (Bray 2014) encoded string. `error` messages have a standard body. Other messages can have bodies that best suit the purpose.

Furthermore, a command can be be either a requests, response, or an event.

Requests

Requests are messages that are expected to receive either `ok` or `error` in response:

`<id> connect / <body>`

Connect message is the first message an agent sends to a server. The message body must contain the name of the joining agent (`agent-id`) and protocol version (`version`). If server receives any other message before `connect` it must terminate the connection.

Only an agent should send a `connect` message. Server only listens for incoming connections.

More details of the handshake sequence are given later (see 3.5.4).

`<id> disconnect / <body>`

Both agent and server should send a `disconnect` message if they are disconnecting normally. The receiving end should respond with an `ok` message.

`<id> read <resource> <body>`

Server can read data from an agent by sending a `read` command. The `resource` field identifies the resource that is requested. `body` can contain additional data

that the message handler can use to process the request.

Agent responds with `ok` if the request was successfully processed. The contents of the response are in the `body` field of the response message.

Agent responds with `error` if the request could not be processed. This can be because the resource does not exist, it timed out, or some other reason.

`<id> write <resource> <body>`

Server can write data to agent by sending a `write` command. The `resource` field identifies the resource that is to be updated and `body` contains the new value.

Agent responds with `ok` when the data is successfully updated.

Agent responds with `error` if the request could not be updated. This can be because the resource does not exist, time-out, or some other internal application error.

Responses

Responses are confirmations that a *request* was received and handled:

`<id> ok <resource> [body]`

A success is indicated by an `ok` message. The message body can provide feedback or it can be empty.

`<id> error <resource> [body]`

A failure to perform a command is indicated by a `error` message. The message should provide enought context to troubleshoot the problem.

The protocol has a configurable timeout for a response. It expects a response to a request within that timeout. If there is no response within the timeout, a request is canceled and timeout error is returned.

Events

Events are messages without a response:

`<id> event <resource> <body>`

Event is a message sent by agent. It has no response. They can be used to push telemetry data to the server.

Events are used for telemetry data that does not require validation by the server, such as, statistics or logs.

There are no guarantees that events are handled properly. While the underlying transport provides some guarantees about delivery, the application might choose to ignore them. (Postel 1981b, 10)

3.5.4 Connection

The connection encapsulates a single point-to-point protocol connection. The implementation uses an underlying TCP transport for its connection-oriented nature and delivery guarantees. (Postel 1981b)

Handshake

A connection is always initiated by an *agent*. Once an agent has established a TCP connection to the server it initiates the protocol handshake.

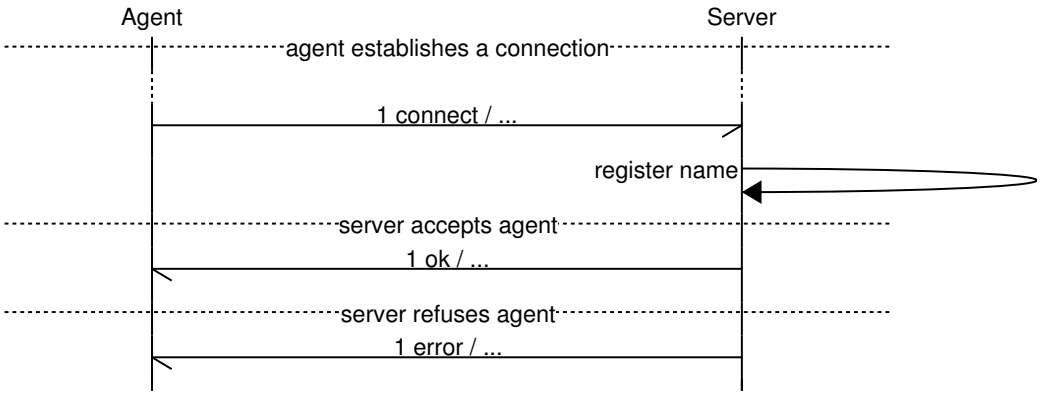

Figure 8: Agent-server handshake

As seen in figure 8, the handshake sequence is very simple. It confirms that agent and server are using the same protocol version (the version is not negotiated) and exchanges the names between the endpoints.

1. Agent sends a `connect` message with an odd message number. The message must have `version` and `agent-id` fields in the body:

```
1 connect / {
        "version": "1.0",
        "agent-id": "marmaduke",
}
```

2. The server responds with `ok` if `version` is supported by the server and `agent-id`

does not conflict with connected agents. `server-id` field identifies the server:

```
1 ok / {
        "version": "1.0",
        "server-id": "shirley",
}
```

3. Otherwise the server responds with `error`:

```
1 error / {
        "error": "agent-id already reserved",
}
```

Similarly an error could indicate that the protocol version is not supported.

If the agent sends any messages other than `connect` before the server has responded with `ok`, the server will close the connection. Likewise, if the server responds with anything but `ok` the agent should close the connection.

Once a connection is established both endpoints agree on the protocol version and have unique names for each other. After this point most of the traffic is initiated by the server, with the exception of `event` messages that are used to deliver performance statistics and logs to the server.

Malformed messages

The protocol is very strict about correctness. Any malformed message [5] causes the recipient to close the connection. In the case of a malfunctioning node, this ensures that the connection won't be flooded with malformed messages. In either case the errors should be logged by the receiving end.

Connection errors

If the connection is lost due to a transport error, the protocol provides means of recovery after a timeout. The recovery mode uses a naive backoff mechanism[6] to avoid flooding the network with `connect` and `error` messages.

5. TCP provides an error correction mechanism so transport errors should not cause broken messages
6. The backoff algorithm uses feedback to incrementally decrease the rate of reconnection, in order to avoid congestion.

Concurrency

The `Conn` interface hides the asynchronous nature of the protocol. This is implemented with a *wait queue* that maps responses to sent requests.

Process from the perspective of the `Conn` is as follows:

1. Request a channel from the *wait queue*.
2. Writes the request to the underlying connection.
3. Wait for a response or the channel, or a timeout.

Operation of the *wait queue* is two fold:

1. Get next request identifier and create a channel for the response.
2. Add the channel to a map using the request identifier as the key.

When a user makes a request using a `Conn` object, it first requests a channel from the wait queue. Once a channel is acquired, `Conn` writes the request to the underlying connection and proceeds to receive on the channel. When a response arrives, a receiving goroutine reads it from the connection and passes it to a wait queue. Wait queue checks the response identifier against known request identifiers: if there is a match, the message is dispatched to the waiting sender.

The protocol allows concurrent requests to the same resources. It is necessary to limit asynchronous access to requests that can't handle it. These situations can be handled manually or by the *request multiplexer* (see 3.5.5).

3.5.5 Request multiplexing

All `read`, `write` and `event` messages received through a connection after a handshake are passed to a `Handler` that is registered when a connection is created.

`Handler` interface

Handler is an interface that receives a connection and a request and returns a JSON encoded message body or an error:

```
type Handler interface {
        Handle(req *Request) ([]byte, error)
}
```

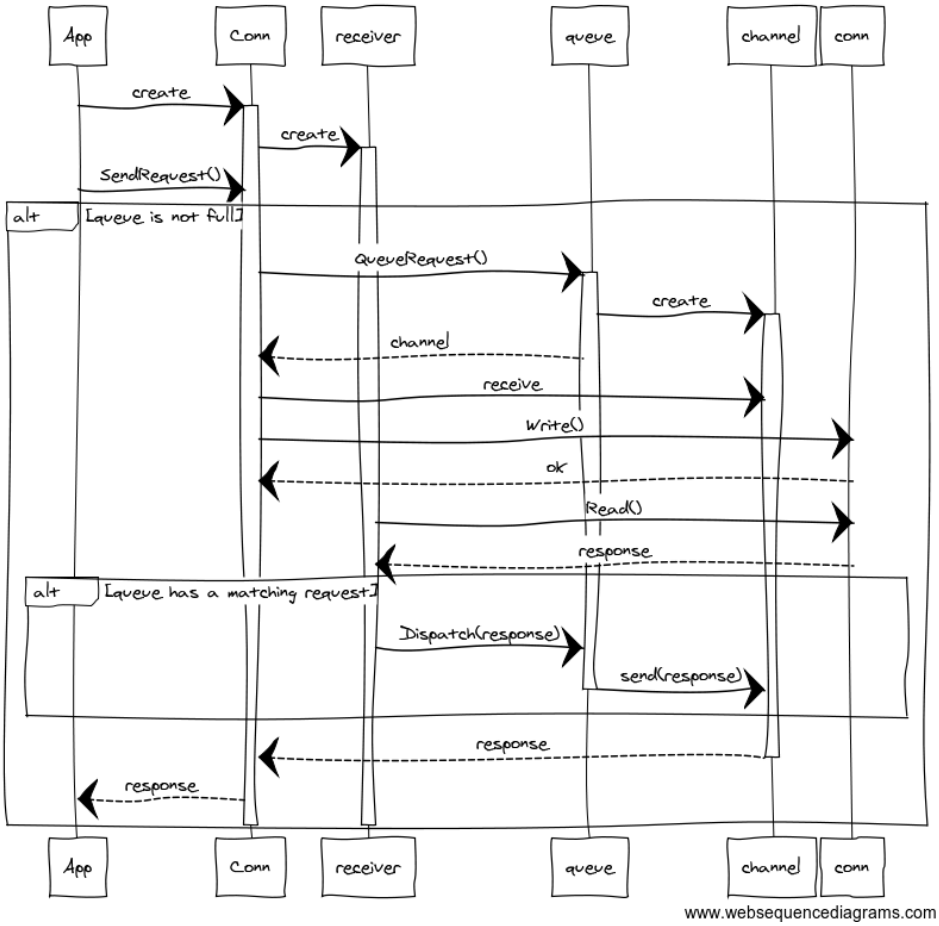

Figure 9: Request-response sequence

Message body is translated back into an `ok` message and sent to the recipient, while an error is translated into an `error` message.

Mux

Typically the default handler would be the `Mux` type that implements a *request multiplexer*. It maps messages by *resource* and *command* fields to a *handler* that knows how to handle the message.

Additionally, a mux knows how to serialize access to synchronous resources, for example the `tc` driver.

Following example shows how the API works:

```
func (a *Agent) createAgentMux() Handler {
        mux := proto.NewMux()
        mux.Handle("get", "/info", a.handleGetInfo)
        mux.Handle("put", "/tc", proto.ErrorOnAsync(a.handlePutTc))
        mux.Handle("get", "/stat", a.handleGetStat)
        return mux
}
```

The example creates a mux that has three handlers. `put:/tc` uses `proto.Sync`. It decorates the `a.handlePutTc` handler with `proto.ErrorOnAsync` that returns an error unless the previous request has returned or timed out. Similar technique could be used to implement caching, request coalescing, or other access modifiers, as needed.

4 Evaluation

Many of the approaches taken in the implementation tend towards specific, rather than generic, solutions. It is a conscious choice. While it can be argued that generic solutions promote code reuse, it is more likely that the solution becomes too broad and harder to maintain if the generic solution is implemented too early.

Functionality

The software has two major flaws inherent to its split functionality. Firstly, it is a specialized configuration management software for network emulation settings. Secondly it is a performance monitoring software to troubleshoot any errors that rise because of the former function.

The configuration management on its own could be achieved by a much simpler setup. A complex system has more modes of failure than a simple system. Incorporating performance monitoring introduces complexity that, while usefull for troubleshooting purposes, is not strictly necessary for configuration purposes.

Additionally, the integrated performance monitoring recreates a small subset of the monitoring services already available in the target environment.

User interface

The user interface exposed by the server is limited at best. It is not nearly as usable as it could be. While it is easier to configure many agents with the current system, as compared to doing it manually, it could be better.

Agent-server protocol

The agent-server protocol mixes *request-response* and *one-way* messaging patterns in a single protocol (Hophe and Woold 2003). While it is possible to do both, it is not a good idea to diverge from the common case unless the benefits outweigh the added complexity. In this particular case, since `event` messages are only used to deliver statistics and logs to the user interface relatively infrequently, having a separate method for one- way messages is not well justified.

The agent-server protocol also supports asynchronous messages. While it would be desirable in a system with high contention, it is not the case with the implementation. Instead, as noted above, the frequency of messages is rather slow: server sends new configurations to agents and agents occasionally send status updates to the server.

A synchronous agent-server protocol that supports only request-response messages would be much simpler to implement and maintain while still delivering the performance required by the system.

In addition to the obvious drawbacks, there is one that is harder to identify. Most, if not all, actions supported by the system are synchronous in nature. A configuration cannot be applied concurrently on a single machine and logs and stats should be delivered in the order they are sent. While the underlying transport makes guarantees about the order of delivery, the application does not make any guarantees that the messages are processed in the order they are sent. In fact there no real guarantees that the messages are even sent in the order the sending application thinks they are sent. It is possible to order statistics and timestamps correctly based on timestamps, and the configuration driver can simply return with an error if it is already applying a configuration, but these are problems that would be easier to avoid with a synchronous protocol.

Summary

In summary, the software would better off if it focused solely on delivering network configurations to the target machines in as simply a manner as possible. It it easier to add features as needed instead of changing them once they are there.

5 Conclusions

Once there was a subject for the work – a software for managing network emulation configurations in a virtual Internet – the first step was to decide how to approach the problem.

An approach was chosen that would involve writing a custom protocol that connects a central server to multiple agents hosted on the network. Over many iterations, the protocol changed from a text-based synchronous protocol with many options, into a text-based asynchronous protocol with very few options.

It was a good choice to move away from more options to less options. If one goes to the length of writing a custom protocol, then it would make little sense to make it broader than the solution requires.

This idea could have been taken even further. From purely pragmatic point of view a synchronous protocol would have sufficed in this case. The amount of traffic between the nodes is relatively small and does not necessarily warrant an asynchronous protocol and the pitfalls that it introduces (see 4).

For purely practical purposes, there is no need for a custom protocol. Any number of existing RPC protocols or simply an HTTP server on each client would be enough. But this conclusion can be taken as a result in itself.

6 Future work

There are parts in the system that could use improvement. From practical standpoint the most lacking part is the user interface. In addition to that the agent-server communication could be remodeled to something simpler.

User interface

The part of the system that is most lacking is the user interface. Currently it supports only one configuration for the entire system. It could support multiple configurations that could be swapped. That would require some logic regarding what to do when the target system does not match the configuration. Also having multiple configurations for a single node would be good. It would be easier to swap them as needed.

Agent-server communication

Depending on other needs in the target system, there are several approaches regarding the agent-server communication. From purely academic standpoint, a binary implementation of the current protocol would simplify the implementation while delivering better performance. The changes would also be mostly isolated from the rest of the application.

A more likely scenario is that the protocol isn't reliable enough in a larger network and with longer uptimes. One such problem could be incorrectly terminated TCP connections that would leave connections "open" on the other end.

References

Arpaci-Dusseau, Remzi H., and C. Andrea. 2014. “Introduction to Distributed Systems.” Accessed April 12, 2015. Retrieved from: http://pages.cs.wisc.edu/~remzi/OSTEP/dist-intro.pdf.

Bray, T. 2014. *The JavaScript Object Notation (JSON) Data Interchange Format.* Technical report. Internet Engineering Task Force.

Fielding, Roy Thomas. 2000. “Architectural Styles and the Design of Network-based Software Architectures.” AAI9980887. PhD diss.

Backus-Naur Form. 1997. Free On-Line Dictionary of Coputing. Accessed May 27, 2015. Retrieved from: http://foldoc.org/Backus-Naur%20Form.

Effective Go. 2015. golang.org. Accessed May 11. Retrieved from: https://golang.org/doc/effective_go.html.

Frequently Asked Questions (FAQ) - The Go Programming Language. 2015. golang.org. Accessed March 24. Retrieved from: https://golang.org/doc/faq.

Getting Started. 2015. golang.org. Accessed April 20. Retrieved from: https://golang.org/doc/install.

Go 1.4 Release Notes. 2015. golang.org. Accessed May 11. Retrieved from: https://golang.org/doc/go1.4.

How to Write Go Code. 2015. golang.org. Accessed April 20. Retrieved from: https://golang.org/doc/code.

Package http. 2015. golang.org. Accessed April 12. Retrieved from: http://golang.org/pkg/net/http/.

Package time. 2015. golang.org. Accessed March 4. Retrieved from: http://golang.org/pkg/time/.

The Go Blog: Share Memory By Communicating. 2015. golang.org. Accessed May 11. Retrieved from: https://blog.golang.org/share-memory-by-communicating.

Hemminger, Stephen. 2015. “Network Emulation with NetEm” (April).

Hoare, C. A. R. 1985. *Communicating Sequential Processes.* Upper Saddle River, NJ, USA: Prentice-Hall, Inc. ISBN: 0-13-153271-5.

Hophe, Gregory, and Bobby Woold. 2003. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions.* October. ISBN: 078-5342200683.

Kleiweg, Peter. 2014. zmq4. Accessed March 17, 2015. Retrieved from: https://github.com/pebbe/zmq4.

Linux User's Manual. 2001. *netem(8) Linux man page.* Linux User's Manual, November. Retrieved from: http://man7.org/linux/man-pages/man8/tc-netem.8.html.

Linux User's Manual. 2001. *tc(8) Linux man page.* Linux User's Manual, December. Retrieved from: http://man7.org/linux/man-pages/man8/tc.8.html.

Postel, J. 1981. *Internet Protocol.* Technical report. Internet Engineering Task Force.

Postel, J. 1981. *Transmission Control Protocol.* Technical report. Internet Engineering Task Force.

Schmidt, Julien. 2013. HttpRouter. Accessed April 12, 2015. Retrieved from: https://github.com/julienschmidt/httprouter.

Ylonen, T., and C. Lonvick. 2006. *The Secure Shell (SSH) Transport Layer Protocol.* Technical report. Internet Engineering Task Force.