UKRAINIAN CATHOLIC UNIVERSITY

BACHELOR THESIS

Research of developing demand-driven services using the Reactive Streams concept and RSocket

Author:
Vladyslav URSUL

Supervisor:
Oleh DOKUKA

A thesis submitted in fulfillment of the requirements for the degree of Bachelor of Science

in the

Department of Computer Sciences
Faculty of Applied Sciences

Lviv 2021

Declaration of Authorship

I, Vladyslav URSUL, declare that this thesis titled, "Research of developing demand-driven services using the Reactive Streams concept and RSocket" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

UKRAINIAN CATHOLIC UNIVERSITY

Faculty of Applied Sciences

Bachelor of Science

Research of developing demand-driven services using the Reactive Streams concept and RSocket

by Vladyslav URSUL

Abstract

The goal of this bachelor's thesis is to investigate the potential of leasing capabilities in the RSocket protocol while developing self-balancing, demand-driven microservices for usage in distributed systems...

Acknowledgements

I want to start by saying how grateful I am to my parents, who have been wonderful and tolerant of all my weirdness for the last 22 years. In my academic pursuit, I would want to express my gratitude to the numerous people who supported me, helping me through my education - Mykola Biliaiev, Danylo Shankovskyy, Yuriy Stasinchuk, Ivan Yurochko, and others. Also great thanks to my supervisor Oleh Dokuka! And last but not least, I would want to thank the individuals who make my studies in UCU a reality, which includes students such as Adrian Slywotzky, Oles Dobosevych, Yaroslav Prytula, and many more. . . .

Contents

List of Figures

List of Abbreviations

SLA	**S**ervice-**L**evel **A**greement
HTTP	**H**yper**T**ext **T**ransfer **P**rotocol
SOA	**S**ervice-**O**riented **A**rchitecture
CSS	**C**ascading **S**tyle **S**heets
TCP	**T**ransmission **C**ontrol **P**rotocol
FCS	**F**rame **C**heck **S**equence
MAC	**M**edium **A**ccess **C**ontrol
CPU	**C**entral **P**rocessing **U**nit
SDK	**S**oftware **D**evelopment **K**it
PaaS	**P**latform **a**s **a** **S**ervice

Dedicated to the first humans on Mars…

Chapter 1

Introduction

1.1 Context

Currently, sophisticated solutions are frequently comprised of a number of components that are integrated into a single platform (microservice architecture or SOA). This results in greater scalability and ease of maintaining underlying logic than monolithic apps do. Despite the numerous disadvantages of this application architecture, there are a few bottlenecks that users frequently experience. When discussing high-load distributed systems, it is critical to consider how our services will interact and also how we will handle demand and reduce response time.

This brings us to the issue of load balancing, which is a critical issue in high-demand distributed systems. There are a few ways that this is often handled in the real world, and in the majority of situations, individuals are now resolving the issue through the use of certain severe constraints. However, certain clients are incapable of buffering enormous amounts of data, and severe limitations are sometimes ineffective for big systems that are heavily laden with data that must be handled with non-blocking back pressure (like video-streaming or real-time AI solutions). This is when the notions of Reactive Streams come into play.

Therefore, it is worthwhile to investigate the possibilities of building demand-driven self-balancing services using Reactive Streams, particularly RSocket, a binary protocol that implements Reactive Streams semantics in inter-service communication and also gives possibilities for developing self-balancing services via leasing, which has not been investigated previously.

1.2 Thesis task

The purpose of this thesis is to investigate the leasing capabilities of RSocket and to develop a demand-driven service that will be used to assess the stability and performance of RSocket under load with and without the Lease method implemented.

Additionally, my objective was to get more knowledge about current trends for building a communication in distributed systems and gather real experience of working with RSocket while developing self-balancing distributed systems.

1.3 Thesis Structure

1. In the first chapter I will describe my thesis context and main tasks

2. In second chapter I will follow-up with covering the background of my thesis, by describing main concepts of Reactive Streams and RSocket as well as some historical introduction and reasoning for HTTP/2 and the need of leasing
3. In the third chapter I will cover my experiment and practical gaining during my work on this thesis
4. In the end I will make a conclusions and summarize results.

Chapter 2

Background and Related Works

2.1 Reactive Manifesto

The Reactive Manifesto was released in 2013 with the following reasoning: "Application requirements have changed dramatically in recent years. Both from a runtime environment perspective, with multicore and cloud computing architectures nowadays being the norm, as well as from a user requirements perspective, with tighter SLAs in terms of lower latency, higher throughput, availability and close to linear scalability. This all demands writing applications in a fundamentally different way than what most programmers are used to." by the creator Jonas Boner in article *Why do we need a reactive manifesto*.

Currently there is already a second version it, that was published in 2014 and it currently has more than 30 thousands of signes on it.

In general The Reactive Manifesto describing main principles of the reactive programming 2.1. There are four of main them with two additional ones as you can see at 2.1

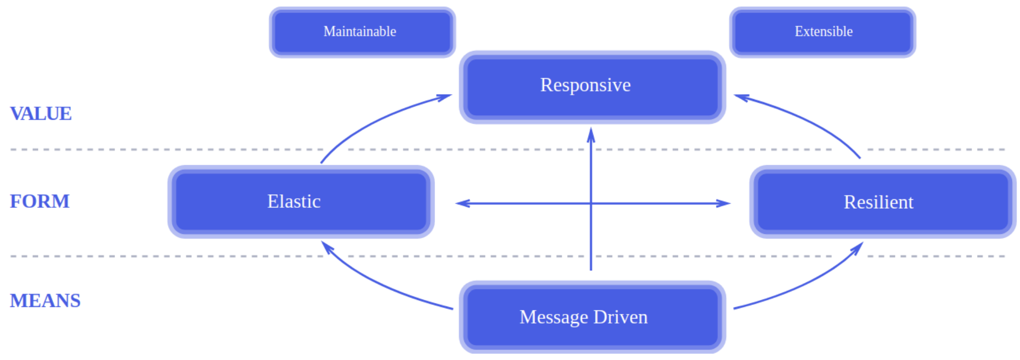


FIGURE 2.1: Characteristics of Reactive Systems from *Reactive Manifesto*

So to summarize, if we want to build a reactive system it should match with thee criteria above, that means:

1. Have an ability to react on different load. (be elastic)

2. Have a proper error handling and have an ability to make disaster recovery in a fast way. Also, failure in one component shouldn't impact other ones. (be resilient)
3. All components should be isolated and should have a proper non-blocking message-driven communication.
4. Minimized response time that also should be consistent predictable. (be responsive)

2.2 Reactive Streams

The specification for Reactive Streams is modeled around the Reactive Manifesto. The notion of Reactive Streams was primarily established to manage the exchange of stream data across an asynchronous boundary—for example, delivering items to another thread or thread-pool—while guaranteeing that the receiving side is not compelled to buffer arbitrary quantities of data (*Reactive Streams*). This is critical because, while we invest time and effort in designing services with asynchronous processing, we frequently operate with a restricted pool of workers or middlewares, which frequently becomes a bottleneck. Thus having an uncontrolled producer leads to a failure since it can overwhelm a slow Consumer with a messages.

For instance, if our Consumer' capacity is insufficient to handle the quantity of computational demands transmitted from other levels, we need queue our requests to avoid missing data, which requires greater concurrency at the worker level. However, asynchrony is required to maximize the utilization of available both computer and network resources.

To solve that problem, Reactive Streams specification introduces a mechanism called Backpressure which allows a Consumer to proactively notify a Producer about its demand. This mechanism allows a consumer to enable resilience since a Producer will never send more data than it was demanded (predictable load). At the same time, Producer can scale down its resources when there is no demand to produce messages and scale up when there is a demand (enables elasticity)

There is 3 main programming interfaces introduced to identify the Reactive Streams protocol in any programming language. First two are Publisher and Subscriber - representation of Producer and Consumer where to received data a Consumer has to allocate a Subscriber and pass it to the Publisher's subscribe method The third interfaces is a Subscription - a contract given by Publisher to a connected Subscriber. To enable Backpressure, Subscription offers a request method through which a Subscriber can asynchronously expose its demand (see *Reactive Streams*)

To simplify that interface could be splitted into a two flows: Control and Data layer (see example in 2.2 . Control layers make sure, that all communication parties are following communication rules and setting all required variable as well as taking care and notifying everyone about it's own capacity.

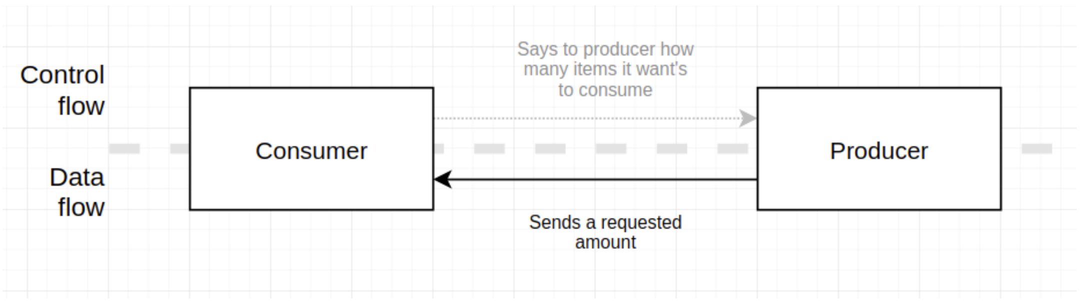


FIGURE 2.2: Demonstration of responsibilities of Control and Data flows

2.3 HTTP/2

Following Tim Berners-Lee' idea, the initial version of HTTP was launched in 1991 as version 0.9. It has been iteratively enhanced over the years, reaching version 1.0 in 1996 and 1.1 in 1997.

Over its first 14 years, HTTP has undergone just a few significant enhancements. HTTP/2 - The protocol's second major version was published in 2015. The cause for this was the emergence of new trends and requirements in people's information consumption experiences. Internet resources grew increasingly resource-intensive and performance-intensive. (*Evolution of HTTP*)

The following critical methods were introduced in HTTP/2: server push, compression, multiplexing, and request priority. These features contribute to the reduction of latency associated with the processing of browser requests.

2.3.1 Multiplexing

In HTTP/1.x we have an ability only to make a single request at once from the same connection, that was rapidly improved HTTP/2 by adding an ability to send multiple of them and receive response back in any order.

Consider the following scenario of a person accessing a website to better understand the implications of multiplexing:

On an HTTP/1.1 connection, you can download only one of those at a time. Thus, the HTML file is downloaded and then the CSS file is requested. Once that is returned, the JavaScript file is requested. When that is returned, the first image file is requested... and so forth.

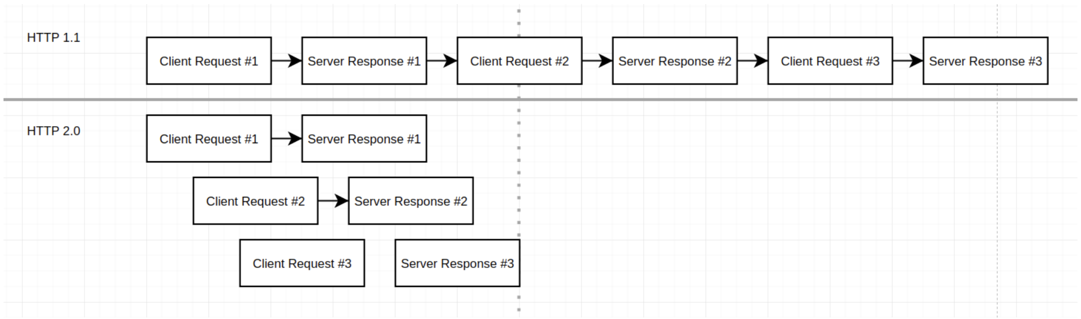


FIGURE 2.3: HTTP/1 and HTTP/2 comparison

HTTP/1.1 is a synchronous protocol, which means that once you send a request, you are stuck waiting for a response. This means that the browser spends the majority of its time idle, having sent a request, waiting for a response, then sending another request, waiting for another response again and again (see 2.3 as an example flow). Of course, complex sites with a lot of JavaScript require the Browser to perform a lot of processing, but that processing is dependent on the JavaScript being downloaded, so the delays inherent in HTTP/1.1 do cause issues in the beginning.

So, one of the primary issues on the web today is the network latency associated with the transmission of requests between the browser and the server. It may only be a few tens or hundreds of milliseconds, but they add up and are frequently the slowest part of web browsing - especially as websites become more complex and majority of users now are using mobile devices.

To bypass this limitation, browsers frequently establish several connections to the web server. This enables a browser to send many requests concurrently, which is significantly more efficient, but at the expense of the complexity associated with setting up and managing many connections (which impacts both browser and server). HTTP/2 enables you to submit many requests over a single connection, eliminating the requirement to start several connections as described above. This obviously improves performance by not delaying the transmission of requests while waiting for a free connection. All of requests go in almost) concurrently across the network to the server. The server answers to each one, and they then begin their way back. Indeed, it is considerably more powerful than that, since the web server may reply to them in any order it wishes, returning files in any order, or even breaking each file requested into parts and recombining them. This has the extra benefit of preventing a single large request from blocking all future requests. The web browser is then charged with reassembling all of the fragments. In the best-case scenario (assumes there are no bandwidth constraints - see below), if all requests are sent in parallel and are instantly responded to by the server, this implies you only have to make one round trip to download all resources.

2.3.2 Server Push

HTTP/2 introduces a new form of communication in which a server can push replies to a client (Section 8.2 of *Hypertext Transfer Protocol Version 2 (HTTP/2)*). Server push enables a server to transmit data to a client speculatively based on what the server predicts the client will require, balancing network utilization against the possibility for latency gain

Throughout history, the typical pattern for inter-service communnication was request-response pattern. To get the data, the requester submits a request to a remote server, which responds with the requested data. 2.4

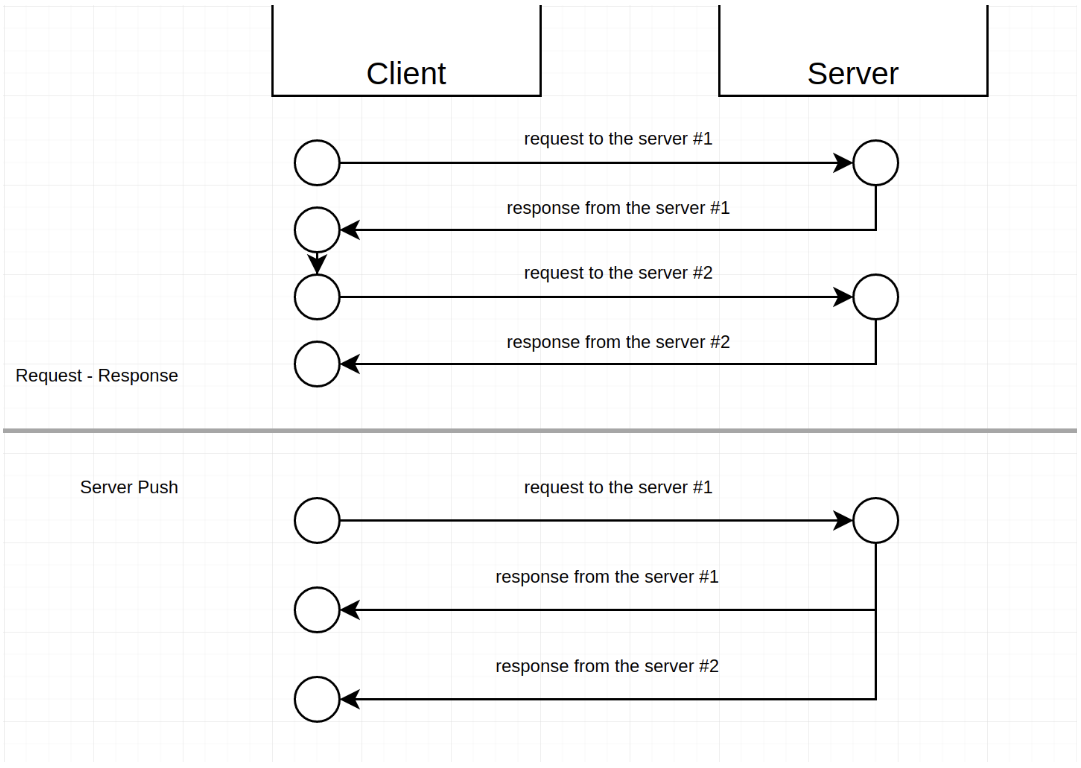


FIGURE 2.4: Request-Response and Server Push comparison

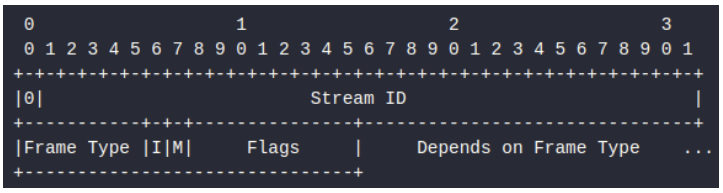
This technique has the potential to be inconvenient, as it requires users to wait for the client to locate and obtain critical elements after retrieving basic date. It can be referred to the old practise of sending hard beats to the servers, just receive notification, etc as well as retrieving single HTML page and only after request for all other resources in the case of a website. This usually is increasing latency and system load time.

By utilizing server push, we may resolve this issue. Server push enables the responder to "push" requester assets to the client in advance of the user initiating a request. When implementing this functionality, we must exercise caution to ensure that we are only transmitting information that the user will needs for the page they requested.

That is quite straightforward to visualize, and it's minimizing the networking time.

2.4 RSocket

RSocket - it's a protocol that implements the Reactive Streams semantics. It is a point-to-point binary communication protocol intended for use in distributed applications. Basically, all comunication in RSocket is build-up from binary Frames, which follow similar paradigm, that HTTP/2 are. But to make RSocket work it utilizes a lower-level transport protocol, and RSocket frames are transported over this protocol. There are a few protocols currently supported, but by specifications of RSocket - transport protocols should meet the following requirements (see *RSocket Protocol* official documentation for more details):



```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0|                         Stream ID                           |
+-----------+-+-+---------------+-------------------------------+
|Frame Type |I|M|     Flags     |     Depends on Frame Type    ...
+-------------------------------+
```

FIGURE 2.5: Frame structure example from *RSocket Protocol* official documentation

- Unicast Reliable Delivery.
- Connection-Oriented and preservation of frame ordering. Frame A sent before Frame B MUST arrive in source order. i.e. if Frame A is sent by the same source as Frame B, then Frame A will always arrive before Frame B. No assumptions about ordering across sources is assumed.
- FCS is assumed to be in use either at the transport protocol or at each MAC layer hop. But no protection against malicious corruption is assumed.

And currently it supports the following framing protocols:

- TCP
- WebSocket
- Aeron
- HTTP / 2 Stream

2.4.1 Frames

When using a transport protocol providing framing, the RSocket frame is simply encapsulated into the transport protocol messages directly. RSocket frames begin with a RSocket Frame Header. The general layout of the Header is given below in 2.5.

Each frame is structurally distinct, as seen in the sample image. There are now 12 Frame types, however this number might expand to 63 if there is a demand.

2.4.2 General characteristics of Rsocket

Multiplexity

While few logical streams are required they all could be incorporated into one single connection.

Communication types

Both parties of communication could request data, there for it makes RSocket to be a bi-directional protocol with the support of "Servers Push" and channeling. Actually, RSocket supports 4 kinds of communication which are:

1. Request-Response - for standard request-response
2. Request-Stream - for a single one-way data stream
3. Chanel - for transmission of streams of data in both directions

4. Fire-and-forget - for pushing data with no response

Backpressure

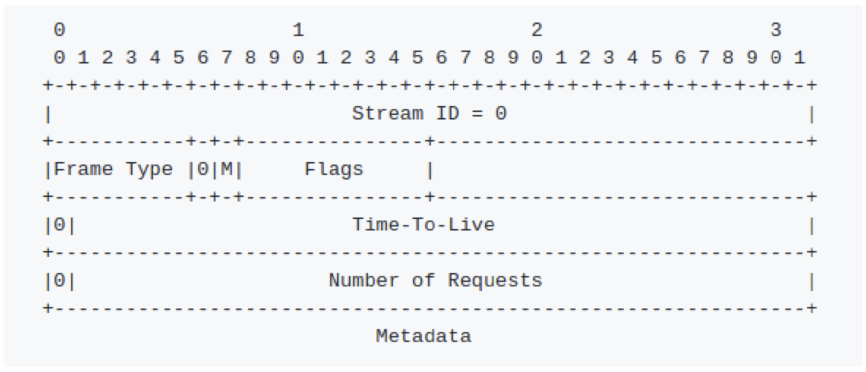
RSocket embraces Reactive Streams specification and implements it as a network protocol. Therefore it enables all the benefits of asynchronous messaging over the network mentioned before.

However, while Reactive Streams backpressure is confined to a single logical stream, RSocket offers another mechanism called leasing to provide resilience over numerous independent requests and connections.

2.4.3 Leasing

By allowing/disabling incoming data requests, leasing enables a service to maintain its stability. There are several leasing methods that may be employed in various situations in order to get the optimal match between a specific service and a leasing method.

RSocket implements Leasing by introducing the LEASE frame (see *RSocket Lease Frame*) As far as Rsocket is bi-directional LEASE frame (see structure in 2.6) could be used on both sides requester and responder.



```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                         Stream ID = 0                         |
+-----------+-+-+---------------+-------------------------------+
|Frame Type |0|M|     Flags     |
+-----------+-+-+---------------+-------------------------------+
|0|                         Time-To-Live                        |
+---------------------------------------------------------------+
|0|                       Number of Requests                    |
+---------------------------------------------------------------+
                           Metadata
```

FIGURE 2.6: *RSocket Lease Frame* structure

The proccess, that describes how does Leasing works is greatly described in *RSocket Connection Establishment* section in the official RSocket docs and is follows:

1. Based on the existence of the L flag in the SETUP frame, the client-side Requester can indicate the server-side Responder whether it will respect LEASEs or not.

2. If the client-side Requester has not set the L flag in the SETUP frame, it may send requests immediately without waiting for the server to LEASE it.

3. Before sending requests, the client-side Requester that has set the L flag in the SETUP frame MUST wait for the server-side Responder to provide a LEASE frame.

4. Accepting the contents of the SETUP frame requires the server to deliver a LEASE frame if the SETUP frame set the L flag. If the L flag is not set in the SETUP frame, the server-side Requester may send requests immediately upon accepting it.

5. If the server does not accept the SETUP frame's contents, the server MUST return an ERROR[INVALID SETUP | UNSUPPORTED SETUP] and then cancel the connection.
6. The server-side Requester replicates the client-side Requester's LEASE requests. If the client-side Requester includes the L flag in the SETUP frame, the server-side Requester MUST wait for the client-side Responder to deliver a LEASE frame before sending a request. Following a SETUP frame with the L flag set, the client-side Responder MUST send a LEASE frame.
7. If a client receives a response to a request, a LEASE frame, or sees a REQUEST type, it thinks the SETUP is approved.
8. If a client receives an ERROR, it believes the SETUP was refused.
9. A Requester MUST NOT transmit any Request frames until the connection has been established.
10. A Responder MUST NOT broadcast any PAYLOAD frames until the connection is established.

As can be seen, we may set up Leases in such a manner that the service receives the precise quantity of data that it can consume by implementing an algorithm (for example *Netflix Concurrency Limits*) that can forecast load of the service in dependence to the further requests.

Worth to mention, that the main difference between client side Request Rate or Concurrency control and Leasing is that all the decision are made on the Responder side which exactly knows about its current capacity and proactively lease it to a remote client. With such algorithm, a client does not have to work in predict-try-fail strategy and can safely rely on given capacity to make requests.

There are actually several leasing strategies that utilize various elements to give Leases with certain constraints in addition to TTL:

1. Max Frames Count
2. Max Requests Count
3. Max Concurrency Limit

Each of these strategies may be optimal for a given service set, but this needs be determined in each situation. So, as we can see adding Leasing to our communication will give service an ability to self-balance load and become demand driven.

Chapter 3

Experiment

3.1 Task Overview

The goal of current thesis is to research Leasing possibilities of RSocket and create a demand-driven service to compare it's stability & performance under load-test with and without Lease strategy applied. Therefore it was decided to run load-test on two different setups of an identical service sets with and without Lease enabled.

3.2 Implementation

3.2.1 Architecture overview

For the experiment I've decided to take popular architecture design consist from three layers of services: Producers, Brokers and Workers (3.2). I have two strategies communications between services, as described above: First - and with Lease enabled, second - without Lease and Rate Limits. Bellow I will describe the role of each service and functionality of them.

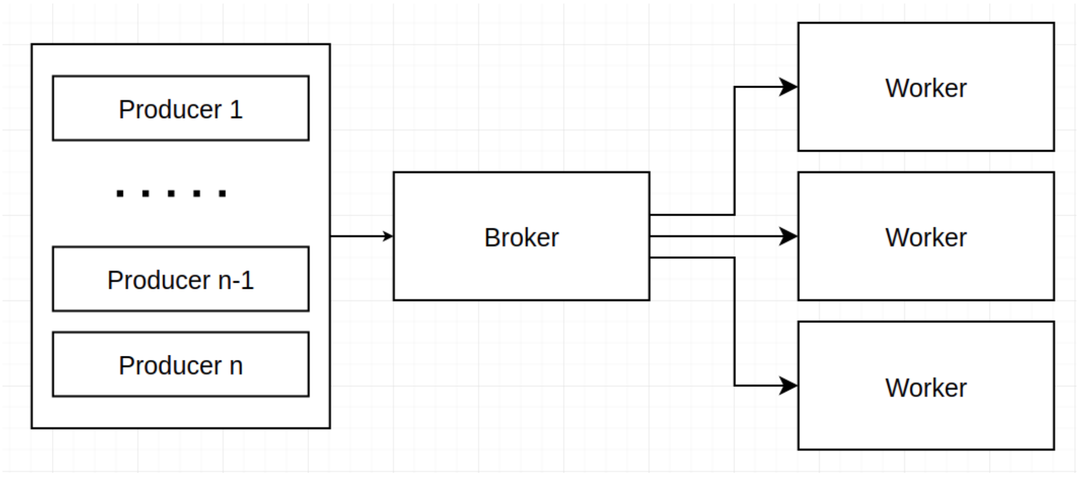


FIGURE 3.1: Solution Architecture

Producer

Producers main task is to produce request's to the Broker with some tasks for the workers. Basically that's a simple client that is throwing requests every random amount of time (In the experiment I've set it from 0 to 100ms).

Broker

Broker is a middleware between Producers and Workers with the main task to divide work gathered from Producers to workers. I've also implemented RoundRobin load balancing algorithm here to split requests among Workers, by gathering IP's of all Worker instances and iterating them, while processing requests.

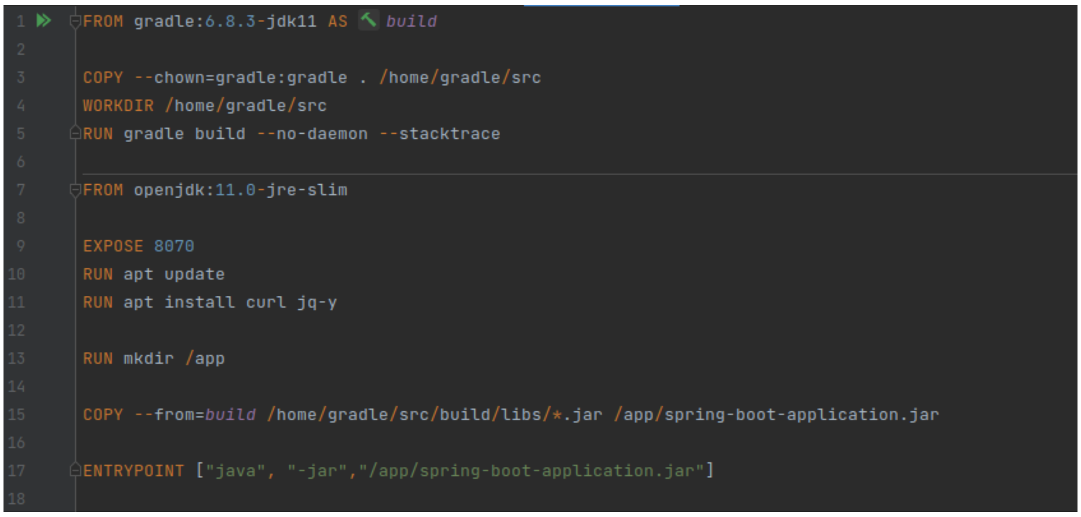
Worker

In that case broker will take some input from Brokers and will imitate CPU work of random duration from 0 to 1000ms, it will also produce logs that will give me an ability to track all tasks.

3.2.2 Selected Tools

Java

There'is a few languages that currently have an SDK, which has RSocket protocol support. So at first I've tried to implement required services in Golang, but I've realized, that RSocket library wasn't implemented there in a required way (Leasing wasn't working there), so I've selected Java to move forward with, since it has the most advance RSocket library.



```
FROM gradle:6.8.3-jdk11 AS build

COPY --chown=gradle:gradle . /home/gradle/src
WORKDIR /home/gradle/src
RUN gradle build --no-daemon --stacktrace

FROM openjdk:11.0-jre-slim

EXPOSE 8070
RUN apt update
RUN apt install curl jq-y

RUN mkdir /app

COPY --from=build /home/gradle/src/build/libs/*.jar /app/spring-boot-application.jar

ENTRYPOINT ["java", "-jar","/app/spring-boot-application.jar"]
```

FIGURE 3.2: Broker Dockerfile

Docker Kubernetes

For the deployment tool I've selected Kubernetes not only for it's ability to handle deployments and scaling, but also for its popularity in the real products. And also I've chosen Docker as an orchestration tool.

In general Kubernetes is all-inclusive PaaS (Platform as a Service) system. It operates at the container level rather than at the hardware level. (*What is Kubernetes?*)

I have setup a local Kubernetes cluster by using minikube with it's dashboard, that gives me an ability to easily debug and monitor my services container resources, while running load tests. You can see an example setup of Deployment and Service resource for Broker mircoservice in 3.2.2

FIGURE 3.3: Broker service kubernetes specification

```yaml
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: broker-deployment
  labels:
    app: broker
spec:
  replicas: 1
  selector:
    matchLabels:
      app: broker
  template:
    metadata:
      labels:
        app: broker
    spec:
      containers:
        - name: broker
          image: broker-test:latest
          ports:
            - containerPort: 8060
          imagePullPolicy: IfNotPresent
      priorityClassName: high-priority

---
apiVersion: v1
kind: Service
metadata:
  name: broker-service
  labels:
    name: broker-service
spec:
  type: NodePort
  selector:
    app: broker
  ports:
    - protocol: TCP
      port: 8080
      targetPort: 8060
```

CLI tool

To optimize my work, during running an experiment I've also created an Bash script, which allows me to build containers and start all services with a one command

Jmeter

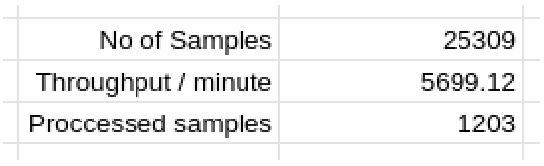
I've used Producer service, while developing services and initial tests, but for an experiment I've required more configurable tool to gather results, so for running load tests, I've used Jmeter with RSocket plugin, that provides an ability to run Fire and Forget requests to the Broker service. That replaced Produced in the figure 3.2.

For tests I've set up a 150 threads, that produced single request every random period maxim of 20ms with an ramp-up period of 300 seconds, that means that Jmeter added new thread every 2 seconds

3.2.3 Tests Results

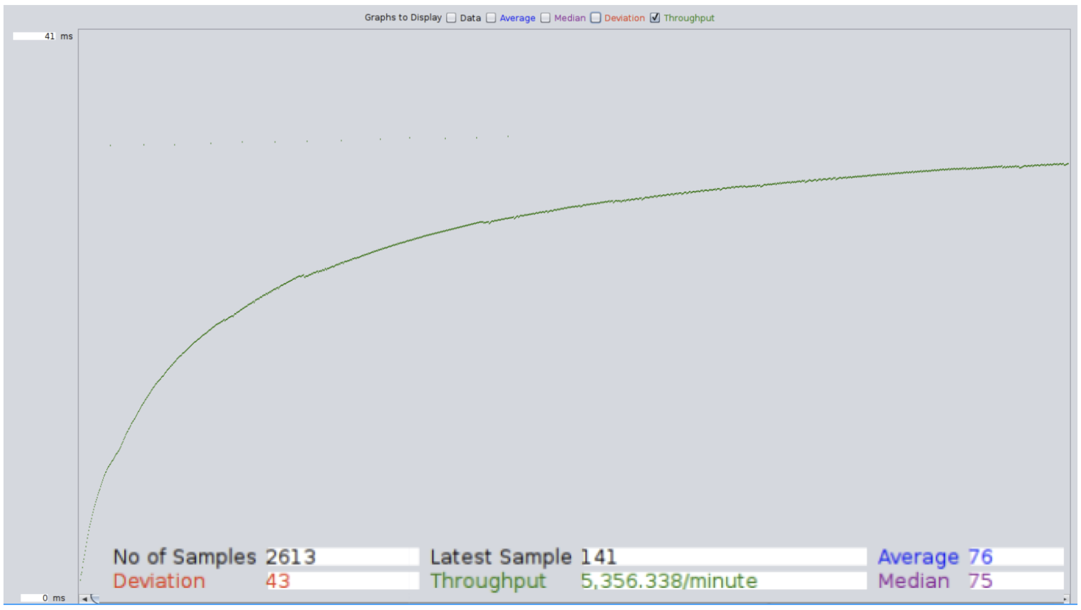
No Lease test

Here you can see a result from a load test with RSocket communication and no lease strategy applied. We see here that our throughput is growing as much as network capacities are (see 3.5), but the capacities of the pool of available workers are not enough to handle such load (see Processed samples in 3.4) delay, when result is going to be returned. That is also will lead to potential increase of memory consumption and running out of memory for larger worker services.



No of Samples	25309
Throughput / minute	5699.12
Proccessed samples	1203

FIGURE 3.4: No lease tests average



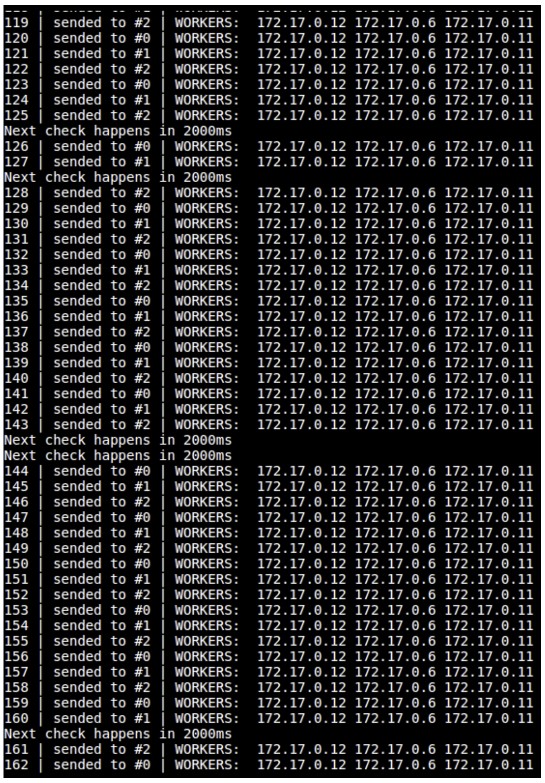
```
No of Samples 2613      Latest Sample 141                   Average 76
Deviation     43        Throughput    5,356.338/minute      Median  75
```

FIGURE 3.5: No lease test. Throughput graph

In the image 3.5, we notice that the number of messages sent by a requester rises even though the receiving party's capabilities remain same. Under actual conditions, this will produce failures and increased delay, since the responder will hold all data in memory and will be unable to process it in a timely manner.

Leasing test

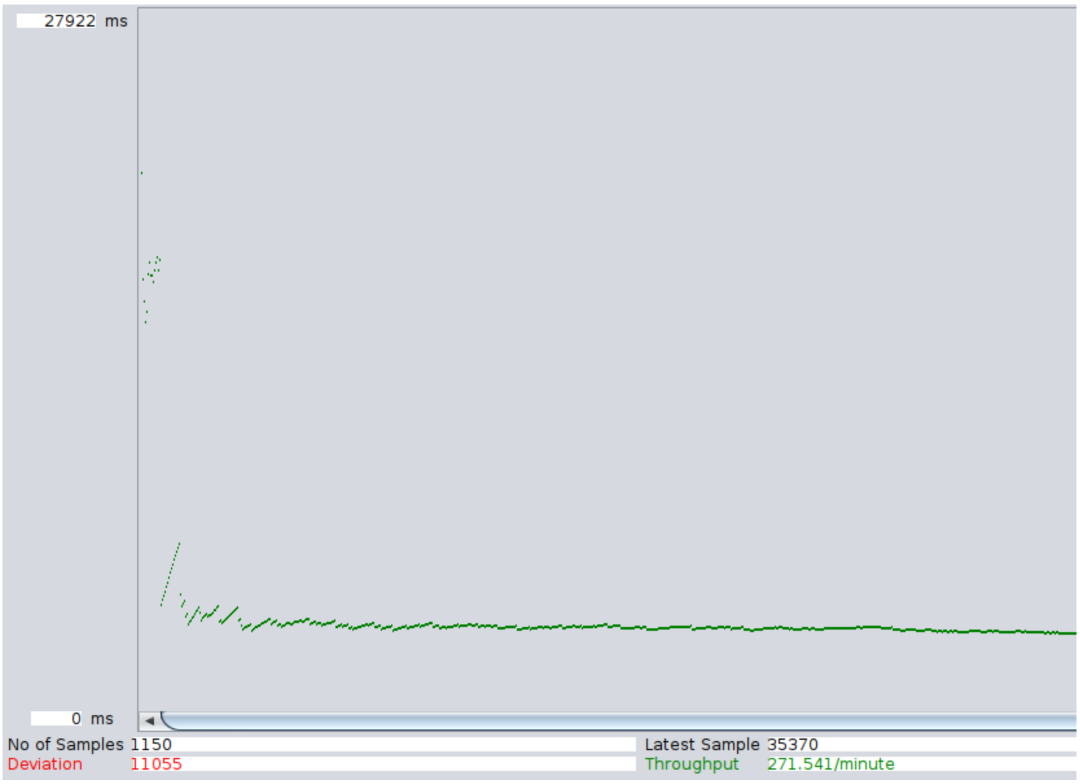
Here is the results of testing our system with Lease enabled, which is using Limit Based Leasing Strategy supported with Netflix Concurrency Limits library (see *Netflix Concurrency Limits*) for predicting Leases. At 3.6 you could see the logs from the Broker service, where it's showing up how does Leasing works in reality. Since our workers are performing a random amount of work for each of requests, which take from 0 to 1sec of CPU time, you could see that we have a different leases to issued. We also could see an empty leases, when our workers doesn't have a capacity to handle new requests in the nearest lease TTL time (2sec in our case).



```
119 | sended to #2 | WORKERS:  172.17.0.12 172.17.0.6 172.17.0.11
120 | sended to #0 | WORKERS:  172.17.0.12 172.17.0.6 172.17.0.11
121 | sended to #1 | WORKERS:  172.17.0.12 172.17.0.6 172.17.0.11
122 | sended to #2 | WORKERS:  172.17.0.12 172.17.0.6 172.17.0.11
123 | sended to #0 | WORKERS:  172.17.0.12 172.17.0.6 172.17.0.11
124 | sended to #1 | WORKERS:  172.17.0.12 172.17.0.6 172.17.0.11
125 | sended to #2 | WORKERS:  172.17.0.12 172.17.0.6 172.17.0.11
Next check happens in 2000ms
126 | sended to #0 | WORKERS:  172.17.0.12 172.17.0.6 172.17.0.11
127 | sended to #1 | WORKERS:  172.17.0.12 172.17.0.6 172.17.0.11
Next check happens in 2000ms
128 | sended to #2 | WORKERS:  172.17.0.12 172.17.0.6 172.17.0.11
129 | sended to #0 | WORKERS:  172.17.0.12 172.17.0.6 172.17.0.11
130 | sended to #1 | WORKERS:  172.17.0.12 172.17.0.6 172.17.0.11
131 | sended to #2 | WORKERS:  172.17.0.12 172.17.0.6 172.17.0.11
132 | sended to #0 | WORKERS:  172.17.0.12 172.17.0.6 172.17.0.11
133 | sended to #1 | WORKERS:  172.17.0.12 172.17.0.6 172.17.0.11
134 | sended to #2 | WORKERS:  172.17.0.12 172.17.0.6 172.17.0.11
135 | sended to #0 | WORKERS:  172.17.0.12 172.17.0.6 172.17.0.11
136 | sended to #1 | WORKERS:  172.17.0.12 172.17.0.6 172.17.0.11
137 | sended to #2 | WORKERS:  172.17.0.12 172.17.0.6 172.17.0.11
138 | sended to #0 | WORKERS:  172.17.0.12 172.17.0.6 172.17.0.11
139 | sended to #1 | WORKERS:  172.17.0.12 172.17.0.6 172.17.0.11
140 | sended to #2 | WORKERS:  172.17.0.12 172.17.0.6 172.17.0.11
141 | sended to #0 | WORKERS:  172.17.0.12 172.17.0.6 172.17.0.11
142 | sended to #1 | WORKERS:  172.17.0.12 172.17.0.6 172.17.0.11
143 | sended to #2 | WORKERS:  172.17.0.12 172.17.0.6 172.17.0.11
Next check happens in 2000ms
Next check happens in 2000ms
144 | sended to #0 | WORKERS:  172.17.0.12 172.17.0.6 172.17.0.11
145 | sended to #1 | WORKERS:  172.17.0.12 172.17.0.6 172.17.0.11
146 | sended to #2 | WORKERS:  172.17.0.12 172.17.0.6 172.17.0.11
147 | sended to #0 | WORKERS:  172.17.0.12 172.17.0.6 172.17.0.11
148 | sended to #1 | WORKERS:  172.17.0.12 172.17.0.6 172.17.0.11
149 | sended to #2 | WORKERS:  172.17.0.12 172.17.0.6 172.17.0.11
150 | sended to #0 | WORKERS:  172.17.0.12 172.17.0.6 172.17.0.11
151 | sended to #1 | WORKERS:  172.17.0.12 172.17.0.6 172.17.0.11
152 | sended to #2 | WORKERS:  172.17.0.12 172.17.0.6 172.17.0.11
153 | sended to #0 | WORKERS:  172.17.0.12 172.17.0.6 172.17.0.11
154 | sended to #1 | WORKERS:  172.17.0.12 172.17.0.6 172.17.0.11
155 | sended to #2 | WORKERS:  172.17.0.12 172.17.0.6 172.17.0.11
156 | sended to #0 | WORKERS:  172.17.0.12 172.17.0.6 172.17.0.11
157 | sended to #1 | WORKERS:  172.17.0.12 172.17.0.6 172.17.0.11
158 | sended to #2 | WORKERS:  172.17.0.12 172.17.0.6 172.17.0.11
159 | sended to #0 | WORKERS:  172.17.0.12 172.17.0.6 172.17.0.11
160 | sended to #1 | WORKERS:  172.17.0.12 172.17.0.6 172.17.0.11
Next check happens in 2000ms
161 | sended to #2 | WORKERS:  172.17.0.12 172.17.0.6 172.17.0.11
162 | sended to #0 | WORKERS:  172.17.0.12 172.17.0.6 172.17.0.11
```

FIGURE 3.6: Broker logs. Lease every 2sec

All above leads to the self-balancing of the Broker Worker services and leads to the nearby the same throughput over the time, with no matter how many clients are currently connected (see 3.7).



No of Samples 1150
Deviation 11055
Latest Sample 35370
Throughput 271.541/minute

FIGURE 3.7: Throughput graph. Lease every 2sec

Chapter 4

Conclusion

In my thesis I have find out what is Reactive Streams and why is that paradigm can be useful in building self-balancing distributed systems. I've settled up a default producer-broker-worker service set and prepared them to be deployed on kubernetes cluster in the cloud, that could be extended to some real product. I've studied the capabilities of those services when Leasing is enabled and when it is not. Additionally, I've done load testing to see how much stress my system could manage and whether or not it would be able to self-balance. And in result of the tests we see, that the servise could really be a demand-driven if we are using RSocket as a transport protocol!

The research and measurements focus exclusively on the comparison of RSocket use when the leasing protocol is activated vs when it is disabled. While this demonstrates that the protocol works, it remains an open question regarding the protocol's efficiency in comparison to the conventional client-side rate-limiting mechanism.

As far as I know, Leasing capabilities of RSocket has never been researched before and we have also had a call with a few engineers from Netflix, and I'm hoping my thesis will be the first step in developing new research, and that the results will lead to the adoption of RSocket in Netflix's production.

Bibliography

Bonér, Jonas. *Why do we need a reactive manifesto*. URL: https://www.lightbend.com/blog/why-do-we-need-a-reactive-manifesto.

contributors, HTTP/2. *Hypertext Transfer Protocol Version 2 (HTTP/2)*. URL: https://httpwg.org/specs/rfc7540.html.

contributors, Rsocket. *RSocket Connection Establishment*. URL: https://rsocket.io/about/protocol#connection-establishment.

– *RSocket Lease Frame*. URL: https://github.com/rsocket/rsocket/blob/enchancement/leasing-extension/Protocol.md#frame-lease.

– *RSocket Protocol*. URL: https://rsocket.io/about/protocol.

Foundation, Cloud Native Computing. *What is Kubernetes?* URL: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/.

Foundation, Reactive. *Reactive Streams*. URL: http://www.reactive-streams.org/.

Inc., Netflix. *Netflix Concurrency Limits*. URL: https://netflixtechblog.medium.com/performance-under-load-3e6fa9a60581t.

Jonas Bonér Dave Farley, Roland Kuhn and Martin Thompson. *Reactive Manifesto*. URL: https://www.reactivemanifesto.org/.

Mozilla and individual contributors. *Evolution of HTTP*. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/Evolution_of_HTTP.