Dynamic load balancing based on latency prediction

FEDERICO PICCININI

Master of Science Thesis
Stockholm, Sweden 2013

TRITA-ICT-EX-2013:255
Dynamic load balancing based on latency prediction

Federico Piccinini

Degree project in Information and Software systems
Stockholm, Sweden 2013

TRITA-ICT-EX-2013:255
Dynamic load balancing based on latency prediction

FEDERICO PICCININI
890614-T415

Supervisor: Javier Ubbilos, Spotify
Examiner: Johan Montelius, KTH
Abstract

Spotify is a music streaming service that offers access to a vast music catalogue; it counts more than 24 million active users in 28 different countries. Spotify’s backend is made up by a constellation of independent loosely-coupled services; each service consists of a set of replicas, running on a set of servers in multiple data centers: each request to a service needs to be routed to an appropriate replica. Balancing the load across replicas is crucial to exploit available resources in the best possible way, and to provide optimal performances to clients.

The main aim of this project is exploring the possibility of developing a load balancing algorithm that exploits request-reply latencies as its only load index. There are two reasons why latency is an appealing load index: in the first place it has a significant impact on the experience of Spotify users; in the second place, identifying a good load index in a distributed system presents significant challenges due to phenomena that might arise from the interaction of the different system components such as multi-bottlenecks. The use of latency as load index is even more attractive under this light, because it allows for a simple black box model where it is not necessary to model resource usage patterns and bottlenecks of every single service individually: modeling each system would be an impractical task, due both to the number of services and to the speed at which these services evolve.

In this work, we justify the choice of request-reply latency as a load indicator, by presenting empirical evidence that it correlates well with known reliable load index obtained through a white box approach. In order to assess the correlation between latency and a known load index obtained though a white box approach, we present measurements from the production environment and from an ad-hoc test environment.

We present the design of a novel load balancing algorithm based on a modified $\varphi$ accrual failure detector that exploits request-reply latency as an indirect measure of the load on individual backends; we analyze the algorithm in detail, providing an overview of potential pitfalls and caveats; we also provide an empirical evaluation of our algorithm, compare its performances to a pure round-robin scheduling discipline and discuss which parameters can be tuned and how they affect the overall behavior of the load balancer.
# Contents

## 1 Introduction

1.1 Background .................................................. 7
1.2 Problem description ........................................... 8
1.3 Purpose ......................................................... 10
1.4 Limitations ..................................................... 11

## 2 Background

2.1 Service architecture .......................................... 13
2.2 ØMQ ............................................................. 17
2.3 Hermes .......................................................... 20

## 3 The $\varphi$ accrual failure detector

3.1 Failure detectors ............................................... 27
3.2 The $\varphi$ accrual failure detector .......................... 30
3.3 Applications of $\varphi$ accrual failure detectors ............ 34

## 4 Dynamic load balancing using an adapted $\varphi$ accrual failure detector

4.1 Defining, measuring and distributing load .................. 38
4.2 Bottlenecks in an N-tier distributed system ............... 41
4.3 Latency as an indicator of load ............................. 46
4.4 Load balancing using an adapted $\varphi$ accrual failure detector ........................................................................... 53
4.5 Pitfalls and caveats .............................................. 65

## 5 Experimental results

5.1 Weight functions ................................................ 75
5.2 Experimental setup ............................................. 78
5.3 Test scenario 1 .................................................... 80
5.4 Test scenario 2 .................................................... 81
## CONTENTS

5.5 Test scenario 3 .................................................. 88
5.6 Test scenario 4 .................................................. 93
5.7 Test scenario 5 .................................................. 99
5.8 Conclusions ..................................................... 99

6 Future developments and conclusions .......................... 103
6.1 Failover ........................................................... 103
6.2 Tackling tail latency ............................................. 104
6.3 Conclusions ..................................................... 106

A Exponentially weighted mean and variance ...................... 109

B Latency distribution ............................................... 115
B.1 Experimental setup ............................................. 115
B.2 Chi-Square Goodness-of-Fit Test ................................ 115
B.3 Conclusions ..................................................... 118

Bibliography ......................................................... 119
Chapter 1

Introduction

1.1 Background

Spotify is a peer-assisted music streaming service that offers instant access to a vast music catalogue of over 20 million tracks. Spotify has gained worldwide popularity in the past few years. It is present in 28 countries and counts over 24 million monthly active users.

Unlike many commercial music streaming services, Spotify transfers data from both its servers and a proprietary Peer-to-Peer (P2P) network. The use of Peer-to-Peer technology reduces server workload and bandwidth requirements while improving scalability.

The Spotify backend consists of a number of individual services, each deployed over multiple servers across 4 data centers globally. Services are typically replicated in each data center, in order to improve fault tolerance, limit inter-site traffic and guarantee low latencies to end users.

While a client is running, it maintains a long-lived TCP connection to a Spotify Access Point. The Access Point acts a reverse proxy: its role is to demultiplex all traffic and distribute requests to the appropriate service replicas. Essentially Access Points act as the single point of access to all backend services. Clients are directed to access points in the closest data center by GeoDNS, which detects the geographic location of the client that issues a DNS query. In response, the client receives a pool of Access Points, and randomizes the order in which it attempts to contact them.

The Access Points and the backend services communicate using Hermes, a high-level proprietary protocol for unified interoperability between services. The Hermes library, Spotify’s implementation of the Hermes protocol, relies on Google’s protocol buffers for message serialization and on ØMQ network communication. ØMQ is an embeddable networking library based on an asyn-
chronous I/O model that provides powerful socket-like abstractions.

Most of the inter-service communication at Spotify is modeled around request-reply interactions. Each Hermes request header is characterized by a service-specific action or verb to be carried out on the requested resource. This use of the Hermes protocol closely mimics an RPC-like semantics, where each service-specific action corresponds univocally to a remote method.

In addition to request-reply interactions, the Hermes library allows for publisher-subscriber interactions. Hermes implementation, in addition to the message-passing framework itself, consist of several other components, including a message router, a message aggregator, a message broker and a message forwarder.

1.2 Problem description

Section 1.1 features a 10,000-foot overview of the backend infrastructure at Spotify. Despite the lack of details, the reader might notice that there are different levels at which routing decisions are made, with different load-balancing requirements.

If we take into consideration the typical interaction flow of a client that connects to Spotify servers to stream a song, the first routing decision is made when the client application is opened and it establishes a session with one of the Access Points. As mentioned in Section 1.1, the choice of the Access Point is based on geographic proximity and the number of active connections per access point. Hence, a DNS record with a set of the closest, least loaded Access Points is returned to the client.

Once a client establishes a session with an access point, it needs to communicate with other backends to access services such as authentication, storage of audio files and related metadata, and so on. As mentioned in the previous section a client never communicates with backend services directly: the access point it connects to is responsible for proxying the client’s requests to the appropriate replica of each service. At this stage, for each request a routing decision has to be made. This kind of routing is performed by the component called Hermes router.

The third and last routing strategy presented here is implemented in a component called Hermes Client. Hermes Client is an abstraction provided as part of the Hermes library. Hermes Client is an asynchronous high-level interface to send and receive messages to any server that supports the Hermes protocol. Hermes Client can be used to communicate with multiple endpoints, and it masks out the complexity of routing messages to a specific backend instance.
This thesis project is focused on the third load balancing strategy, which Hermes client relies upon.

**Measuring load**

The notion of load in computer science is intuitively related to some amount of work to be performed. The *New Oxford American Dictionary* defines load as *the amount of work to be done by a person or machine*. The two major problems to address when devising a load balancing algorithm are *a*) defining a unit of work, and *b*) devising a criteria to distribute work units across multiple servers.

In general, on individual servers, system performance depends on how efficiently a system’s resources are applied to the demand by various jobs in the system. The most important system resources from a performance perspective are CPU, memory, and disk and network I/O (although sometimes other device I/O could also be relevant). How well a system performs at any given moment is the result of *a*) the total demand for system resources with respect to their availability, and *b*) and how efficiently the resources are allocated to competing processes. Accordingly, performance problems on a system can arise from a number of causes, mainly both a lack of needed resources and inefficient resource allocation.

The performance of a distributed system is an even more complex topic. The same problems of resource availability and allocation extend not only to single nodes, but also to the system seen as a whole: how should processes be mapped to nodes? How should individual nodes’ resources be allocated to multiple processes? How should units of work be distributed across processes? Additional complexity arises from interactions: the dynamic performance of a distributed system depends both on the performance of the individual components that make up the system and on the cooperation between individual systems. [Dean and Barroso] present strong evidence of this phenomenon in [5], where they analyze the request-reply latency distribution for high-fan-out reads in Google backends. High-fan-out reads are particularly interesting with this respect, because the lower bound for the execution time of a read operation is limited by the slowest service instance.

[Malkowski et al.] present an approach to analysis of N-tier systems with multiple bottlenecks in [13]. In presenting their findings, they reach the conclusion that in several of the configurations taken into account during their

---

Footnotes:

1. In this context the term *load* refers loosely to the notion of work. There is no direct relation to the definition of *load* and *load average* in UNIX terms. Nonetheless load in UNIX systems (as reported by *uptime*) can be considered as a special case of load index. More on this topic is presented in Section 4.1.
study, all system components show average resource utilization — significantly below saturation, but the overall throughput remains limited despite adding more resources. In other words, there are cases of multiple bottlenecks, where the average resource utilization is low, but where several resources saturate alternatively, indicating a clear lack of independence in their utilization. Most backend systems at Spotify are essentially an N-tier systems in their own right. The Playlist backend used a system model in this work is an example of an N-tier system: there is a first tier that consists of the access points, a second tier that consists of the actual Playlist backend, and a third layer for storage in a columnar database (Cassandra).

To sum it up, modeling the load of a system by analyzing its resource usage and bottlenecks would be impractical at the scale and evolution rate of Spotify services. Spotify’s backend infrastructure is made up by a large number of individual services, that interact with each other and evolve at a fast pace. In this work we investigate a potential way to implement a load balancing strategy less naïve than a pure round-robin scheduling discipline. Our attention has been focused on using latency as a load index to build a dynamic feedback load balancing algorithm.

1.3 Purpose

As mentioned, the main aim of this work is exploring the possibility of developing a load balancing algorithm based solely on request-reply latencies as a load index. Latency has a significant impact on the experience of Spotify users, hence distributing the requests based on which replica of a service can provide shorter response times is an interesting possibility. Identifying a good load index in a distributed system presents significant challenges due to phenomena that might arise from the interaction of the different system components: we have introduced multi-bottlenecks, presented by [Malkowski et al.] in [13]. The use of latency as load index is even more attractive under this light, because it allows for a simple black box model where it is unnecessary to know resource usage patterns and bottlenecks of every single service. We pointed out that modeling each individual system manually would be an impractical task, due both to the number of individual services and to the speed at which these services evolve: every time a substantial code change is made, the behavior of a service in terms of resource usage might be affected.

In this work, we justify the choice of request-reply latency as a load indicator, by presenting empirical evidence that in a context where we have a known reliable load index, such as CPU utilization, latency is correlated to that index. In order to assess the correlation between latency and a known
1.4 Limitations

Load balancing, failover, overload protection, and QoS differentiation are closely related aspects of traffic management. Load balancing is often used to implement failover — the continuation of a service after the failure of one or more of its components. Typically this works by keeping track of unhealthy components through a failure detector. When a component is suspected to be failed, it is removed from the pool of active components. Similarly, a component that comes back online is reinstated in the pool of active components and the load balancer begins to route traffic to the component again. It is out of the scope of this master thesis to address failover, but a possible approach based on a \( \varphi \) accrual failure detector is presented in Section 6.1.

This master thesis presents experimental evidence of a statistical correlation between load and request-reply latency for a production system at Spotify. It is outside the scope of this work to demonstrate the general validity of this statement, or to delimit the scope under which this statement might be generally valid. It is worth noting, though, that some of the sources in the bibliography are in agreement with our experimental findings (cf. \[18\], \[13\]). This suggests that there is a group of systems for which latency correlates with load.
Chapter 2

Background

2.1 Service architecture

Spotify is a peer-assisted music streaming service that low-latency access to a music catalogue of over 20 million tracks. The service is present in 28 countries and counts over 24 million monthly active users. Unlike many other commercial music streaming services, Spotify transfers data from both its servers and a proprietary Peer-to-Peer network. Using P2P technology considerably increases Spotify’s scalability and reduces server workload and bandwidth requirements.

Spotify uses a proprietary protocol designed for on-demand music streaming. The protocol supports both streaming from Spotify’s data centers and peer-assisted playback, even though access to the peer-to-peer overlay is only available from desktop clients, due to power consumption and computational constraints on mobile devices. Spotify clients cache content they have recently accessed, to decrease bandwidth consumption. Around 55% of the playbacks come directly from the playing client’s local cache, and Peer-to-Peer accounts for a further 35% of the data volume [12]: cache eviction is based on a Least Recently Used (LRU) policy. Client software is available for Windows, OSX, and Linux, as well as several smartphone operating systems; a web-based player is available at play.spotify.com. Additionally, several versions of the Spotify client are integrated into set-top boxes and hardware music players, such as Sonos, Bang & Olufsen, and others.

Users can search and browse Spotify’s catalogue, and select tracks to play. Tracks can be organized into playlists, which can be edited collaboratively, and are kept synchronized across all of a user’s devices; additionally playlists support a subscription model, that allows users to follow playlists of their social contacts and be notified when the playlist they follow are modified. In
ordered to log in into the client, customers are required to own a Spotify or a Facebook account linked to their subscription, which can be free (financed via advertisement), or paid-for. Paid-for premium accounts allow users to synchronize tracks from the Spotify catalogue for offline playback.

While UDP is commonly used in streaming applications, Spotify is based on TCP. In the first place, having a reliable transport protocol simplifies the application protocol design and implementation. Furthermore, TCP's congestion control is friendly to other applications using TCP, and the explicit connection signaling helps stateful firewalls.

The Spotify backend

The Spotify backend consists of a number of individual services, each deployed over multiple servers across 4 data centers globally. Services are typically replicated in each data center, in order to improve fault tolerance, limit inter-site traffic and guarantee low latencies to end users. Figure 2.2 is a simplified depiction of the Spotify backend architecture.

Spotify services are designed around the Unix philosophy: “write programs that do one thing and do it well”; each service is a specialized network-accessible daemon. Simple, loosely coupled, multi-tiered applications are easier to scale horizontally, which allows for services to be hosted on many machines, avoiding single points of failure. Most of the services are written in Python and Java, with some C++ code for performance critical systems. Hermes, a multi-language communication framework, acts as the scaffolding that keeps all the services together. Hermes is a thin layer on top of ØMQ that supports message serialization through Google’s protocol buffers.

From click to play

On startup, a client is directed to the geographically closest data center by detecting the country it connects from and returning an appropriate set of IPs through GeoDNS. While the client is running, it keeps an open long-lived TCP connection to a Spotify server called Access Point to keep down playback latency and reduce connection setup overhead; the access point is chosen from a pool of DNS record received when initially connecting to the service. If the client detects that it was disconnected from its Access Point, it will try to reconnect. It may either detect a disconnection via the TCP socket, or by the server not responding to a heartbeat message the client sends every 2 minutes.

Users are initially prompted for their credentials, which can be stored and reused for automatic log in. Users can actively search for songs, albums, or artists; they can browse artist and album pages, or discover new music
using Discover — a personalized recommendation system based on individual listening habits and other criteria. A radio feature is also available: it produces a seemingly infinite song feed based on an initial seed, such as an artist or a specific song.

Audio streams are encoded using Vorbis with a default quality of $q_5$, which has variable nominal bitrate averaging roughly 160 kbps. Users with a premium subscription can choose to instead receive higher quality streams, encoded using Vorbis $q_9$, with an average bitrate of roughly 320 kbps. Both types of files are served both from Spotify’s own servers and the peer-to-peer network and since no re-encoding is done by the peers, to the purpose of streaming the $q_5$ and $q_9$ versions of a song are effectively two different and independent files. Additionally, the library is available in Vorbis $q_2$, with an average bitrate of 96 kbps: this is the default quality on mobile clients due to storage space and bandwidth constraints.

Figure 2.2 shows the ratio for each data source: Spotify’s own servers, peer-to-peer and local caches. We present an overview of the inner workings of each data source in the following sections, where most of the content refers to desktop clients.
CHAPTER 2. BACKGROUND

Servers

We distinguish between random access and predictable track selection: random access occurs when the user selects an unpredictable track; about 39% of the playbacks in Spotify start by random access [12]. Every time the user plays a track by random access and the track is not cached, the client contacts the Access Point requesting the first 15 seconds of the track; meanwhile it searches the peer-to-peer overlay for the remainder of the file and it switches back and forth between the servers and the peer as needed: clients model their play-out buffers as a Markov chains in order to determine whether they can handle the playback without stutter. The connection to the server is assumed to be more reliable than peer-connections, so if a client’s buffer levels are low, it requests data from the server. As long as the client’s buffers are sufficiently full and there are peers to stream from, the client only streams from the peer-to-peer overlay.

While streaming, desktop clients avoid downloading data from the server unless it is necessary to maintain playback quality or keep down latency. This doesn’t apply to the web player or mobile clients, that rely solely on servers to fetch data, due to limited resource availability. The client handles predictable playback sequences by prefetching the next track before the current track is played to completion. Predictable playback occurs when the previous track played to its end, or because the user pressed the forward button. Clients begin prefetching the next track before the currently playing track has been played to completion.

The protocol supports an “emergency mode” to compensate for connections with asymmetric connection capacity: in these situations, ACK compression can occur and degrade TCP throughput [20]; to prevent such phenomena, if the buffer becomes critically low, the client temporarily stops uploading data to its peers.

Peer-to-peer

Spotify’s peer-to-peer overlay is an unstructured network, the construction and maintenance of which is assisted by a centralized tracker. All peers that participate in the network are equals: there is no such thing as supernodes performing special functions. A client will connect to a new peer when it wishes to download a track it thinks the peer has, and this is the only mechanism by which new connections are added to the overlay. There are two ways for a client to discover peers: i) receiving a list of peers from the tracker, ii) querying its neighbors at distance one or two in the overlay network.

The tracker keeps track of 20 online peers per track and returns a list of
10 peers when it receives a request for a specific file; the use of a tracker bears a similarity to BitTorrent, but the tracker strategy in Spotify has a few substantial differences: a) there is a unified overlay for all of the tracks (instead of a per-file overlay), b) blocks are always downloaded in order, c) fairness is not enforced (unlike BitTorrent, where a tit-for-tat policy applies), d) requests are prioritize, based on the estimate of the client that issues them. Clients report to the tracker each time they play a track, rather than periodically reporting the contents of their caches, or notifying the tracker when a track is evicted the local cache. In addition to tracker-based peer searches, Spotify clients also support a Gnutella-like flooding algorithm: they query each per at distance 1 or 2, searching for tracks they need. Peers keep track of the most recent search queries they received in order to ignore duplicated message. Keeping the state required to maintain a large number of TCP connections to peers can be expensive, in particular for home routers acting as stateful NATs. Thus, the maximum number of peers is limited: there is a soft limit, set to 50 peers and a hard limit set to 60 peers: clients are not allowed to make connections when they reach the soft limit, but they can still accept incoming connections until they hit the hard limit, and they periodically prune the list of peers to keep its size below the soft limit.

Cache

Caching is important for two reasons: a) it exploits the regularity in users’ listening habits, b) cached music data can be served in the peer-to-peer overlay, reducing traffic to Spotify servers. The cache can store partial tracks, so if a client only downloads a part of a track, that part will generally be cached, but only clients that have a full track advertisement it on the peer to peer overlay. Cached content is encrypted and cannot be used by other players. The size of the The default cache size is at most 10% of free disk space (excluding the size of the cache itself), not less than 50 MB and not more than 10 GB. The cache size can be configured and files are evicted using a least recently used policy.

2.2 ØMQ

ØMQ (read ZeroMQ) is a high-performance asynchronous messaging library developed for use in scalable distributed or concurrent applications. It is based on a message queue abstraction, but unlike message-oriented middleware, a ØMQ system doesn’t need a message broker. In fact, the zero in ØMQ originally stood for “zero broker”. ØMQ is impronted to minimalism on the
CHAPTER 2. BACKGROUND

Figure 1. The weekly usage pattern of the Spotify service. Data has been normalized to a 0-1 scale.

(a) Tracks played
(b) Users connected

Figure 3. Playback latency and music stutter over a week.

Figure 2. Sources of data used by clients

IV. PERFORMANCE EVALUATION

In this section, we will present and discuss measurements indicating the performance of the Spotify system with focus on the peer-to-peer network performance. For business reasons, some data is presented as ratios rather than absolute volumes.

A. Measurement Methodology

Both Spotify clients and servers perform continuous instrumentation and monitoring of the system. Most client measurements are aggregated locally before being sent to the server. For instance, reports on connection statistics are sent every 30 minutes to the server.

The raw log messages are collected and stored on log servers and in a Hadoop cluster (an open-source map-reduce and distributed storage implementation), where they are available for processing. There is also a real-time monitoring system, based on the open-source Munin monitoring system, storing aggregated data and generating graphs based on the log messages and instrumentation of Spotify servers. Most of our graphs are based on the aggregated Munin databases, while most aggregate statistics (e.g., median playback latency) were computed from the raw log files.

In the graphs presented in this paper, min and avg gives the minimum and average (per time unit) values taken over the measurement period, and cur denotes the current value when the measurement was made. Values below 1 will be denoted with units m or u, denoting milli \(10^{-3}\) and micro \(10^{-6}\), respectively.

![Data source - ratio - by week](image)

<table>
<thead>
<tr>
<th>Data Source</th>
<th>Cur</th>
<th>Min</th>
<th>Avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>Server</td>
<td>10.86</td>
<td>6.76</td>
<td>9.62</td>
</tr>
<tr>
<td>P2P</td>
<td>33.90</td>
<td>23.78</td>
<td>33.86</td>
</tr>
<tr>
<td>Cache</td>
<td>55.24</td>
<td>48.47</td>
<td>56.53</td>
</tr>
</tbody>
</table>

Figure 2.2: Ratio of data sources used by clients [12].

grounds of reducing latency, complexity and administrative cost. Despite the lack of need for a message broker, ØMQ provides all the abstractions necessary to build one. As we shall see in Section 2.3 the Hermes library — built on top of ØMQ — provides a router device and a message broker device. ØMQ exposes a familiar socket-style API, and takes care of the concurrency aspects behind the scenes.

Why ØMQ?

When using raw TCP sockets for inter-process communication, there are a number of design decisions involved:

- **blocking versus non-blocking I/O**: should our application block, or should it handle I/O in the background? There is a trade-off between scalability and complexity involved: blocking I/O is not recommended to build scalable architectures; on the other hand it’s harder to get non-blocking I/O right. ØMQ offers non-blocking I/O support off-the-shelf: I/O is transparently managed by background threads that communicate with application threads by means of lock-free data structures;
2.2. ØMQ

- **failure model**: in a large distributed systems components come and go dynamically, either due to failures or due to capacity requirements, for example when scaling the pool of replicas of a service in order to cope with varying loads; ØMQ supports automatic retry on failure, which allows for components to start in any order (not necessarily server-first) and to come and go dynamically;

- **message representation**: ØMQ only provides a simple framing on the wire; it is totally agnostic when it comes to actual payload encoding, so the user is free to layer any serialization library on top of ØMQ in order to encode application-specific messages;

- **message delivery**: ØMQ delivers whole messages exactly as they were sent, using a simple framing on the wire. The recipient of a message doesn’t need to take care of explicitly reassembling messages on its end (unlike with plain TCP sockets).

**Protocol support and built-in core patterns**

ØMQ offers support for a variety of transport protocols, in addition to TCP: INPROC and IPC for local communication, and PGM or EPGM for multicast transport.

**INPROC** is used for local in-process (inter-thread) communication transport, messages are passed via memory directly between threads.

**IPC** is used for local inter-process communication; it transports messages between local processes using a system-dependent IPC mechanism (the current implementation is based on UNIX domain sockets).

**PGM** (Pragmatic General Multicast) is a protocol for reliable multicast transport of data over IP networks. PGM is a standard defined by RFC 3208 [16]. PGM datagrams are layered directly on top of IP datagrams.

**EPGM** (Encapsulated Pragmatic General Multicast) is an encapsulated version of PGM, where PGM datagrams are encapsulated inside UDP datagrams.

Unlike raw TCP sockets, ØMQ deals with messages, not streams. Whole messages are delivered as byte blobs using a simple framing on the wire. This means that no constraint on message format is enforced. ØMQ allows composing a message out of several frames as a ‘multipart message’. Multipart
messages can be used both for wrapping messages with address information and for simple serialization.

Several built-in core patterns are available:

**Request-reply** connects a set of clients to a set of services. This is a remote procedure call and task distribution pattern.

**Pub-sub** connects a set of publishers to a set of subscribers. This is a data distribution pattern.

**Pipeline** connects nodes in a fan-out/fan-in pattern that can have multiple steps and loops. This is a parallel task distribution and collection pattern.

**Exclusive pair** connects two sockets exclusively. This is a pattern used specifically for connecting two threads in a process.

### 2.3 Hermes

Hermes can mean two things: Hermes is the high-level protocol for unified interoperability between services that glues together the constellation of independent loosely-coupled services that make up the Spotify backend. Hermes is also Spotify’s implementation of the Hermes protocol. Bindings exist for multiple languages, including the main languages used at Spotify (Python, Java and C++), together with Go, PHP, Scala and more.

Hermes is essentially a thin layer on top of ØMQ that takes care of serializing the payload of each message as a protocol buffer. Services define their API as part of the protocol buffer message: each Hermes request header is characterized by a service-specific *action* or *verb* to be carried out on the requested resource. This use of the Hermes protocol closely mimics an RPC-like semantics, where each service-specific action corresponds univocally to a remote method. On top of that, Hermes supports publisher-subscriber interactions analogously to ØMQ. Formally, an Hermes message is a single ØMQ multi-parted message consisting of at a *URI* frame and a *preamble*, and an additional body frame that can carry additional payload. Figure 2.3 depicts the structure of a Hermes message. The address and Body frames have variable lengths.

**Request-response flow**

Most of the inter-service communication at Spotify is modeled around request-reply interactions. A typical scenario involves a client sending a request to
2.3. HERMES

a Hermes based backend service and expecting a response back. In order to accomplish this, at the very minimum the client needs to know:

- A URI that specifies which resource is requested and is used to route the message. Table 2.1 contains a few examples of URIs, from which we can infer that the general structure of a URI is:

  hermes_URI = "hm:" // service [ "/" abs_path ]

- A verb or method describing how to act on the URI. This could be for example GET, PUT or DELETE, in analogy with RESTful APIs. Note that the choice of verb names doesn’t necessarily have to mimic RESTful APIs: any arbitrary name is valid;

- optionally, the request may carry a payload. For example, a PUT request may have an associated payload that is the data to set. A payload consists of one or more ‘frames’, where each frame has arbitrary, service-specific data, treated as binary blobs.

The client fills in an Hermes message using the information described above and sends it to the appropriate server. Generally speaking, Hermes clients don’t resolve the URIs themselves, so the message typically goes through an Hermes Router that resolves the URI on behalf of the requester and forwards the message to an appropriate replica of the service. Additional layers of indirections, such as Hermes Broker, are possible: for Python services, it is common to have \( N - 1 \) (where \( N \) is the number of available cores) replicas of the service running on the same machine on different TCP ports; a Hermes Broker stands in front of the service replicas and makes them accessible through a single TCP port, taking care of forwarding and load balancing requests.

A typical backend service at Spotify has a worker implemented, which makes it easy to map the URI and verb of the incoming request to a class method. For example, in an hypothetical Foo service, a request handler might look similar to the content of Listing 2.1. A decorator binds a Hermes verb to one of the worker’s handler method, optionally parsing the URI and passing tokens over to the handler method as arguments. The reply message returned by the handler method reaches the requester going through the reverse path it traversed to reach the service.

**Publisher-subscriber flow**

Hermes supports publish/subscribe interactions, analogously to ØMQ. The publish/subscribe pattern has a somewhat similar flow to the request-reply
Figure 2.3: Fields of a Hermes message. Each block wrapped in curly braces represents a ØMQ frame; additional user-defined frame can appear after the command payload. Note that this diagram does not represent the actual wire format, but rather a high level view of ØMQ frames; byte alignment is not necessarily accurate due to variable-length non padded fields.
### 2.3. HERMES

<table>
<thead>
<tr>
<th>Service</th>
<th>URI</th>
</tr>
</thead>
<tbody>
<tr>
<td>User</td>
<td><code>hm://presence/user/&lt;user-name&gt;/</code></td>
</tr>
<tr>
<td>Playlist</td>
<td><code>hm://playlist/user/&lt;user-name&gt;/playlist/&lt;playlist-id&gt;/</code></td>
</tr>
<tr>
<td>Artist</td>
<td><code>hm://notifications/feed/artist-id%notication-type/</code></td>
</tr>
<tr>
<td>Social</td>
<td><code>hm://notifications/feed/username%notication-type/</code></td>
</tr>
</tbody>
</table>

Table 2.1: Several examples of Hermes URI and their related services.

**Listing 2.1:** Sample code of a Hermes server written in Python. Note the `handle_request` decorator that binds a class method to a matching to a `(URI, method)` pair, in an RPC-like fashion.

```python
class FooService(spotify.hermes.Worker):
    ...
    @handle_request('/user/<string:user>/', methods=['PUT'])
    def user_put(self, request, user=None):
        ...
```

The flow described in Section 2.3. Subscriptions in Hermes are based on URIs. A pub/sub client may, for example, subscribe to the URI `hm://presence/user/ foo/`; the client will subsequently receive a message when a service publishes to this URI.

We examine the publisher/subscriber interaction that occurs in real time between the user clients and backend services, depicted in Figure 2.4. To initiate a Hermes subscription, a client sends a subscription request to the publisher service (via Hermes Router). The service responds with a set of subscriptions that are intercepted by the Access Point [13]. Access Points set up the required subscription filtering and forward each subscription request to the Hermes Aggregator, transparently from the client.

When a backend service publishes, it sets up a message similar to a Reply in the context of a request-reply interaction as described in Section 2.3. After the Aggregator has completed its subscription processing, it will listen for and forward published messages to the appropriate Access Point, which will in turn forward messages to the client.

### Reliability and guarantees

Hermes itself offers no guarantees that a published message will end up to the subscribing client. For example, if the Aggregator goes down and the Access Point is subsequently restarted, published messages will be lost. In order to
provide reliable delivery guarantees, a more complex infrastructure is needed. The need for persistent delivery is driven by a) reliable delivery of publications, b) offline delivery and future retrieval of publications, and c) delivery of publication events to the same user but on different clients (i.e. desktop, mobile). Reliable persisted delivery to online clients is needed to deliver critical publications, such as a ‘like’ on an album release or a Facebook friend joining Spotify; the requirement for such publications is that the related notification should be delivered reliably, at least once across all devices. The Notification Module handles this requirement by storing the publication events. An offline client can retrieve the publication events from the Notification Module, by sending a PULL request with the timestamp of the last-seen notification.

Designing services around shallow queues

This might sound like a pretty counter-intuitive notion, but both long queues and high-concurrency are often undesirable in services that are built to handle high throughputs. Generally speaking, the intuitive idea that higher concurrency implies higher throughput is far from reality.

Anything that allows several requests to be in-flight at the same time should be carefully considered. We discuss queues since they are a common way to build a high-concurrency service, either implicitly or explicitly: for example, a naive multi-threaded network server that spawns one thread per incoming connection without any explicit queue, will nonetheless degenerate into a queue as threads compete to access resources concurrently. A related
well-known phenomenon in the field of networking is \textit{buffer bloat}: increasing buffer sizes, hence queue lengths, has the possibly counter-intuitive effect of causing higher latency and uneven throughput.

A queue does not increase throughput. It can help for short spikes but a long queue is almost as bad as very high concurrency. If a service is able to handle fewer requests than it receives, a queue only postpones the time when the overload manifests itself to its clients. Hiding an overload, or postponing the moment where this becomes evident from the client’s point of view, makes it harder and slower for them to back off or cope with the problem.

For this reasons, the backend systems at Spotify are designed around the notion that the endpoints should have shallow queues and fail fast. Fail-fast is a feature of a system with respect to its response to failures. Failing fast is key to scaling, and it’s a crucial property to handle failures gracefully: for instance, if a non-essential service is not available, it’s better to ignore its functionalities and provide a slightly degraded service than to shut down music playback altogether, which would heavily affect the user experience.
Chapter 3

The $\varphi$ accrual failure detector

In this chapter we present the failure detector abstraction. We discuss failure detector properties and illustrate some common failure detection algorithms presented in literature. After laying down concisely the theoretical foundations that support failure detection algorithms, we present the $\varphi$ accrual failure detector, introduced by Hayashibara et al. in [10]. Language conventions, terminology and formal definitions in this chapter follow Coulouris et al. [4].

3.1 Failure detectors

A failure detector is an abstraction that encapsulates timing assumptions in distributed systems. In terms of timing, we distinguish three types of distributed systems: a) asynchronous systems, b) synchronous systems, and c) partially synchronous systems.

Asynchronous distributed systems do not allow for any kind of timing assumptions about processes and links.

Synchronous distributed systems are characterized by the following properties:

1. synchronous computation: there is a know upper bound $M$ on processing delays. In other words the time taken by any process to execute a step is always less than $M$. Each step consists of a) the delivery of a message (possibly nil) sent by some other process, b) a local computation, c) and the sending of a message to some other process (possibly omitted); This model of process is captured in Figure 3.1.

2. synchronous communication: there is a known upper bound on message transmission delays;
3. synchronous physical clocks: every process is equipped with a local physical clock which deviates from a hypothetical global real-time clock at a rate limited by a known upper bound.

Partially synchronous distributed systems Most of the times, distributed systems appear to be synchronous. For most practical applications it is relatively easy to define physical time bounds that are respected most of the time. However, from time to time, such timing assumptions do not hold: a network overload, a garbage collection cycle in garbage collected languages or a variety of other phenomena can generate sudden latency spikes that cause the timing assumptions not to hold for some time. In this sense, practical systems are partially synchronous. Partial synchrony is a property of systems in which timing assumptions eventually hold. Hence, there exists a time $t_0$ such that timing assumptions will hold forever after $t_0$. Note that this definition of eventual synchrony neither means that a) there is a time after which the underlying components of the communication stacks (processes, hardware, physical links) become synchronous forever, nor b) that the system is initially asynchronous, only to become synchronous after time $t_0$. Our definition, borrowed from [9], captures the fact that — despite the system not always being synchronous — it behaves synchronously over periods long enough for algorithms to do something useful.

How to take into account some level of timing assumptions, in order to encompass synchronous and partially synchronous systems? The answer to our question is ‘failure detectors’. Failure detectors are abstractions that provide information about which processes have crashed and which are correct, and
define the conditions under which the information they provide is reliable, allowing that this information is not necessarily accurate.

We present failure detectors that encapsulate the timing assumptions of synchronous systems and failure detectors that encapsulate the timing assumptions of partially synchronous systems. Unsurprisingly, the stronger the timing assumptions, the more accurate is the information produced by the failure detector. We consider a classification of failure detectors introduced in [9] that is based both on the timing assumptions on the underlying system (synchronous systems, asynchronous systems) and the properties of the failure detector itself. We examine the properties of failure detectors in Section 3.1.

Properties of failure detectors

Failure detectors are characterized by two properties: completeness and accuracy. Completeness is related to the ability of a failure detector to detect actually crashed processes; accuracy is related to the complementary ability to not suspect correct processes. Both properties come in two variants: strong and weak, based on which level of guarantees they offer. Here is some definitions:

- **strong completeness**: eventually, every process that crashes is permanently detected by every correct process;
- **weak completeness**: every crashed node is eventually detected by some correct node;
- **strong accuracy**: no correct node is ever suspected;
- **weak accuracy**: there exists a correct node which is never suspected by any node.

More relaxed requirement for accuracy exist: a failure detector is eventually strongly accurate if it provides strong accuracy after some finite time; a failure detector is eventually weakly accurate if it provides weak accuracy after some finite time.

Implementation

Using heartbeat messages is a common approach to implementing failure detectors. Let’s assume that processes have access to a local physical clock they can use to measure time. Clocks need to be correct, but not necessarily synchronized. A correct clock is characterized by a) monotonicity, the property
that time always moves forward, and b) low drift, i.e. the speed at which the clock drifts apart from a reference time source should be negligible.

The basic idea in heartbeat-based failure detectors is that the monitored process periodically sends a heartbeat message to monitor $q$, informing process $q$ that $p$ is still alive. The period is called the heartbeat interval $\Delta_i$. Process $q$ suspects $p$ if it doesn’t receive any heartbeat message from $p$ before a timeout value $\Delta_{to}$, with $\Delta_{to} \geq \Delta_i$.

An alternative implementation of failure detectors is based on a request-reply interaction rather than periodic heartbeats. Algorithm 3.1 is presented in [9] and shows how a failure detector can be built out of a request-reply message exchange: the monitor process sends periodic heartbeat requests to the monitored processes and expects a respect a reply within a given timeout. The initial timeout value $\Delta_{to}$ must be greater than the round trip time $\Delta_{rt}$. If the monitor doesn’t receive a reply before the timeout expires, it marks the non-responding processes as suspected and notifies the upper layer. Each time one or more suspicions turn out to be wrong (because the monitor receives a reply from one or more suspects), the timeout is increased by $\Delta_{to}$ and the upper layer is notified that the process in question is no longer a suspect.

Traditional $\diamond P$ failure detectors, such as Algorithm 3.1, output a boolean information: suspected or not-suspected. One of the major drawbacks of this model is that a single failure detector doesn’t support scenarios where different applications have multiple quality-of-service requirements to be handled separately. Furthermore, some classes of distributed applications require different levels of confidence in the suspicion that a service has failed to trigger different reactions. For instance, an application that supports graceful degradation might take some precautionary measures when the confidence in a suspicion reaches a certain level, and take more drastic measures when the level of confidence reaches a higher threshold.

### 3.2 The $\varphi$ accrual failure detector

Accrual failure detectors are a different class of failure detectors, in which a monitor service outputs a value on a continuous scale rather than boolean information. The higher the value, the higher the chance that the monitored process has crashed: roughly speaking the output of an accrual failure detector captures the degree of confidence that a process has crashed. If an actual crash occurs, the output is guaranteed to accrue over time and tend towards infinity, hence the name.

This model allows for a greater degree of flexibility, since it is left to the application processes to decide an appropriate suspicion threshold according
3.2. THE $\varphi$ ACCRUAL FAILURE DETECTOR

Algorithm 3.1: Increasing Timeout.

**Implements:**
EventuallyPerfectFailureDetector, instance $\diamond P$.

**Uses:**
PerfectPointToPointLinks, instance $pl$.

**Upon event** $\langle \diamond P, \text{Init} \rangle$ **do**
\begin{itemize}
  \item \texttt{alive} := $\Pi$;
  \item \texttt{suspected} := $\emptyset$;
  \item \texttt{delay} := $\Delta_{to}$;
  \item \texttt{starttimer}(\texttt{delay});
\end{itemize}

**Upon event** $\langle \text{Timeout} \rangle$ **do**
\begin{itemize}
  \item if $\texttt{alive} \cap \texttt{suspected} \neq \emptyset$ then
    \begin{itemize}
      \item \texttt{delay} := \texttt{delay} + $\Delta$;
      \item foreach $p \in \Pi$ do
        \begin{itemize}
          \item if $(p \notin \texttt{alive}) \wedge (p \notin \texttt{suspected})$ then
            \begin{itemize}
              \item \texttt{suspected} := \texttt{suspected} $\cup \{p\}$;
              \item \texttt{trigger} $\langle \diamond P, \text{Suspect} \mid p \rangle$;
            \end{itemize}
          \item else if $(p \in \texttt{alive}) \wedge (p \in \texttt{suspected})$ then
            \begin{itemize}
              \item \texttt{suspected} := \texttt{suspected} $\setminus \{p\}$;
              \item \texttt{trigger} $\langle \diamond P, \text{Restore} \mid p \rangle$;
            \end{itemize}
        \end{itemize}
        \item \texttt{trigger} $\langle pl, \text{Send} \mid p, [\text{HeartbeatRequest}] \rangle$;
        \item $\texttt{alive} := \emptyset$;
        \item \texttt{starttimer}(\texttt{delay});
    \end{itemize}
\end{itemize}

**Upon event** $\langle pl, \text{Deliver} \mid q, [\text{HeartbeatRequest}] \rangle$ **do**
\begin{itemize}
  \item \texttt{trigger} $\langle pl, \text{Send} \mid q, [\text{HeartbeatReply}] \rangle$;
\end{itemize}

**Upon event** $\langle pl, \text{Deliver} \mid p, [\text{HeartbeatReply}] \rangle$ **do**
\begin{itemize}
  \item $\texttt{alive} := \texttt{alive} \cup \{p\}$;
\end{itemize}
to their own quality-of-service requirements. There is a tradeoff in tuning the
threshold, whereby a low threshold ensures a quick detection in the event of
a real crash at the price of increasing the likeliness of wrong suspicions. Con-
versely, a high threshold is less mistake-prone, but makes the failure detector
slower to detect actual crashes. Note that accrual failure detectors can be
used to implement failure detectors of class $\diamond P$.

Hayashibara et al. present the $\varphi$ accrual failure detector in [10]. The
principle is that the arrival time of heartbeats is sampled and maintained in a
sliding window to estimate a statistical distribution of the inter-arrival times
of heartbeats. The distribution of past samples is used as an approximation
for the probabilistic distribution of future heartbeat messages. With this
information, it is possible to compute a value $\varphi$ with a scale that changes
dynamically to match recent network conditions. $\varphi$ is defined as follows:

$$
\varphi \overset{\text{def}}{=} -\log_{10}(P_{\text{later}}(t_{\text{now}} - T_{\text{last}}))
$$

where $P_{\text{later}}$ is defined as:

$$
P_{\text{later}}(t) = \frac{1}{\sigma \sqrt{2\pi}} \int_{t}^{+\infty} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \, dx
$$

$$
= 1 - F(t)
$$

$F(t)$ represents the cumulative distribution function of a normal distribution
with mean $\mu$ and variance $\sigma^2$. Let $\Phi(x)$ be the cumulative distribution function
of the standard normal distribution $\mathcal{N}(0, 1)$:

$$
\Phi(t) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{t} e^{-\frac{x^2}{2}} \, dx
$$

hence,

$$
F(t) = \Phi\left(\frac{t - \mu}{\sigma}\right)
$$

Intuitively, computing the value $P_{\text{later}}(t_{\text{now}} - T_{\text{last}})$ is equivalent to answ-
ering the question: “given the distribution of previous inter-arrival times, and
given that the last heartbeat arrived at time $T_{\text{last}}$, what is the probability of
the next heartbeat arriving in the future, considering that $t_{\text{now}} - T_{\text{last}}$ has
already elapsed?”.

Conceptually, $\varphi$ takes the following meaning: given a threshold value $\Phi$,
and assuming that we decide to suspect $p$ when $\varphi \geq \Phi = 1$, then the likeliness
that we will make a mistake (i.e., the decision will be contradicted in the
future by the reception of a late heartbeat) is about 10%. The likeliness is about 1% with \( \Phi = 2 \), 0.1% with \( \Phi = 3 \), and so on. Figure 3.2 represents the information flow described in the paper.

![Diagram](image)

**Figure 3.2:** Information flow in the original implementation of the \( \varphi \) failure detector [10].

### Extending the \( \varphi \) accrual failure detector

A variant of the algorithm presented in [10] based on request-reply interactions is possible. Instead of waiting for heartbeats arriving at regular intervals, the monitor can send heartbeat requests, analogously to Algorithm 3.1 and tag them with the current timestamp reported by its physical clock; the monitored process replies with a message tagged with the same timestamp as the request. Instead of building a distribution based on inter-arrival times, the monitoring process computes the difference between the timestamp of the request and the current timestamp reported by its physical clock; the monitored process can send heartbeat requests, analogously to Algorithm 3.1 and tag the time when an heartbeat was last received, we consider the difference between the current time and the time when an heartbeat request was last sent.

This variant of the \( \varphi \) accrual failure detector is used as a foundation for the load-balancing algorithm that we introduce in Chapter 4.
3.3 Applications of $\varphi$ accrual failure detectors

In this section we present two different applications of $\varphi$ accrual failure detectors seen in Cassandra. Apache Cassandra is an open source distributed database management system designed to handle large amounts of data across many commodity servers, providing tunable consistency and high availability. Cassandra’s replication model is based on Amazon Dynamo [6] and its data model is based on Google Bigtable [2].

Failure detection

Cassandra determines if a node in the system is up or down from gossip state. Detecting failed nodes is important to avoid routing requests to unreachable nodes whenever possible.

Failure detection is based on a gossip variant of the $\varphi$ accrual failure detector. The gossip process tracks heartbeats from other nodes both directly (other nodes send their heartbeats to the monitor) and indirectly (nodes heard about secondhand, thirdhand, and so on). During gossip exchanges, every node maintains a sliding window of inter-arrival times of gossip messages from other nodes in the cluster, analogously to the original implementation presented in [10]. The value of $\phi$ is based on the distribution of heartbeat inter-arrival times across all nodes in the cluster. A threshold $\texttt{phi\_convict\_threshold}$ defines after which value of $\varphi$ a node is suspected, analogously to the threshold $\Phi$ described in Section 3.2.

Dynamic snitching

In Cassandra replicas of each row are stored on multiple nodes in the cluster, according to a tunable replication factor. To determine the physical location of nodes and their proximity to each other, the replication strategy also relies on a component called snitch. A snitch maps nodes’ IPs to racks and data centers. It defines how the nodes are grouped together within the overall network topology. Cassandra uses this information to route inter-node requests as efficiently as possible. For example, if a cluster is deployed over multiple data centers, it might be desirable to distribute the replicas in order to a) satisfy reads locally, without incurring cross-datacenter latency, and b) tolerate failure scenarios.

Snitches can be based on explicit topology files, or infer the topology based
on a range of IP addresses. For example, given an IP address such as

\[ 110. \text{a}. 200. \text{b}. 105 \] (3.7)

each octet could be interpreted as the data center octet, \( b \) as the rack octet and \( c \) as the node octet.

By default, all snitches in Cassandra are augmented with a dynamic snitch layer that monitors read latency and, when possible, routes requests away from poorly-performing nodes [17]. This comes into play for read operation: when doing reads, Cassandra only asks one node for the actual data, and it asks remaining replicas for checksum depending on the consistency level. Choosing the fastest replica for requesting actual data has a significant impact over latency, since it is not possible to reconstruct a reply out of the checksums received by additional replicas.

The dynamic snitch layer builds up a statistically significant, exponentially decaying random sample of request-reply latencies. The snitch normalizes the vector of latencies and the vector of times elapsed since last reply; each vector contains a measurement for each host. A badness score is computed for each host as the sum of three components: \( a \) its median request-reply latency, \( b \) the time elapsed since its last reply (the longer the time, the higher the penalization), \( c \) the severity of the host. Severity is a measurement of the compaction activity relative to the host’s own load and the size of the task causing the severity; compactions can often penalize reads. The badness scores are recomputed periodically, and nodes are ordered by decreasing badness. Each time a node forwards a read request for actual data, it chooses the ‘least bad’ replica.

\[ ^1 \text{The online sampling algorithm used by the dynamic snitch is based on [3].} \]
Module 3.1: Interface of the eventually perfect failure detector.

Module:
  Name: EventuallyPerfectFailureDetector, instance \( \diamond P \).

Events:
  \( \langle \diamond P, \text{Suspect} \mid p \rangle \): Notifies that process \( p \) is suspected to have crashed.
  \( \langle \diamond P, \text{Restore} \mid m \rangle \): Notifies that process \( p \) is not suspected anymore.

Properties:
  \( \text{EPFD1: Strong completeness} \): Eventually, every process that crashes is permanently suspected by every correct process.
  \( \text{EPFD2: Eventual strong accuracy} \): Eventually, no correct process is suspected by any correct process.
Chapter 4

Dynamic load balancing using an adapted \( \varphi \) accrual failure detector

In Section 4.1 we discuss the importance of the choice of a good load index. We present the conclusions reached by Ferrari and Zhou in [7], where a comparison of different load indices in a UNIX system is conducted. In particular we will refer to the well known UNIX load index `loadavg`.

In Section 4.2 we show that identifying a good load index in a distributed system presents significant challenges due to phenomena that might arise from the interaction of the different system components. We adopt the classification schema of multi-bottlenecks proposed by Malkowski et al in [13]. The difficulty of detecting distributed bottlenecks arises from the impossibility of evaluating with certainty a non-stable predicate from the observation of a consistent global state in an asynchronous distributed system [4, p. 620]. Additionally we present the occurrence of a concurrent bottleneck in Spotify’s backend systems that resulted in a user-facing outage that affected European datacenters on April 27, 2013.

In Section 4.3 we justify the choice of request-reply latency as a load indicator. We present the challenges involved and an overview of the trade-offs. We present empirical evidence that in a context where we have a known reliable load index, such as CPU utilization, latency is correlated to that index. Experimental findings presented in [13] and the theoretical study conducted by Yanggratoke et al in [18] appear to validate our experimental results.

In Section 4.4 we present the motivations and the design goals for a novel load balancing algorithm based on a modified \( \varphi \) accrual failure detector. We analyze the algorithm in detail, providing an overview of potential pitfalls and caveats.
4.1 Defining, measuring and distributing load

The *New Oxford American Dictionary* defines load as the amount of work to be done by a person or machine. It is a rather non-trivial task to move from this abstract definition to a measurable quantity. In Section 1.1 we make the case that devising a load balancing algorithm requires solving two problems: a) defining a criterion for measuring the amount of work being performed by a system, b) devising a criterion to distribute units of work across multiple servers.

A unit of work is a self-contained request can be satisfied with a given amount of work and that can’t be split further. For example, a unit of work might be: a) a layer 2 or layer 3 datagram in the context of routing traffic in a network; b) a TCP connection in the context of routing HTTP traffic to a web server; c) individual HTTP requests.

Table 4.1 classifies a number of load balancing strategies found in [21] and in the *mod_proxy_balancer* Apache HTTP server module using the two criteria expressed above: unit of measurement of workload and type of unit of work.

From the perspective of devising a criteria to distribute work units across multiple servers, load balancing and job scheduling are analogous problems. In the following section we will present a well known definition of load, the UNIX system load value. In [7], *Ferrari and Zhou* present an empirical study about the quality of different load indices in the context of dynamic load balancing (scheduling). They underline a correlation between the efficiency...
of load balancing and the choice of the load indices, given the same load balancing algorithm.

In Section 4.1 we refer to Ferrari and Zhou’s paper to present some of the challenges posed by load balancing in a non-distributed environment. Section 4.2 deals more specifically with aspects related to dynamic load balancing in a distributed environment and is largely based on the work presented by Malkowski et al. in [13].

**Load in UNIX systems**

In UNIX systems, the system load is a measure of the amount of computational work that a computer system performs.

The instantaneous load on Linux systems is computed as the sum of the processes in `TASK_RUNNING` and `TASK_UNINTERRUPTIBLE` states. The `TASK_RUNNING` state qualifies processes that are either executing on a CPU or waiting to be executed. The `TASK_UNINTERRUPTIBLE` state qualifies processes that are suspended (sleeping) until some condition becomes true. Delivering a signal to a process in `TASK_UNINTERRUPTIBLE` state leave the state of the process unchanged.

The values reported as load average by `uptime` or in `/proc/loadavg` are computed as follows:

- the instantaneous value of load is the sum of all processes in `TASK_RUNNING` and `TASK_UNINTERRUPTIBLE` states;
- the instantaneous value of load is sampled periodically, every 5 seconds (as of Linux kernel 3.11);
- the sampled values are averaged using exponentially damped averages with time constants 1 minute, 5 minutes and 15 minutes. These are the three values reported as `loadavg`.

Ferrari and Zhou study the behaviour of the same load balancing algorithm using different load indexes, namely: 

1. CPU queue length;
2. linear combination of CPU, I/O and memory queue lengths;
3. CPU utilization;
4. load average.

They provide an empirical comparison of the effectiveness of each load balancing index. According to their results, the use of load average and other resource queue lengths provides better performances than indexes based on resource utilization. A possible explanation for this behavior is that when a host is heavily loaded, the CPU utilization tends to saturate rapidly, thus making a bad indicator for the actual load level. Two heavily loaded hosts

---

1. Linux v3.11. See definition of `LOAD_FREQ` in `include/linux/sched.h`
with that present 100% CPU utilization but different values of load average should be considered differently for the purpose of load balancing, according to Ferrari and Zhou’s results.

The performance of a system depends both on the availability of resources versus the demand for resources, and on how efficiently the available resources are allocated to the competing. The most important system resources from a performance perspective are CPU, memory, and disk and network I/O. Load indexes based on resource queue lengths seem to provide better performances than indexes based on resource utilization, since queue lengths are still meaningful even when resource utilization metrics saturate.

Past the point of saturation of a metric, we don’t have any information to compare the state of two systems. As a rule of thumb metrics with a wide dynamic range should be preferred over metrics with a narrower dynamic range because they are able to provide more information close to the point of saturation.

Resource availability is not necessarily what limits the throughput of a system. Bottleneck analysis provides a better picture of the phenomena that affect a system’s performances. A bottleneck lies on a system’s critical path and provides the lowest throughput. Thrashing in a multiprogramming environment, and lock contention in the context of concurrent access to a shared resource are examples of phenomena where the throughput of a system is limited in spite of the availability of resources.

Figure 4.1 shows a quantitative representation of thrashing. The degree of multiprogramming represents the amount of processes (or threads) that a system can accommodate. Past a critical degree of multiprogramming the throughput drops abruptly, due to the overhead imposed by context switching. This happens in spite of a larger availability of CPU capacity. If the operating system scheduler uses CPU utilization as a load index, a positive-feedback effect occurs: the scheduler will try to increase the degree of multiprogramming since it perceives a significant CPU under-utilization; this, in turn, will decrease the CPU utilization even further by adding extra overhead due to additional context switching. Taking load average into account would mitigate this effect, since load average actually is the sum of the number of tasks that are either allocated to a CPU, or waiting on the running queue, together with those in state TASK_UNINTERRUPTIBLE.

In Section 4.2 we will present an overview of bottleneck analysis in an N-tier distributed system, such as the Spotify backend. The next Section is largely based on the work of Malkowski et al.
4.2 Bottlenecks in an N-tier distributed system

N-tier applications are characterized by non-stationary workloads (e.g. peak load several times the average operating load) and complex dependencies among the components. Spotify’s backend system architecture as a whole can be considered as a complex N-tier system. Often individual subsystems can be seen as N-tier distributed systems themselves.

In this Section we refer to the work of Malkowski et al. that present an experimental evaluation and classification of bottlenecks in N-tier distributed systems [13].

In traditional bottleneck analysis single-bottlenecks are considered predominant. In classical queueing theory, the set of bottlenecks in a system is defined as:

\[
B = \left\{ i \mid \lim_{N \to \infty} U_i(N) = 1 \right\}
\]

(4.1)

where \( B \) is the set of resources \( i \) that reach full utilization, when the number of jobs \( N \) tends to infinity under stable conditions. This definition assumes independent utilization of resources.

Single bottlenecks are characterized by an average resource utilization that grows near-linearly with load, until a point of saturation is reached past a certain workload. This phenomenon can be recognized in Figure 4.1.

Malkowski et al. show that the characteristic of bottlenecks changes significantly for N-tier systems with heterogeneous workloads. In [13] they present a classification of multi-bottlenecks that is based on: a) resource saturation frequency (i.e. resources that saturate for a part of the observation...
CHAPTER 4. DYNAMIC LOAD BALANCING USING AN ADAPTED $\phi$

ACCRUAL FAILURE DETECTOR

period vs. resources that saturate for the whole observation period), and b) resource usage dependence (i.e. the correlation between the saturation of any two resources).

More formally, accepting the definitions given in [13]:

**Critical saturation** A system resource $i \in I$ (where $I$ is a set of resources) is critically saturated for an interval $t$ of length $\lambda > 0$ if its utilization exceeds a saturation threshold $\alpha \in [0, 1]$.

**Fully saturated resource** A system resource is said to be fully saturated if it is critically saturated for a whole observation period $\Lambda$.

**Partially saturated resource** A system resource is said to be partially saturated if it is saturated for $N$ observation periods $\lambda_j$ such that $f_i = \frac{1}{\Lambda} \sum_{j=1}^{N} \lambda_j$; $f_i$ is the saturation frequency of resource $i$.

**Resource usage dependence** A bottleneck resource $i_k \in I$ is said to be resource usage dependent, if there exists another bottleneck resource $i_l \in I$ such that their critical saturation states have a mutually exclusive relationship for any given interval $t$. Note that the saturation state $S_i(t)$ of any resource $i$ at time $t$ is a binary property: $S_i(t) \in \{0, 1\}$, $\forall t \in \Lambda$. This means that $P(S_k(t) = 1 \land S_l(t) = 1) \leq \varepsilon$, with $0 < \varepsilon \ll 1$.

**Resource usage independence** A bottleneck resource $i_k \in I$ is said to be resource usage independent if there exists no bottleneck resource $i_l \in I$ such that their critical saturation states have a mutually exclusive relationship for any given interval $t$. This means that $P(S_k(t) = 1 \land S_l(t) = 1) \geq \varepsilon$, with $0 < \varepsilon \ll 1$. Unlike dependent bottleneck resources, independent bottleneck resources show a clear increase in the overlap of critical saturation intervals with growing workload; e.g., $P$(overlap$) > 5\%$.

Malkowski et al. propose a classification of bottlenecks based on resource saturation frequency and resource usage dependence. They distinguish between three different types of multi-bottlenecks.

**Concurrent bottlenecks** happen if multiple resources saturate independently of each other, for a fraction of the observation period: $f_i < 1$, $\forall i$.

**Oscillatory bottlenecks** consist of multiple resources forming a combined bottleneck. In an oscillatory bottleneck resource usage dependencies cause multiple resources to saturate alternatively (i.e. there is no overlapped saturation).
4.2. **BOTTLENECKS IN AN N-TIER DISTRIBUTED SYSTEM**

**Simultaneous bottlenecks** consist of multiple independent resources that saturate together. According experimental results in [13] if resources exhibit full saturation, their utilization is not dependent on any other saturated resource (in other words, multiple fully saturated resources implies resource usage independence).

A common assumption in computer system performance analysis is that low average resource utilization implies absence of saturation. This not true in the case of oscillatory bottlenecks, where multiple resources form a combined bottleneck which can only be analyzed in union. When an oscillatory bottleneck occurs, system components show average resource utilization significantly below saturation, but the overall throughput is limited despite addition of more resources, due due alternating resource saturation.

![Classification of bottlenecks according to Malkowski et al.](image)

**The Popcount incident**

On April 27, 2013 a concurrent bottleneck that involved two different services manifested itself in one of the biggest incidents in Spotify’s history.

For a few hours on a Saturday night a significant number of European users experienced very high latencies on music playback and some were unable to login. Saturday night is usually a very busy night for Spotify’s servers.

Popcount is a service that takes care of storing the list of subscribers for each of the more than 1 billion playlists that Spotify stores for its users. All the backend services and clients that communicate with Popcount are designed to fail fast.

Fail-fast is a feature of a system with respect to its response to failures. Fail-fast systems are designed to stop normal operation rather than attempt
to continue a possibly flawed process. This is a common design feature of most of Spotify’s systems. Failing fast is key to scaling. It’s a crucial property to handle failures gracefully: for instance, if the Popcount service is not available, it’s better to discard the list of playlist subscribers altogether than to delay showing playlist data, which would heavily affect the user experience.

In a previous incident with the Popcount backend that occurred two months before the outage, a legacy piece of code in the desktop client implemented neither fail-fast logic, nor exponential back-off, which caused the desktop client to retry Popcount requests as soon as one timed out. This put a lot of pressure on the Popcount service, by leading to a positive-feedback effect that caused more and more requests to time out due to full request queues and the service being unhealthy; in turn those requests were retried, which put even more pressure on the service and caused more failed requests. As a side note, despite reaching CPU saturation on the Popcount servers, the Access Points where perfectly healthy. According to the definitions given in Section 4.2 CPU on Popcount servers and disk I/O bandwidth on Access Points are usage independent.

On the night of April 27, 2013, the Popcount service started misbehaving again. The reason for Popcount’s unhappiness was a newly introduced dependency between the Discovery service (see Figure 4.4) and Popcount. The Discovery page was fetching data from Popcount in order to show playlist counts. In this case the fast-fail logic introduced in response to the previous accident did not help, since the service was plainly under-provisioned.

By the time Popcount machines were firewalled off — as a mitigation — the Access Points had built up an excessive amount of logging, which saturated their I/O resources. In this case resource saturation on Pop Count machines cause in turn I/O saturation on Access Point machines, making the service unhealthy as a whole.

This little summary of the Popcount incident is aimed at showing how tricky multi-bottlenecks are to identify and how the interaction between services can be a source of bottlenecks that those same services would not exhibit individually. Figure 4.3 shows the latency distribution of Popcount requests-replies as seen by Access Points. Notice the progressive degradation of the service: before week 19 the predominant latency classes are $\Delta t_{RR} < 5$ ms and $5$ ms $\leq \Delta t_{RR} < 10$ ms (red and green distributions), where $\Delta t_{RR}$ is the request-reply latency. Note the deterioration after week 19: the lower latency classes are barely visible in the stacked distribution, which means that there is an overall latency increase. On week 20 a big red spike is visible, due to requests being fast-failed when the service is overwhelmed by incoming requests.
4.2. BOTTLENECKS IN AN N-TIER DISTRIBUTED SYSTEM

Figure 4.3: Popcount request delay. Distribution by latency class.

Figure 4.4: A screenshot of the Discovery page on the web client, available at play.spotify.com.
CHAPTER 4. DYNAMIC LOAD BALANCING USING AN ADAPTED ACCRUAL FAILURE DETECTOR

4.3 Latency as an indicator of load

As previously discussed, finding a meaningful load indicator is a key to distribute the work successfully across a set of servers. Due to the presence of multi-bottlenecks in N-tier systems, identifying the critical paths can be tricky.

In this section we will discuss the possibility to use request-reply latency as a load index. In [18], Yanggratoke et al. model and evaluate the performance of the Spotify storage system, by introducing an analytical model based on queueing theory. The research is focused specifically on the Storage backend, where access to memory and storage is the only potential bottleneck, and CPUs and network bandwidth are dimensioned to never reach saturation before I/O resources saturate. Yanggratoke et al. reach two conclusions, among others, that are relevant in the context of our dissertation:

- they propose an analytical model of the Spotify storage architecture that allows to estimate the distribution of the response time of the storage system as a function of the load, thus implying that $\Delta t_{RR} = f(\text{load})$, at least when a number of preconditions are satisfied;
- they provide an experimental evaluation that shows the model predictions are within 11% of the measurements, for all system configurations and load patterns within the range of the confidence range of the model (in other words, when the system load is reasonably lower than the saturation point).

Evidence of a relationship between system load and response time $\Delta t_{RR}$ also emerges from [13], though this aspect is not explicitly analyzed in the paper. For this purpose, cf. Figure 4.5a and Figure 4.5b, both produced from data reported in [13]. Also in this case, it appears that the correlation holds when the system’s resources are not saturated.

In order to produce other meaningful indicators regarding the hypothesis that there is a relationship between load and request-reply latency, we have conducted two different experiments. Both experiments are based on a white-box approach, where load is modeled with CPU usage, and matched against request-reply latency. Section 4.3 describes the experimental setup and the results of running the experiment in an ad-hoc test environment. Section 4.3 describes the methodology used for measuring request-reply latency in the production environment, and the empirical conclusions that can be drawn from the measurements.

In particular, quoting [18]: “we restrict the applicability of the model to systems with small queues — they contain at most one element on average. In other words: our model is accurate for a lightly loaded storage system.”
Correlation in the test environment

We have adapted a white-box approach to compute the correlation between a known load index (in our case, CPU usage) and a request-reply latency. The first instance of the experiment was run in an ad-hoc test environment, consisting of 2 virtual machines and a pool of machines used for running a test Cassandra cluster.

Experimental setup

Figure 4.6 shows the experimental setup used for investigating the load-latency correlation in the test environment. The test environment is made up by two separate virtual machines, hosted in the same datacenter: a load generator runs on one, and a test instance of the Playlist backend runs on the other. The test instance of the Playlist backend is supported by a small shared Cassandra test cluster made up by three machines.

Load generator The load is represented by a uniform query mix that is generated out of the set of supported methods and playlist types. Let $M$ be the set of supported methods $M = \{\text{HEAD, GET, DIFF}\}$; let $L$ be the set of supported playlist types $L = \{l_0, l_1, l_2, \ldots, l_n\}$. The query mix is the cartesian product of $M$ and $L$, $M \times L = \{(m, l) \mid m \in M \land l \in L\}$. Each pair $(m, l)$ is generated with equal probability $P(m, l) = \frac{1}{|M| \cdot |L|}$. The load generator supports rate limiting, which means that the query rate can optionally be set programmatically.
Figure 4.6: Experimental setup used to assess load-latency correlation in the test environment.

**Playlist backend** The Playlist backend is a production-like instance of the Playlist service that runs on a virtual machine in the test environment. It is supported by the test Cassandra cluster.

**Cassandra test cluster** A small Cassandra cluster that runs on a set of three virtual machines in the testing playground. This is a shared service, meaning that potentially other testing instances of the Playlist service could access the cluster.

The load test consists in setting a fixed query rate, and running the synthetic load generator continuously for 5 minutes. The load generator collects data about the request-reply latency, using an online algorithm\(^3\) to compute the latency average, standard deviation, minimum and maximum values. The data collected during the first 30 seconds are ignored.

On the same virtual machine where the playlist backend instance runs, a script launches periodically the `ps` command, `ps -eo pcpu, id, comm` (the result of which is shown in Snippet 4.1), grepping for the Playlist process PID. The CPU usage is sampled every 20 seconds, and an on-line algorithm\(^4\) is used to compute the cpu usage average, standard deviation, minimum and maximum values. Also in this case, the data collected during the first 30 seconds are ignored. The reason for ignoring the first 30 seconds is to let the transient response of the system stabilize. This process is repeated for a range of query rate values.

\(^3\)The numerically stable online algorithm used for estimating variance is proposed in [11, p. 232].
\(^4\)Ibid.
4.3. LATENCY AS AN INDICATOR OF LOAD

Snippet 4.1: A snippet showing the output of `ps -eo -eo pcpu,id,comm | head -n 6`

<table>
<thead>
<tr>
<th>%CPU</th>
<th>PID</th>
<th>COMMAND</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>1</td>
<td>init</td>
</tr>
<tr>
<td>0.0</td>
<td>2</td>
<td>kthreadd</td>
</tr>
<tr>
<td>0.0</td>
<td>2415</td>
<td>sshd</td>
</tr>
<tr>
<td>0.0</td>
<td>2416</td>
<td>bash</td>
</tr>
<tr>
<td>0.0</td>
<td>5554</td>
<td>dhclient</td>
</tr>
</tbody>
</table>

Results

The results of the first experiment are shown in Figures 4.7 and 4.8. Note that all values are normalized, meaning that given a vector $V$ of $n$ values $v_i$, with $0 \leq i < n$,

$$v_{i,\text{norm}} = \frac{v_i}{\|V\|_\infty}, \quad \forall i \in [0, n)$$  \tag{4.2}

where $\|V\|_\infty$ is the infinity norm of vector $V$, $\|V\|_\infty = \max(V)$.

Figure 4.7 illustrates the correlation between request rate and CPU usage. The $x$ axis of the scatter plot represents the normalized value of the request rate $rr_{\text{norm}}$, and the $y$ axis represents the normalized CPU usage $cpu_{\text{norm}}$. Y-error bars are given by the standard deviation of the CPU usage value. Note that the test Playlist service reaches a saturation point for $rr_{\text{norm}} > 0.7$. The Person sample correlation coefficient for two random variables $X$ and $Y$, of which a series of $n$ measurements is know is computed as:

$$r_{x,y} = \frac{\sum_{i=1}^{n} (x_i - \mu_x)(y_i - \mu_y)}{(n-1)\sigma_x\sigma_y}$$  \tag{4.3}

where $\mu_x$ and $\mu_y$ are respectively the sample means of $X$ and $Y$, and $\sigma_x$, $\sigma_y$ are their respective sample standard deviations, and $r$ is in the range $[-1, +1]$.

The population Pearson correlation $r$ between $cpu$ and $rr$ is $r_{cpu, rr} = 0.99$.

Figure 4.7 illustrates the correlation between CPU usage, our load index of choice, and the request-reply latency. The $x$ axis of the scatter plot represents the normalized value of the CPU usage $cpu_{\text{norm}}$, and the $y$ axis represents the normalized value for request-reply latency $\Delta t_{RR,\text{norm}}$. Error bars are given by the standard deviation for both samples. Note that the scale for CPU usage is consistent with the one adopted in Figure 4.7, so latency values past the point of saturation are discarded ($rr_{\text{norm}} \leq 0.7$). The population Pearson correlation $r$ between $cpu$ and $rr$ is $r_{cpu, rr} = 0.95$. Figure
Correlation in production

A white box approach similar to the one described in Section 4.3 has been adopted to compute a value for the Pearson correlation of the service running in production: again, the CPU is adopted as a reliable load index. Since the data collection in this case happens directly in the production environment, there is a strong requirement to keep the overhead to a minimum.

At Spotify, all services are monitored with a 5-minute granularity. Every 5 minutes, when the minute figure is a multiple of 5, a daemon running on each machine collects all relevant metrics, that are in turn collected and aggregated by the monitoring system. Clocks are kept in synchrony using NTP. Increasing the granularity is not an option, and neither is running a custom script on all Playlist production machines, so latency metrics are collected with a 5-minute granularity to take advantage of the existing CPU usage data made available by the monitoring system.

Design goals

As mentioned before, the main goal in this scenario is keeping the overhead to a minimum, since increasing the load would have the potential to affect the production environment and cause undesirable user-visible disruptions. Another important design goal is using the existing infrastructure as much as possible.

No such information as the latency with a per-request granularity is available from the monitoring system. The only information that the monitoring system provides in terms of latency is a distribution in terms of latency classes as measured by the Access Point, for each service as a whole or each individual service replica, something similar to what is shown in Figure 4.4. Unfortunately the granularity of the information already available from the monitoring system is too coarse for reconstructing the mean value and variance.

One could always compute an approximate value of latency mean and variance and average based on the latency classes. Let $x$ be a population of latency values, and let $C$ be the set of latency classes, where each class $C_i$ is characterized by its upper bound value $u_i$ and a number of items $|C_i|$, $C = \{C_i \mid \forall \ i \in [0, n]\}$, with $u_0 = 0$ by definition:

$$\bar{x} = \frac{1}{\sum_{i=1}^{n} |C_i|} \sum_{i=1}^{n} \frac{u_i - u_{i-1}}{2} |C_i|$$  \hspace{1cm} (4.4)

$$s_x^2 = \frac{1}{\sum_{i=1}^{n} |C_i|} \sum_{i=1}^{n} |C_i| \left( \frac{u_i - u_{i-1}}{2} - \bar{x} \right)^2$$  \hspace{1cm} (4.5)
As already mentioned, the estimated values of $\bar{x}$ and $s^2_x$ are not accurate enough to be directly used for evaluating the correlation between latency and CPU usage, but they are still usable for comparison.

The available CPU usage data is another source of inaccuracy, since the CPU usage is collected by running `mpstat` periodically (a sample output of which is shown in Snippet 4.2). `Mpstat` reports the aggregated percentage of CPU time spent in each usage category (such as `usr` for CPU utilization that occurs while executing at the user level, or `sys` for CPU utilization that occurs while executing at the kernel level). Since the only service running on the machines we have considered is Playlist, we consider the CPU time spent executing at user level to be a meaningful approximation of the Playlist process CPU usage. This is inaccurate, but the other values reported by `mpstat` are negligible with respect to the time spent in user mode (usually a couple of order of magnitude smaller), so we choose to ignore them.

Snippet 4.2: A snippet showing the output of `mpstat 2 2`. The two parameters represent respectively the reporting interval and the total number of records to be shown.

<table>
<thead>
<tr>
<th>Linux 2.6.32-5-amd64 (dvm)</th>
<th>09/18/2013</th>
<th><em>x86_64</em></th>
<th>(2 CPU)</th>
</tr>
</thead>
<tbody>
<tr>
<td>02:00:20 PM CPU %usr %nice %sys %iowait %irq %soft %steal %guest %idle</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>02:00:20 PM CPU</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>02:00:22 PM all</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>02:00:24 PM all</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
</tbody>
</table>

**Experimental setup**

The basic setup is shown in Figure 4.10. All the requests directed to the Playlist service are captured from a single Access Point, and stored in binary format in the same cluster where the Access Point resides, to minimize inter-datacenter traffic. The `hmcat` tool allows to easily snoop and dump Hermes traffic. Unfortunately, the information about which specific instance of the Playlist service will process the request we are interested in is not readily available (more on this in the following paragraphs).

The next step is to reconstruct the information about which backend instance takes care of processing a request. This is achieved by dumping all the incoming traffic to all the instances of the Playlist service to binary files. Also in this case, all data is stored in the same datacenter where the instance resides, so that no inter-datacenter traffic is generated.

A Python script later processes all data offline to establish which instance processed each request and to reconstruct and aggregate latency information. Aggregation produces average, standard deviation, minimum and maximum values and latency histograms for each machine. The process of collecting
traffic is automated through cron, that takes care of launching a set of hmcat processes with the appropriate parameters every 5 minutes, for a 2-minute timespan, in synchrony with the CPU usage value collection.

As mentioned in Section 2.3, the Hermes library allows services to expose a snooping socket, a ØMQ PUB-SUB socket where all the traffic in transit through a service can be published, mostly for debugging purposes. A snooping socket is a non-intrusive way to listen to traffic in transit through a process, especially useful in the production environment. As with all ØMQ PUB-SUB sockets, if the subscriber is slow, messages will queue up on the publisher until the user-defined high watermark value is reached, at which point point messages will simply be dropped. The value for high watermark is usually kept very low — in the order of a few tens of messages, as it is with most services at Spotify. The consequence is in turn that if the subscriber is too slow, messages will simply be dropped without taking up resources on the production service. We had to make sure that the consumer was fast enough, in order to guarantee that the sample size was significant. This is done by comparing the information about request rate available from the monitoring system and the number of requests processed by the consumer, over the same time span.

The percentage of messages recorded at the access point that are correctly matched to the Playlist instance they are processed by is approximately 90%. The main causes for a message not to be matched are: a) the corresponding message is dropped by the hmcat process that captures the requests from the Playlist service side, b) the skew between clocks on different servers causes the hmcat processes to be started at slightly skewed times, c) the requests that are still in-flight when the capture stops are dropped.

As already mentioned, the information about which backend replica processes a given request is unavailable from the data captured through the snooping port on the Access Points. This happens because Hermes Router, in charge of routing requests to the appropriate replica of the service, forwards the request directly to the appropriate ØMQ socket, and in turn to the appropriate underlying TCP socket, while a copy of the request is forwarded to the snooping socket. The request itself doesn’t contain any hint about which service instance it is directed to (except for the URI, which qualifies a resource as a whole but not it’s location). Now, it would be possible to snoop the TCP traffic using tcpdump, which would make the destination of each packet easily accessible. The problem with this approach is that tcpdump would consume precious resources on production machines, and it could possibly interfere with the stability of the service. Forwarding packets on the snoop sockets adds very little overhead and mainly it consumes additional network bandwidth, which — given the total amount of available network bandwidth — doesn’t constitute a problem.
4.4. LOAD BALANCING USING AN ADAPTED $\varphi$ ACCRUAL FAILURE DETECTOR

Results

The results of the first experiment are shown in Figures 4.11 and 4.12. Note that all values are normalized. Figure 4.11 shows the CPU usage value and average request-reply latency, as measured during our experiment in a 30-hour timespan. The continuous line represents latency and the dashed line represents CPU usage. Interestingly, even when the CPU usage time series doesn’t overlap with the CPU usage series, it follows the same shape.

On a totally unrelated note, Figure 4.11 shows a common usage pattern seen at Spotify during weekdays: the traffic volume drops at night, there is a first spike in the morning during commute time, followed by a plateau and another spike in late-afternoon, that leads to sustained traffic till the evening. More information on the service’s usage patterns can be found in [19].

Figure 4.12 is a scatter plot obtained from the same dataset used to generate Figure 4.11, in order to illustrate the correlation between the normalized value of CPU usage $cpu_{norm}$ and the normalized value for request-reply latency $\Delta t_{RR,norm}$. Error bars are omitted. In this case, the Pearson’s correlation between $X = \Delta t_{RR,norm}$ and $y = cpu_{norm}$ is $r_{x,y} = 0.83$. Despite having a lower correlation value in production — that is, in comparison with the value found in the test environment under more controlled conditions — we believe that there is enough evidence to accept the hypothesis that load (in the form of CPU utilization, in this specific case) and request-reply latency are correlated in the context of the Playlist service.

4.4 Load balancing using an adapted $\varphi$ accrual failure detector

In this section we present the design goals and the properties of a novel load balancing algorithm based on the $\varphi$ accrual failure detector. The $\varphi$ accrual failure detector is an algorithm for probabilistic, adaptive failure detection and has been presented in Chapter 3.

We propose an algorithm based on modeling the statistical distribution of latencies for each instance in a set of equivalent backends, and using the information on the latency distribution to predict future latencies for each instance. In Section 4.4 we discuss the motivations for using latency as a load index, in Section 4.4 we provide the pseudocode for the algorithm with comments, and in Section 4.4 we provide more information on the implementation in the context of the Hermes library.
CHAPTER 4. DYNAMIC LOAD BALANCING USING AN ADAPTED $\varphi$ ACCRUAL FAILURE DETECTOR

Motivations

There are three main reasons for which the perspective of using latency as a load indicator sounds attractive.

In the first place, the experimental evidence presented in Sections 4.3 and 4.3 suggests that latency might be a good load index. In general, load indices are strictly related to the specific system they refer to: a load index might be one of the resources available on the system (e.g. CPU usage, as in [21], disk I/O as in [13], network I/O, memory occupation, a linear combination of these resources, any of the other criteria listed in Table 4.1, and many more). Generally speaking, deciding which load index is most suitable for a system requires a detailed knowledge of the dynamic characteristic of the systems and its bottlenecks. Bottleneck analysis in the context of distributed systems is made particularly tricky by the presence of distributed multi-bottlenecks, discussed in Section 4.2. Performing a complete bottleneck analysis of all backend systems at Spotify would be a particularly unpractical task, due to the relatively large number of services involved and the speed at which these services evolve.

We are not suggesting that latency can be used as a universal load index, but evidence in Sections 4.3 and 4.3 confirm that there is a category of systems for which a correlation between load (as defined through a white-box approach) and latency (or a function thereof) exists, in a given confidence range (i.e. before saturation of critical resources is hit).

The second reason that makes latency a potentially attractive load index in a service like Spotify is that latency per se has a huge impact on user experience, the idea being that the user should be able to interact with the service as if everything was stored locally on their PC. This is partially achieved through smart use of caching in the client applications, but backend latency still makes a difference for content that is not cacheable, has been evicted from caches, or has never been fetched before. In other words, trying to minimize latency is often a desirable goal, regardless of the actual correlation between a known and proven load index and latency itself.

The third reason that makes latency an attractive load index, is that it allows for a fully fault-tolerant distributed design, with no support on the upstream side. Designs such as the one proposed in [21] require some support on the upstream servers, which in its minimum form comes in the shape of a daemon running on each upstream server, acting as a monitoring agent. The monitoring agent collects load metrics (such as CPU usage, load average, disk

5This statement applies if the user is not connecting to the service through a mobile connection. The latency on mobile connections is in the order of magnitude of backend response time, if not higher, which makes the backend request-reply latency negligible.
and network I/O bandwidth, memory usage, etc.) and exposes them to the load balancer through an API. A similar setup is also adopted by Linux Virtual Server to implement dynamic feedback load balancing scheduling in its IP Virtual Server component. Apart for the need to make those metrics available in a fault-tolerant fashion, so that the component that collects metrics doesn’t act as a single point of failure, care needs to be taken not to overload the metric collector with requests from the load balancers. Latency metrics can be collected independently by each load balancer without the need for any upstream component support. Despite not having a single authoritative source for metrics, the distributions built up with samples taken by each load balancer independently tend to converge and manifest statistical compatibility.

**Design space**

As far as load balancing based on request-reply latency is involved, several design choices are possible.

**Layer** Load balancers are generally classified into two categories, depending on which ISO OSI layers they act upon: Layer 4 and Layer 7. Layer 4 load balancers act upon data found in network and transport layer protocols (IP, TCP, UDP). Layer 7 load balancers distribute requests based upon data found in application layer protocols such as HTTP. As mentioned in Section 2.3, Hermes requests usually travel on long-lived TCP connections in order to avoid the cost of setting up a connection each time, therefore it makes little sense to implement Layer 4 load balancing. The algorithm proposed in Section 4.4 acts upon individual Hermes requests, so it is a Layer 7 load balancing algorithm.

**Choice of statistical distribution** When presenting the ϕ accrual failure detector in Section 3.2, we stated that heartbeat inter-arrival time is modeled as a Gaussian process. This assumption was made by Hayashibara et al. in the original ϕ accrual failure detector paper, but no justification for this choice has been presented to the best of our knowledge. The two main candidate distributions for request-reply latency that we have considered are in fact the normal and log-normal distribution, due to the bell-shaped resemblance of the latency histogram. In Appendix B we present the result of the chi-square goodness-of-fit test that was performed on a sample distribution observed in the test environment.

---

6Layer 2 load balancing is also supported by many vendors, and commonly known as direct routing, SwitchBack or nPath. In this case the load balancer determines the real server to forward each frame to and performs MAC address translation.
The conclusion of the chi-square test is that both distributions can approximate the sample distribution observed during our experiment, but this result might be partially biased by the inherent limitations of the chi-square goodness-of-fit test, as we motivate in Appendix B. In Figures 4.15 and 4.16 we show respectively some sample normal and log-normal distributions. The support of the log-normal distribution is $\mathbb{R}^+$, which reflects that latency is by definition positive; additionally it presents a rather long tail that reflects the observed behavior of actual latency distributions (see [5] with regards to this aspect). In Chapter 5 we test and compare the behavior of our algorithm against both distributions. Detailed conclusions regarding the choice of the distribution are presented in Appendix B.

**Request-reply latency vs heartbeat inter-arrival time** The original $\varphi$ accrual failure detector described in [10] is based purely on heartbeat inter-arrival times. In Section 3.2 we propose an extension to the failure detector that allows for measurement of request-reply latency instead. In analogy with the failure detector implementation, we have considered whether is it better to take into account request-reply latency or to send periodical heartbeats and measure the time it takes to get a reply. These two metrics reflect fundamentally different aspects: heartbeat inter-arrival time substantially reflects network conditions and communication queue lengths. These two parameters are unfortunately not particularly relevant to our purpose, since a) network conditions are rather controlled, especially when it comes to intra-datacenter traffic. Under normal operating conditions inter-datacenter network bandwidth doesn’t constitute a bottleneck and we explicitly decided not to take care of failures (which can be easily done using a regular $\varphi$ accrual failure detector); b) communication queue lengths are bound to rather small numbers by setting low values of ØMQ socket high-watermark levels: as [7] suggests, metrics that saturate rapidly do not make good load indices. On the other hand, measuring request-reply latency for actual requests reflects not only network conditions and queue lengths, but also service time, which is a far more relevant metric. There is a trade-off involved: using actual requests to compute the weights of each backend server eventually leads to starvation if one of the servers performs poorly: the server that exhibits higher-than-normal latency values might be assigned a nil weight, which in turn causes requests to stop being forwarded to the server in question. This prevents the load balancer from acquiring new informations on the state of the underperforming server, so it won’t assigning a new non-zero weight to the server. Measuring request-reply
4.4. LOAD BALANCING USING AN ADAPTED $\varphi$ ACCRUAL FAILURE DETECTOR

latency on actual requests requires dealing with starvation, a topic that is treated in detail in Section 4.5.

**Weight function** The main idea behind our use of the $\varphi$ accrual failure detector is to build a statistical distribution for each of the individual backend instances $s_i$ that the upstream server communicates with, and a global distribution that models all of the $n$ backend instances as if they were a single service. An upstream server builds $n + 1$ statistical distributions for each of the services it communicates with, each distribution being identified by two parameters $\mu$ and $\sigma^2$, where $\mu$ is the population mean and $\sigma^2$ is the population variance. Once the statistical distribution is known we need a function to translate it into a weight, so that we can express the weight of each backend $i$ as a function $w_i$ of the backend’s own distribution and the global distribution, $w_i(t) = f(w_i(t-1), \mu_i, \sigma_i^2, \mu_{\text{global}}, \sigma_{\text{global}})$. This is analogous to the $\varphi$ accrual failure detector, where the probability of a host $i$ being failed is a function of the mean and variance associated to its heartbeat inter-arrival delays: $\varphi_i = f(\mu_i, \sigma_i^2)$. In addition, we keep into account the mean and variance of the system as a whole as a reference point, hence the weight of a server depends both on its past performance and on the performance of the whole system.

The actual routing decision is made using weighted round-robin as a scheduling discipline, with weights $w_i$, $i \in [0, n)$. The choice of the weight function is an extremely delicate matter, because it heavily affects the performances of the load balancing algorithm in terms of stability and reactivity. As stated in Section 1.4, it is outside of the scope of this work to prove the optimality or sub-optimality of a given function $f$, nonetheless we have tested two different functions that we discuss in Section 4.4 and compare in Chapter 5.

**Algorithm**

In this section we present the main ideas that characterize our load balancing algorithm and the related pseudocode. Conceptually, the load balancing component collects request-reply latencies for each of the upstream servers and computes their average and standard deviation, that are used respectively as location and scale parameters for modeling the distributions. Basing on individual servers’ distribution and the distribution shown by the system as

$^{\text{7}}$Regardless of whether we consider normal or log-normal distributions, since they are both qualified by $\mu$ and $\sigma^2$. 
a whole, the load balancer periodically computes each server’s weight, and requests are forwarded in a weighted round-robin fashion according to those weights.

**Latency sampling** For each downstream server \(i\), the load balancing component keeps track of the latency distribution. In order to compute the latency for a single request, the load balancing layer tags a rate-limited number of requests with the timestamp at which the request was processed. A timestamp is simply the number of milliseconds expired since the Unix Epoch. As we shall see in Section \[4.4\] the load balancing layer is part of the Hermes stack and it runs on a dedicated thread pool as part of the downstream service, hence each reply is processed by the same service that issued the corresponding request. This simplifies the latency sampling logic, since we can always rely on the local clock of the machine where the service runs. As soon as a reply for a timestamped request is received by the service that issued the corresponding request, the difference is computed. Obviously this strategy doesn’t take into account the processing time at lower layers in the stack, which we ignore on the ground that it’s approximately 3 orders of magnitude lower than the time spent by requests on the wire and the processing time on the downstream server (tens of milliseconds vs tens of nanoseconds). Timestamp-tagging is rate limited, meaning that we only tag up to \(n\) requests per downstream backend, per unit of time, in order to limit the overhead of heavily loaded services.

**Moving averages** Every collected sample is used to update the exponential moving average and moving variance of the replica that processed the related request, and the global moving average and variance of the system as a whole. Adding a new sample to the population has cost \(O(1)\) thanks to the incremental algorithm presented in Appendix \[A\]. Tuning the value of the time constant involves a trade-off between the reactivity of the algorithm and the stability of the weights: a shorter time constant makes the load balancer more reactive to variations in latency, however it makes weights more sensitive to noise (i.e. the weights are more likely to suffer from spurious oscillations). More about tuning the time constant follows in Chapter \[5\].

**Updating the weights** We can think of the weights for each downstream server as a normalized vector \(w\) of length \(n\), where each weight is generically a function of \(i\) the previous weight, \(ii\) the moving average and standard deviation of the latency of the specific replica the weight belongs to, \(iii\) the moving average and standard deviation of the latency of
4.4. LOAD BALANCING USING AN ADAPTED $\varphi$ ACCRUAL FAILURE DETECTOR

the system as a whole. Hence, $w_i(t) = f(w_i(t-1), \mu_i, \sigma^2_i, \mu_{global}, \sigma^2_{global})$.

A concurrent lock-free map contains the mapping $id_i \mapsto w_i$, which allows to update the weights in a non-blocking fashion. Depending on the computational cost of $f$, updating the weights might be an expensive operation. In order to limit the overhead from weight updates, this is done periodically with a tunable weight refresh period $T_{\text{refresh}}$.

Updating the list of downstream servers In order to allow for a failure detector (or a discovery service, or an operator) to update the list of active connections, our load balancer’s interface provides a way to update the list of downstream servers. A typical usage for this feature is failover, to be implemented in conjunction with a failure detector: when a component is suspected to be failed, it is removed from the pool of active components. Similarly, a component that comes back online is re-instated in the pool of active components and the load balancer begins to route traffic to the component again. Updating the weights atomically requires locking, which unfortunately adds a lot of overhead for a rather uncommon case. In Section 4.4 we propose a locking scheme in order to minimize such overhead.

The scheduling algorithm implemented in our load balancing layer, will not send any new requests to a server after it has been removed from the pool of active servers, but in-flight requests still get served by said server. System administrators can use this feature to make a server quiescent. When all the in-flight requests to this server have finished or timed-out, they can switch the server out of the cluster for system maintenance.

Implementation

Module 4.1 and Algorithm 4.1 present respectively the properties and the pseudocode of our load balancing algorithm, expressed with the same notation used by Guerraoui and Rodrigues in [9]. Algorithm 4.1 has a dependency on the PerfectPointToPointLink abstraction defined in [9, p.37] and which has been included here as Module 4.2 for the reader’s convenience.

The LoadBalancer component $lb$ provides a similar interface to a PerfectPointToPointLink, except that the $\text{Send}$ method is substantially split in two parts: $i)$ providing the LoadBalancer with a set of correct processes, and $ii)$ sending a message to any of the processes in the set. In a PerfectPointToPointLink, $\text{Send}$ accepts both a message and a recipient process as parameters. On the contrary the $\text{Send}$ event of a LoadBalancer accepts a message as its
Figure 4.7: A scatter plot showing the correlation between request rate and CPU usage in the test environment. The methodology is reported in Section 4.3. Note that all values are normalized. The population Pearson correlation \( r \) between \( X \) and \( Y \) is \( r_{x,y} = 0.99 \).

**Module 4.1**: Interface and properties of load balancers.

**Name**: LoadBalancers, instance \( lb \).

**Events**:
- **Request**: \( \langle pl, UpdateServers \mid S \rangle \): Requests to update the set of correct known servers with \( S = \{s_1, \ldots, s_n\} \).
- **Request**: \( \langle pl, Send \mid m \rangle \): Requests to send message \( m \) to any process in \( S \).
- **Indication**: \( \langle pl, Deliver \mid p, m \rangle \): Delivers message \( m \) sent by process \( p \).

**Properties**:
- **LB1**: Reliable delivery: If a correct process \( p \) sends a message \( m \) to a correct set of processes \( S \), then one process \( q \in S \) eventually delivers \( m \).
- **LB2**: No duplication: No message is delivered by a process more than once.
- **LB3**: No creation: If some process \( q \) delivers a message \( m \) with sender \( p \), then \( m \) was previously sent to \( q \) by process \( p \).
4.4. LOAD BALANCING USING AN ADAPTED $\varphi$ ACCRUAL FAILURE DETECTOR

Note that all values are normalized. The population Pearson correlation $r$ between $X$ and $Y$ is $r_{x,y} = 0.95$

---

**Module 4.2:** Interface and properties of perfect point-to-point links.

**Module:** PerfectPointToPointLinks, instance $pl$.

**Events:**
- **Request:** $\langle pl, Send \mid q, m \rangle$: Requests to send message $m$ to process $q$.
- **Indication:** $\langle pl, Deliver \mid p, m \rangle$: Delivers message $m$ sent by process $p$.

**Properties:**
- **PL1:** Reliable delivery: If a correct process $p$ sends a message $m$ to a correct process $q$, then $q$ eventually delivers $m$.
- **PL2:** No duplication: No message is delivered by a process more than once.
- **PL3:** No creation: If some process $q$ delivers a message $m$ with sender $p$, then $m$ was previously sent to $q$ by process $p$. 

---

Figure 4.8: A graph showing the correlation between request rate and CPU usage in the test environment. The methodology is reported in Section 4.3. 

Module 4.2: Interface and properties of perfect point-to-point links.
Figure 4.9: A three-dimensional representation of the same dataset shown in figure 4.8. The CPU usage and latency values are normalized. The surface represents the probability density function as a function of CPU usage and latency. Notice how higher CPU usage corresponds to higher values of latency standard deviation.

Figure 4.10: Experimental setup used to assess load-latency correlation in the production environment.
4.4. LOAD BALANCING USING AN ADAPTED ϕ ACCRUAL FAILURE DETECTOR

Algorithm 4.1: Load balancing algorithm.

**Implements:**
Loadbalancer, instance lb.

**Uses:**
PerfectPointToPointLinks, instance pl.

**Upon event** \(\langle lb, \text{Init} \rangle\) do

\[\text{activeServers} := \emptyset;\]
\[\text{properties} := [];\]
\[\mu_{\text{global}}, \sigma_{\text{global}} := [0, 1];\]
\[\text{starttimer}(\Delta);\]

**Upon event** \(\langle \text{Timeout} \rangle\) do

\[
\text{foreach server in keyset(properties) do}
\]
\[
\text{weight}, \mu, \sigma := \text{properties}[server];
\]
\[
\text{newWeight} := f(\text{weight}, \mu, \sigma, \mu_{\text{global}}, \sigma_{\text{global}});
\]
\[
\text{properties}[server] := [\text{newWeight}, \mu, \sigma];
\]

**Upon event** \(\langle lb, \text{UpdateServers} | servers \rangle\) do

\[\text{activeServers} := \text{servers};\]
\[\text{properties} := [];\]
\[
\text{foreach server in activeServers do}
\]
\[\text{properties}[server] := [\text{defaultWeight}, \mu = 0, \sigma = 1];\]

**Upon event** \(\langle lb, \text{Send} | message \rangle\) do

\[t_s = \text{now}();\]
\[\text{server} = \text{wrrSelect}(\text{activeServers}, \text{properties});\]
\[\text{trigger} \langle pl, \text{Send} | server, [message, t_s] \rangle;\]

**Upon event** \(\langle pl, \text{Deliver} | p, [message, t_s] \rangle\) do

\[t_r = \text{now}();\]
\[\text{properties}[p] := \text{updateProperties}(p, t_r - t_s);\]
\[\text{trigger} \langle lb, \text{Deliver} | p, message \rangle;\]
only parameter and it embeds the logic to choose a recipient out of the activeServers set of correct processes. The set of correct processes is retained by the LoadBalancer as part of its state.

**Correctness** Properties LB2 and LB3 are guaranteed directly by the underlying PerfectPointToPointLink, and reflect properties PL2 and PL3. In particular LB3 has an additional implication with respect to its homologous PL3: the process q that receives a message m gets to known which process p received the original request that triggered m as a reply, only when the LoadBalancer delivers m (that is, if we consider a request-reply semantics); in other words, the load balancer is transparent to the receiving process, which sees each message as if it were coming from the originating process, but the originating process has no knowledge of what process in activeServers will be delivered its messages. Property LB1 states that reliable delivery is guaranteed if and only if every process in activeServers is correct. A monitor based on a failure detector can be used to update the set of correct servers.

**A note on locking** The notation used for Algorithm 4.1 is consistent with [9] and is based on the notion that event handlers are atomical, hence they do not interfere with each other. This is not the case in the Hermes library, where event handlers are executed by a thread pool, are not executed atomically, and can be interrupted at any time by the scheduler. Furthermore there are shared data structures that must handle concurrent access. As mentioned before, the mapping id 
\[ \mapsto w \]
where id is the ID of the \( i \)th server and \( w \) is its weight, is kept in a concurrent lock-free map. The lock-free map makes it possible to update each individual weight without locking up the whole map for the entire duration of the weight refresh operation. As a consequence of this design, there is a short time frame during which the weights are not consistent and the weight vector \( w \) is not normalized (i.e. \( |w| \neq 1 \)), since they are not being refreshed atomically. The resulting overhead from this solution is minimal, at the cost of a brief inconsistency that doesn’t have any noticeable effect on condition that the refresh period is long enough: let \( T_{\text{refresh}} \) be the refresh period and \( \text{\bar{t}}_{\text{update}} \) the duration of the update itself (i.e. the amount of time it takes from the first weight being modified, to the last weight being modified in accordance); it is required that \( T_{\text{refresh}} \gg \text{\bar{t}}_{\text{update}} \).

However, it is necessary to guarantee the atomicity of the server list update if we want to make sure that a happened before relationship holds between a call to the method that updates the list of downstream servers and every subsequent send operation. In order to avoid every call to the send method being blocked on acquiring the lock, we can use a read-write lock. The idea
is that regular weight updates and usual send calls will only acquire the read
lock, thus giving the same guarantees described above at the cost of a little
overhead (so that the calls are non-blocking, at least in the common case). On
the other hand, updating the list of downstream servers will require acquiring
the write lock. This has two implications: a) while the lock is being held,
all the send calls will be blocked when entering the critical section protected
by the read-write lock, and b) an happened-before relationship holds between
the call to the update method and any subsequent send operation: once the
update call returns, all messages are guaranteed to be forwarded to one of the
backends in the new list.

Updating the list of downstream servers is quite expensive, since every
send call will be blocked for the entire time it takes to remove the old items
and insert the new ones, but this is an extremely unfrequent case.

4.5 Pitfalls and caveats

In this section we discuss some pitfalls and caveats related to the load balanc-
ing algorithm we have proposed. In the first place, we show that the choice
of measuring the latency of actual requests (instead of heartbeats) has the
unpleasant side-effect of making our algorithm susceptible to starvation. We
propose two different strategies to avoid this problem.

In Section 4.5 we propose two different strategies to handle faulty servers
that reply with low-latency error messages.

Avoiding starvation

The choice of request-reply latency instead of heartbeat inter-arrival times to
build latency distributions makes our load balancing algorithm susceptible to
starvation. Request-reply latency allows to keep into account not only the
effect of communication queue lengths and network latencies, it also accounts
for the processing time of each request. However, if a server is assigned a
nil weight because it performs poorly compared to the rest of the replicas, no
requests will be forwarded to that server, making it impossible to subsequently
update its latency distribution and thus its weight.

Building latency distributions based on heartbeat inter-arrival times doesn’t
cause any starvation problems, because we can obviously keep sending heart-
beats regularly, independently of a server’s weight.

As an alternative, we could send duplicated requests to multiple backends,
serving back the reply that comes first to the requesting server while keeping
into account all the replies to compute their latencies. This option is inves-
tigated in Chapter 6 and based on some of the ideas presented by Dean and Barroso in [5], but it is unfortunately not applicable to Hermes requests since there is no semantic indication about their idempotence.

We have investigated and implemented two solutions, one based simply on setting a minimum weight, and one inspired by the optimistic unchoking algorithm used in BitTorrent.

**Minimum weight**

The simplest option to avoid starvation is imposing a minimum weight. By doing so the weight of a single server cannot fall under a pre-determined threshold, making it possible for each server that performs poorly to still receive requests in order to eventually improve its weight. Having a separate failure detector for failover makes this a viable option even in case of failures: after all, the responsibility of excluding a server from the active server pool should be on the failure detector, not on the load balancer.

**Optimistic unchoking**

The idea of optimistic unchoking is inspired by the optimistic unchoking algorithm used in BitTorrent. BitTorrent clients are affected by starvation in the bootstrapping phase, when the users don’t have any files to share in exchange for their downloads. BitTorrent allocates a finite number of download slots using a tit-for-tat strategy, also known as equivalent retaliation. The main principles of equivalent retaliation are:

- unless provoked, the agent will always cooperate;
- if provoked, the agent will retaliate;
- the agent is quick to forgive;
- the agent must have a good chance of competing against the opponent more than once.

Cooperation is achieved when upload bandwidth is exchanged for download bandwidth. Hence, when a peer is not uploading in return to our own peer uploading, the BitTorrent client will choke the connection with the uncooperative peer. Now, this constitute a problem for clients that have no content to upload in return for downloads (e.g. new users). BitTorrent solves this problem with optimistic unchoking, by periodically unchoking a randomly selected interested peer.\(^8\)

---

\(^8\) In BitTorrent terminology, peer A is interested in peer B if peer B owns pieces of a file that peer A does not have and wishes to download.
4.5. PITFALLS AND CAVEATS

We have implemented a similar strategy by making sure that after a configurable number of weight refresh periods $T_{\text{refresh}}$, a server with a weight under a configurable threshold $w_{\text{min}}$ has a configurable probability $p > 0$ of receiving requests. This is similar to the minimum weight strategy, except that it allows for an hysteresis cycle where the weight can fall until $w_{\text{min}}$ is reached, and it is then restored to $p$ after some time, giving a potentially overloaded backend some extra time to process its backlog. In all the tests that we have conducted optimistic unchoking doesn’t seem to provide significant improvements or different behaviors when compared to the minimum weight strategy, despite being significantly more complex to implement and despite requiring more state information to be stored by the load balancer (namely, the load balancer needs to keep track for how many $T_{\text{refresh}}$ periods each of the servers has a weight $w_i < w_{\text{min}}$).

Handling faulty replicas

In Section 4.2 we argued about the importance to have a fail-fast strategy in place, in order for services to handle graceful degradation when failures occur. Figure 4.4 shows the fail-fast behavior, triggered by the popcount incident. The effect of the fast fail behavior is the large spike of low-latency replies that is visible around week 20.

The load balancing algorithm we propose weights replicas based on the latency distribution that their replies exhibit, so we need to take into account the status code of each reply to account for low-latency, error replies, that might mislead the load-balancer into improving the weight of faulty service replicas. Replies of Spotify services are qualified by a status code, analogous to those defined in the HTTP standard. We propose two ways to deal with failures: consider them as high latency replies without modifying the load balancing algorithm, or take errors into account separately.

High-latency replies The simplest way to penalize replicas that serve low-latency error replies is to ‘fake’ the latency sample so that it appears much higher than reality; as an example we might decide to consider each error as a reply with latency equal to $n$ standard deviations, with $n > 1$. If a sufficient proportion of the replies are errors, the average latency value of the faulty backend will converge to $n$ standard deviations as fast as the time constant of latency sampler allows. The global latency distribution is not supposed to be updated with the artificial latency samples, in order not to influence the real, system-wide latency value.

Error distribution A second option for handling fast error replies, is to count the number of errors that occur in a given time window and com-
pute the error rate over said time window. The weight of the faulty backend is artificially set to a pre-determined value when the error rate goes above a certain threshold value, and is not restored to the value computed from samples until the error rate drops below the threshold. The threshold can either be set to a fixed value, or to $n$ standard deviations of the error rate distribution across all backend replicas.
Figure 4.11: Two time series that show side by side the normalized value for CPU usage and request-reply latency in the Playlist backend, over a time span of approximately 30 hours. Error bars are not plotted to avoid clutter. Data was captured between September 18, 2013 and September 20, 2013.
Figure 4.12: A scatter plot obtained from the same data set plotted in Figure 4.11 showing the correlation between CPU usage and request-reply latency in production. The methodology is reported in Section 4.3. Note that all values are normalized. The population Pearson correlation \( r \) between \( X \) and \( Y \) is \( r_{x,y} = 0.83 \).
Figure 4.13: Normalized value for CPU usage and request-reply latency in the Playlist backend between September 17, 2013 and September 18 2013. Note how the latency spike around 20:00 corresponds to a CPU usage spike that happens at the same time.
Figure 4.14: A scatter plot analogous to Figure 4.12 obtained from the same data set plotted in Figure 4.13 showing the correlation between CPU usage and request-reply latency in production. The population Pearson correlation $r$ between $X$ and $Y$ is $r_{x,y} = 0.80$. 
4.5. PITFALLS AND CAVEATS

Figure 4.15: Probability density function of a normal distribution $\mathcal{N}(\mu, \sigma^2)$. In the picture $\mathcal{N}(0, 0.25)$, $\mathcal{N}(0, 0.5)$ and $\mathcal{N}(0, 1)$ are depicted. $\mathcal{N}(0, 1)$ is also called standard normal distribution. $\mathcal{N}(\mu, \sigma^2)$ has two inflection points in $x_{1,2} = \mu \pm \sigma$. Note that supp($\mathcal{N}$) = $\mathbb{R}$.

Figure 4.16: Probability density function of a log-normal distribution $\ln\mathcal{N}(\mu, \sigma^2)$. In the picture $\ln\mathcal{N}(0, \sqrt{0.25})$, $\ln\mathcal{N}(0, \sqrt{0.5})$ and $\ln\mathcal{N}(0, \sqrt{1})$ are depicted. Note that supp($\ln\mathcal{N}$) = $\mathbb{R}^+$. 

\[
\frac{1}{\sqrt{2\pi \sigma^2}} \exp\left\{-\frac{(x - \mu)^2}{2\sigma^2}\right\}
\]

\[
\frac{1}{\sqrt{2\pi \sigma^2}} \exp\left\{-\frac{(\ln x - \mu)^2}{2\sigma^2}\right\}
\]
Chapter 5

Experimental results

In Section 5.1 we provide an overview of two different weight functions that we have considered as part of our empirical evaluation; we discuss the properties of both functions and the expected behaviour in the context of our load balancing algorithm.

Section 5.2 provides an overview of the experimental setup used to obtain the data presented in this chapter. In order to operate in a completely deterministic environment and to produce comparable results, we have chosen to simulate a number of test scenarios, rather than using the test environment to capture metrics on real systems. Nonetheless, a few configurations for each test scenario have also been evaluated in the test environment to guarantee the reliability of our simulations; we have chosen to report solely on data obtained through simulation to avoid repetition: the general conclusion is that data obtained through simulations is consistent with the behavior observed in the test environment.

From Section 5.3 until the end of this chapter, we describe a list of the test scenarios we have evaluated and the outcome of each scenario for a meaningful set of configurations.

5.1 Weight functions

As mentioned in Section 4.4, the choice of the weight function is an important design decision that affects the performances of our algorithm. In Section 1.4 we stated that it is outside of the scope of this work to provide analytical insight into the optimality or suboptimality of different weight functions. For the purpose of evaluating our algorithm we have conducted a series of simulations, using two different weight functions. A weight function is a function
in the form:

\[ w_i(t) = f(w_i(t-1), \mu_i, \sigma_i^2, \mu_{\text{global}}, \sigma_{\text{global}}^2) \]  \hspace{1cm} (5.1)

where \( w_i(t) \) is the weight of backend \( i \) at step \( t \) in the context of a weighted round robin scheduling discipline, \( \mu_i \) and \( \sigma_i^2 \) are respectively the latency average and standard deviation of backend instance \( i \), and \( \mu_{\text{global}} \) and \( \sigma_{\text{global}}^2 \) are respectively the latency average and standard deviation of the system seen as a whole. The weight function is used to recompute the weight of each backend instance periodically, with period \( T_r \).

The first weight function we have considered is:

\[ w_i(t) = e^{-\frac{T_r}{\tau}} w_i(t-1) + (1 - e^{-\frac{T_r}{\tau}}) \frac{1}{2\pi \sigma_i^2} \mu_{\text{global}} \int_{-\infty}^{\infty} e^{-\frac{(x-\mu_i)^2}{2\sigma_i^2}} \, dx \]  \hspace{1cm} (5.2)

where \( \tau \) is the time constant of the exponential smoothing and \( T_r \) is the refresh period, i.e. the time interval between any two subsequent weight updates. The weight of backend instance \( i \) at time \( t \) is the exponentially smoothed probability that the response time of backend instance \( i \) is lower than the average response time of the system as a whole, assuming a normally distributed variable:

\[ P(x_i < \mu_{\text{global}}) = F\left(\frac{\mu_{\text{global}} - \mu_i}{\sigma_i}\right) \]  \hspace{1cm} (5.3)

\[ = \int_{-\infty}^{\mu_{\text{global}}} e^{-\frac{(x-\mu_i)^2}{2\sigma_i^2}} \, dx \]  \hspace{1cm} (5.4)

We will refer to this weight function as weight function \#1. Let us consider the derivative of (5.4) with \( \mu_i = 0 \) and \( \sigma_i^2 = 1 \). We obtain the probability density function of a standard normal distribution:

\[ f(x) = \frac{1}{2\pi \sigma^2} e^{\frac{x^2}{2\sigma^2}} \]  \hspace{1cm} (5.5)

Equation (5.5) is plotted in figure 5.1. \( f(x) \geq 0, \forall x \in \mathbb{R} \), as imposed by the fact that the related cumulative distribution function must be monotonically increasing by definition. Furthermore \( f(x) \) has its absolute maximum in \( x_0 = 0 \) \( (x_0 = \mu \) for a generic distribution), which has implications on the reactivity of the load balancing algorithm: the relative variation of the weights is more pronounced (has a higher derivative value) when the latency value of the backend instance we’re evaluating is close to the global average, and it tends to be smoother the further the average latency value is from the global average.
5.1. WEIGHT FUNCTIONS

We have considered a second weight function, inspired by Red Hat’s cluster management tool Piranha. Piranha integrates a dynamic feedback load balancing algorithm that exploits the value of server load average as a load index, using the following weight function:

\[
\begin{align*}
    w_i(t) &= \begin{cases} 
        w_{i,c} = w_i(t-1) + A \sqrt{1 - \text{load}_i} & \text{if } w_{i,c} \in [w_{\text{min}}, w_{\text{max}}] \\
        w_{\text{min}} & \text{if } w_{i,c} \in (-\infty, w_{\text{min}}) \\
        w_{\text{max}} & \text{if } w_{i,c} \in (w_{\text{max}}, +\infty)
    \end{cases}
\end{align*}
\]  

(5.6)

where  \( \text{load}_i \) is the load average on server  \( i \), and  \( w_{i,c} \) is the candidate weight at the current step. if the candidate weight is out of the range  \([w_{\text{min}}, w_{\text{max}}]\), the weight saturates at one of the extremes.  \( A \) is a tunable coefficient that acts as a scale factor with respect to each unique increment. The function written in this form implies that a server is underloaded when its weight is less than 1, it is optimally loaded with a weight exactly equal to 1, and it is overloaded with a load greater than 1. The value of optimal load can be tuned by applying the substitution  \( \text{load}_i \rightarrow \frac{\text{load}_i}{\text{load}_{\text{opt}}} \) with optimal load  \( \text{load}_{\text{opt}} \). We have used a similar approach to define a weight function that allows us to take into account the ratio between the latency of a specific server instance and the latency of the
system as a whole:

\[
    w_i(t) = \begin{cases} 
        e^{-\tau} w_i(t-1) + (1 - e^{-\tau}) & \text{if } w_{i,c} \in [w_{\min}, w_{\max}] \\
        w_{\min} & \text{if } w_{i,c} \in (-\infty, w_{\min}) \\
        w_{\max} & \text{if } w_{i,c} \in (w_{\max}, +\infty) 
    \end{cases}
\]

(5.7)

where again \(\tau\) is the time constant of the exponential smoothing and \(T_r\) is the refresh period. The candidate weight is exponentially smoothed and the increment at each step is proportional with proportionality coefficient \(A\) to the cube root of the average latency manifested by the specific backend instance we’re considering divided by the global latency. Hence, a server is underloaded when it’s average latency is lower than the average latency of the system as a whole, it is optimally loaded when the two values coincide, and it is overloaded when the average latency is higher than the average latency of the whole system. We will refer to this weight function as \textit{weight function #2}. Let us consider the derivative of \(f(x) = \sqrt[3]{1-x}, \ x \in \mathbb{R}\):

\[
    \frac{\partial}{\partial x} \sqrt[3]{1-x} = -\frac{1}{3(1-x)^{\frac{2}{3}}}
\]

(5.8)

the result is plotted in figure 5.2. The derivative has an essential discontinuity in \(x_0 = 1\):

\[
    \lim_{x \to 1^-} f' = -\infty \neq \lim_{x \to 1^+} f' = +\infty
\]

(5.9)

With respect to its derivative, \textit{weight function #2} has similar properties to \textit{weight function #1}: the relative variation of the weights is more pronounced (has a higher derivative value) in a neighborhood of the global average value. On one hand this makes the algorithm more reactive to any deviation from the global average, on the other it increases the sensitivity of the weight function to spurious noise: as we’ll see in following sections the time constant \(\tau\) that regulates the exponential smoothing of weights plays a crucial role in contrasting spurious high-frequency weight variations.

5.2 Experimental setup

Due the difficulty of obtaining uniform and deterministic testing conditions in the test setup that was used for the measurements presented in Section 4.3, we have decided to simulate the behavior of the load balancer instead. The
5.2. EXPERIMENTAL SETUP

The main sources of non-determinism in our test setup are garbage collection cycles, periodic tasks run by chron and other tasks running on the same virtual machines and competing for resources: all of these factors contribute to generating a non-reproducible bias in some of the measurement. While mitigations are possible, we felt that a reproducible latency distribution was important in order to provide a reliable and unbiased comparison of the performances of our load balancer with respect to a pure round robin scheduler, and to illustrate the performances of our load balancer using different configurations. Most importantly, using a simulator allows us to test a number of behaviors in total independence, having a guarantee that the observed effect depends solely on the configuration of the system and its input.

The simulation environment is written in Python and runs as a single process. It features a Python implementation of Algorithm 4.1; the response time of backend services is modeled as a normal or log-normal distribution where latency and variance are configurable functions of the query rate (i.e., the number of requests received in a time unit). In all of our experimental scenarios, the average latency is a linear function of the request rate $r_r$, and variance is a constant:

$$\mu = ar_r + b \quad \sigma^2 = c \quad (5.10)$$

A random number generator is initialized with the same seed at the beginning of each simulation. The incoming request rate is a function of elapsed simulation time $t$. The time granularity of all simulations is $T_s = 10^2$ s and no

![Figure 5.2: Plot of $\frac{\partial}{\partial x} \sqrt{1-x}$](image)
rate limiting is applied to latency sampling (i.e., the latency of every incoming request is sampled). In order to guarantee the reliability of our simulations, at least a couple of configurations for each test scenario have been reproduced in the test environment. We have found the results to be compatible, so we omit the measurement from the test environment to avoid repetition.

We have run each test scenario against a pool of 2, 5 and 10 simulated backend instances. Since no inconsistencies in the results have emerged, the graphs in the following sections only refer to the minimum pool size of two servers to avoid cluttering.

5.3 Test scenario 1

In test scenario 1 we measure the rise time for the sampled latency value. The rise time is the time taken by a signal to change from a specified low value to a specified high value: in our case the average value of every latency distribution is initialized to 0, and the final value we expect to obtain is 1. In other words, we measure how long it takes for a latency distribution to converge to the actual value of latency in static conditions. Since every latency distribution is exponentially damped, and as such behaves as a first-order LTI system, it is easy to compute the rise time analytically; we will consider the rise time from 10% to 90%: as shown in Appendix A, the step response of a first order LTI system is:

\[ y(t) = y_f \ast (1 - e^{-\frac{t}{\tau}}) \] (5.11)

where \( y_f \) is the final value (in our case \( y_f = 1 \)), and \( \tau \) is the time constant of the system. Solving Equation 5.11 for \( t \)'s, we obtain:

\[ t = -\tau \ln \left( 1 - \frac{V(t)}{V_0} \right) \] (5.12)

Let \( t_{10\%} \) be the time needed to go from 0% to 10%, and let \( t_{90\%} \) the time needed to go from 0% to 90%:

\[ t_{10\%} = -\tau \ln(1 - 0.1) = -\tau \ln(0.9) \] (5.13)
\[ t_{90\%} = -\tau \ln(1 - 0.9) = -\tau \ln(0.1) \] (5.14)

Subtracting \( t_{10\%} \) from \( t_{90\%} \):

\[ t_{10\%} - t_{90\%} = t_{90\%} - t_{10\%} \] (5.15)
\[ = -\tau \ln \left( \frac{9}{10} \right) + \tau \ln \left( \frac{1}{10} \right) \] (5.16)
\[ = -\tau (\ln(1) - \ln(10)) + \tau (\ln(9) - \ln(10)) \] (5.17)
\[ = \tau \ln(9) \approx \tau \cdot 2.197 \] (5.18)
5.4. TEST SCENARIO 2

As shown this result is easily obtainable analytically: we want to verify that the behavior of the system actually matches the specifications.

Results

Results for test scenario 1 are shown in Figure 5.3. We have performed the test against a six different exponentially smoothed sample latencies, with time constants \( \tau \in \{1 \text{ s}, 5 \text{ s}, 10 \text{ s}, 20 \text{ s}, 50 \text{ s}, 100 \text{ s}\} \). The results are entirely consistent with our expectations.

5.4 Test scenario 2

In this test scenario we compare the performances of our load balancing algorithm against pure round robin scheduling in the context of backend services that have the same latency profile. Request-reply latency is modeled for each backend instance as a normal distribution, where the average latency value is a linear function of the request rate \( r \): \( \mu(r) = a \cdot r + b \), hence \( Y = \mathcal{N}(\mu(r), \sigma^2) = \mathcal{N}(a \cdot r + b, \sigma^2) \). In such conditions, we would ideally expect that the weights converge to the same value and exhibit minimum fluctuations. We have chosen a time constant for the exponential smoothing of weights of 2 seconds. We have performed the test against a six different exponentially smoothed sample latencies, with time constants \( \tau \in \{1 \text{ s}, 5 \text{ s}, 10 \text{ s}, 20 \text{ s}, 50 \text{ s}, 100 \text{ s}\} \). The main criteria we consider for evaluation:

- stability of weights: amplitude and frequency of weight oscillations;
- effect of time constant of exponentially decaying average on the stability of weights;

Results

Figures 5.4–5.5 and Figures 5.6–5.7 show the outcome of test scenario 2, applying respectively latency function #1 and weight function #2. Both figures represent the values of normalized weights over time, for a total running time of 200 s. The output exponential smoothing time constant on weights is consistently set to 2 s for all the scenarios, across all configurations.

Configurations in Figure Figures 5.4–5.5, where weight function #1 is applied, exhibit relative fluctuations in the range \( \pm 4\% \) on normalized weights (with respect to the equilibrium value \( w_0 = w_1 = 0.5 \)). This observation holds for all values of \( \tau \) except \( \tau = 1 \) s: the width of oscillations is approximately in a \( \pm 8\% \) range of equilibrium values. As expected, the period of oscillation
Figure 5.3: Test scenario 1: rise time for estimated mean average $\mu$. All the values are supposed to converge to the actual value 1. Note that the horizontal scale is different in each subfigure.
5.4. TEST SCENARIO 2

Figure 5.4: Test scenario 2: average latency and weights for weight function #1 and $\tau \in 1\text{ s}, 5\text{ s}, 10\text{ s}$. 

(a) Average latency, $\tau = 1\text{ s}$

(b) Weights, $\tau = 1\text{ s}$

(c) Average latency, $\tau = 5\text{ s}$

(d) Weights, $\tau = 5\text{ s}$

(e) Average latency, $\tau = 10\text{ s}$

(f) Weights, $\tau = 10\text{ s}$
Figure 5.5: Test scenario 2: average latency and weights for weight function #1 and $\tau \in 20 \text{ s}, 50 \text{ s}, 100 \text{ s}$. 
Figure 5.6: Test scenario 2: average latency and weights for weight function #2 and $\tau \in 1 \text{ s}, 5 \text{ s}, 10 \text{ s}$. 
Figure 5.7: Test scenario 2: average latency and weights for weight function #2 and $\tau \in 20$ s, 50 s, 100 s.
appears to be related to the value of $\tau$: the larger $\tau$, the larger the oscillation period. This is due to the bass-pass behavior of the exponentially decaying average: higher values of $\tau$ reduce the bandwidth of the filter:

$$f_{3dB} = \frac{1}{2\pi \tau} \quad (5.19)$$

where $f_{-3dB}$ is the conventional way of specifying the bandwidth of a system: bandwidth is defined as the frequency range where power drops by less than half (or, in decibels, at most -3 dB). $f_{3dB}$ is also know as cutoff frequency. There is a tradeoff that will appear recurrently in the following test scenarios, regarding the choice of $\tau$: larger values of the time constant (both on the exponentially smoothed latency averages and the exponentially smoothed weights) guarantee less oscillations (due to reduced bandwidth of the system), but at the same time make the load balancer slower to react to changes in the controlled system’s conditions: as we have shown both in Section 5.3 and in Appendix A, the rise time of a first order LTI system is directly proportional to its time constant, hence higher values of $\tau$ make the system ‘slower’.

Configurations in Figures 5.6–5.7, where weight function #2 is applied, exhibit much narrower fluctuations compared to the other weight function: the width range is approximately one order of magnitude lower. This is in general a desirable behavior if the two backends have equivalent latency profiles, since sudden variation in the weights, hence in the load distributions, can compromise the stability of the overall system: an underloaded server could suddenly become too loaded to cope with all requests if all the downstream servers redirect their traffic at once. This in turn would render the server in question overloaded, and cause the traffic to be redirected to other server, thus generating unstable load oscillations. The weight oscillation frequency appears to be approximately two orders of magnitude lower than in the configurations where weight function #1 is used. This is behavior is aligned with our goal to obtain a system that behaves as round robin scheduler if the latency profile of the upstream servers are equivalent.

An additional note: we have repeated the same simulations using different values of $\sigma^2$ in the latency distributions: we have considered $\sigma^2 = \alpha \ast \mu$, with $\alpha \in \{0.1, 0.2, 0.5, 1, 2\}$ and we have found that the value of variance doesn’t affect the stability of weights. The graphs related to the results of this test scenario in function of the value of variance in latency distributions have been omitted.
5.5 Test scenario 3

In this scenario we simulate having two servers, one located on US west coast, one located in Europe. We achieve this result by imposing a fixed offset on the latency of the server that is further from us of approximately 250 ms. The latency distribution of the backend closer to us is \( Y_0 = \mathcal{N}(a_0 \cdot r + b_0, \sigma^2_0) \), and the furthest server has distribution latency \( Y_1 = \mathcal{N}(a_1 \cdot r + b_1, \sigma^2_1) \), with \( a_0 = a_1, b_0 = 0, b_1 = 250 \) and \( \sigma_0 = \sigma_1 \). We compare the performances of our load balancing algorithm based on the \( \varphi \) accrual failure failure detector with the performances of round robin scheduling, focusing on the aggregated latency of the whole system and the stability of weights. Once again, we consider six different exponentially smoothed sample latencies, with time constants \( \tau \in \{1 \text{ s}, 2 \text{ s}, 5 \text{ s}, 10 \text{ s}, 20 \text{ s}, 50 \text{ s} \} \). We set the value of the time constant for the exponential smoothing of weights of 2 seconds.

We handle starvation using the minimum weight strategy presented in Section 4.5: the weight of each instance never drops below a minimum value set to 20% of the equilibrium weight. In this specific case the equilibrium weight is \( w_{eq} = \frac{1}{\#\text{servers}} = \frac{1}{2} \), hence the minimum weight for each server is \( w_{min} = 0.1 \).

Results

Figures 5.8–5.9 and Figures 5.10–5.11 show the outcome of test scenario 3, applying respectively latency function #1 and weight function #2. Both figures represent the values of normalized weights over time, for a total running time of 1000 s. The exponential smoothing time constant on weights is consistently set to 2 s for all the scenarios, across all configurations.

Configurations in Figures 5.8–5.9, where weight function #1 is applied, exhibit relative fluctuations in the range ±10% of normalized weights (with respect to the equilibrium value \( w_0 \approx 0.23, w_1 \approx 0.77 \)). As expected, the time for weights to reach the equilibrium value is related to the value of \( \tau \): the larger \( \tau \), the longer time is needed for weights to settle, for the same reasons we have observed in Section 5.4 and Appendix A. More specifically we observe that weights tend to diverge so that the closest server is rewarded with a higher weight and the furthest server is penalized with a lower weight. Weights reach saturations, so the minimum-weight strategy that is triggered to avoid starvation.

Figure ?? represents the average latency value for the same configuration set that has been tested to produce Figure 5.8 this time using a round robin scheduling discipline. The average latency value over the whole 1000 s time frame is \( \mu_\varphi \approx 180 \text{ ms} \) with a \( \varphi \)-accrual-failure-detector-based load balancer,
Figure 5.8: Test scenario 3: average latency and weights for weight function #1 and $\tau \in 1 \text{ s}, 5 \text{ s}, 10 \text{ s}$. 
Figure 5.9: Test scenario 3: average latency and weights for weight function #1 and $\tau \in 20 \text{ s, 50 s, 100 s}$. 
5.5. TEST SCENARIO 3

Figure 5.10: Test scenario 3: average latency and weights for weight function #2 and $\tau \in 1 \text{s}, 5 \text{s}, 10 \text{s}$.
Figure 5.11: Test scenario 3: average latency and weights for weight function \#2 and $\tau \in \{20 \text{ s}, 50 \text{ s}, 100 \text{ s}\}$. 
whereas the average latency value using a round robin load balancer is $\mu_{rr} \approx 215\text{ms}$. This denotes a 16.3% improvement on the average latency value when using a $\varphi$-accrual-failure-detector-based load balancer over a round robin load balancer.

Configurations in Figures 5.10–5.11, where weight function #2 is applied, exhibit a slightly higher settling time, but no oscillations, due to the fact that weights reach saturation and the starvation-protection algorithm is triggered. Note the the speed at which weight converge to their final value is tunable by acting on the value of the scale constant $A$, in Equation 5.7: the higher $A$, the shorter the settle time. Weight function #2 tends to saturates quickly in the current scenario: a decrement to the weights is applied each time the average latency of a server is found to be higher than the global average value. In any setting such as the present scenario, where a fixed offset make one of the servers consistently slower than the average, saturation is inevitable.

Also in this scenario, we have repeated the simulations using different values of $\sigma^2$ in the latency distributions: we have considered $\sigma^2 = \alpha \times \mu$, with $alpha \in \{0.1, 0.2, 0.5, 1, 2\}$ and we have found that the value of variance doesn’t affect the stability of weights. The graphs related to the results of this test scenario in function of the value of variance in latency distributions have been omitted, as in the previous section.

## 5.6 Test scenario 4

In this scenario we simulate having two servers, both with the same latency distribution. One of the two servers suffers from a latency spike of the duration of 30 seconds, from time $t_0 = 60$ s to time $t_1 = 90$ s. The average latency function during the spike has the same slope as the average latency function in stationary conditions, but its offset is approximately 1000 ms. Also in this scenario, we consider six different exponentially smoothed sample latencies, with time constants $\tau \in \{1$ s, $2$ s, $5$ s, $10$ s, $20$ s, $50$ s$\}$. The value of the time constant for the exponential smoothing of weights of is consistently 2 seconds for each configuration. In this case our analysis is focused on:

- stability of weights: amplitude and frequency of weight oscillations;
- reactivity to the latency spike if function of weight function and latency exponential average time constant;
- median and 99-percentile latency of requests routed using latency-based load balancing vs requests routed using round robin.
Figure 5.12: Test scenario 4: average latency and weights for weight function #1 and \( \tau \in 1 \text{s}, 5 \text{s}, 10 \text{s} \).
Figure 5.13: Test scenario 4: average latency and weights for *weight function #1* and \( \tau \in 20 \text{ s}, 50 \text{ s}, 100 \text{ s} \).
Figure 5.14: Test scenario 4: average latency and weights for weight function #2 and $\tau \in 1 \text{s}, 5 \text{s}, 10 \text{s}$. 
Figure 5.15: Test scenario 4: average latency and weights for weight function #2 and $\tau \in 20$ s, 50 s, 100 s.
Results

Figures 5.12–5.13 and Figures 5.14–5.15 show the outcome of test scenario 4, applying respectively latency function #1 and weight function #2. Both figures represent the values of normalized weights over time, for a total running time of 1000 s. The exponential smoothing time constant on weights is consistently set to 2 s for all the scenarios, across all configurations; the scale constant $A$, (weight function #2, Equation 5.7) is consistently set to $A = 0.5$.

Configurations in Figures 5.12–5.13, where weight function #1 is applied, exhibit relative fluctuations in the range $\pm 5\%$ of normalized weights (with respect to the equilibrium value $w_0 = w_1 = 0.5$), except for Figure , where the time constant $\tau = 1$ s makes the weights less stable (oscillation range $\pm 10\%$ of stationary value). The value of $\tau$ has a crucial impact on the performances of the load balancing algorithm in non-stationary conditions: the adaptation of weights to the latency spike is fast enough to impact the measured overall average latency only for $\tau \in \{1 \text{ s}, 5 \text{ s}\}$; other values of $\tau$ exhibit a similar variation in the weights, delayed sufficiently to occur after the latency spike (hence, there is no visible effect on the average latency of the system). Weight saturation is not reached.

When using a time constant $\tau = 5$ s for the exponentially smoothed latency, the average value of latency for the duration of the spike — between time $t_0$ and $t_1$ — is $\mu_\varphi = 462.30$ ms; for comparison, the average latency using a round robin scheduling discipline is $\mu_{rr} = 537.20$ s, leading to a 14.1\% improvement. Most importantly, around 25\% of the requests are affected from the latency spike when using a $\varphi$-accrual-failure-detector-based load balancer, as opposed to 50\% of requests served by the slow server when using a round robin scheduling discipline; this is reflected by the value of the 75\textsuperscript{th} percentile of latency: $l_{\varphi,75\text{th}} = 845.14$ ms, $l_{rr,75\text{th}} = 1036.63$ ms, which corresponds to a 18.43\% improvement. Configurations in Figures 5.14–5.15, where weight function #2 is applied, exhibit relative fluctuations in the range $\pm 0.5\%$ of normalized weights (with respect to the equilibrium value $w_0 = w_1 = 0.5$): once again weight function #2 proves to be more reliable in terms of oscillations. Analogously to the configurations where weight function #1 is used, $\tau$ impacts the speed at which weights adapt to mutate conditions: only configurations with $\tau \in \{1 \text{ s}, 5 \text{ s}\}$ react to the latency spike timely; Weight saturation is reached for $\tau \in \{10 \text{ s}, 20 \text{ s}, 50 \text{ s}\}$: as stated in Section 5.5, weight function #2 tends to saturate more easily than weight function #1.

When using a time constant $\tau = 5$ s for the exponentially smoothed latency, the average value of latency for the duration of the spike — between time $t_0$ and $t_1$ — is $\mu_\varphi = 502.30$ ms; leading to a mere 6.1\% improvement over the average latency obtained with round robin scheduling, $\mu_{rr} = 537.20$ ms. The
5.7. TEST SCENARIO 5

In this test scenario we simulate a latency distribution mismatch: we first assume the actual latency distribution to be normal, but model it with a log-normal distribution in the load balancer. Secondly, we assume the actual latency distribution to be log-normal, but model it with a normal distribution in the load balancer. This test is only run against weight function #1, since it is the only one dependent on the choice of distribution, whereas weight function #2 only depends on the average value of latency, which is independent from the choice of the distribution.

Previously, in Section 4.4, we have debated whether the latency distribution observed in the test environment is most likely normal, or log-normal. In Appendix B we try to answer the question by means of a chi-square goodness-of-fit test. This test scenario is aimed at observing whether in practice choosing a normal distribution or a log-normal distribution leads to different performances. We consider a fixed value of the time constant for the exponentially smoothed latency average and variance $\tau = 5$ s. We adopt the same workloads and conditions analyzed in Sections 5.4–5.6. The evaluation parameters are the stability of weights (amplitude and frequency of weight oscillations under stationary conditions) and the overall reactivity of the load balancer.

The adapted version of weight function 1 when the latency distribution is modeled as a log-normal distribution is (from Equation 5.2):

$$w_i(t) = e^{-\frac{\tau_r}{\tau}} w_i(t-1) + (1 - e^{-\frac{\tau_r}{\tau}}) \frac{\mu_{\text{global}}}{\mu_{\text{global}}\sqrt{2\pi\sigma_i^2}} \int_{-\infty}^{\ln x - \mu_i} e^{-\frac{(\ln x - \mu_i)^2}{2\sigma_i^2}} \, dx \quad (5.20)$$

5.8 Conclusions

In all of the experimental scenarios described in this Section we have consistently observed that weight function #2 outperforms weight function #1 in terms of weight stability in stationary conditions. On the other hand, weight function #1 appears to be more reactive to dynamic load variations, as we have seen by simulating a sudden latency spike in Section 5.6.

The main parameters that can be tuned in order to adjust the performances of the load balancing algorithm are: a) the value of the time constant of
CHAPTER 5. EXPERIMENTAL RESULTS

Figure 5.16: Test scenario 5. Same setup as in Test 2, source of latency: normal; \( \tau = 10 \) s.

Figure 5.17: Test scenario 5. Same setup as in Test 3, source of latency: normal; \( \tau = 10 \) s.

Figure 5.18: Test scenario 5. Same setup as in Test 4, source of latency: normal; \( \tau = 10 \) s.
5.8. CONCLUSIONS

Figure 5.19: Test scenario 5. Same setup as in Test 2, source of latency: log-normal; $\tau = 10$ s.

Figure 5.20: Test scenario 5. Same setup as in Test 3, source of latency: log-normal; $\tau = 10$ s.

Figure 5.21: Test scenario 5. Same setup as in Test 4, source of latency: log-normal; $\tau = 10$ s.
the exponentially weighted means and variances $\tau_{in}$, b) the value of the time constant for the exponential smoothing of weights $\tau_{out}$, c) the weight function, and d) — for weight function #2 — the value of the scale constant $A$ (see Equation 5.7).

Longer time constant make the weights more stable and the system more resilient to noise, at the price of decreasing the overall reactivity of the system. The choice of time constants effectively affects the granularity of events (in terms of duration) to which the load balancer can react. A compromise can be reached by tuning sensibly the value of the time constant of exponentially weighted means and variances and the time constant used for the exponential smoothing of weights: since weights tend to exhibit high frequency oscillations for low values of $\tau_{in}$, we can reduce the oscillations with relatively low values of $\tau_{out}$, thus reasonably preserving reactivity while maintaining oscillations in a limited range. As mentioned in Section 5.4, oscillation are not desirable because they have the potential to make the whole system unstable.
Chapter 6

Future developments and conclusions

In this section we present an overall summary of potential future developments. As stated in the introduction, our work on load balancing does not deal with failover, despite failover being a tightly related topic. In Section 6.1 we present an approach to extend our load balancer with a failure detector.

Our approach to load balancing is specifically focused on reducing the average request-response latency of a replicated service, but doesn’t address tail-latency. Dean and Barroso deal with the topic of tail latency in [5], propose an analysis of the phenomenon in the context of high-fanout requests and an effective solution. We will sum up the findings of [Dean and Barroso] in Section 6.2 and discuss the applicability of the solution presented in [5] to the architecture of Spotify’s backend systems.

In Section 6.3 we provide an overview of our conclusions and wrap up the findings presented in this work.

6.1 Failover

Load balancing, failover, overload protection, and QoS differentiation are closely related aspects of traffic management. Load balancing is often used to implement failover — the continuation of a service after the failure of one or more of its components. Typically this works by keeping track of unhealthy components through a failure detector. When a component is suspected to be failed, it is removed from the pool of active components. Similarly, a component that comes back online is reinstated in the pool of active components and the load balancer begins to route traffic to the component again.

The load balancer interface described in Module 4.1 supports an event
of type \textit{UpdateServers} that takes a set of correct server processes as input, and updates the set of correct known server that the load balancer stores internally. Note that the reliable delivery property \textbf{PL1} is satisfied only if the set of server processes know to the load balancer is correct, i.e., every process in the set is correct. A way to make sure that this conditions is guaranteed, is to implement a weak\footnote{As opposed to Consensus-Based Group Membership.} group membership abstraction. A weak group membership algorithm would simply keep track of which processes are alive by means of a failure detector, and would notify the upper layer with an updated set of processes each time a process is suspected or that a process is added. Note that no notion of consistency or view-synchronization is implied: each instance of the weak group membership module keep tracks individually. Nonetheless, if an eventually perfect failure detector is used, all the group membership instances operating together would eventually converge to the same set of alive processes thanks to the strong completeness and eventual strong accuracy properties of the eventually perfect failure detector, provided that when a new process is added, all the group membership instances are notified.

### 6.2 Tackling tail latency

The research presented by \cite{Dean2012} is focused on the effect of high-tail latency on high-fan-out requests, how such effects are seen in production systems at Google and what mitigations can be applied to prevent occasional latency spikes from impacting the performances of a system and disrupting the user experience.

Temporary high-latency episodes, that are usually negligible in moderate-size systems, may come to dominate overall service performance at large scale. This effect is especially pronounced in highly parallel systems, where the result of a single operation is obtained by fanning out a single request in a large number of sub-requests from a root server to a set of leaf servers and merging back responses. Variability in request-reply latency exists for many reasons:

- \textit{not all requests are made equal}: the source of latency might be inherent in the particular request. In other words, some requests might effectively need more resources than others to be processed;

- \textit{shared local and global resources}: machines might be shared by different applications contending for shared resources; different requests within the same application might contend for resources and, last but not least,
applications running on different machines might contend for global resources, such as network switches and shared file systems;

- **background activities and periodic tasks**: background daemons, such as those used for handling configurations (e.g. puppet-agent) or monitoring (e.g. collectd) may use limited resources on average, but they can generate latency hiccups when they are scheduled; the same considerations apply to periodic tasks, such as those run under the supervision of chron, or garbage collection in the Java virtual machine;

- **energy management**: power-saving modes in many devices can add substantial latency when moving from inactive to active modes; think of a hard drive in power save mode that needs to spin up before it can read or write data.

A simple way to curb latency variability is to issue the same request to multiple replicas and use the results from whichever replica responds first. Dean and Barroso name this kind of requests *hedged requests* because a client first sends one request to the most appropriate replica, but falls back on sending a duplicate request if it doesn’t receive a reply after some brief delay: tuning the delay appropriately allows to exploit the best trade off between containing tail latency aggressively and increasing the load on servers. In order to prevent multiple servers from executing the same request unnecessarily, duplicated requests can be tagged with each other’s IDs (what Dean and Barroso call *tied requests*). As soon as the first request is processed, it cancels the second request. There is still a window of uncertainty, when the cancelation message is in-flight and both requests could still be executed. This can be mitigated by sending out the duplicated request one network round-trip time after the original request, so that it can still be canceled even if queues on the receiving sides are completely empty.

The techniques that Dean and Barroso propose are mostly applicable to read-only operations, which covers a broad range of data-intensive services. Tolerating latency variability for operations that mutate state consistently is somewhat easier for at least a couple of reasons: a) updates can be performed off the critical path, after showing the effect to the user to give the impression of responsiveness, b) consistent updates performed on multiple replicas with a quorum-based algorithm like Paxos are inherently tail-tolerant: when data are duplicated across $N$ replicas, only the fastest $\frac{N}{2} + 1$ must commit in order for the write operation to succeed.
Safety and idempotence of requests

The main obstacle in applying a similar strategy generically for Hermes requests is that the execution semantics of requests is completely obscure in terms of safety. RFC 2616, “Hypertext Transfer Protocol – HTTP/1.1”, defines safety as the property of certain operations that can be applied to a system without changing its state. An operation is safe if it doesn’t have side effects, and for said reasons is also called *nullipotent* [8, Section 9.1]. Furthermore, a request is said to be idempotent if, aside from error or expiration issues, the side-effects of \( N > 0 \) identical requests is the same as for a single request; hence, safe requests are a subset of idempotent requests. The Hypertext Transfer Protocol specifications mandates that methods GET and HEAD ought to be considered safe. PUT and DELETE are idempotent; analogous specifications apply to RESTful APIs.

Unlike HTTP or RESTful APIs, Hermes requests have user-defined verbs, that might occasionally mimic the same methods defined in HTTP, but offer no guarantee with respect to the safety and idempotence of requests. Since said constraints are necessary in order to be able to duplicate requests, implementing *hedged requests* as part of the Hermes framework is error-prone and might potentially lead to buggy services, if for some reason a method that was previously implemented to have no side-effects is modified so that it doesn’t satisfy safety any longer. In alternative to providing *hedged requests* as part of the Hermes framework, specific self-contained services that have direct control over their dependencies might choose to implement request duplication in a less generic way, in order to reduce the risk of misuse.

6.3 Conclusions

This master thesis explores the possibility of building a load balancing algorithm based on request-reply latency. Request-reply latency is a crucial aspect that affects the experience of Spotify’s users, as such a major focus of this report is investigating the suitability of request-reply latency as a load index. Systems that respond to user actions quickly (within 100 ms) feel more fluid and natural to users than those that take longer [1]. In Section 4.1 we have presented different load indices used both in load balancing and job scheduling — two different aspects of the same problem — together with an overview of the conclusions presented in [7] regarding the properties of effective load indices.

Modeling the load in a distributed system is a complex task due to the presence of multi-bottlenecks that arise from interactions between various sys-
tem components [13]: individual components might show below-saturation resource utilization, but the overall throughput remains limited in spite of the availability of more resources. If latency alone could be used as a load index it wouldn’t be necessary to model resource usage in every system and keeping into account non-linearities that arise from distributed bottlenecks.

In order to evaluate the reliability of request-reply latency as a load index, we sample latency distributions in function of a known load indicator for a system, both in a controlled ad-hoc test environment and in production. We compute the Pearson correlation coefficient for both data sets and found that a strong evidence of correlation exists. As a future development of this thesis, it would be interesting to go beyond correlation by investigating causation and building a theoretical model that would allow to delimit the scope of validity of latency-load correlation. The foundations for such development have been laid in [18].

Spotify’s backend is made up by a constellation of different services, each of which is embodied by different replicas. A component known as access point proxies client requests to a specific replica of the appropriate service; similarly, when two services communicate with each other, the initiator of the communication selects one of the replicas of the recipient service. The choice of the most suitable replica to forward a request to is crucial in order to maximize reliability and performances.

We prove that latency can be reliably used as a load index by construction. We propose a load balancing algorithm based on a modified $\varphi$ accrual failure detector that models the latency behaviour of each service replica and routes requests by predicting which replica will offer the fastest reply, in analogy to what a $\varphi$ accrual failure detector does to predict that an heartbeat from the monitored process will be received. By means of experimental evaluation, we show that such latency-prediction routing algorithm outperforms naive round robin routing in terms of overall average latency of the system, while proving extremely stable and tolerant to spurious noise. We simulate the behavior of our load balancing algorithm both in stationary and dynamic conditions, and discuss the choice of parameters that provides the best trade-off between reactivity and stability of the system.
Appendix A

Exponentially weighted mean and variance

In this Appendix we derive formulae for incremental calculation of the standard deviation of exponentially weighted moving average, starting from the known formula to incrementally compute the mean. It is crucial to us to be able to compute the mean and variance of a sample efficiently, since this action is repeated for a good proportion of all requests. The algorithm presented here guarantees O(1) complexity for value added to the sample. We show that the exponentially weighted moving average behaves as a first order LTI system, and we analyze its time-domain behavior. Let us consider the exponentially weighted moving average:

\[ \mu_n = \mu_{n-1} + \alpha(x_n - \mu_{n-1}) \]  \hspace{1cm} (A.1)

Let \( a = 1 - \alpha \):

\[ \mu_n = \mu_{n-1} + \alpha(x_n - \mu_{n-1}) \]  \hspace{1cm} (A.2)
\[ \mu_n = (1 - \alpha)\mu_{n-1} + \alpha x_n \]  \hspace{1cm} (A.3)
\[ \mu_n = a\mu_{n-1} + (1 - a)x_n \]  \hspace{1cm} (A.4)

Expanding the inductive definition of the mean we obtain:

\[ \mu_n = a\mu_{n-1} + (1 - a)x_n \]  \hspace{1cm} (A.5)
\[ = a^2\mu_{n-2} + a(1 - a)x_{n-1} + (1 - a)x_n \]  \hspace{1cm} (A.6)
\[ = a^3\mu_{n-3} + a^2(1 - a)x_{n-2} + a(1 - a)x_{n-1} + (1 - a)x_n \]  \hspace{1cm} (A.7)
\[ = \ldots \]  \hspace{1cm} (A.8)
\[ \mu_n = a^nx_0 + \sum_{i=1}^{n} a^{n-i}(1 - a)x_i \]  \hspace{1cm} (A.9)
We can explicitly derive the weights:

\[ w_{n,0} = a^n \]  \hspace{1cm} (A.10)

\[ w_{n,i} = a^{n-1}(1-a) = 1 - a^n, \text{ for } 1 \leq i \leq n \]  \hspace{1cm} (A.11)

\[ w_{n,n} = 1 - a = \alpha \]  \hspace{1cm} (A.12)

Note that \( w_{n,n} \) is independent of \( n \). Furthermore:

\[ \lim_{n \to \infty} w_{n,0} = \lim_{n \to \infty} a^n = 0 \hspace{1cm} \forall a \in [0, 1) \]  \hspace{1cm} (A.13)

We can see that the sum of all weights \( W_n = 1, \forall n \) by summing the geometric series:

\[ \sum_{i=1}^{n} a^{n-1} = \sum_{j=0}^{n-1} a^j = \frac{1 - a^n}{1 - a} \]  \hspace{1cm} (A.14)

From A.11 we have:

\[ \sum_{i=1}^{n} w_{n,i} = \sum_{i=1}^{n} a^{n-1}(1-a) \]  \hspace{1cm} (A.15)

Substituting A.14 in A.15

\[ \sum_{i=1}^{n} w_{n,i} = (1-a) \sum_{i=1}^{n} a^{n-1} \]  \hspace{1cm} (A.16)

\[ = (1-a) \frac{1 - a^n}{1 - a} = 1 - a^n \]  \hspace{1cm} (A.17)

Thus

\[ W_n = \sum_{i=0}^{n} w_{n,i} = w_{n,0} + \sum_{i=1}^{n} w_{n,i} = a^n + (1 - a^n) = 1 \]  \hspace{1cm} (A.18)

The old and the new weights are consistent because:

\[ \frac{w_{n,j}}{\sum_{i=1}^{n-1} w_{n,i}} = \frac{aw_{n-1,j}}{\sum_{i=1}^{n-1} w_{n-1,i}} = \frac{w_{n-1,j}}{\sum_{i=1}^{n-1} w_{n-1,i}} \]  \hspace{1cm} (A.19)

Given a random variable \( x \), The variance can be computed from the definition:

\[ \sigma_n^2 = E_n((x - \mu_n)^2) \]  \hspace{1cm} (A.20)

\[ = E_n(x^2 - 2xE_n(x) + E_n(x^2)) \]  \hspace{1cm} (A.21)

\[ = E_n(x^2) - 2E_n(x)E_n(x) + (E_n(x))^2 \]  \hspace{1cm} (A.22)

\[ = E_n(x^2) - (E_n(x))^2 \]  \hspace{1cm} (A.23)

\[ = E_n(x^2) - \mu^2 \]  \hspace{1cm} (A.24)
For a generic $f(x)$ the incrementally-computable value of $E(f(x))$, where each $x_i$ is exponentially weighted, is:

$$E_n(f(x)) = E_{n-1}(f(x)) + \frac{w_{n,n}}{W_n}(f(x_n) - E_{n-1}(f(x))) \quad (A.25)$$

$$E_n(f(x)) = E_{n-1}(f(x)) + \alpha(f(x_n) - E_{n-1}(f(x))) \quad (A.26)$$

We can express weighted variance with another notation:

$$\sigma^2 = \frac{1}{W_n} \sum_{i=1}^{n} w_i(x_i - \mu)^2 = \frac{1}{W_n} \sum_{i=1}^{n} w_ix_i^2 \quad (A.27)$$

Let $S_n = W_n\sigma^2$

$$S_n = \sum_{i=1}^{n} w_ix_i^2 - W_n\mu^2 \quad (A.29)$$

By computing the value of $S_n$ in function of $S_{n-1}$ it can be shown that:

$$S_n = S_{n-1} + w_n(x_n - \mu_{n-1})(x_n - \mu_n) \quad (A.30)$$

Let $f(x) = x^2$. Substituting $f(x)$ in $A.26$:

$$\mu_n = a\mu_{n-1} + (1-a)x_n \quad (A.31)$$

The transfer function of $A.31$ can be computed through its Z-transform:

$$\mu(z) = Z\{\mu_n\} = az^{-1}\mu(z) + (1-a)X(z) \quad (A.32)$$

$$H(z) = \frac{\mu(z)}{X(z)} = \frac{1-a}{1-az^{-1}} \quad (A.33)$$

We can obtain the step response by multiplying the transfer function $H(z)$ by the Z-transform of the Heaviside step function $X(z) = \frac{1}{1-z^{-1}}$:

$$Y(z) = H(z)X(z) = \frac{1}{1-z^{-1}} \frac{1-a}{1-az^{-1}} \quad (A.34)$$

$$y_n = Z^{-1}(Y(z)) = \frac{(1-a)(a^{n+1} - 1)}{a - 1} \quad (A.35)$$

$$= 1 - a^{n+1} \quad (A.36)$$

Note that the exponent $n + 1$ depends on the fact that we have considered $n \geq 0$. We can obtain a first-order LTI-system-like response by setting $a = e^T$. 

---

$^1$ $S_n$ can be computed in function of $S_{n-1}$ by computing the value of $S_n - S_{n-1}$, substituting both values with $A.29$ and moving $S_{n-1}$ to the left member.
Furthermore, let time $t = nT_s$, where $T_s$ is the sample period, we can rewrite [A.37] as:

$$y(t) = 1 - e^{(\frac{T_s}{\tau}t+1)}$$  \hspace{1cm} (A.38)

$$= 1 - e^{-\frac{t}{\tau}+1}$$  \hspace{1cm} (A.39)

by applying the substitution $a = e^{-\frac{T_s}{\tau}}$, where $\tau$ is our selected time constant in s and $T_s$ is the sample period, with $\tau > 0$. The system’s transfer function [A.33] with no zeros and a single pole in $z = a$ represents a bass-pass filter, which appears evident from the Bode plot of a system with $\tau = 0.3$ s (Figure A.1) and its step response in Figure A.1.
Figure A.1: Bode plot of a first order LTI system with $\tau = 0.3$ s, obtained from A.33 with $T_s = 0.1$ s and $\tau = 0.3$ s. The dotted line represents the cutoff angular frequency.
Figure A.2: Step response of a first order LTI system with obtained from A.33 with $T_s = 0.1 \text{ s}$ and $\tau = 0.3 \text{ s}$.
Appendix B

Latency distribution

In this section, we conduct a chi-square goodness-of-fit test in order to determine whether the most suitable distribution to represent the latency samples measured in the test environment is a normal or a log-normal distribution.

B.1 Experimental setup

The test environment is made up by two separate virtual machines, hosted in the same datacenter: a load generator runs on one, and a test instance of the Playlist backend runs on the other. The test instance of the Playlist backend is supported by a small shared Cassandra test cluster made up by three machines. The load generator is configured to emit a random query mix at a fixed rate. Latency values are bucketed in 15 ms buckets; the result of the experiment is represented in Figure B.1.

B.2 Chi-Square Goodness-of-Fit Test

The goodness-of-fit chi-square test is used to test if a sample of data is extracted from a population with a specific distribution. The chi-square goodness-of-fit test is applied to binned data. The chi-square test is defined for the hypothesis:

- $H_0$: the data follow a specified distribution.

- $H_a$: the data do not follow the specified distribution.
For the chi-square goodness-of-fit computation, the data are divided into $k$ bins and the test statistic is defined as:

$$
\chi^2 = \sum_{i=1}^{k} \frac{(O_i - E_i)^2}{E_i}
$$  \hspace{1cm} (B.1)

where $O_i$ is the observed frequency for the $i^{th}$ bin and $E_i$ is the expected frequency for the $i^{th}$ bin. Given the cumulative distribution $F(x)$ of the expected probability distribution defined in $H_0$, the expected frequency is for the $i^{th}$ bin is:

$$
E_i = N \cdot (F(Y_u) - F(Y_l))
$$  \hspace{1cm} (B.2)

where $Y_l$ is the lower limit for class $i$, and $N$ is the sample size. The chi-square test is sensitive to the choice of bins. There is no optimal choice for the bin width, but most reasonable choices should produce similar — not identical — results. For the chi-square approximation to be valid, the expected frequency should be at least 5 or 10. In order to be valid, this test should be applied to large samples.

The test statistic follows, approximately, a chi-square distribution with $(k - c)$ degrees of freedom where $k$ is the number of non-empty cells and $c$ is the number of estimated parameters for the distribution + 1. For example, for a normal or log-normal distribution, we take into account the location parameter $\mu$ and the scale parameter $\sigma^2$, hence $c = 3$. Therefore, the hypothesis that the data are from a population with the specified distribution can be rejected if

$$
\chi^2 > \chi^2_{1-\alpha, k-c}
$$  \hspace{1cm} (B.3)

where $\chi^2_{1-\alpha, k-c}$ is the chi-square critical value with $k - c$ degrees of freedom and significance level $\alpha$. In order to reject the null hypothesis we consider $\alpha = 0.05$

**Normal distribution**

Let us assume the following hypothesis:

- $H_0$: the data follow a normal distribution.
- $H_a$: the data do not follow a normal distribution.

The parameters of the normal distribution in question are computed as:

$$
\hat{\mu} = \frac{1}{N} \sum_{i=1}^{N} x_i \hspace{1cm} \hat{s}^2 = \frac{1}{N-1} \sum_{i=1}^{N} (x_i - \hat{\mu})^2
$$  \hspace{1cm} (B.4)
In our case, $\hat{\mu} = 347.37$, $s^2 = 506.27$. The samples have been partitioned in 15 ms wide bins. We adopt a significance level $\alpha = 0.05$, and $k - c = 31 - 3 = 29$. The critical value is $\chi^2 > \chi^2_{1-\alpha, k-c} = 41.34$, hence we can reject $H_0$ if $\chi^2 > 41.34$. The value of $\chi^2$ assuming a normal distribution $\mathcal{N}(347.37, 506.27)$ is $\chi^2 = 39.60$, thus we accept $H_0$ and reject the alternative hypothesis.

**Log-normal distribution**

Let us assume the following hypothesis:

- $H_0$: the data follow a log-normal distribution.
- $H_a$: the data do not follow a log-normal distribution.

The parameters of the normal distribution in question are computed as:

$$\hat{\mu} = \frac{1}{N} \sum_{i=1}^{N} \ln x_i \quad \hat{\sigma}^2 = \frac{1}{N} \sum_{i=1}^{N} (\ln x_i - \hat{\mu})^2$$  \hspace{1cm} (B.5)
In our case, $\hat{\mu} = 5.42$, $s^2 = 0.47$. The samples have been partitioned in 15 ms wide bins. We adopt a significance level $\alpha = 0.05$, and $k - c = 31 - 3 = 29$. The critical value is $\chi^2 > \chi^2_{1-\alpha, k-c} = 41.34$, hence we can reject $H_0$ if $\chi^2 > 41.24$. The value of $\chi^2$ assuming a normal distribution $\ln N(5.42, 0.47)$ is $\chi^2 = 37.89$, thus we accept $H_0$ and reject the alternative hypothesis.

### B.3 Conclusions

Both chi-square goodness-of-fit tests lead to accepting the null hypothesis $H_0$, so the sample latency distribution could either follow a normal distribution or a log-normal distribution. Note that due to the constraint on the minimum expected frequency, the chi-square test underestimates the prominence of the distribution tail. Also, note that we have only considered classes of the distribution greater where the lower bound is greater than 0, considering that latency is by definition greater or equal than 0.
Bibliography


