



Assignment of master's thesis

Title:	Implementation of a Service Discovery for Enterprise Platform
Student:	Bc. Josef Havelka
Supervisor:	Ing. Marek Suchánek, Ph.D. et Ph.D.
Study program:	Informatics
Branch / specialization:	Software Engineering
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2024/2025

Instructions

Service discovery is an essential element of service-oriented architecture and distributed systems in general. There are currently various services, technologies, tools, and libraries available and it becomes challenging to select the right one. The goal of this thesis is to research existing solutions and propose replacement of current implementation of service discovery used by the specified enterprise customer.

- Describe briefly the principles of service discovery in distributed systems, related patterns and theorems, and other relevant topics for the thesis.
- Analyze and describe the current architecture of the service discovery used by the customer and summarize the requirements.
- Research relevant technologies, tools and libraries (such as HashiCorp Consul, HashiCorp Serf, Netflix Eureka, ZooKeeper).
- Design the architecture for the new implementation of the discovery service.
- Implement the solution prototype based on the design with use of selected technologies, test and document the solution properly.
- Demonstrate the functionality of the prototype, evaluate it and propose future development.

Master thesis

IMPLEMENTATION OF A SERVICE DISCOVERY FOR ENTERPRISE PLATFORM

Josef Havelka

Czech Technical University in Prague, Faculty of Information Tech-
nology
Department of Software Engineering
Supervisor: Ing. Marek Suchánek Ph.D. et Ph.D.
May 7, 2024

Czech Technical University in Prague
Faculty of Information Technology

© 2024 Josef Havelka. All rights reserved.

This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Link to the thesis: Josef Havelka. *Implementation of a Service Discovery for Enterprise Platform*.
Master thesis. Czech Technical University in Prague, Faculty of Information Technology, 2024.

Contents

Acknowledgements	viii
Declaration	ix
Abstract	x
1 Introduction	1
1.1 Company	1
1.1.1 Team	1
1.2 Summary	1
1.2.1 Motivation	1
1.2.2 Goal	1
1.2.3 Steps	2
1.2.4 Results of the thesis	2
1.2.5 Conclusion	2
2 Theoretical background	3
2.1 Requirements prioritisation MoSCoW	3
2.2 Key-value store	3
2.3 Failure tolerance and high availability	3
2.4 Microservice architecture	4
2.4.1 Pros	4
2.4.2 Cons	4
2.5 Load balancing	5
2.6 Service discovery	5
2.7 Distributed systems	5
2.7.1 Gossip protocol	6
2.7.2 Leader-follower pattern	6
2.7.3 Consistency models	6
2.7.4 CAP theorem	6
2.7.5 Consensus algorithms	7
2.7.6 Raft algorithm	8
2.8 Linux container	9
3 Current service discovery architecture	11
3.1 Service definition and meaning	11
3.2 Actors	11
3.2.1 Serf agent	11
3.2.2 AB agent	12
3.2.3 Distributed store	12
3.2.4 Business applications	12
3.3 Workflow	12

4	Requirements	15
4.1	Main focus	15
4.2	CAP theorem	16
4.3	Pricing	16
4.4	Java API	16
4.5	Maximal size of service registry	16
4.6	Maximal size of one entry	16
4.7	Documentation	16
4.8	Subscription vs repeatedly polling the data	17
4.9	GUI for admin	17
4.10	Data revision history	17
5	Comparison of existing service discovery solutions	19
5.1	Existing solutions	19
5.1.1	Consul	19
5.1.2	Netflix Eureka	19
5.1.3	EtcD	21
5.1.4	ZooKeeper	21
5.1.5	Redis	22
5.2	Comparison	24
5.2.1	Main focus	24
5.2.2	Comparison based on CAP theorem	25
5.2.3	Pricing	25
5.2.4	Implemented the Java API client	26
5.2.5	Subscription vs repeatedly polling the data	26
5.2.6	Maximal reliable size of the storage	26
5.2.7	Maximal size of one database entry	27
5.2.8	Quality of the documentation	27
5.2.9	Has a GUI?	27
5.2.10	Has data revision history?	27
5.3	Conclusion	28
6	Consul analysis	29
6.1	Philosophy	29
6.2	Use cases	30
6.3	Architecture	30
6.4	Fault tolerance	30
6.5	Consistency modes	32
6.5.1	Default mode	32
6.5.2	Consistent mode	32
6.5.3	Stale mode	33
6.5.4	Consistency mode per HTTP request	33
6.6	Multiple datacenters	34
6.6.1	WAN federation	34
6.6.2	WAN gossip pool	34
6.7	Deployment	36
6.7.1	Hashicorp cloud platform	36
6.8	Performance fine-tuning	36
6.8.1	Anti-entropy	36
6.8.2	Agent request caching	36
6.9	Reactive environment	37
6.9.1	Watches	37

6.9.2	Templates	37
6.10	Conclusion	38
7	Design	41
7.1	Using Consul service catalogue	41
7.2	Using Consul for health checking	41
7.3	Using Consul key-value store and other features	44
8	Implementation	45
8.1	Start	45
8.2	Stop	46
8.3	Configuration management	47
8.4	Health checking	47
8.5	Integration with other systems using Java API client	49
8.5.1	Register service	49
8.5.2	Deregister service	50
8.5.3	Subscribe for a service	50
9	Demonstration of the prototype	55
9.1	Configuration for production environment	55
9.2	Deployment schema	55
9.3	Consul GUI	57
9.4	Fault tolerance and stale mode testing	60
9.4.1	Testing scenario	60
9.4.2	Results	61
9.5	Performance measurements	61
9.5.1	Scraping telemetry with Prometheus	61
9.5.2	Test scenario	62
9.5.3	Results	62
10	Conclusion	65
10.1	Future work	65
	Contents of the enclosed media	73

List of Figures

2.1	Replicated log with finite state machine [14]	8
3.1	Current architecture	13
5.1	Consul architecture [35]	20
5.2	Netflix Eureka architecture [12]	21
5.3	Zookeeper architecture [32]	22
5.4	Redis architecture [41]	23
6.1	Consul architecture [35]	31
6.2	Consul default mode [42]	32
6.3	Consul consistent mode [42]	33
6.4	Consul stale mode [42]	34
6.5	Consul cross datacenter forwarding [72]	35
6.6	Consul cross datacenter gossip [72]	35
6.7	Consul long polling [77]	37
7.1	The first phase – Using Consul service catalogue	42
7.2	The second phase – Using Consul for health checking	43
9.1	Prototype deployment schema	56
9.2	Consul GUI nodes	57
9.3	Consul GUI services	58
9.4	Consul GUI service detail	59
9.5	Memory allocation in MB	62
9.6	Leader server response times	63
9.7	Count of registration calls	64

List of Tables

4.1	Requirements prioritisation	15
5.1	Comparison of existing frameworks suitable for implementing the service discovery	24
5.2	Java API client comparison	26
5.3	Maximal size of one database entry	27
5.4	GUI comparison	27

List of Listings

6.1	agent-config.json script watch example [52]	38
6.2	agent-config.json HTTP watch example [52]	38
6.3	Find address template example find_address.tpl [79]	38
6.4	Activate the template [79]	38
6.5	Put the value to the corresponding key [79]	38
6.6	Showing content of the rendered file [79]	38
8.1	Consul starting	46
8.2	Consul stop	47
8.3	Consul configuration template file example	48
8.4	Consul configuration management	48
8.5	Consul health checking	49
8.6	Consul register service	51
8.7	Consul deregister service	52
8.8	Consul subscribe for services	53
9.1	Performance fine-tuning	56
9.2	Enabling the GUI	57
9.3	Commands for fault tolerance testing	60
9.4	Consul telemetry with Prometheus	61
9.5	Prometheus download data	62

I want to thank my supervisor Ing. Marek Suchánek, Ph.D. et Ph.D. for his invaluable guidance and insightful feedback throughout the development of the thesis. His prompt responses to my questions and openness to discussions significantly contributed to the success of my work.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis. I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Czech Technical University in Prague has the right to conclude a licence agreement on the utilization of this thesis as a school work pursuant of Section 60 (1) of the Act.

In Prague on May 5, 2024

.....

Abstract

The thesis aims to identify an optimal service discovery solution to replace the existing one in the current implementation by the real-world enterprise customer. The thesis explores service discovery and distributed systems, beginning with a foundational understanding of relevant theory. Abstract requirements are gathered from the company team and refined through research on existing frameworks. Concurrently, the current implementation is studied to inform the requirements. Following the finalisation of requirements, a comparative analysis of existing solutions guides the selection of the most appropriate technology. Finally, a solution is designed and a prototype developed to showcase its functionality. Towards the conclusion of the thesis, recommended preparatory steps are outlined before initiating the current implementation's refactoring process.

Keywords service discovery, distributed systems, comparative analysis, technology selection, solution design, prototype development, HashiCorp Consul

Abstrakt

Cílem této práce je nalézt optimální řešení pro service discovery, které nahradí existující implementaci u reálného podnikového zákazníka. Práce zkoumá service discovery a distribuované systémy, začíná základním porozuměním příslušné teorie. Abstraktní požadavky jsou shromážděny od firemního týmu a upřesněny prostřednictvím rešerše existujících řešení. Současně je studována stávající implementace, která pomáhá formulovat požadavky. Po finalizaci požadavků následuje komparativní analýza existujících řešení, která usměřuje výběr nejvhodnější technologie. Nakonec je navrženo řešení a vyvinut prototyp, který ukazuje jeho funkčnost. Ke konci práce jsou doporučeny přípravné kroky před zahájením refaktorizace stávající implementace.

Klíčová slova service discovery, distribuované systémy, komparativní analýza, výběr technologie, návrh řešení, implementace prototypu, HashiCorp Consul

Glossary

- AB** AB is an anonymised placeholder name for the real company for which the thesis is written. 1, 2, 11, 12, 19, 41, 45–47, 50, 55, 62
- AP** Available and Partition tolerant. 25, 33
- Apache Curator** Apache Curator is a Java library that provides a high-level API and utilities for working with Apache ZooKeeper [1]. 22
- Apache ZooKeeper** ZooKeeper is an open-source distributed coordination service designed for maintaining configuration information, naming, providing distributed synchronisation, and group services in large-scale distributed systems [2]. 8, 21, 22, 25–28
- API** Application Programming Interface. 4, 16, 21, 26, 27, 29, 41, 45, 46, 49, 61
- AWS** Amazon Web Services. 19, 28
- CAP theorem** The CAP theorem states that in a distributed system, it is impossible to simultaneously achieve consistency, availability, and partition tolerance, and a trade-off must be made between these three characteristics [3]. 6, 16, 25, 33
- Consul** Consul is a software tool designed for service discovery, configuration management, and maintaining the availability and reliability of distributed systems [4]. 2, 19, 21, 22, 24–26, 28–30, 32, 33, 36–39, 41, 44–50, 55–57, 60–63, 65
- CP** Consistent and Partition tolerant. 16, 21, 25, 28, 33
- CPU** Central Processing Unit. 9, 36
- CSP** Communicating Sequential Processes. 29
- CTO** Chief Technology Officer. 1, 15
- DevOps** DevOps is a collaborative approach that bridges the gap between development and operations teams, emphasising automation, communication, and shared responsibility to streamline software delivery. It aims to improve agility, efficiency, and reliability throughout the software development lifecycle [5]. 1, 5, 15, 45
- Dos** A DoS (Denial of Service) attack is when a malicious actor floods a system, network, or service with excessive traffic or requests, rendering it unavailable to legitimate users [6]. 5
- Etcd** etcd is an open-source distributed key-value store designed for storing and managing configuration data, feature flags, and other distributed system metadata [7]. 21, 24–28
- FSM** Finite State Machine. 8
- Git** Git is a distributed version control system used for tracking changes in software development, facilitating collaboration among developers, and managing code revisions efficiently [8]. 45, 47

- Go** Go, or Golang, is a statically typed and compiled programming language developed by Google [9]. 29, 37
- gRPC** Google Remote Procedure Call. 11, 12, 30, 55
- GUI** Graphical User Interface. 17, 27, 28, 55, 57
- HashiCorp** HashiCorp is a software company that provides infrastructure automation software to help organisations manage and secure their infrastructure in a dynamic and scalable way. The company is known for developing a suite of open-source tools and commercial products that facilitate various aspects of infrastructure management, including provisioning, security, networking, and application deployment [10]. 2, 11, 19, 25, 29, 30, 36, 38, 65
- HCP** HashiCorp Cloud Platform. 36
- HTTP** Hypertext Transfer Protocol. 4, 21, 25, 26, 30, 32, 33, 36, 37, 45, 47, 55, 60, 61
- JSON** JavaScript Object Notation. 47
- LXC** Linux Containers. 55, 62
- MoSCoW** MoSCoW prioritisation categorises project requirements into four levels: Must-have, Should-have, Could-have, and Won't-have, helping teams focus on critical needs first [11]. 15
- Netflix Eureka** Netflix Eureka is an open-source service registry and discovery service primarily used for microservices-based architectures [12]. 19, 24–28
- PID** Process ID. 41, 46, 48
- Prometheus** Prometheus is an open-source systems monitoring and alerting toolkit that collects and stores metrics as time series data, initially developed at SoundCloud and maintained independently since joining the Cloud Native Computing Foundation in 2016 [13]. 61
- Raft algorithm** Raft is a consensus algorithm for managing replicated logs in a distributed system, ensuring fault tolerance and consistency among nodes [14]. 6, 8, 12, 15, 16, 19, 21, 25, 26, 30, 32, 46, 50, 55, 60
- RAM** Random Access Memory. 26, 61, 62
- Redis** Redis is an open-source, in-memory data structure store [15]. 22, 25–28
- REST** Representational State Transfer. 27, 41, 46, 49
- RPC** Remote Procedure Call. 41, 61
- SaaS** Software as a Service. 36
- Serf** Serf is a decentralised and highly available cluster membership and gossip protocol designed for providing fault tolerance and coordination in distributed systems [16]. 11, 12, 30
- TCP** Transmission Control Protocol. 30
- TTL** Time To Live. 19, 26
- UDP** User Datagram Protocol. 30
- ZAB** Zookeeper Atomic Broadcast Protocol. 7, 22, 25

Introduction

1.1 Company

The thesis is done for a medium-sized financial technology vendor in a real-world corporate setting. The anonymised placeholder AB will refer to the company.

1.1.1 Team

Whenever the author of this thesis refers to the team, he means a group of people across AB company teams that were selected to discuss the thesis and attend regular meetings. The team consists of a single CTO, a team leader of a back-end team and two DevOps specialists.

1.2 Summary

1.2.1 Motivation

In today's cloud-centric era, seamless deployment relies heavily on a reliable and user-friendly service discovery tool. Although building from scratch is an option, there are existing solutions available. The key lies in choosing the solution that best suits the needs of a company. Cloud-native service registry tools exist, but for legacy reasons, it may be beneficial to have a solution that can run on on-premise hardware and in a cloud environment.

Service discovery is inherently designed to be fault tolerant, given its critical role in any platform. Distributed systems are often used to address this need for resilience. However, while distributed systems effectively mitigate fault tolerance issues, they also introduce additional complexity.

A robust service discovery solution paves the way for the adoption of a microservice architecture, a path that many companies choose for its scalability advantages.

1.2.2 Goal

The primary objective is to find the optimal service discovery solution that will facilitate the transition of the current platform to a microservice architecture and cloud environment tailored to the AB company's needs.

1.2.3 Steps

The initial step involved getting familiar with service discovery, distributed systems, and related theory.

The first abstract requirements were obtained from the team, followed by research into existing frameworks to identify key aspects. The team discussed the acquired knowledge to refine the requirements. As part of the requirements gathering, it was helpful to study the current implementation.

After finalising all requirements, a comparison of existing solutions was conducted. This led to the selection of the most suitable technology.

Subsequently, a solution was designed, and a prototype was prepared to demonstrate the functionality.

1.2.4 Results of the thesis

The outcome of the thesis is a recommendation for a framework to use. Based on the research, the best-fitting solution is the HashiCorp Consul software.

The author of the thesis designed a service discovery solution that could replace the existing implementation inside the AB company's platform. He implemented a prototype using HashiCorp Consul, performed simple tests and suggested future performance tests. The prototype can serve as a core for implementing service discovery if the AB company chooses the recommended option.

1.2.5 Conclusion

The thesis focuses mainly on research, gathering requirements, and creating a prototype. These phases of the project are interconnected and influence each other. For example, discovering important aspects of a technology during research affects the discussion about requirements.

The prototype has been directly added to the AB company's source code. The thesis only reveals important code related to the prototype. Similarly, when describing the current design of the service discovery in the platform, there was a significant emphasis on not revealing confidential information.

The project is part of the AB company's long-term vision of moving to the cloud.

Theoretical background

This section introduces a theoretical background essential for understanding the upcoming content. The reader will understand the following discussions better by laying down these fundamental concepts.

2.1 Requirements prioritisation MoSCoW

The MoSCoW method prioritises requirements into four categories: Must Have, Should Have, Could Have, and Won't Have (this time). [11]

Must Have: These are essential requirements that must be implemented for the project to be considered successful. Failure to meet these requirements would result in the project being deemed a failure. [11]

Should Have: These requirements are important but not critical. They are considered to be of high value, but their absence would not necessarily result in project failure. [11]

Could Have: These requirements are desirable but not necessary for the project's core functionality. They are typically features or enhancements that would be nice to have if resources and time allow. [11]

Won't Have (this time): These are requirements that are explicitly identified as not being included in the current project scope. They may be considered for future phases or releases but are not part of the immediate project goals. [11]

2.2 Key-value store

Key-value store or database is a data structure similar to a map or dictionary in programming. It consists of a collection of key-value pairs with quick and efficient access to the value given the corresponding key. [17]

2.3 Failure tolerance and high availability

High availability strategies prioritise minimising system or application downtime by implementing measures such as load balancing, redundant servers, automatic failover, and monitoring systems to seamlessly transition between components in case of failure, aiming to maximise uptime and ensure uninterrupted operation despite potential disruptions. [18]

A basic explanation of fault tolerance is that if a component or sub-system fails, it does not disrupt the overall functioning of the system. Fault tolerance is a specific aspect of high availability, focussing on the system's ability to withstand and recover from component failures while maintaining continuous operation. [19]

2.4 Microservice architecture

The microservice architectural style is a method of building a single application by breaking it down into smaller services, each operating independently in its own process and communicating through lightweight mechanisms, commonly using HTTP resource APIs. [20] “*These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralised management of these services, which may be written in different programming languages and use different data storage technologies.*” [20]

2.4.1 Pros

2.4.1.1 Strong module boundaries

Microservices with strong module boundaries refer to an architectural approach where individual microservices are encapsulated with clear and defined interfaces, minimising dependencies and promoting independent development, deployment, and scalability. [21]

Another aspect involves distributing data ownership and management responsibilities across individual microservices, improving autonomy, flexibility, and resilience within the system architecture. [21]

2.4.1.2 Independent deployment

Each microservice operates independently, allowing changes and updates to be made without affecting other services. This isolation ensures that deployment is more straightforward, with fewer dependencies and a reduced risk of unintended consequences. Additionally, microservices typically have smaller codebases compared to monolithic applications, making them easier to manage and deploy. [21]

2.4.1.3 Technology diversity

The use of separate processes for each microservice allows the flexibility to employ any programming language or technology that is customised to the specific needs of each service. This versatility enables teams to optimise for performance and scalability while mitigating the risk of technology lock-in. [21]

2.4.2 Cons

2.4.2.1 Distribution

Distributed software, including microservices, introduces complexities such as performance degradation due to slow remote calls and concerns about reliability and consistency. Remote calls can accumulate latency, particularly in systems with multiple service interactions, necessitating strategies to mitigate this, such as increasing call granularity or using asynchronous communication. Additionally, the potential for remote call failures in distributed systems poses reliability challenges, requiring developers to design for failure and handle the consequences of network partitions or call failures. These issues underscore the careful consideration required before adopting distributed architectures like microservices, as they come with inherent costs and challenges. [21]

2.4.2.2 Eventual consistency

Eventual consistency ensures that data across distributed systems will eventually converge to a consistent state, allowing temporary inconsistencies during the propagation of updates [22]. Services may exhibit inconsistencies due to their decentralised nature, which can lead to variations in data states and behaviours across distributed components [21].

However, certain parts of the system may require consistency despite its decentralised nature, which prompts the use of various consensus algorithms to ensure agreement among distributed components [23].

2.4.2.3 Operational complexity

The transition to microservices can significantly increase operational complexity due to the proliferation of numerous small services, demanding robust continuous delivery practices and enhanced monitoring capabilities. While microservices offer the advantage of smaller, more manageable components, the complexity is often shifted to interservice connections, necessitating careful consideration of service boundaries to mitigate operational challenges. Managing this complexity requires new skills and tools, emphasising the importance of a DevOps culture for effective collaboration between development and operations teams. Without adequate training and cultural change, monolithic and microservice applications may impede their development and maintenance. [21]

2.5 Load balancing

“The goal of load balancing is improving the performance by balancing the load among these various resources (network links, central processing units, disk drives...) to achieve optimal resource utilisation, maximum throughput, maximum response time, and avoiding overload.” [24]

The improvement of overall performance is the primary goal of load balancing. However, other motivations could be improved security by mitigating the risk of a Dos attack. [6]

2.6 Service discovery

Service discovery is related to the architecture of microservices. Typically, it involves clients seeking connection details to access services centrally stored in a service registry. This registry is often replicated for fault tolerance. Services are required to register with the registry, and clients query it to obtain connection information. This process may also involve load balancing, where multiple instances of the same service are deployed for improved performance or high availability. In this setup, the service registry can either return a list of services that match the client’s requirements, allowing the client to choose (client-side architecture), or the registry server itself can select and return the service to the client (server-side architecture). The latter approach may improve performance, as the server can make informed decisions about service selection. [25]

2.7 Distributed systems

Distributed systems are a collection of independent nodes that cooperate to achieve a common goal. There are several reasons to use a distributed system. In general, the main reasons are to improve the system’s performance, scalability and reliability. [26]

2.7.1 Gossip protocol

A gossip protocol is a communication protocol used by distributed systems to spread information across the network in an efficient and decentralised manner. The gossip protocol operates similarly to how news spreads in a social network - from one node to another, gradually reaching the entire network. [27].

2.7.2 Leader-follower pattern

The leader-follower pattern, also known as the master-slave architecture, is a distributed computing design in which a node, known as the leader or master, is designated as the primary authority for decision making and coordination. Other nodes, called followers or slaves, typically replicate the actions of the leader and follow its directives. This pattern is commonly used in systems that require coordination and replication, such as databases, messaging systems, and distributed computing frameworks. [28]

In the context of data replication, the leader-follower pattern dictates that only the leader node is responsible for initiating updates to the shared data. Followers passively replicate the updates received from the leader, ensuring that they maintain a consistent copy of the data. This approach simplifies coordination and consistency management by centralising write operations to a single authoritative source. [28]

However, it is essential to prevent multiple nodes from acting simultaneously as leaders, as this can lead to inconsistencies and conflicts in the replicated data. When two or more nodes behave as leaders concurrently, it is referred to as a split-brain situation. To mitigate this risk, leader election mechanisms are often used to ensure that only one node assumes the role of the leader at any given time. These mechanisms use various algorithms and criteria to select a leader node, such as distributed consensus algorithms like Raft algorithm. [26]

2.7.3 Consistency models

There are many consistency models, most of which are described in detail in [29]. The most important models in the context of this thesis are sequential consistency and linearisability.

2.7.3.1 Sequential consistency

“The result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program. A multiprocessor satisfying this condition will be called sequentially consistent.” [30] In the context of distributed systems, all write operations are guaranteed to occur in identical order across each system.

2.7.3.2 Linearisability

Linearisability, also called a strong consistency model, ensures that all read and write operations behave as if executed on a single instance despite multiple replicas. [26].

“The behavior of this model is like the sequential consistency model. However, the order of the operations is set by the whole processes based on their time of occurrence.” [29]

2.7.4 CAP theorem

CAP theorem states the distributed system has three important properties [3].

C – Consistency – From the client’s perspective, it looks like one main server is performing all the tasks in an ordered way. [3]

A – Availability – Each request eventually receives a response. [3]

P – Partition tolerance – The systems continue to function even in case of network issues. [3]

When designing a distributed system, one cannot guarantee all three at once. Only two are possible. The combinations are as follows:

CA – The system blocks in case of a network partition, ensuring consistency and availability. (e.g. banking application) [3]

CP – When the system becomes available again, writes are processed (e.g., When writing into the Google Doc, the client has its local copy and saves only when the server is available.). [3]

AP – During a network partition, each node handles client requests without synchronising with other nodes. The system remains available, but may return stale data. (e.g. When using social media, users may come across older content, but it does not affect their ability to use the platform.) [3]

It is important to note that the following is just a theoretical grouping, and in practice, there are several strategies for handling partitions. The primary concern is the trade-off between consistency and potentially stale data. For better performance, one may accept a solution in which a distributed system is consistent in most cases, except for rare events. [23]

Some authors find that the definition of the CAP theorem needs to be more specific and propose using different terminology. “*Consistency refers to a spectrum of different consistency models (including linearisability and causal consistency), not one particular consistency model. When a particular consistency model such as linearisability is intended, it is referred to by its usual name. The term strong consistency is vague, and may refer to linearisability, sequential consistency or one-copy serializability.*” [31] However, the abstract notion of the dilemma between availability and consistency is sufficient to understand the thesis. The thesis uses consistency, strong consistency, and linearisability as synonyms.

2.7.5 Consensus algorithms

The problem of consensus is defined as an agreement between distributed systems on which data are correct at a given time. The goal is to ensure that all nodes in the system agree on a particular value or decision. [26]

2.7.5.1 Zookeeper custom algorithm

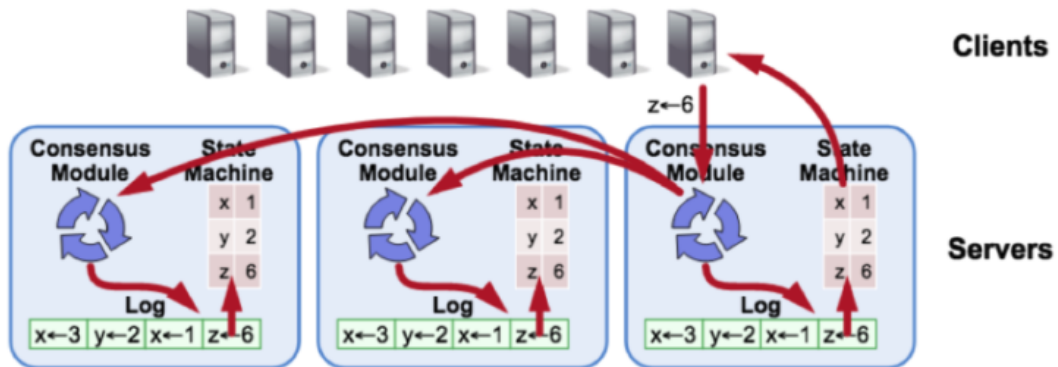
The Zookeeper Atomic Broadcast Protocol (ZAB) operates based on several key guarantees and properties:

Reliable Delivery – Messages sent by one server are eventually delivered by all servers within the system, ensuring that all servers receive the same information. [32]

Total Order – Messages are delivered in the same order across all servers. If a message a is delivered before message b by one server, it will be delivered in the same order by all servers, ensuring consistency of message ordering. [32]

Causal Order – Messages are ordered based on causal relationships. If message b is sent after message a has been delivered by the sender of b , message a must be ordered before b . Furthermore, if a sender sends a message c after sending b , c must be ordered after b . [32]

■ **Figure 2.1** Replicated log with finite state machine [14]



Apache ZooKeeper uses a simple, reliable messaging system to keep servers in sync. It uses direct communication channels that ensure messages are received in the same order they are sent. The system works in two main steps: first, it picks a leader who has the most up-to-date information. This leader then makes sure all servers have the same data, either by sending updates or a full copy of the current system status. Once a leader is elected, it sends out messages with updates. All servers get these messages in order and let the leader know that they have received them. When enough servers have confirmed receipt, the leader finalises the update. This method ensures that all messages are delivered correctly and keeps the system consistent and in sync. [32]

“The consistency guarantees of ZooKeeper lie between sequential consistency and linearisability.” [32] Reads do not rely on a quorum operation, which could result in stale data even without a partition. [32]

2.7.6 Raft algorithm

The original article [14] describes all the details of Raft algorithm. The exact implementation details are not crucial for the thesis. This section will introduce the main idea and concepts of Raft algorithm.

The idea is to create a set of identical Finite State Machine (FSM). A sequence of commands (requests) is saved in a replicated log. These FSMs need to be deterministic, meaning that when each FSM executes the sequence of commands, they should end up in the same state. The log is typically persisted on a hard drive to enable the restoration of each member’s state in case of a crash. [14] For an illustration, see Figure 2.1.

It consists of 3 main components:

Leader election – In a cluster, each node is either a follower, candidate, or leader. The leader periodically announces that it is the leader. If a follower does not hear from the leader within a specific time, it becomes a candidate and tries to set up a new election. [14]

Log replication – The leader receives a client’s request and replicates it to the followers. Followers append the request to their log. [14]

Log commitment – The leader waits until it receives acknowledgments from a majority of its followers for a specific log entry before considering it committed. Once committed, the leader notifies its followers, and they apply the entry to their FSM. After committing and applying the entry, it is possible to clear some resources by truncating the logs. [14]

Consistency – There are specific rules for leader election and log replication to prevent inconsistencies, but it is not necessary to go into detail to understand the rest of the thesis. [14]

2.8 Linux container

“A system container is one that delivers the same user experience as a virtual machine. It runs the entire operating system, minus the shared kernel. The current major implementation on Linux is LXC (Linux Containers).” [33]

The kernel of an operating system acts as the essential intermediary between software applications and computer hardware, managing fundamental tasks like hardware management and resource allocation, serving as the foundation for the entire system. [34]

Control Groups (cgroups), a feature of the Linux kernel, enable the management of system resources like CPU, memory, disk I/O, and network bandwidth by organising processes into hierarchical groups with individual resource limits and policies, thus facilitating container isolation while sharing the kernel. [33]

Current service discovery architecture

This chapter provides an overview of the existing architecture. Understanding the current solution, evaluating its pros and cons, and identifying places for enhancement is crucial. A good understanding of the current architecture will serve as a foundational element during the requirement-gathering phase and help design the new solution.

The current architecture is described in a generalised and abstract approach, ensuring no confidential information is revealed. The author of this thesis reviewed the internal documentation and cross-checked it with the source code. Whenever in doubt, he consulted with his team leader.

3.1 Service definition and meaning

From the AB company's perspective, a service is primarily defined by its connection details, which can vary depending on the protocol used. Usually, this involves an IP address combined with a port number. However, additional information, such as stream ID, becomes essential in some scenarios. Furthermore, services can be represented by straightforward gRPC or database connection strings, which may not always explicitly specify the IP or port.

3.2 Actors

This section outlines the system actors and their respective roles within the platform.

3.2.1 Serf agent

It is a HashiCorp Serf agent. Serf agents form a cluster using the gossip protocol. It serves as a tool for firing and subscribing to events or queries. A Serf event is a fire-and-forget message that gets broadcast across the cluster. The query is a request that is also broadcast, but the difference is that it expects a response from each node. [16]

Local datacenter Serf agent (local DC Serf) – The Serf agent operates on all nodes within the local datacenter. A node is considered inactive if the Serf agent is not running on it.

Cross datacenter Serf agent (cross DC Serf) – It is used only to find distributed stores and AB agents from another datacenter.

3.2.2 AB agent

The AB agent performs health checks of all applications. The AB agent fires an event to Serf in case some application goes down. There are two ways AB agents communicate with each other. The first is to directly use the gRPC. The second option is to fire a query to the Serf agent. All AB agents that subscribe to the particular query type will receive it. If there is a request for all datacenters, the AB agent that receives the request broadcasts it to all AB agents within its datacenter and sends the request to one AB agent per other datacenter. The AB agent from the other datacenter then broadcasts within its datacenter.

3.2.3 Distributed store

As the name suggests, this system stores data in a distributed way. It forms a fault tolerant cluster. It uses the Raft algorithm for data replication through the cluster. The cluster is within the scope of one datacenter. For a deployment with multiple datacenters, the distributed store sends the data to only one of the distributed store systems from another datacenter. It picks up the distributed store system registered as last by the cross DC Serf. The distributed store from another datacenter replicates this within its own Raft algorithm cluster.

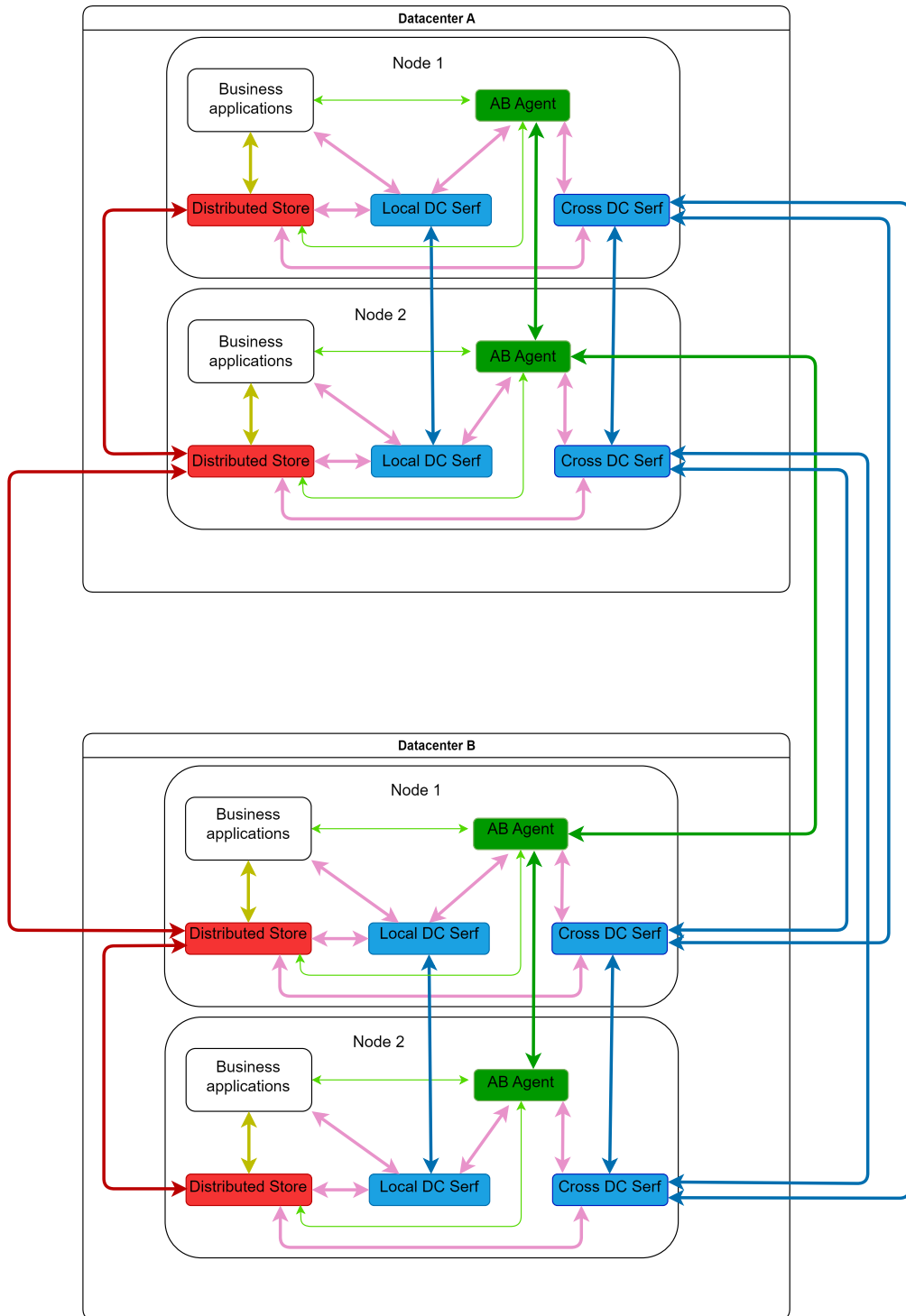
3.2.4 Business applications







These applications handle business logic and are the main product. An application can register its available services to the distributed store or query it for service configurations. Also, it is possible to subscribe for a service, and the distributed store will send a notification in case the service is down or the connection details of the service change.

3.3 Workflow

An internally developed distributed store is in place, leveraging the Raft algorithm for leader election and ensuring consistency. This store maintains connection details for all services. External systems establish connections directly with the leader, locating it through Serf. The leader tags a Serf instance on its node with connection details, facilitating communication. In addition, on each node, an AB agent process performs health checks. In the event of an application failure, the agent triggers an event via the Serf cluster. Consequently, the discovery service plugin within the distributed store can unregister failed applications. However, if the entire server crashes, the discovery service remains aware, as the Serf instance on that box becomes unreachable. The communication schema is depicted in Figure 3.1.

■ **Figure 3.1** Current architecture



- | | | | | | |
|---|---|---|-------------------------------------|---|-------------------------------|
|  | Aeron, or TCP connection – apps query data from store |  | Gossip protocol between Serf agents |  | gRPC calls between AB agents |
|  | Aeron connection – data replication between stores |  | RPC calls to Serf agents |  | PID health checks by AB agent |

Requirements

This chapter shows the main goals and lists all requirements. There is a section for each requirement that describes why it is important. The upcoming Chapter 5 will refer to these requirements when comparing the existing frameworks. The author of this thesis attended meetings with his colleagues almost weekly. The meeting’s participant list included the company’s CTO, leader of a back-end team and two DevOps specialist. The primary goals were established at the beginning. Later, during the research phase, more specific requirements were added.

The primary objective of the refactoring is to replace the custom implementation of the Raft algorithm with an established solution to improve reliability. Other aims include preparing the platform for microservice architecture and migration to the cloud. Having the legacy option to run on on-premise infrastructure is also essential.

However, it is crucial to retain certain aspects, such as subscription for service updates, fault tolerance, consistency, the versatility of the distributed store for storing various information, maintaining a lightweight implementation, and ensuring the new version works across datacenters.

The Table 4.1 summarises the priority of each requirement using the MoSCow methodology described in Section 2.1.

■ **Table 4.1** Requirements prioritisation

Requirement	Importance
Main focus	Must have
CAP theorem	Must have
Pricing	Must have
Java API client	Should have
Maximal size of database	Should have
Maximal size of one entry	Should have
Documentation	Could have
GUI for admin	Could have
Data revision history	Will not have

4.1 Main focus

The framework’s primary focus has to be a service discovery solution or a technology that can be used to implement service discovery. This is why the palette shrinks to service discovery solutions and distributed stores.

4.2 CAP theorem

The CAP theorem, described in Section 2.7.4, offers valuable information when discussing consistency in distributed systems. Although the current implementation relies on the Raft algorithm, adhering to this specific algorithm is not mandatory. However, to maintain the same functionality, it is essential to configure the new solution to operate in a consistent mode, as defined by the CAP theorem, denoted CP.

4.3 Pricing

Ideally, there should be a free alternative, as is typical in the software industry, where a free version is often accompanied by a paid version offering additional features. Ensuring that the free version meets all the requirements is crucial. Once a suitable use case arises, it is worth considering using the paid or enterprise version.

4.4 Java API

Given that most systems in our platform are written in Java, it is crucial to have a seamless integration method through embedded Java code or a project with an existing Java API client.

4.5 Maximal size of service registry

Another crucial requirement pertains to the technical limits of the framework, particularly with regard to the maximum size of the service registry. It is imperative to ascertain whether the framework can accommodate all services the current implementation registers without encountering issues. Understanding this limit ensures that the framework can effectively handle the expected workload.

With an expected amount of several hundreds of services, each with relatively small metadata, even under a pessimistic estimate of 100 kB per service, a framework should comfortably accommodate the registry's storage needs, totalling several hundreds of megabytes. This ensures that the storage capacity of the framework is sufficient to handle the anticipated workload without encountering capacity constraints.

4.6 Maximal size of one entry

In addition to the overall storage capacity, it is common for frameworks to impose limits per entry. Although individual services in the current architecture only require a little space, choosing a framework with a maximum entry size above 200 kB ensures that each entry can comfortably accommodate any future expansion or increased metadata requirements.

4.7 Documentation

Clear documentation reduces the learning curve, enabling developers to grasp and utilise the framework's functionalities quickly. Critical aspects of documentation include well-described internal implementation details, comprehensive API documentation, and tutorials covering usage and deployment processes.

4.8 Subscription vs repeatedly polling the data

The best approach would be to keep the same approach with the service registry, which proactively sends notifications to clients about updates. It is essential to know whether and how this subscription for updates is implemented internally and how it affects performance. In the worst case, it is possible to achieve the same effect as the subscription by repeatedly polling.

4.9 GUI for admin

Once the discovery service is established, it is essential to monitor the state of the registry to ensure its health and accuracy and validate the correctness of the data in the service registry. A simple Graphical User Interface (GUI) for querying data from the service registry would be beneficial, eliminating the need to build one from scratch.

4.10 Data revision history

A data revision history in a service discovery solution provides transparency by allowing users to track changes made to service metadata over time. This feature improves accountability and facilitates troubleshooting by identifying when and by whom changes were made. In addition, it improves reliability by enabling users to revert to previous versions in case of accidental data deletion or corruption.

Comparison of existing service discovery solutions

5.1 Existing solutions

This section will introduce established frameworks suitable for service discovery. Each technology is initially presented with a concise overview of its essential characteristics, advantages, and drawbacks. Then, a comprehensive table will summarise and compare all the technologies.

For each aspect derived from the requirements Chapter 4, a dedicated section is provided in the detailed comparison, offering a comprehensive evaluation of how each framework meets these criteria. Finally, a conclusion will guide the selection of the most appropriate solution for a specific use case and the reasoning behind the choice.

The author of this thesis researched discovery services and distributed stores, recognising that the latter could be the foundation for building discovery services. Colleagues from the AB company team provided tips on specific applications, while others surfaced during Internet research, often compared to frameworks he already had in a scope.

5.1.1 Consul

Consul, developed by HashiCorp, is an open-source tool tailored for service discovery and configuring distributed systems. Its purpose is to facilitate the coordination and configuration of services within large and dynamic infrastructure setups. [4]

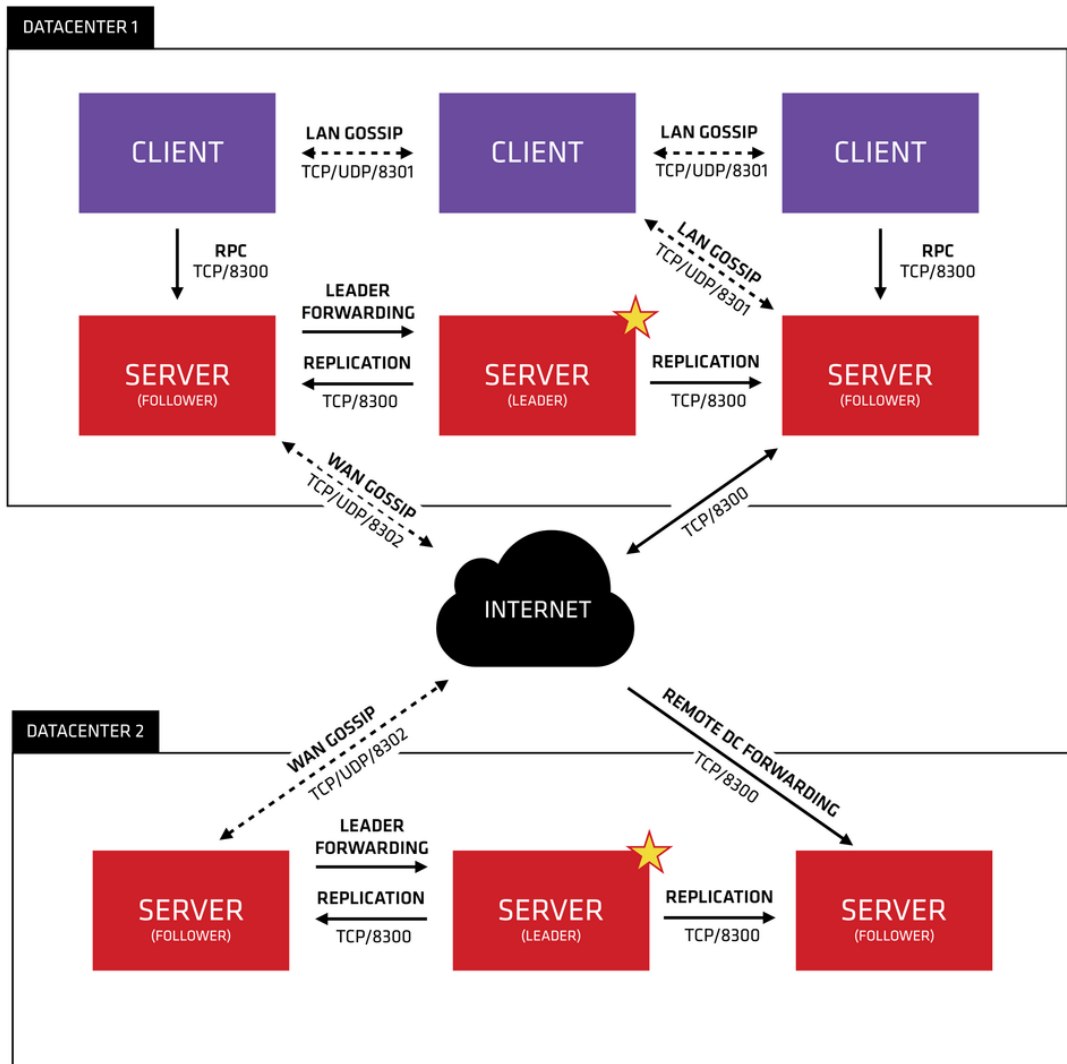
The Figure 5.1 shows the architecture of the Consul system. It is composed of a Raft algorithm cluster made up of Consul servers, where the data is primarily stored. The Consul client is a separate process that acts as a proxy. All Consul processes communicate with each other using a gossip protocol.

Consul has a whole Chapter 6 dedicated to its functionalities and features.

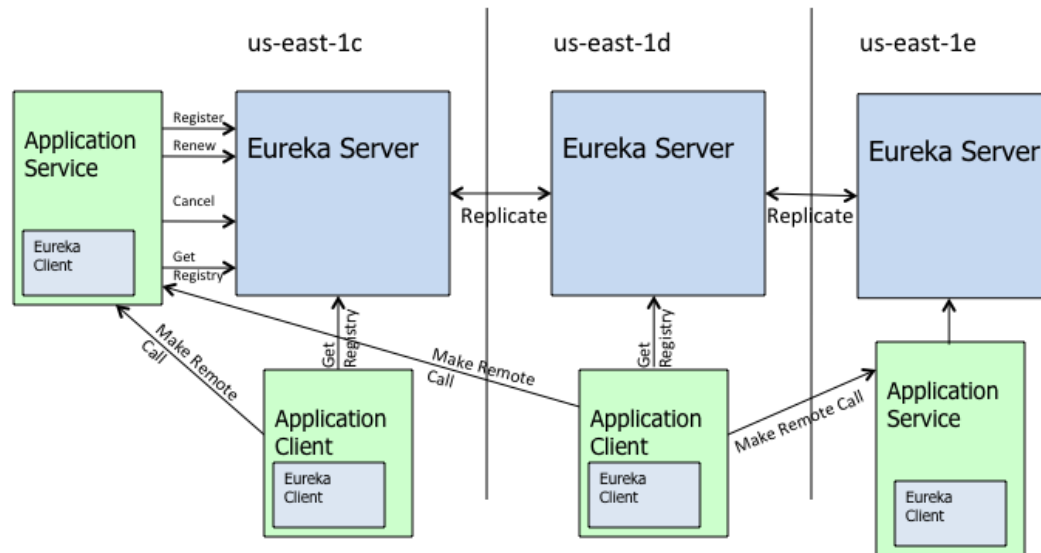
5.1.2 Netflix Eureka

Netflix Eureka [36], akin to Consul, is a service discovery tool designed for environments with dynamic changes, particularly suited for AWS deployments. It specialises in helping services discover and communicate with each other in a dynamic and scalable manner. In Netflix Eureka, registered services have defined Time To Live (TTL) and must be renewed every 30 seconds to maintain accurate and up-to-date service discovery information. [12]

■ **Figure 5.1** Consul architecture [35]



■ **Figure 5.2** Netflix Eureka architecture [12]



The Figure 5.2 illustrates the architecture of the Netflix Eureka design, which consists of a cluster of Eureka servers. Unlike consensus-based clusters, this design does not guarantee consistency across the cluster but follows a best-effort approach prioritising availability over consistency. The client is embedded into an application, and the Eureka server exposes an HTTP endpoint. Both the server and the client are written in Java. [12]

5.1.3 Etcd

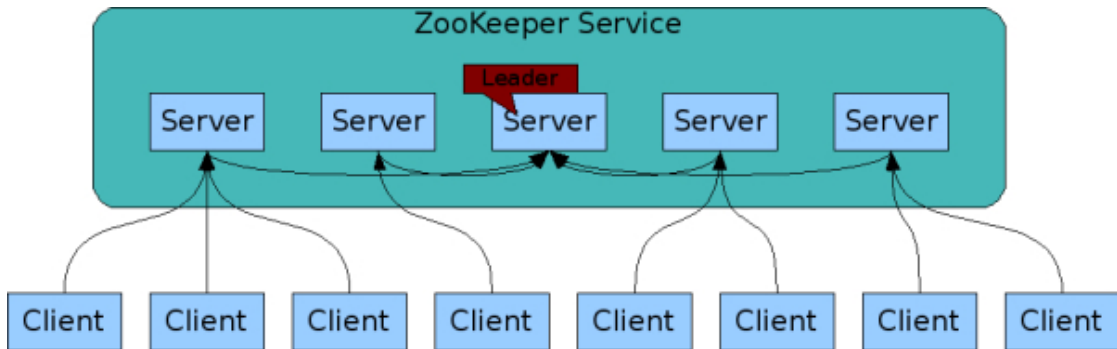
Etcd is a performant distributed key-value. Etcd does not inherently implement service discovery. Users can build their service discovery solutions on top of its versatile key-value store foundation. Like the Consul, Etcd uses the Raft algorithm, ensuring a CP system. Etcd servers also expose their API via an HTTP endpoint. [7]

The official Etcd website [37] makes comparisons between Etcd, Apache ZooKeeper, and Consul. Etcd is endorsed for its efficiency as a distributed key-value store, while Consul is preferred for tasks such as service discovery and service mesh deployment. [37]

5.1.4 ZooKeeper

Apache ZooKeeper primarily functions as a hierarchical key-value store, with service discovery being one of its potential applications. Unlike Consul, which offers a built-in service discovery feature, Apache ZooKeeper requires manual implementation for service discovery.

■ **Figure 5.3** Zookeeper architecture [32]



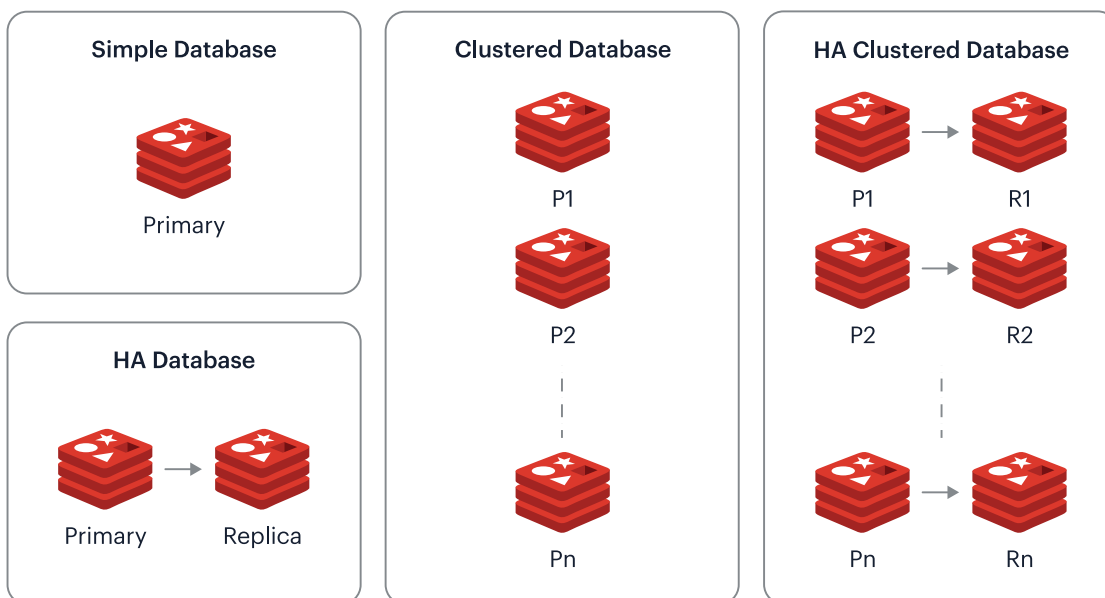
Both Apache ZooKeeper and Consul feature a similar architecture, comprising fault-tolerant server cluster to which clients connect. The custom consensus algorithm, Zookeeper Atomic Broadcast Protocol (ZAB), is in place, as described in Section 2.7.5.1. Apache ZooKeeper lacks a proxy (in opposite to Consul) between the client and the server. The Apache ZooKeeper architecture is shown in Figure 5.3. The client component in Apache ZooKeeper is provided as a Java package, Apache Curator being among the recommended client libraries for Apache ZooKeeper. [32].

5.1.5 Redis

“Redis is an open-source in-memory data structure store that can be used as a database, cache, and message broker.” [38] It is known for its high performance, versatility, and simplicity. Redis has gained widespread adoption due to its efficient handling of data structures like strings, hashes, lists, and sorted sets. [15]

The Figure 5.4 depicts all of the possibilities for the Redis deployment. Redis can be configured to use a leader-follower (primary-replica) pattern cluster for data replication and high availability [39]. This is called a sentinel [40]. Additionally, it is possible to chain sentinels to a cluster of sentinels [40]. Data replication is not a quorum operation, implying that data consistency is not guaranteed [39].

■ **Figure 5.4** Redis architecture [41]



5.2 Comparison

Table 5.1 shows a summarised comparison. The comparison of each aspect is described in more detail in the following sections.

■ **Table 5.1** Comparison of existing frameworks suitable for implementing the service discovery

(a) Consul and Netflix Eureka

	Consul	Netflix Eureka
Main focus	Service discovery, service mesh	Service discovery in AWS
CAP theorem	CP, AP, almost CP	AP
Pricing	Open-source, premium	Open-source
Java API client	Yes	Yes
Maximal size of service registry	100s of MB	Unknown
Maximal size of one entry	512 kB	Unknown
Documentation	Very good	Poor
Possible subscription for updates	Yes	No
GUI for admin	Yes	No
Data revision history	No	No

(b) Key-value stores Etcd, ZooKeeper and Redis

	etcd	ZooKeeper	Redis
Main focus	Distributed key-value store	Distributed key-value store	In-memory key-value store
CAP theorem	CP, AP	AP, almost CP	AP
Pricing	Open-source	Open-source	Open-source, premium
Java API client	Yes	Yes	Yes
Maximal size of service registry	Up to 8 GB, 2 GB by default	100s of MB	In-memory, limited by RAM
Maximal size of one entry	1.5 MB	1 MB	512 MB
Documentation	Very good	Very good	Very good
Possible subscription for updates	Yes	Yes	Yes
GUI for admin	Yes	Yes	Yes
Data revision history	Yes, built-in	Yes, built-in	No

5.2.1 Main focus

Consul – Consul primarily focuses on service discovery, service mesh capabilities, and providing a distributed key-value store. It enables dynamic service registration and discovery, health checking, and advanced features for service-to-service communication within modern microservices architectures. [4]

Netflix Eureka – Netflix Eureka is designed for service discovery and registration, particularly for mid-tier load balancing in cloud-native environments. It enables services to register themselves and locate other services without hard-coded dependencies, thus enhancing the resilience and scalability of applications. [12]

Etcd – Etcd serves as a distributed key-value store primarily focusing on configuration management. It provides a reliable way to store configuration data across distributed systems and

facilitates coordination among different components, making it suitable for building highly available and scalable applications. [7]

ZooKeeper – Apache ZooKeeper is a distributed coordination service primarily focusing on maintaining configuration information, providing naming services, distributed synchronisation, and supporting group services. It is widely used for building distributed systems that require reliable coordination and consensus among multiple nodes. [2]

Redis – Redis is an in-memory data structure store that serves various purposes, including acting as a database, cache, and message broker. Its main focus is on providing high-performance data storage and manipulation capabilities, making it a versatile tool for various use cases, from caching to real-time analytics. [15]

5.2.2 Comparison based on CAP theorem

Consul – Writes are always CP. There are three modes for a read request in Consul. By default, Consul uses a mode that is consistent in most cases, except the edge case, with a trade-off for better performance. It can also be strictly configured to use CP, AP, or a default approach. In addition, it is possible to choose the strategy per each HTTP request separately. [42]

Netflix Eureka – Netflix Eureka emphasises availability, acknowledging the dynamic nature of cloud environments and their inherent variability. Eureka’s design assumes frequent network partitions or system failures, including those affecting the Netflix Eureka cluster, particularly in large-scale deployments. Regarding the CAP theorem, the framework leans towards being AP in its design and functionality. It operates without employing any consensus algorithm, instead opting for a “best-effort” approach to achieve its objectives. [12]

Etcd – Etcd provides the flexibility to be configured for either CP or AP modes. Similarly to Consul, it internally utilises Raft algorithm as its consensus algorithm. [43]

Zookeeper – There is a unique consensus protocol developed by Apache ZooKeeper called Zookeeper Atomic Broadcast Protocol (ZAB). This algorithm is already introduced in the Section 2.7.5.1. Strong consistency (CP) is not guaranteed, but it is close to it. It maintains sequential consistency, defined in Section 2.7.3.1, meaning that updates from a client will be applied in the order they were transmitted, but it is not linearisable (CP) [32]

Redis – Redis prefers availability, AP. The Redis cluster implements asynchronous replication, leading to potential data loss scenarios. Even with synchronous writes enforced through the `WAIT` command, strong consistency is not achieved, and there is still a risk of data loss. It is worth noting that the `WAIT` command significantly reduces performance. [39]

5.2.3 Pricing

Each framework provides an open-source version, with Consul and Redis offering an enterprise edition with additional features.

Consul – Open-source, with enterprise options for easier server cluster management. Consul Enterprise Edition presents a range of advanced functionalities designed for larger-scale deployments, including enhanced support for network segmentation, resource isolation through namespaces, and automated backup capabilities. Additionally, organisations can entrust the management of Consul servers to HashiCorp, streamlining operational tasks and ensuring optimal performance and reliability. [44]

Redis – The core functionality is open-source. Redis Enterprise, developed by Redis, Inc., is a robust commercial product that enhances the core Redis Engine with enterprise-grade features. It enables linear scaling to handle large workloads and ensures high availability with up to 99.999% uptime. Redis Enterprise ensures data integrity and protection with geo-replication capabilities and advanced security measures. Redis Enterprise is supported by 24/7 assistance for seamless implementation and maintenance, offering flexible deployment options, including on-premise, cloud-based, and hybrid configurations. [45]

5.2.4 Implemented the Java API client

A Java API client exists for each framework. The summary is depicted in Table 5.2. All of the clients are open-source.

■ **Table 5.2** Java API client comparison

Framework	Java API Client
Consul	Rickfast [46]
Netflix Eureka	EurekaClient [36]
Etcd	Jetcd [47]
ZooKeeper	Curator [1]
Redis	Jedis [48]

5.2.5 Subscription vs repeatedly polling the data

For Etcd [49], Apache ZooKeeper [50] and Redis [51], it is possible to register a callback for data changes.

Consul – It is possible to subscribe for events with watches [52]. The Consul agent triggers a script or sends a request to the HTTP endpoint [52]. The Rickfast [46] client implements a registering of a callback by the long-polling mechanism. Section 6.8.2.2

Netflix Eureka – There is not mentioned anything about callbacks for data changes in the documentation [12]. In the worst case, it is possible to implement repeated polling on the client side to achieve the same.

5.2.6 Maximal reliable size of the storage

Consul – It is not designed to store much data in the key-value store. Up to several hundreds of MB can be stored using disk storage for the Raft algorithm. According to the documentation, it is recommended to have 2 to 4 times more RAM available than the size of the storage. [53]

Netflix Eureka – Without an official benchmark, accurately determining the RAM requirements for Netflix Eureka’s operation entails conducting empirical testing and performance analysis. Since Netflix Eureka is an in-memory database with a TTL for each service, usually around 30 seconds before automatic removal, the RAM size needed would vary depending on the stored data volume [12]. It would be prudent to allocate additional RAM to handle fluctuations effectively.

Etcd – Scales well with big key-value stores, up to several GB. By default, it is set to 2 GB. The maximum recommended size is 8 GB. [54]

ZooKeeper – Assuming the Etcd benchmark [37], Apache ZooKeeper scales well with big key-value stores, up to several hundreds of MB.

Redis – In-memory database works well with data that fits into memory. It can be configured to persist data, supporting up to several GB. [55]

5.2.7 Maximal size of one database entry

The author of this thesis could not find a maximum size for Netflix Eureka.

■ **Table 5.3** Maximal size of one database entry

Framework	Maximal size of one entry
Consul	512 kB [53]
Netflix Eureka	Unknown
Etcd	1, 5 MB [54]
ZooKeeper	1 MB [32]
Redis	512 MB [56]

5.2.8 Quality of the documentation

Except for Netflix Eureka, all other frameworks have comprehensive documentation, providing thorough descriptions of their internal implementation, APIs, and tutorials.

Netflix Eureka – The documentation for Netflix Eureka [12] is notably concise, offering brief explanations of its internal implementation, deployment tutorials, and REST API. However, while tutorials on configuration and the REST API are sufficiently described, the internal implementation of the cluster may lack detailed documentation. Nonetheless, users can still benefit from practical guidance on the deployment and use of the REST API for their projects.

5.2.9 Has a GUI?

All solutions have an existing GUI, with variations in whether it is an official or third-party offering. All of them are free except for Redis, which is a paid service.

■ **Table 5.4** GUI comparison

Framework	Note	GUI software	Pricing
Consul	Official	Consul [57]	Free
Netflix Eureka	Third party	Eureka Spring Cloud [58]	Free
Etcd	Third party	Etcdmanager [59]	Free
ZooKeeper	Third party	Zoonavigator [60]	Free
Redis	Official	Redis [61]	Paid

5.2.10 Has data revision history?

Consul – Revision history tracking is not directly supported, but it offers options to log changes or capture snapshots of services. [4]

Netflix Eureka – The documentation [12] does not mention the built-in support for revision history tracking.

Etc – Revisions can be monitored programmatically or managed through the GUI. [62]

ZooKeeper By default, Apache ZooKeeper retains historical snapshots of data and maintains both a transactional log and snapshots. [32]

Redis – Revisions in Redis would require manual implementation as it is could not be found in the documentation [15].

5.3 Conclusion

Consul emerges as the optimal choice to meet the customer’s requirements. Its versatility allows it to operate seamlessly in cloud environments and on-premise infrastructure. Its primary focus on service discovery and robust documentation makes it the most promising fit for the project.

Although Etc and Apache ZooKeeper are viable alternatives, they would require additional implementation efforts to achieve service discovery using the key-value store. In contrast, Consul offers built-in capabilities for service discovery, streamlining the development process.

The decision to rule out Netflix Eureka is primarily due to its limitation of running exclusively in the AWS cloud, which does not align with the requirement for flexibility across different environments. In addition, the documentation is inadequate.

As for Redis, although it excels as a key-value database, it may not meet the project’s specific needs, mainly if consistency and partition tolerance (CP) are essential requirements.

In summary, Consul stands out as the preferred solution due to its compatibility across various deployment environments, strong focus on service discovery, and comprehensive documentation, making it an optimal choice for the project.

Consul analysis

Consul is an open-source tool for service discovery and distributed system configuration made by HashiCorp, a company that provides infrastructure automation software [4]. It is designed to help coordinate and configure services in large-scale dynamic infrastructure environments [4]. Consul is primarily written in the Go programming language [63]. The tool was first conceptualised in 2013 and introduced as a service discovery and configuration tool by HashiCorp in 2014, reaching version 0.1 [64].

6.1 Philosophy

The main philosophy outlined on their website [65] describes the main points to be followed.

Workflow focus – HashiCorp prioritises workflows to easily adopt new tools. [65]

Simple, modular, composable – HashiCorp products are built using simple, modular, and composable components, allowing for innovation by combining smaller, well-defined parts, following the Unix philosophy. [65]

Communication sequential processing – HashiCorp believes that Communicating Sequential Processes (CSP) are crucial for managing complexity and building robust, scalable systems in a service-oriented architecture. Each service is treated as an autonomous process that communicates via APIs. [65]

Immutability – HashiCorp promotes immutability in infrastructure, extending it to app source code, versions, and server states, resulting in more robust and straightforward systems to operate. [65]

Versioning through codification – The belief in codification involves writing processes as code, storing, and versioning them to promote knowledge sharing and prevent data loss during operations. [65]

Automation through codification – HashiCorp emphasises automation by promoting codification to enable machine execution while remaining readable by operators, increasing productivity, and reducing human errors. [65]

Resilient systems – HashiCorp prioritises building durable systems that handle unexpected inputs and outputs. This is achieved through maintaining a desired state, collecting real-time information, and self-healing mechanisms. [65]

Pragmatism – HashiCorp values practicality, adapting principles such as immutability, codification, and automation to meet requirements and encourage innovation. [65]

6.2 Use cases

This section explains the primary characteristics and practical applications of Consul.

Service discovery – Automatic detection and cataloguing of available services within a network, keeping the service registry [66]

Health-checking – Several types of health checks: HTTP, TCP, UDP, gRPC... [67]

Key-value store – For storing key-value pairs [66]

Leader election within services – Implementation of a leader election that other applications can use [68]

Service mesh – Management of communication between services (authentication, load balancing, data encryption, sets rules for service-to-service communication) [66]

In the thesis context, service discovery and the key-value store are the most important features.

6.3 Architecture

There are two types of Consul agents: server and client. Each agent exposes an HTTP endpoint [69].

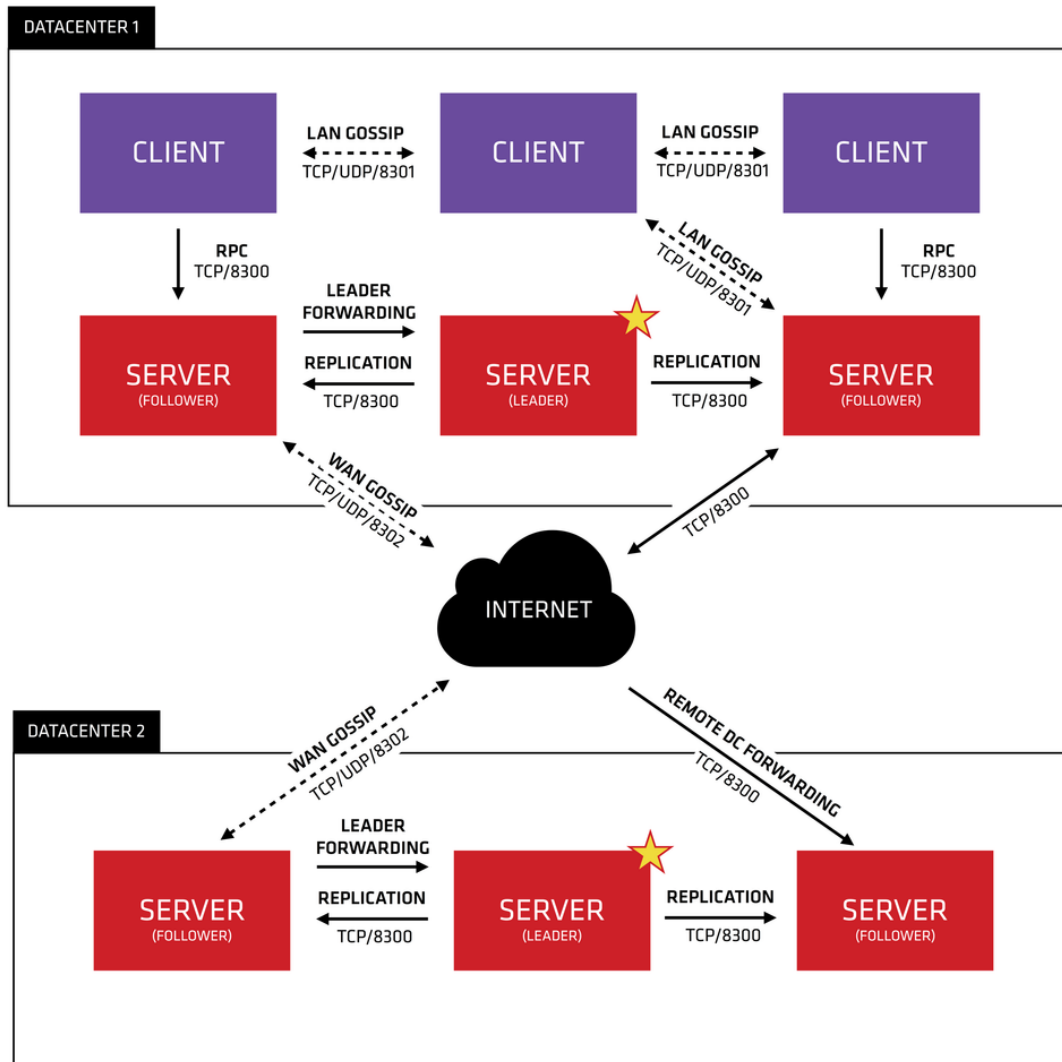
The server stores all data, such as the service catalogue and key-value store, while the client is a proxy to forward requests to the server. Clients maintain a cache, which allows them to handle some requests without contacting the servers. Generally, each box has one Consul agent (server or client). In addition, the client is responsible for managing the health checks of the services. Registering a check for a service is possible, which triggers a bash script or a custom HTTP endpoint when the service is down. The servers that provide the core functionality are usually in a cluster to ensure fault tolerance. The Raft algorithm ensures consistency, with Serf being used as the underlying technology. All Consul instances (clients and servers) communicate with each other using the gossip protocol. [70]

In the image shown in Figure 6.1, the architecture of Consul is depicted. This tool also allows for deployment across multiple datacenters. According to Consul, a datacenter is defined as follows: “*We define a datacenter to be a networking environment that is private, low latency, and high bandwidth. This excludes communication that would traverse the public internet, but for our purposes multiple availability zones within a single EC2 region would be considered part of a single datacenter.*” [71].

6.4 Fault tolerance

According to Raft algorithm, a cluster can continue to operate as long as a majority (also known as quorum) of servers remains alive. However, there is a trade-off between fault tolerance and performance, as a larger quorum can tolerate more server failures but at the cost of decreased performance due to increased gossip communication. The Consul documentation recommends a cluster size of three or five members, where a three-member cluster can tolerate one server failure and a five-member cluster can tolerate two server failures without affecting the cluster functionality. [42]

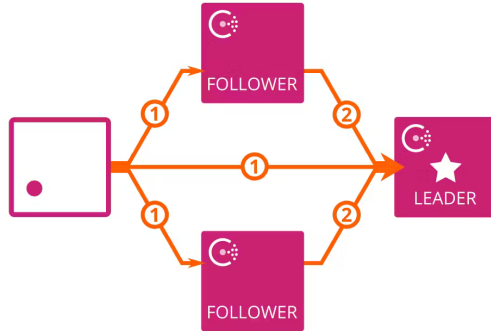
■ **Figure 6.1** Consul architecture [35]



■ **Figure 6.2** Consul default mode [42]

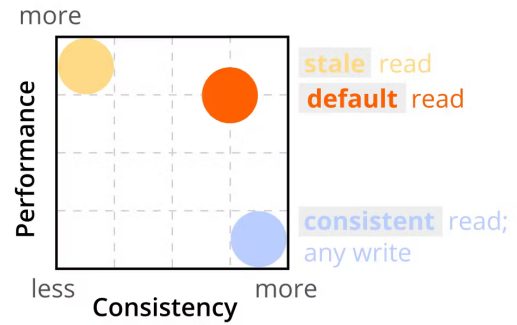
default read request path




PROCESS



- ① client may contact any server agent
- ② followers forward to their known leader

TRADE-OFFS | performance vs consistency



System Components		Communication Paths	
	Consul Server: Leader		Consul Client Agent
	Consul Server: Follower		Control Plane

6.5 Consistency modes

When discussing consistency modes in Consul, we only refer to read requests. This is due to the fact that write requests must always be replicated and committed by the Raft algorithm leader, which means they are always consistent. [42]

A user can select from three consistency modes. Additionally, each mode can be set for each HTTP request. Full details can be found on the Consul official documentation [42].

6.5.1 Default mode

“It is strongly consistent in almost all cases. However, there is a small window in which a new leader may be elected, during which the old leader may respond with stale values. The trade-off is fast reads but potentially stale values. The condition resulting in stale reads is hard to trigger, and most clients should not need to worry about this case. Also, note that this race condition only applies to reads, not writes.” [42] This mode is shown in Figure 6.2.

It is an excellent example of how sacrificing a little consistency can significantly improve performance. This fits to the Consul’s pragmatical philosophy.

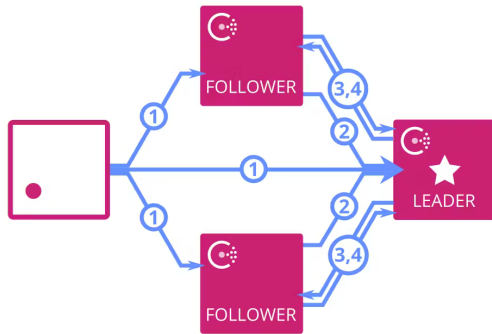
6.5.2 Consistent mode

For users prioritising strong consistency over availability, the **consistent** mode offers a stricter approach. The leader checks its leadership status before responding to a read request, resulting in an additional round-trip for each request as seen in Figure 6.3. In other words, this mode is

■ **Figure 6.3** Consul consistent mode [42]

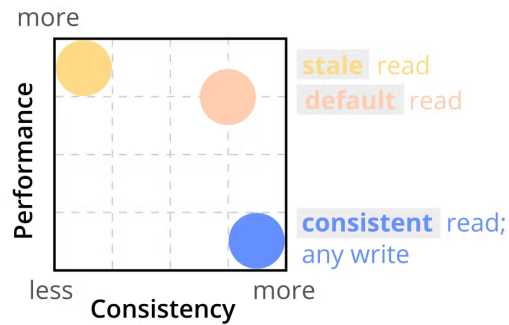
consistent read, any write request path

PROCESS



- ① client may contact any server agent
- ② followers forward to their known leader
- ③ leader asks followers if it's still the leader
- ④ followers respond to leader

TRADE-OFFS | performance vs consistency



System Components



Consul Server: Leader



Consul Client Agent



Consul Server: Follower

Communication Paths



Control Plane

the same as the default mode with the addition of solving the edge case when the leader may send old values. The reads are consistent and partition-tolerant (CP) based on the CAP theorem. [42]

6.5.3 Stale mode

The **stale** consistency mode is designed for users who prioritise availability over strong consistency. In this mode, any Consul server can handle a read request, regardless of whether it is the leader. This mode is best suited when performance is more important than potentially stale values. One can say that reads are available and partition-tolerant (AP) based on the CAP theorem. [42]

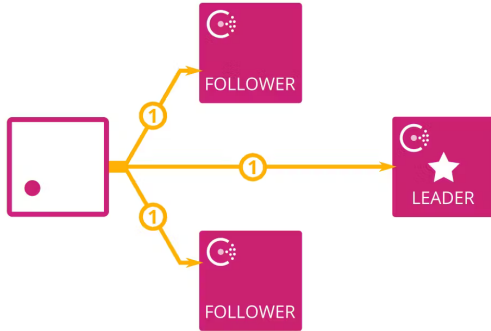
6.5.4 Consistency mode per HTTP request

Consul provides fine-grained control over consistency modes for individual HTTP requests. This flexibility allows users to tailor the consistency level based on specific requirements for different parts of their application. To select the desired request mode, use one of the HTTP query parameters: `?stale`, `?consistent`, `?default`. [42]

■ **Figure 6.4** Consul stale mode [42]

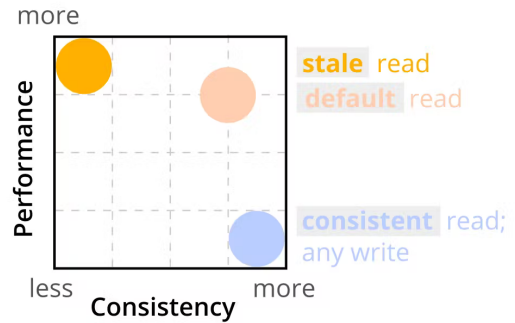
stale read request path



PROCESS



① client may contact any server agent

TRADE-OFFS | performance vs consistency



System Components		Communication Paths	
	Consul Server: Leader		Consul Client Agent
	Consul Server: Follower		Control Plane

6.6 Multiple datacenters

Cross-DC communication is the transfer of data between geographically dispersed datacenters. It differs from inner DC communication within a single datacenter. The main differences are scale, distance, and latency. Cross-DC communication experiences higher latency and lower bandwidth due to longer physical distances and multiple network hops. The design and optimisation of networking infrastructure for these two communication types require different considerations. [72]

There are two ways to set up communication between datacenters. The first option is to set up a WAN gossip pool. The second one is to set up the Remote datacenter forwarding. Both have their pros and cons. [72]

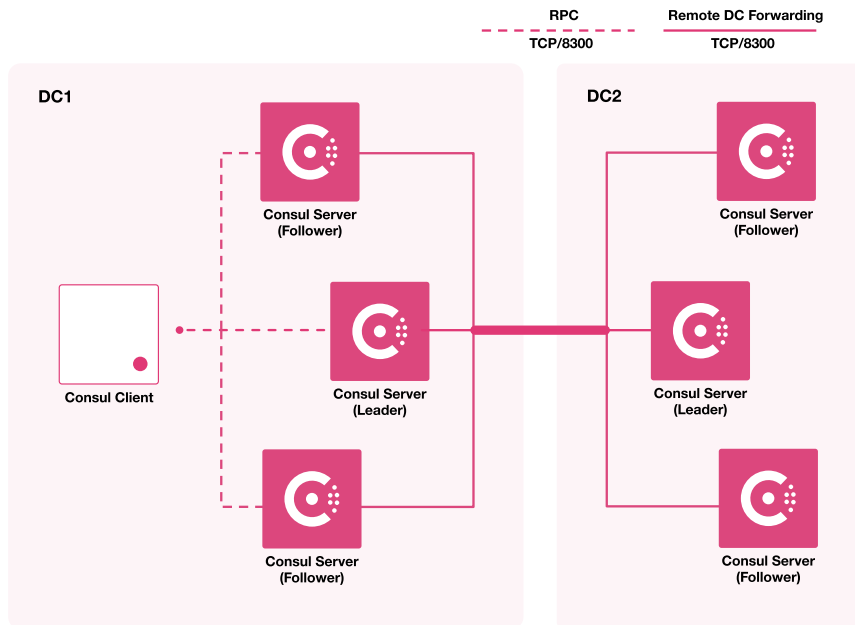
6.6.1 WAN federation

The local server from the datacenter will send a request to a server from the secondary datacenter when data is needed. The schema is depicted in Figure 6.5.

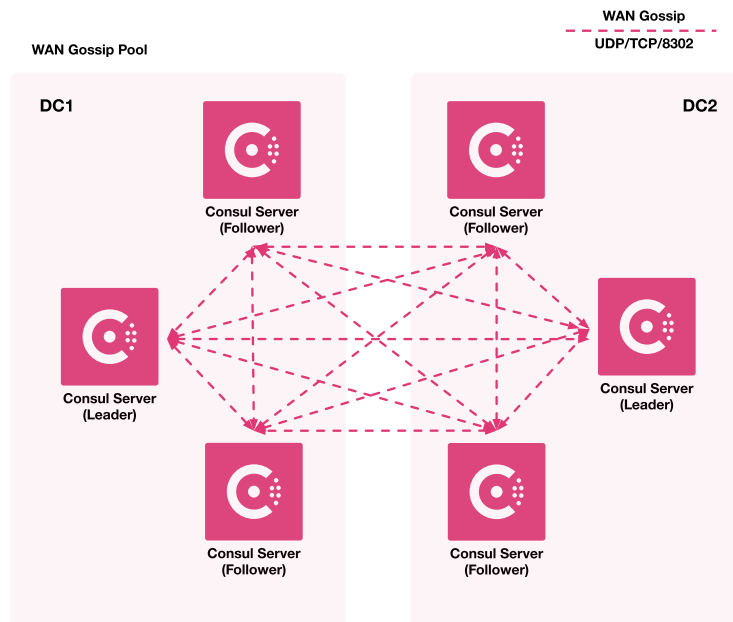
6.6.2 WAN gossip pool

The way it works is similar to a single datacenter. All servers are connected through gossip communication, which allows messages to be sent across the datacenter. The schema is depicted in Figure 6.6.

■ **Figure 6.5** Consul cross datacenter forwarding [72]



■ **Figure 6.6** Consul cross datacenter gossip [72]



6.7 Deployment

The deployment process for a Consul cluster is relatively simple and lightweight. There are three ways to install a Consul agent. website. The easiest method is to download a precompiled binary and run it. [73]

6.7.1 Hashicorp cloud platform

Consul offers a paid feature called HashiCorp Cloud Platform (HCP) cluster. It is a fully managed server cluster by HashiCorp that saves user's time as it is used in a Software as a Service (SaaS) way. [44]

6.8 Performance fine-tuning

To optimise Consul for better performance, administrators can fine-tune parameters related to the gossip protocol, such as adjusting the gossip interval and managing the number of nodes involved in gossip. Additionally, carefully allocating resources for Consul servers and agents based on the specific deployment characteristics, like adjusting memory and CPU allocations, can contribute to a more efficient and scalable Consul infrastructure. [74]

The following sections describe some of Consul's internal performance enhancements. This knowledge can be used to further improve performance by configuring the agent or requests.

6.8.1 Anti-entropy

Each Consul client stores information about services and health checks that are registered locally on the same node. The agent sends this data to the service catalogue, a collection of global states held by the consul servers. Some requests can be handled from the client's local state, usually containing information about services from the local node. The catalogue and the client's state are synchronised periodically. *"The anti-entropy mechanism reconciles these two views of the world: anti-entropy is a synchronization of the local agent state and the catalogue."* [75]

6.8.2 Agent request caching

The agent can fulfil certain requests without contacting the servers every time. To achieve this, two caching strategies can be employed - Simple caching and Background Refresh caching. Each strategy is supported by different HTTP endpoints. [76]

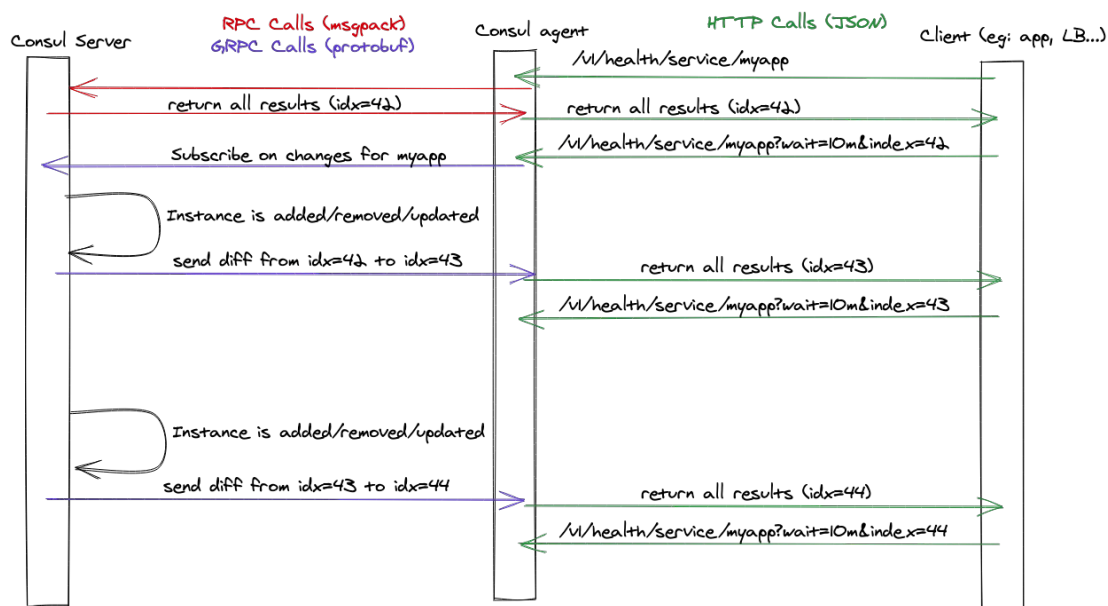
6.8.2.1 Simple caching

The caching mechanism will be employed when the HTTP `?cached` query parameter is set, and the required tags are added to the `Cache-Control` header. The value for `max-age` should be specified in seconds. The response includes a flag indicating whether the cache was hit or missed. [76]

6.8.2.2 Background refresh caching

Some endpoints in Consul support an active caching feature that helps keep the data synchronised with the catalogue. To achieve this, the server and the client use a technique called long polling [77]. Instead of the client repeatedly asking the server for updates, it sends a request to the server and holds it open until new data is available or a timeout occurs. When new data is

■ **Figure 6.7** Consul long polling [77]



ready, the server responds to the client, which processes the information and immediately sends another request to the server to maintain the connection [78].

The `idx` and `wait` HTTP query parameters are crucial for Consul's long polling. The `idx` parameter corresponds to the data version, while the `wait` parameter specifies the interval for how long the connection can be kept open [46].

An example of how it may look is depicted in Figure 6.7.

6.9 Reactive environment

Consul provides two ways to specify actions for incoming events based on changes in the key-value store, service catalogue, list of nodes, etc.

6.9.1 Watches

“Watches are a way of specifying a view of data (e.g. list of nodes, KV pairs, health checks) which is monitored for updates. When an update is detected, an external handler is invoked.” [52]. A handler can be a script (see example in Listing 6.1) or a HTTP endpoint (see example in Listing 6.2). [52]

6.9.2 Templates

It is basically a daemon that watches for changes in the key-value store and based on that renders a configuration file on a local node. This is essentially just syntactic sugar. The same result could be achieved through the use of a watch. Templates have their own scripting language similar to Go. The demo is shown in the Listing 6.3 to Listing 6.6. [79]

■ **Listing 6.1** agent-config.json script watch example [52]

```

1  {
2    "watches": [
3      {
4        "type": "key",
5        "key": "foo/bar/baz",
6        "handler_type": "script",
7        "args": ["/usr/bin/my-service-handler.sh", "-redis"]
8      }
9    ]
10 }

```

■ **Listing 6.2** agent-config.json HTTP watch example [52]

```

1  {
2    "watches": [
3      {
4        "type": "key",
5        "key": "foo/bar/baz",
6        "handler_type": "http",
7        "http_handler_config": {
8          "path": "https://localhost:8000/watch",
9          "method": "POST",
10         "header": { "x-foo": ["bar", "baz"] },
11         "timeout": "10s",
12         "tls_skip_verify": false
13       }
14     }
15   ]
16 }

```

■ **Listing 6.3** Find address template example find_address.tpl [79]

```
{{ key "/hashicorp/street_address" }}
```

■ **Listing 6.4** Activate the template [79]

```
1 consul-template -template "find_address.tpl:hashicorp_address.txt"
```

■ **Listing 6.5** Put the value to the corresponding key [79]

```

1 consul kv put hashicorp/street_address "101 2nd St"
2
3 Success! Data written to: hashicorp/street_address

```

■ **Listing 6.6** Showing content of the rendered file [79]

```

1 cat hashicorp_address.txt
2
3 101 2nd St

```

6.10 Conclusion

Consul by HashiCorp is an open-source tool designed to facilitate service discovery and distributed system configuration. It is a powerful and flexible tool that is easy to use and provides a range of features such as consistency modes, support for multiple datacenters, and performance

fine-tuning options.

Consul is built on the principles of simplicity, modularity, and pragmatism. Its architecture is based on a gossip-based cluster, which combines server and client agents for scalability and fault tolerance. Consul is not just a distributed key-value store, but also serves as a crucial component for creating reactive environments through watches and templates. Additionally, it offers solutions for managing databases, including health checking, leader election, and other coordination tasks.

In summary, Consul is an invaluable asset for organisations dealing with modern infrastructure and distributed systems. It simplifies the complexities of managing such systems and offers a practical solution.



Chapter 7

Design

This section exclusively illustrates actors and processes relevant to service discovery to simplify the view. The current architecture is described in Chapter 3 with the schema Figure 3.1.

The refactoring process will be divided into several phases for the sake of smoother implementation. In the first phase, the existing distributed store will be deployed alongside Consul. During this phase, the AB agent will continue to handle health checks. The primary modification will involve transferring the responsibility of the discovery service from the distributed store to the Consul while retaining the rest of the current implementation.

Subsequently, in the second phase, the Consul will take over the health check responsibility from the AB agent. Upon completion of these two phases, the migration of the discovery service to the new implementation will be finalised.

As a bonus step, the final phase will involve utilising Consul's key-value store instead of the distributed store, allowing for the complete removal of the distributed store from the platform.

7.1 Using Consul service catalogue

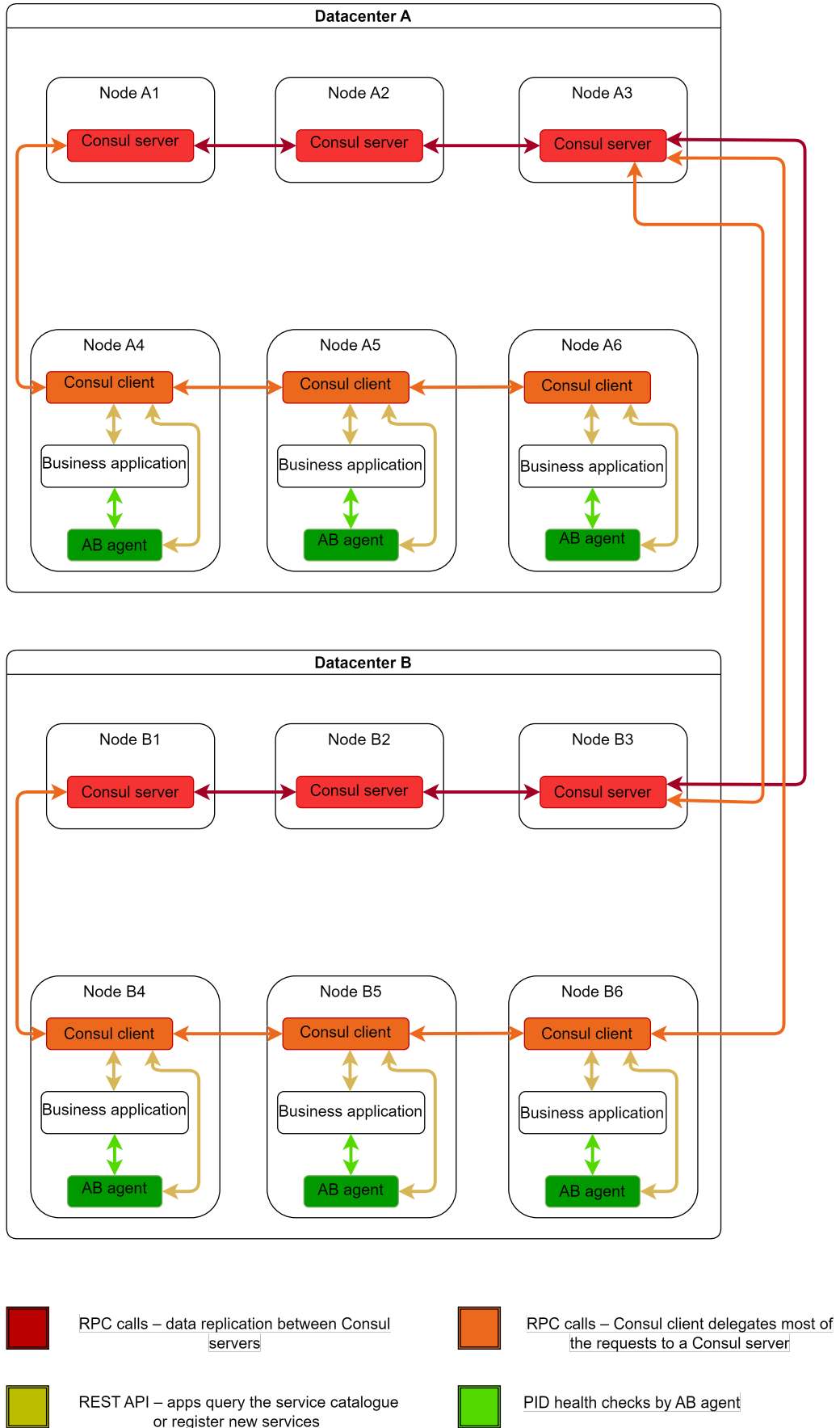
The Consul server cluster is typically deployed on dedicated nodes, separate from other applications, comprising three to five nodes to ensure tolerance against failures of one or two nodes. Each node hosting a business application has a Consul client as a proxy. Requests are routed through REST API calls. Upon receiving a request, the Consul agent may promptly respond or delegate the request via RPC call to the Consul server cluster. The Consul's internal communication is described in more detail in Section 6.3 and Section 6.8.2.

Applications register their available services by sending a REST request to the locally running Consul client. The application queries the Consul cluster when needing connection details from the service registry. Consul's service registry and service catalogue are synonyms. The AB agent continues to perform PID health checks and deregisters the application in case of a stoppage or crash. This phase is illustrated in Figure 7.1.

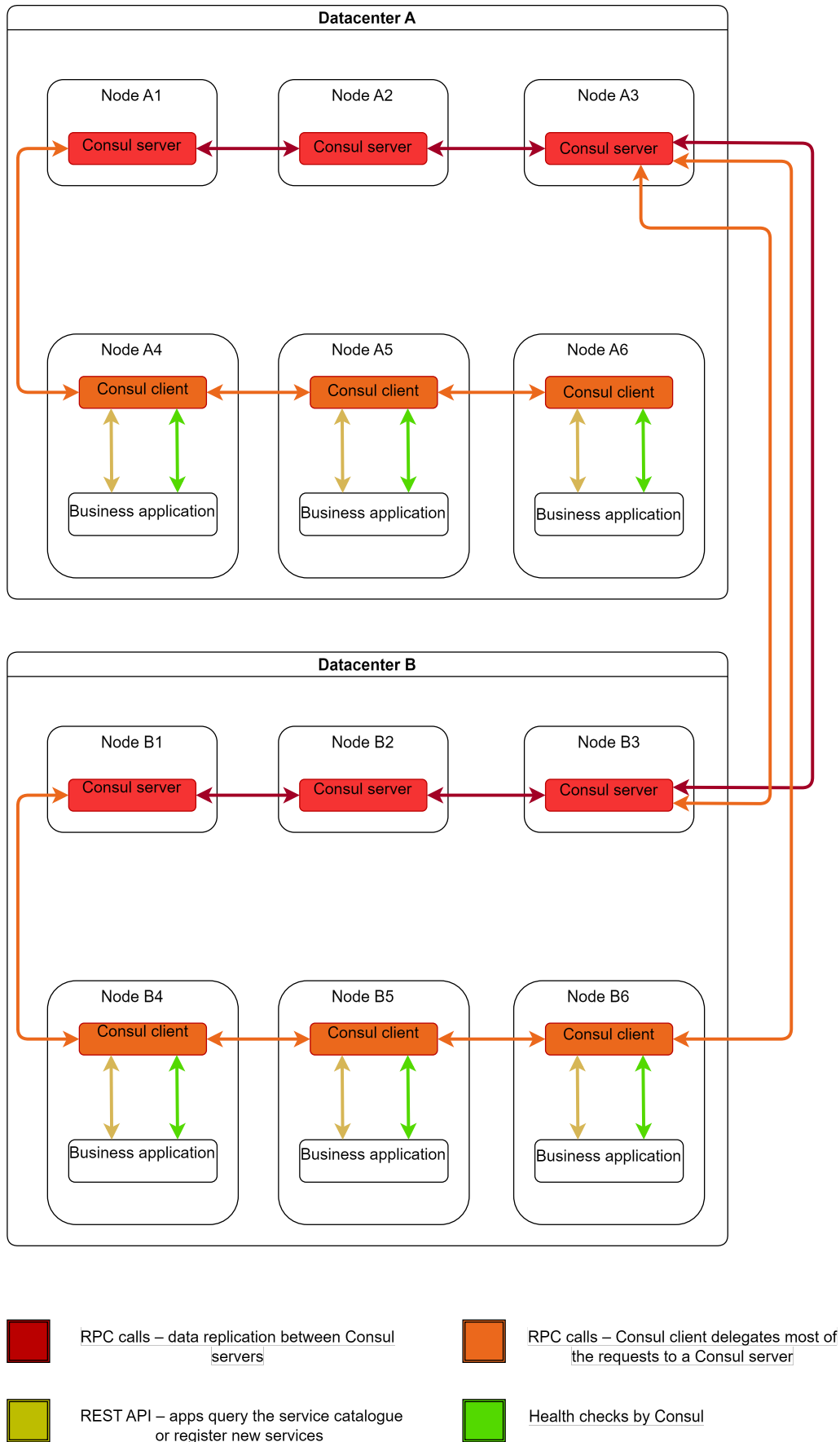
7.2 Using Consul for health checking

In the second phase, the Consul instances perform all health checks. An application registers health checks for itself and all the services it exposes. The client may subscribe for the service metadata and health updates. The Section 6.8.2.2 outlines the subscription mechanism. This phase is depicted in Figure 7.1.

■ **Figure 7.1** The first phase – Using Consul service catalogue



■ **Figure 7.2** The second phase – Using Consul for health checking



7.3 Using Consul key-value store and other features

This phase is considered a bonus phase as it is not related to service discovery. Instead, it focuses on leveraging Consul's key-value store to replace the existing one within the platform. Consul provides many supplementary features, among which the most notable for integration are the service mesh, Consul metrics, and leader election functionalities. All use cases for Consul are listed in Section 6.2.

Implementation

It is important to note that the current implementation is designed only for Linux-based operating systems. The code is written in a simple manner and can be improved later if needed. The author of this thesis prioritised understanding and using the Consul framework over writing the code as a final product. It is considered more of a prototype and there will likely be improvements once it is fully implemented. However, the existing code should serve as a good starting point and a core implementation.

The implementation encompassed several facets, primarily orchestrated by the AB agent. This included the deployment of Consul, configuration management, and health checking. The DevOps framework ensures that the binary file is downloaded to each box where the AB agent is running. The AB agent then takes the binary and initiates the Consul process. The AB agent handles the entire lifecycle of the Consul process and its configuration management. The main code about the implementation of the service discovery API is described in Section 8.5.

8.1 Start

A Consul agent needs configuration upon start-up. The easiest way is to provide a configuration file. All available configuration settings are explained in the official documentation [80]. For demonstration purposes, refer to the configuration file provided in Listing 8.3 as an example. The Consul needs a data directory to store some data to function correctly.

As visible in Listing 8.1, the `start` method performs the following steps in order: Firstly, it creates the `data` and `config` directories. Then, it removes the existing configuration file to replace it with the latest updated properties. The agent takes these properties from its own configuration, which is pulled from the Git repository. The exact values that are populated into the configuration file are further explained in Section 8.3. The AB agent does the online reload of the configuration in case the Consul process is already running. Only some parts of the configuration are reloadable. To reload configuration, the AB agent sends an HTTP PUT request to the `agent/reload` endpoint. Periodic health checks are scheduled to verify that the Consul is running. If not, the AB agent attempts to restart it. The health checking itself is further described in Section 8.4. The `InteractiveProcessRunner` is a wrapper for `java.lang.ProcessBuilder`, which allows starting a new process with arguments and redirecting the output to a log file. The AB agent passes the `-config-file` option with the file path to the Consul binary.

■ Listing 8.1 Consul starting

```

1  ...
2  @Singleton
3  public class ConsulStarter {
4  ...
5      public void start() {
6          try {
7              Files.createDirectories(this.consulDataFolder);
8              Files.createDirectories(this.configFile.getParent());
9              Files.deleteIfExists(this.configFile);
10             Files.createFile(this.configFile);
11             copyAndPopulateConfig(
12                 applicationConfiguration.getApplicationFile(consulConfigTemplateName)
13             );
14             startConsul();
15         } catch (final Exception e) {
16             logger.atSevere().withCause(e).log("Unable to start consul.");
17         }
18     }
19     ...
20     private void startConsulProcess() {
21         backupOldLog(logFile);
22         final String[] startCmd = new String[] {
23             "./consul", "agent", String.format("-config-file=%s", this.configFile)
24         };
25
26         logger.atInfo().log("Starting Consul Agent");
27         final InteractiveProcessRunner runner =
28             new InteractiveProcessRunner(workingDirectory.toFile(), startCmd);
29         try {
30             runner.startRedirectOutput(logFile);
31         } catch (final IOException e) {
32             throw new IncorrectStartException(
33                 String.format("Cannot start consul in directory %s.", workingDirectory),
34                 IncorrectStartException.CANNOT_EXECUTE_CONSUL,
35                 "Check that the consul is correctly installed."
36             );
37         }
38     }
39     ...
40 }

```

8.2 Stop

Stopping the Consul process is an important consideration. Initially, using the `leave` command [81] via the REST API seemed promising. However, this approach has a distinct drawback. The `leave` command stops the Consul instance and reduces the size of the Raft algorithm cluster. The primary reason for wanting to stop Consul is to allow it to restart in case of malfunctions or when a configuration change that cannot be reloaded online occurs. Therefore, for this use case, stopping Consul without affecting the existing cluster size is desirable. The implementation is depicted in Listing 8.2.

The AB agent creates a backup of the current log file by appending a suffix with the current date and time. This helps in a quicker identification and investigation of any issues.

The current implementation of stopping Consul could be improved by using a PID file to read the process ID instead of calling the `pgrep consul` command.

■ Listing 8.2 Consul stop

```

1  ...
2  @Singleton
3  public class ConsulProcessKiller implements IConsulProcessKiller {
4  ...
5      public boolean tryToKillConsul() {
6          try {
7              final int pid = findConsulPid();
8              if (pid == -1) {
9                  flogger.atInfo().log("Consul process is already not running.");
10             }
11             return killProcess(pid);
12         } catch (final Exception e) {
13             flogger.atSevere().withCause(e).log("Unable to stop consul agent");
14             return false;
15         }
16     }
17 ...
18     private static boolean killProcess(final int pid)
19         throws IOException, InterruptedException {
20         final ProcessBuilder processBuilder =
21             new ProcessBuilder("kill", "-9", Integer.toString(pid));
22         final Process process = processBuilder.start();
23         final int exitCode = process.waitFor();
24
25         if (exitCode == 0) {
26             flogger.atInfo().log("Process killed successfully.");
27             return true;
28         } else {
29             flogger.atSevere().log("Failed to kill process. Exit code: %d", exitCode);
30             return false;
31         }
32     }
33 }
34

```

8.3 Configuration management

The AB agent retrieves its configuration files from a Git repository, which also contains configuration file templates for the Consul server and the client. Each node has a different Consul configuration, and the AB agent fills in the templates with local node-specific information based on its configuration. The template is stored in JSON format. An example of such a file is in Listing 8.3.

The fields populated by the AB agent include the advertise address, node name, datacenter name, Consul data directory name, and retry join addresses. The retry-join addresses refer to the addresses of other members of the Consul cluster. The rest of the fields are self-explanatory, but refer to Consul's documentation for more detailed information. The code for this is in the method `enrichConfig` in Listing 8.4.

8.4 Health checking

The health checking strategy involves executing a sequence of HTTP requests. If a specified number of these requests fail, the Consul process is considered to be not running, and it is restarted. A special case occurs right after starting the Consul, where a longer delay is required before the first health check can be performed to ensure that the Consul is fully operational. The timeouts are currently hard-coded, but it would be better to make them configurable in the

■ Listing 8.3 Consul configuration template file example

```

1  {
2  "server": true,
3  "bootstrap_expect": 3,
4  "client_addr": "0.0.0.0",
5  "datacenter": "Will be populated by AB agent",
6  "data_dir": "../data",
7  "domain": "consul",
8  "advertise_addr": "Will be populated by AB agent",
9  "node_name": "Will be populated by AB agent",
10 "retry_join": ["Will be populated by AB agent"],
11 "..."
12 }

```

■ Listing 8.4 Consul configuration management

```

1  ...
2  @Singleton
3  public class ConsulStarter {
4  ...
5      private void copyAndPopulateConfig(final Path consulConfigPath)
6          throws IOException {
7          final JsonObject config = this.loadConsulConfig(consulConfigPath);
8          this.enrichConfig(config);
9          Files.writeString(this.configFile,
10                          this.gson.toJson(config),
11                          StandardOpenOption.WRITE,
12                          StandardOpenOption.CREATE
13          );
14      }
15  ...
16  /**
17   * Populates the config with node data etc.
18   * @param consulConfig template config to enrich
19   */
20  private void enrichConfig(final JsonObject consulConfig) {
21      final JSONArray retryJoinAddresses = new JSONArray();
22      this.localDeployments
23          .getNodesInCurrentDC()
24          .stream()
25          .map(NodeConfiguration::getSystemListenInterface)
26          .filter(a -> !Objects.equals(a, this.localDeployments.getSystemListenInterface()))
27          .forEachOrdered(retryJoinAddresses::add);
28
29      consulConfig.remove("data_dir");
30      consulConfig.addProperty("data_dir", this.consulDataFolder.toAbsolutePath().toString());
31      consulConfig.remove("advertise_addr");
32      consulConfig.addProperty("advertise_addr", this.localDeployments.getAdvertiseAddress());
33      consulConfig.remove("node_name");
34      consulConfig.addProperty("node_name", this.localDeployments.getLocalNodeName());
35      consulConfig.remove("retry_join");
36      consulConfig.add("retry_join", retryJoinAddresses);
37      consulConfig.remove("datacenter");
38      consulConfig.addProperty("datacenter", this.localDeployments.getDataCenterName());
39  }
40  ...
41  }

```

future. Additionally, just like killing the Consul process, it may be helpful to use PID checks to determine if a process is running correctly. The current implementation is depicted in Listing 8.5.

■ Listing 8.5 Consul health checking

```

1  ...
2  @Singleton
3  public class ConsulConnectionChecker {
4  ...
5      public void scheduleHealthCheck(final int initialDelay, final Runnable startConsulTask) {
6          scheduleNext.set(true);
7          this.scheduledExecutorPool.schedule(
8              initialDelay,
9              () -> this.runAndScheduleCheck(startConsulTask)
10         );
11     }
12     ...
13     public boolean isConsulRunning() {
14         try {
15             return performHttpHealthCheck(healthCheckUrl);
16         } catch (final IOException e) {
17             logger.atSevere().withCause(e).log("Unable to ping ConsulAgent.");
18             return false;
19         }
20     }
21     ...
22     private void runAndScheduleCheck(final Runnable startConsulTask) {
23         int actualDelay = delay;
24         try {
25             if (!isConsulRunning()) {
26                 logger.atWarning().log("Unable to ping Consul local Agent");
27                 if (failedHealthChecksCounter.incrementAndGet() > this.allowedFailedPings) {
28                     startConsulTask.run();
29                     actualDelay = 10_000;
30                     failedHealthChecksCounter.set(0);
31                 }
32             } else {
33                 logger.atFine().atMostEvery(10, TimeUnit.SECONDS)
34                     .log("Able to ping Consul local Agent");
35                 // resetting the counter
36                 failedHealthChecksCounter.set(0);
37             }
38         } finally {
39             if (scheduleNext.get()) {
40                 scheduledExecutorPool.schedule(
41                     actualDelay,
42                     () -> runAndScheduleCheck(startConsulTask)
43                 );
44             }
45         }
46     }
47 }

```

8.5 Integration with other systems using Java API client

This section will explore the process of interacting with the Consul REST API via Rickfast client [46]. The operations that can be performed include registering, deregistering, and subscribing to the service definition. To keep things straightforward, each service definition in the prototype is associated with a unique integer ID, which will be utilised to illustrate the prototype in Section 9.5.

8.5.1 Register service

There exist two techniques for registering a service with the Consul. The initial technique involves directly registering the service into the service catalogue via the `/catalog/register`

endpoint. The request is completed after the changes are committed to the Raft algorithm log and a response is received from the servers. The corresponding operation is called "REGISTER-SERVICES_CATALOG" and it can be found in Listing 8.6.

The second and more recommended technique employs the `/agent/register` endpoint. This technique registers the service through the local agent instance and takes advantage of Consul's anti-entropy mechanism as detailed in the Section 6.8.1. The implemented operation, "REGISTER-SERVICES_LOCAL_AGENT", is depicted in Listing 8.6.

The author of this thesis has incorporated an AB agent operation for both endpoints, in addition to the choice of sending a stream of such services. This strategy ensures that the registrations are spread over time, as a limit of requests per second can be designated as an argument. The procedures closely resemble "REGISTER-SERVICES_CATALOG" and "REGISTER-SERVICES_LOCAL_AGENT", with the sole distinction being the application of the rate limiter prior to each request, indicating a per-second throttle. Methods that are rate limited end with a "FLOW" suffix.

8.5.2 Deregister service

There are corresponding `deregister` methods for each `register` method. Each service is identified based on the string id. During the prototype, the service is identified by the index variable in the for loop to keep things simple. The code for methods is in Listing 8.7.

8.5.3 Subscribe for a service

It is possible to subscribe to changes in the catalogue when a service definition changes in the catalogue. In RickFast [46], this subscription mechanism is referred to as a cache. Its usage is illustrated in Listing 8.8. Internally, it uses the long polling mechanism already mentioned in Figure 6.7. It is possible to remove the subscription.

■ **Listing 8.6** Consul register service

```

1  ...
2  @Singleton
3  @OperationContainer
4  public class ConsulOperationContainer {
5      @Operation(name = "REGISTER_SERVICES_CATALOG")
6      public String registerServices(
7          @Parameter(name = "indexFrom") final int indexFrom,
8          @Parameter(name = "indexTo") final int indexTo) {
9          return foreachDo(indexFrom, indexTo, i -> {
10             final var catalogRegistration
11                 = ImmutableCatalogRegistration.builder()
12                     .node("someNode")
13                     .address("localhost")
14                     .service(generateServiceRegistration(i))
15                     .build();
16             client.catalogClient().register(catalogRegistration);
17         });
18     }
19     ...
20     @Operation(name = "REGISTER_SERVICES_LOCAL_AGENT")
21     public String registerServicesLocal(
22         @Parameter(name = "indexFrom") final int indexFrom,
23         @Parameter(name = "indexTo") final int indexTo) {
24         return foreachDo(indexFrom, indexTo, i -> {
25             client.agentClient().register(
26                 ImmutableRegistration.builder()
27                     .name("service-" + i)
28                     .id("" + i).build()
29             );
30         });
31     }
32     ...
33     @Operation(name = "REGISTER_SERVICES_CATALOG_FLOW")
34     public String registerServicesCatalogFlow(
35         @Parameter(name = "indexFrom") final int indexFrom,
36         @Parameter(name = "indexTo") final int indexTo),
37         @Parameter(name = "rateLimit") {
38         if (rateLimit <= 0) {
39             return "Invalid argument, \"rateLimit\" should be positive.";
40         }
41         final RateLimiter rateLimiter = RateLimiter.create(rateLimit);
42         return foreachDo(indexFrom, indexTo, i -> {
43             final var catalogRegistration
44                 = ImmutableCatalogRegistration.builder()
45                     .node("someNode")
46                     .address("localhost")
47                     .service(generateServiceRegistration(i))
48                     .build();
49             rateLimiter.acquire();
50             client.catalogClient().register(catalogRegistration);
51         });
52     }
53     ...
54     @Operation(name = "REGISTER_SERVICES_LOCAL_AGENT_FLOW")
55     public String registerLocalAgentServicesFlow(...) {
56         // Using ratelimiter analogically as in the REGISTER_SERVICES_CATALOG_FLOW.
57         ...
58     }
59     ...
60 }

```

Listing 8.7 Consul deregister service

```
1 ...
2 @Singleton
3 @OperationContainer
4 public class ConsulOperationContainer {
5     ...
6     @Operation(name = "DEREGISTER_SERVICES_CATALOG")
7     public String deRegisterServices(
8         @Parameter(name = "indexFrom") final int indexFrom,
9         @Parameter(name = "indexTo") final int indexTo) {
10        return foreachDo(indexFrom, indexTo, i -> {
11            final var catalogDeRegistration = ImmutableCatalogDeregistration.builder()
12                                                    .serviceId("" + i)
13                                                    .node("someNode")
14                                                    .build();
15            client.catalogClient().deregister(catalogDeRegistration);
16        });
17    }
18    ...
19    @Operation(name = "DEREGISTER_SERVICES_LOCAL_AGENT")
20    public String deRegisterServicesLocal(
21        @Parameter(name = "indexFrom") final int indexFrom,
22        @Parameter(name = "indexTo") final int indexTo) {
23        return foreachDo(indexFrom, indexTo, i -> {
24            client.agentClient().deregister("service-" + i);
25        });
26    }
27    ...
28 }
29
30
```

■ **Listing 8.8** Consul subscribe for services

```

1  ...
2  @Singleton
3  @OperationContainer
4  public class ConsulOperationContainer {
5  ...
6      @Operation(name = "SUBSCRIBE_SERVICES")
7      public String subscribeServices(
8          @Parameter(name = "indexFrom") final int indexFrom,
9          @Parameter(name = "indexTo") final int indexTo) {
10         return forEachDo(indexFrom, indexTo, i -> {
11             final int finalI = i;
12             final ServiceCatalogCache cache = ServiceCatalogCache.newCache(
13                 client.catalogClient(),
14                 "service-" + i
15             );
16             cache.addListener(new ConsulCache.Listener<String, CatalogService>() {
17                 @Override
18                 public void notify(final Map<String, CatalogService> newValues) {
19                     final var serviceInfo = newValues.get("" + finalI);
20                     if (serviceInfo != null) {
21                         logger.atInfo().log("Received info about service: %s", serviceInfo);
22                     }
23                 }
24             });
25             this.caches.compute(finalI, (key, existingCache) -> {
26                 if (existingCache != null) {
27                     existingCache.stop();
28                 }
29                 return cache;
30             });
31             cache.start();
32         });
33     }
34     ...
35     @Operation(name = "UNSUBSCRIBE_SERVICES")
36     public String unsubscribeServices(
37         @Parameter(name = "indexFrom") final int indexFrom,
38         @Parameter(name = "indexTo") final int indexTo) {
39         return forEachDo(indexFrom, indexTo, i -> this.caches.get(i).stop());
40     }
41     ...
42 }

```


Demonstration of the prototype

The prototype demonstration primarily serves as a proof of concept, showcasing the deployment of a Consul cluster onto LXC's and subsequent interactions. Its core objective is to validate whether the deployed Consul instances function as described in the official documentation and fulfil the requirements mentioned in Chapter 4. Key areas of focus include assessing the failure tolerance of the Raft algorithm, estimating the storage capacity of Consul, measuring response times for requests to the Consul cluster, and exploring the capabilities of GUI.

9.1 Configuration for production environment

The author of the thesis investigated how to configure the Consul cluster for a production environment from the documentation [74]. The framework seems opinionated, so there is no need to specify many settings, and defaults are usually the best choice. The main configuration factor appeared to be the `raft_multiplier` specifying timeouts related to gossip protocol heartbeats and leader election. The recommended value is 1. *“The trade off is between leader stability and time to recover from an actual leader failure. A short multiplier minimises failure detection and election time but may be triggered frequently in high-latency situations. This can cause constant leadership churn and associated unavailability. A high multiplier reduces the chances that spurious failures will cause leadership churn, but it does this at the expense of taking longer to detect real failures and thus takes longer to restore cluster availability.”* [74]

Another crucial factor is the rate limitation of client requests. The `read_rate` and `write_rate` refer to the limits of client requests per second. When the `enforcing` mode is set, the Consul server will stop accepting new requests once the limit is reached. The `permissive` mode is the softest option, with the Consul server only writing information about hitting the limit into the log. The `disabled` mode turns off the rate limit, which is the default option. [74]

The configuration is depicted in Listing 9.1.

9.2 Deployment schema

On-premise servers were utilised to deploy systems, with all hosts being provisioned within LXC's. In subsequent phases, exploring deploying the system in a cloud environment would be beneficial, mainly in exploring the multiple datacenter deployment. However, the current setup suffices for the initial implementation and gaining hands-on experience with the Consul.

The prototype consists solely of the AB agent and Consul cluster, where operations on each agent can be initiated via a gRPC call. The agent operation then triggers an HTTP call to the localhost Consul client using the Rickfast implementation [46]. The code for agent methods is

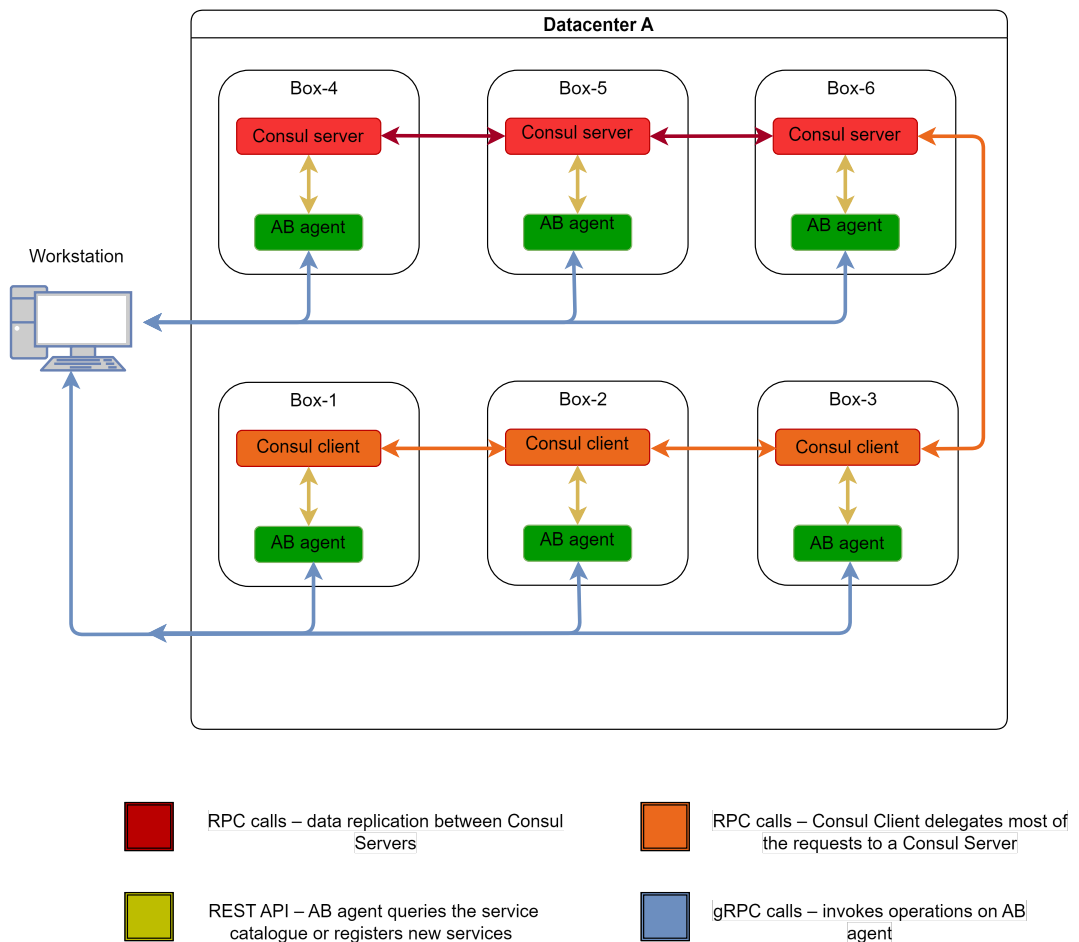
■ **Listing 9.1** Performance fine-tuning

```

1  "performance": {
2    "raft_multiplier": 1
3  },
4  "limits": {
5    "request_limits": {
6      "mode": "enforcing",
7      "read_rate": 10000,
8      "write_rate": 10000
9    }
10 }
    
```

discussed in the preceding Section 8.5. The agent will interact with the Consul cluster, which consists of three servers – the minimum size for a fault-tolerant cluster. An equivalent number of boxes with Consul clients are also deployed in the same datacenter. The complete schema is visible in Figure 9.1.

■ **Figure 9.1** Prototype deployment schema



9.3 Consul GUI

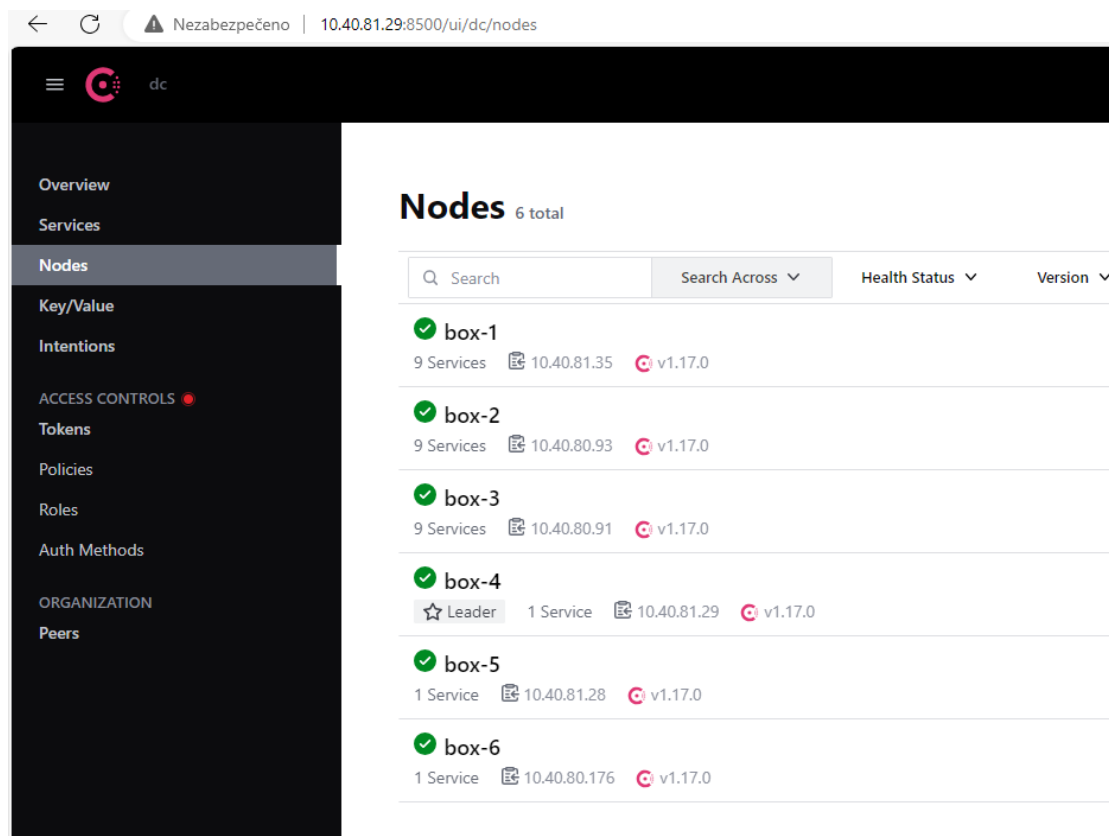
The GUI needs to be enabled from the config file as presented in Listing 9.2. The GUI is accessible only from Consul servers, not from clients. The GUI is accessible only from a Consul server, not a client.

■ Listing 9.2 Enabling the GUI

```
1  "ui_config": {  
2    "enabled": true  
3  }
```

The GUI allows users to access different functionalities such as viewing the cluster's state, browsing stored data, monitoring leader election status, and more. In Figure 9.2, the cluster's state and the marked leader node are visible. In Figure 9.3, there is a list displaying available services, along with information about their health checks' status and the number of instances for each service. Users can click on a service to view more details in Figure 9.4, such as which node each instance runs on, service tags, and routing rules.

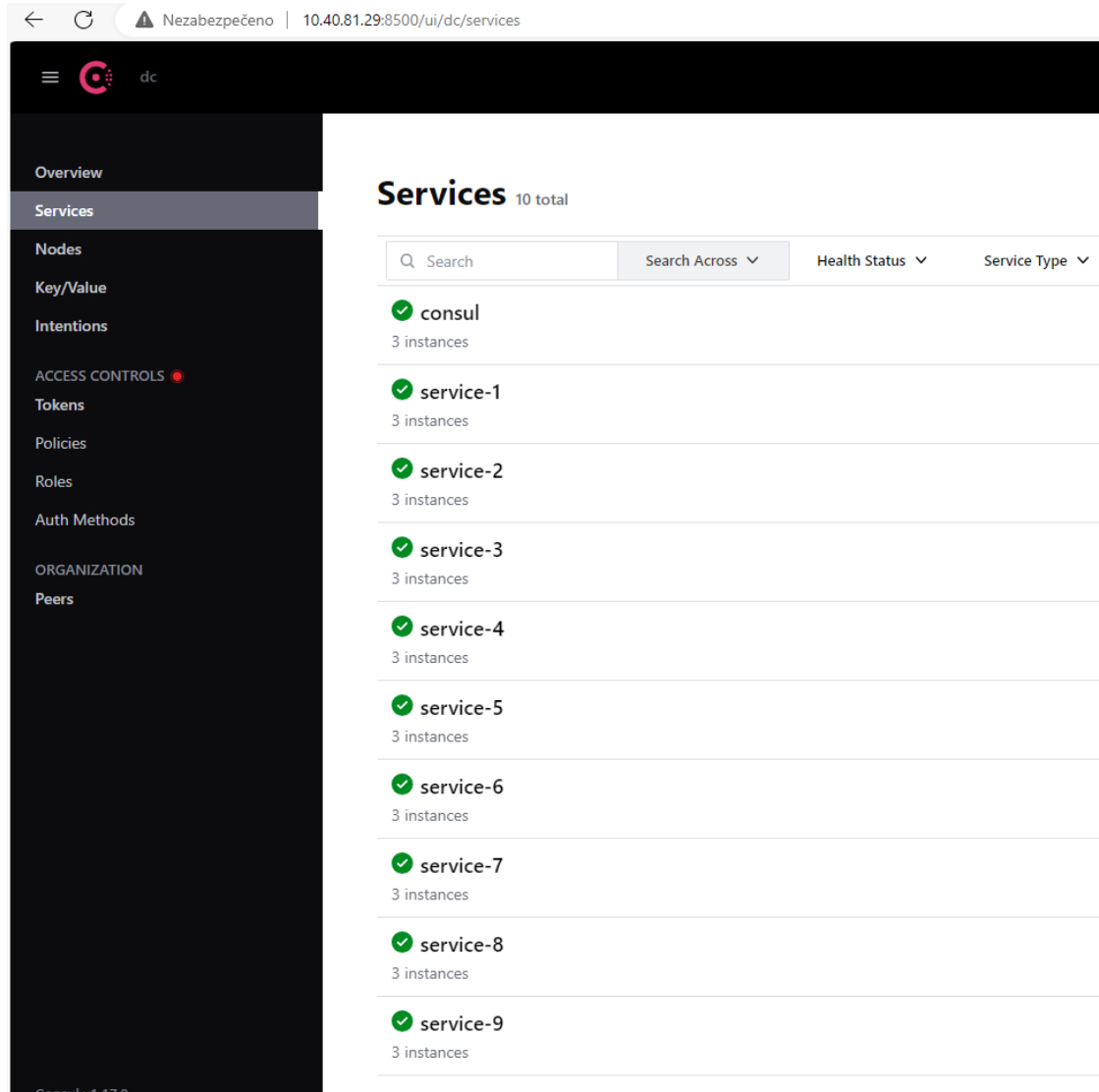
■ Figure 9.2 Consul GUI nodes



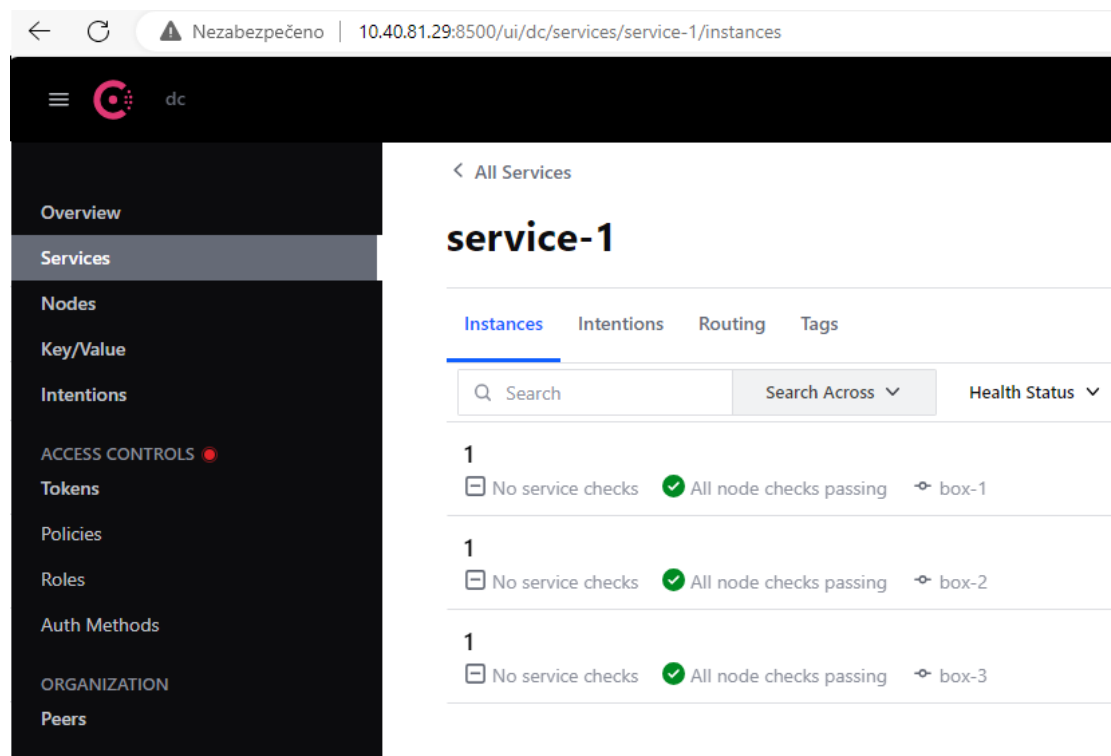
The screenshot shows the Consul GUI interface for the 'dc' datacenter. The left sidebar contains navigation options: Overview, Services, Nodes (selected), Key/Value, Intentions, ACCESS CONTROLS, Tokens, Policies, Roles, Auth Methods, ORGANIZATION, and Peers. The main content area is titled 'Nodes 6 total' and features a search bar and filters for 'Search Across', 'Health Status', and 'Version'. The nodes are listed as follows:

Node Name	Health Status	Services	IP Address	Version
box-1	✓	9	10.40.81.35	v1.17.0
box-2	✓	9	10.40.80.93	v1.17.0
box-3	✓	9	10.40.80.91	v1.17.0
box-4	✓	1 (Leader)	10.40.81.29	v1.17.0
box-5	✓	1	10.40.81.28	v1.17.0
box-6	✓	1	10.40.80.176	v1.17.0

Figure 9.3 Consul GUI services



■ **Figure 9.4** Consul GUI service detail



9.4 Fault tolerance and stale mode testing

The fault tolerance comes from the properties of the Raft algorithm, and the documentation ensures that it should work well. However, it is better to test as the testing is not that time-consuming, and this is a critical feature of the service discovery system.

The testing was done to assert that the leader election, data replication, and consistency work as stated in the Consul documentation. Another task was to check how the consistency mode per HTTP request works.

To test the fault tolerance of the cluster, boxes containing Consul servers were turned on and off repeatedly to check whether each node maintained consistent data. The `stale` request method was used to verify the content of each server, as consensus from the leader is not required to retrieve the data. The entire test scenario is described step by step in following Section 9.4.1.

9.4.1 Testing scenario

1. Set the default consistency mode in the config file and start all three Consul servers.
2. Register some service from any node.
3. **Assert** that all Consul servers keep the same data by sending a `stale` request to each server and comparing the results.
4. Stop the box-1.
5. **Assert** that Consul cluster works with two instances running by registering and unregistering another service
6. Start box-1 and wait for the Consul server to rejoin the cluster
7. **Assert** the same as in step 3. **Assert** that the server on box-1 got synced with the rest of the cluster.
8. Stop box-1 and box-2.
9. **Assert** that normal requests are rejected, only stale requests work.
10. Stop box-3.
11. Start box-1, box-2, box-3.
12. **Assert:** the same as in step 3.

Commands for retrieving and posting the data are shown in Listing 9.3. For simplicity, the `curl` command has been used for interacting with the HTTP endpoint instead of the Rickfast java client [46].

■ Listing 9.3 Commands for fault tolerance testing

```

1 # Command for getting the services using the stale mode
2 curl http://10.40.81.28:8500/v1/catalog/services?stale
3
4 # Command for registering a service into the service catalog
5 curl --request PUT --data @payload.json http://10.40.81.28:8500/v1/catalog/register

```

9.4.2 Results

The test scenario was successful and the cluster operated as expected. The Consul cluster functioned well with only two servers, showing its fault tolerance. With one server, standard requests were rejected because consensus could not happen, and the stale request feature still worked. When a stopped node was restarted, data replication was smooth.

9.5 Performance measurements

Consul makes its metrics available through the `/agent/metrics` endpoint, with data aggregated every 10 seconds. This feature greatly aids in performance assessment and will be valuable for administrators in monitoring the health and efficiency of the Consul cluster based on these metrics. There is a wealth of metrics available, but for now, the author of this thesis has singled out `consul.rpc.server.call` as the most significant metric, as it closely reflects server workload. This metric measures the time taken for an RPC call to respond, measured in milliseconds. Additionally, RAM usage `consul.runtime.alloc_bytes` is of interest since Consul operates in memory. Finally, another aspect worth considering is the number of HTTP requests `consul.api.http` for each endpoint across all nodes.

9.5.1 Scraping telemetry with Prometheus

To gain a comprehensive view of the metrics over time, Prometheus [13] can be used to periodically scrape data from the endpoint. The data can be visualised in the dashboard. There is also a way to download the data from dashboards using the Prometheus API [82]. The example of API usage is in Listing 9.5. Configuring Prometheus on the Consul side is simple. It involves adjusting a few properties in the Consul configuration file. The key aspect is filtering out the important metrics. The exact configuration is shown in Listing 9.4.

■ Listing 9.4 Consul telemetry with Prometheus

```
1  "telemetry": {
2    "prometheus_retention_time" : "300s",
3    "filter_default": false,
4    "enable_host_metrics": true,
5    "prefix_filter": [
6      "-consul",
7      "+consul.api.http",
8      "+consul.rpc.server.call",
9      "+consul.box-1.runtime.alloc_bytes",
10     "+consul.box-1.runtime.sys_bytes",
11     "+consul.box-2.runtime.alloc_bytes",
12     "+consul.box-2.runtime.sys_bytes",
13     "+consul.box-3.runtime.alloc_bytes",
14     "+consul.box-3.runtime.sys_bytes",
15     "+consul.box-4.runtime.alloc_bytes",
16     "+consul.box-4.runtime.sys_bytes",
17     "+consul.box-5.runtime.alloc_bytes",
18     "+consul.box-5.runtime.sys_bytes",
19     "+consul.box-6.runtime.alloc_bytes",
20     "+consul.box-6.runtime.sys_bytes"
21   ]
22 }
```

■ Listing 9.5 Prometheus download data

```
1 curl -g 'http://<prometheus-ip>:<prometheus-port>/api/v1/query_range?query=consul_rpc_server_call
2 &start=2024-04-15T8:25:30.781Z&end=2024-04-15T16:00:30.781Z&step=10s'
```

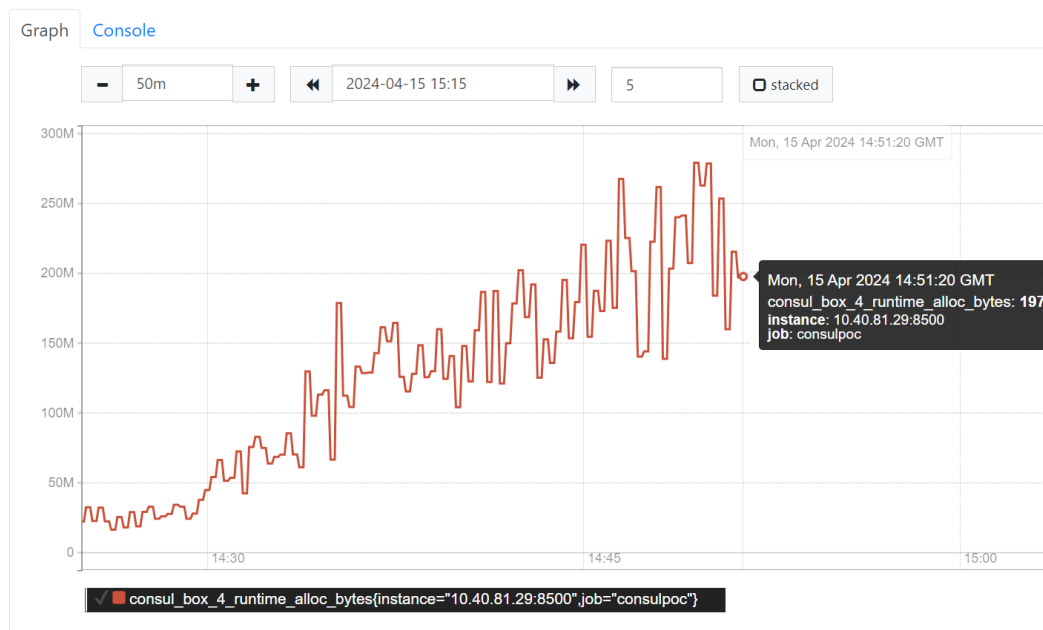
9.5.2 Test scenario

The test scenario was straightforward: 20,000 registration requests, at a rate of 1,000 per second, were simultaneously triggered on box-1 and box-2 Consul clients.

9.5.3 Results

As visible in Figure 9.5, the RAM usage remains within reasonable limits, hovering around 200 MBs even after registering 40,000 services.

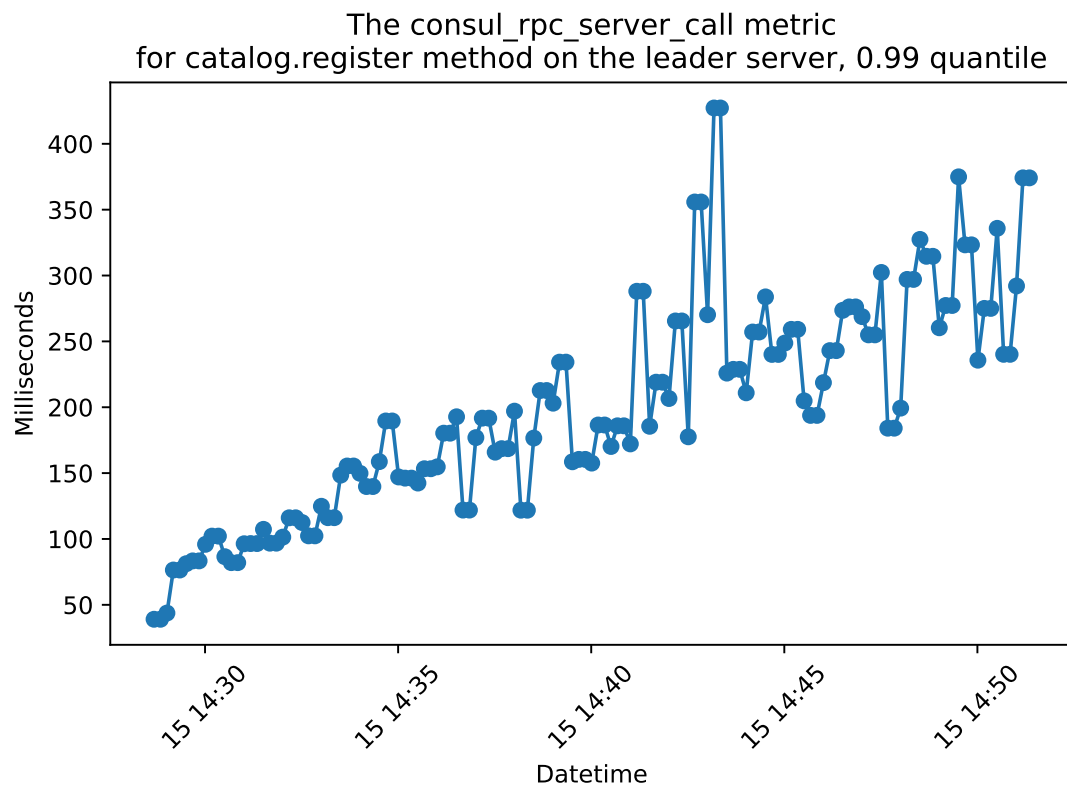
■ Figure 9.5 Memory allocation in MB



As expected, the server response time increases over time. The 0.99 quantile indicates that 99 % of requests are executed within a certain number of milliseconds. The full diagram is shown in Figure 9.6.

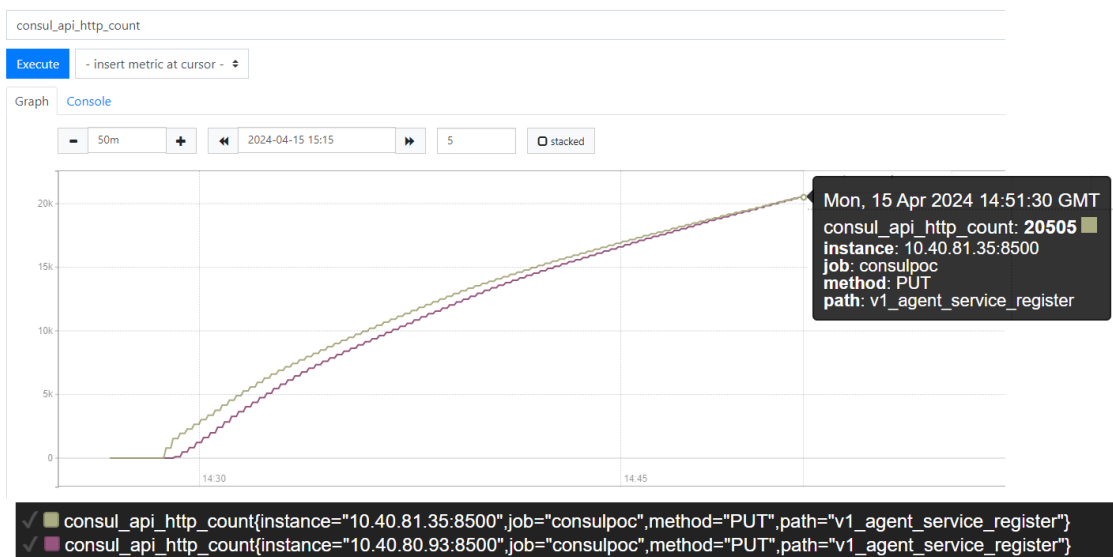
As apparent from Figure 9.7, the number of requests reported by the telemetry module did not match the expected rate of 2,000 requests per second. During 23 minutes, 40,000 services were registered, which resulted in an average of 30 requests per second. At the start, there were 800 requests every 5 seconds, equivalent to 160 requests per second. The official documentation states that workflows that require heavy writing tend to be limited by disk I/O [74]. Discussions online suggest that Consul is designed to favour reads over writes. Conducting further performance testing in a cloud environment where resources can be better controlled may be beneficial. Resource utilisation evaluation on the current LXC setup, which is shared with other LXCs, is challenging. Additionally, it is important to investigate potential problems in the current implementation, such as rate limiting in the AB agent, within the Rickfast client [46], or

■ **Figure 9.6** Leader server response times



on the Consul client side, which could contribute to slowdowns. The current numbers should be sufficient for the existing platform, but scalability concerns may arise with larger deployments, requiring additional performance testing and investigations.

■ **Figure 9.7** Count of registration calls



Conclusion

The thesis aimed to identify the optimal service discovery solution to facilitate the transition of the current platform to a microservice architecture and cloud environment.

The steps involved familiarising with the theory of distributed systems and service discovery, gathering abstract requirements and researching existing frameworks while simultaneously refining requirements through team discussion. Following this, a thorough comparison of existing solutions was undertaken to justify the selection of HashiCorp Consul as the most suitable technology. Subsequently, a solution was designed, and a prototype was developed, showcasing Consul's functionality and potential within the company's platform.

The outcome of the thesis was a recommendation for Consul, supported by the implementation of a prototype integrated into the company's source code, paving the way for future performance testing and aligning with the company's long-term vision of migrating to the cloud.

10.1 Future work

As mentioned in the previous sections, several future steps need to be taken before refactoring the current service discovery solution can be considered complete. The ideal next steps are to expand the prototype to cover multiple datacenters, conduct further performance tests to ensure it meets the platform's requirements, and deploy it in a cloud environment to understand its behaviour better there.

After this final round of testing, it should be ready to receive a green light for refactoring. This will involve replacing the current implementation of the service discovery with Consul and utilising the code already written for a prototype.

Bibliography

- [1] Apache. *Apache Curator*.
URL: <https://github.com/apache/curator> (visited on 03/21/2024).
- [2] ZooKeeper Authors. *ZooKeeper Internals*. Mar. 12, 2024.
URL: <https://zookeeper.apache.org/doc/r3.9.2/zookeeper0ver.html> (visited on 03/23/2024).
- [3] Seth Gilbert and Nancy Lynch. “Perspectives on the CAP Theorem”.
In: *Computer* 45.2 (2012), pp. 30–36.
- [4] HashiCorp Authors. *What is Consul?* HashiCorp, Inc.
URL: <https://developer.hashicorp.com/consul/docs/intro> (visited on 03/23/2024).
- [5] Ramtin Jabbari et al.
“What is DevOps? A systematic mapping study on definitions and practices”.
In: *Proceedings of the scientific workshop proceedings of XP2016*. 2016, pp. 1–11.
- [6] Adaoma Ezenwe, Eoghan Furey, and Kevin Curran.
“Mitigating Denial of Service Attacks with Load Balancing”.
In: *Journal of Robotics and Control (JRC)* 1.4 (2020), pp. 129–135.
- [7] EtcD Authors. *What is etcd?* Apr. 20, 2023.
URL: <https://etcd.io/docs/v3.5/faq/> (visited on 03/21/2024).
- [8] Scott Chacon and Ben Straub. *Pro git*. Springer Nature, 2014.
- [9] Go Authors. *The Go Programming Language*.
URL: <https://go.dev/> (visited on 04/25/2024).
- [10] HashiCorp Authors. *HashiCorp*. HashiCorp, Inc.
URL: <https://www.hashicorp.com/> (visited on 04/25/2024).
- [11] Janet Kuhn. “Decrypting the MoSCoW analysis”.
In: *The workable, practical guide to Do IT Yourself* 5 (2009).
- [12] Netflix Eureka Authors. *What is Eureka?* Netflix Eureka. Dec. 18, 2014.
URL: <https://github.com/Netflix/eureka/wiki/Eureka-at-a-glance> (visited on 03/23/2024).
- [13] Prometheus Authors. *What is Prometheus?*
URL: <https://prometheus.io/docs/introduction/overview/> (visited on 04/16/2024).
- [14] Diego Ongaro and John Ousterhout.
“In search of an understandable consensus algorithm (extended version)”.
In: *Proceeding of USENIX annual technical conference, USENIX ATC*. 2014, pp. 19–20.

- [15] Redis Authors. *Introduction to Redis*. URL: <https://redis.io/docs/about/> (visited on 03/21/2024).
- [16] HashiCorp Authors. *Documentation*. HashiCorp, Inc. URL: <https://www.serf.io/docs/index.html> (visited on 04/25/2024).
- [17] MongoDB Authors. *Key-Value Databases*. MongoDB, Inc. URL: <https://www.mongodb.com/databases/key-value-database> (visited on 12/11/2023).
- [18] Ben Lutkevich and Alexander S. Gillis. *Definition of High Availability (HA)*. TechTarget. Apr. 1, 2021. URL: <https://www.techtarget.com/searchdatacenter/definition/high-availability> (visited on 04/22/2024).
- [19] R.V. White and F.M. Miles. “Principles of fault tolerance”. In: *Proceedings of Applied Power Electronics Conference. APEC '96*. Vol. 1. 1996, 18–25 vol.1. DOI: 10.1109/APEC.1996.500416.
- [20] Martin Fowler and James Lewis. *Microservices*. Mar. 25, 2014. URL: <https://martinfowler.com/articles/microservices.html> (visited on 03/25/2024).
- [21] Martin Fowler. *Microservice Trade-Offs*. July 1, 2015. URL: <https://martinfowler.com/articles/microservice-trade-offs.html> (visited on 03/26/2024).
- [22] Rishabh Batra. *Eventual Consistency in Distributed Systems — Learn System Design*. Feb. 21, 2024. URL: <https://www.geeksforgeeks.org/eventual-consistency-in-distributive-systems-learn-system-design/#what-is-eventual-consistency> (visited on 03/27/2024).
- [23] Eric Brewer. *CAP Twelve Years Later: How the “Rules” Have Changed*. June 30, 2012. URL: <https://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed/> (visited on 12/23/2023).
- [24] A. Khiyaita et al. “Load balancing cloud computing: State of art”. In: *2012 National Days of Network Security and Systems*. 2012, pp. 106–109. DOI: 10.1109/JNS2.2012.6249253.
- [25] Vishva Desai, Yash Koladia, and Suvarna Pansambal. “Microservices: architecture and technologies”. In: *Int. J. Res. Appl. Sci. Eng. Technol* 8.10 (2020), pp. 679–686.
- [26] Dr. Martin Kleppmann. *Concurrent and Distributed Systems*. Michaelmas term. University of Cambridge. 2021. URL: <https://www.cl.cam.ac.uk/teaching/2122/ConcDisSys/dist-sys-notes.pdf> (visited on 04/24/2024). Lecture videos available at <https://www.youtube.com/playlist?list=PLeKd45zvjcDFUEvohrHdUFe97RItdiB>.
- [27] Prateek Gupta. *Gossip Protocol in distributed systems*. Apr. 23, 2022. URL: <https://medium.com/nerd-for-tech/gossip-protocol-in-distributed-systems-e2b0665c7135> (visited on 12/25/2023).
- [28] Martin Kleppmann. *Designing data-intensive applications*. 2019.
- [29] Hesam Nejati Sharif Aldin et al. “Consistency models in distributed systems: A survey on definitions, disciplines, challenges and applications”. In: *arXiv preprint arXiv:1902.03305* (2019).
- [30] Lamport. “How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs”. In: *IEEE Transactions on Computers* C-28.9 (1979), pp. 690–691. DOI: 10.1109/TC.1979.1675439.

- [31] Martin Kleppmann. “A Critique of the CAP Theorem”. In: *arXiv preprint arXiv:1509.05393* (2015).
- [32] ZooKeeper Authors. *ZooKeeper Internals*. July 18, 2024. URL: <https://zookeeper.apache.org/doc/r3.9.2/zookeeperInternals.html> (visited on 02/05/2024).
- [33] Tomas Vondra. *Containers - principles and Docker*. Czech technical university in Prague. Apr. 3, 2022. URL: <https://courses.fit.cvut.cz/NIE-VCC/> (visited on 04/22/2024).
- [34] Petr Zemanek. *Introduction to (Linux) Kernel*. Czech technical university in Prague. URL: <https://courses.fit.cvut.cz/NI-OSY/> (visited on 04/22/2024).
- [35] HashiCorp Authors. *Consul Vocabulary*. HashiCorp, Inc. URL: <https://developer.hashicorp.com/consul/tutorials/production-deploy/deployment-guide> (visited on 12/11/2023).
- [36] Netflix Eureka Authors. *Netflix Eureka*. URL: <https://github.com/Netflix/eureka> (visited on 02/05/2024).
- [37] Etcd Authors. *etcd versus other key-value stores*. URL: <https://etcd.io/docs/v3.5/learning/why/> (visited on 02/05/2024).
- [38] Heather Bennett. *Is Redis a Data Structure?* July 1, 2023. URL: <https://serverlogic3.com/is-redis-a-data-structure/> (visited on 04/27/2024).
- [39] Redis authors. *Scale with Redis Cluster*. URL: <https://redis.io/docs/management/scaling/> (visited on 03/11/2024).
- [40] Redis authors. *Redis Sentinel Documentation*. URL: <https://cndoc.github.io/redis-doc-cn/cn/topics/sentinel.html> (visited on 03/11/2024).
- [41] Redis authors. *Cluster architecture*. URL: <https://redis.io/redis-enterprise/technology/redis-enterprise-cluster-architecture/> (visited on 03/11/2024).
- [42] HashiCorp Authors. *Consistency modes*. HashiCorp, Inc. URL: <https://developer.hashicorp.com/consul/api-docs/features/consistency> (visited on 12/26/2023).
- [43] Etcd Authors. *Design learner*. July 23, 2021. URL: <https://etcd.io/docs/v3.5/learning/design-learner/> (visited on 02/05/2024).
- [44] HashiCorp Authors. *Consul Enterprise*. HashiCorp, Inc. URL: <https://developer.hashicorp.com/consul/docs/enterprise> (visited on 03/24/2024).
- [45] Redis Authors. *Redis Enterprise*. Redis Ltd. URL: <https://redis.io/docs/about/redis-enterprise/> (visited on 03/24/2024).
- [46] Rick Fast. *Consul Client for Java*. URL: <https://github.com/rickfast/consul-client> (visited on 03/21/2024).
- [47] Etcd Authors. *jetcd - A Java Client for etcd*. URL: <https://github.com/etcd-io/jetcd> (visited on 03/21/2024).
- [48] Redis Authors. *Jedis*. URL: <https://github.com/redis/jedis> (visited on 03/21/2024).
- [49] Etcd Authors. *Interacting with etcd*. June 14, 2021. URL: https://etcd.io/docs/v3.5/dev-guide/interacting_v3/ (visited on 03/21/2024).
- [50] Zookeeper Authors. *ZooKeeper Programmer’s Guide*. URL: https://zookeeper.apache.org/doc/r3.9.2/zookeeperProgrammers.html#ch_zkWatches (visited on 03/21/2024).

- [51] Redis authors. *Keyspace triggers*. URL: https://redis.io/docs/latest/develop/interact/programmability/triggers-and-functions/concepts/triggers/keyspace_triggers/ (visited on 03/11/2024).
- [52] HashiCorp Authors. *Watches Overview and Reference*. HashiCorp, Inc. URL: <https://developer.hashicorp.com/consul/docs/dynamic-app-config/watches> (visited on 12/29/2023).
- [53] HashiCorp Authors. *Memory requirements*. HashiCorp, Inc. URL: <https://developer.hashicorp.com/consul/docs/install/performance#memory-requirements> (visited on 03/21/2024).
- [54] Etcd Authors. *Storage size limit*. Apr. 20, 2023. URL: <https://etcd.io/docs/v3.5/dev-guide/limit/#storage-size-limit> (visited on 03/21/2024).
- [55] Redis Authors. *Database memory limits*. Feb. 1, 2024. URL: <https://docs.redis.com/latest/rs/databases/memory-performance/memory-limit/> (visited on 03/21/2024).
- [56] Redis Authors. *Keyspace*. URL: <https://redis.io/docs/manual/keyspace/> (visited on 03/21/2024).
- [57] HashiCorp Authors. *Watches Overview and Reference*. HashiCorp, Inc. URL: <https://developer.hashicorp.com/consul/tutorials/certification-associate-tutorials/get-started-explore-the-ui> (visited on 12/29/2023).
- [58] Netflix Eureka Authors. *Spring Cloud Netflix*. HashiCorp, Inc. URL: <https://github.com/spring-cloud/spring-cloud-netflix> (visited on 12/29/2023).
- [59] Tamas Geschitz. *etcdmanager*. URL: <https://github.com/gtamas/etcdmanager> (visited on 03/21/2024).
- [60] Lubos Kozmon. *zoonavigator*. URL: <https://github.com/elkozmon/zoonavigator> (visited on 03/21/2024).
- [61] Redis Authors. *Redis insight*. Redis Ltd. URL: <https://redis.com/redis-enterprise/redis-insight/> (visited on 03/21/2024).
- [62] Etcd Authors. *Compacted revisions*. June 14, 2021. URL: https://etcd.io/docs/v3.5/dev-guide/interacting_v3/#compacted-revisions (visited on 03/25/2024).
- [63] HashiCorp Authors. *Consul*. HashiCorp, Inc. URL: <https://github.com/hashicorp/consul> (visited on 04/28/2024).
- [64] HashiCorp Authors. *7 Years On: Remembering the Origins of HashiCorp Consul*. Hashicorp, Inc. Oct. 21, 2020. URL: <https://www.hashicorp.com/resources/7-years-on-remembering-the-origins-of-hashicorp-consul> (visited on 12/25/2023).
- [65] HashiCorp Authors. *The Tao of HashiCorp*. Hashicorp, Inc. URL: <https://www.hashicorp.com/tao-of-hashicorp> (visited on 12/25/2023).
- [66] HashiCorp Authors. *Consul use cases*. HashiCorp, Inc. URL: <https://www.hashicorp.com/products/consul/use-cases> (visited on 04/28/2024).
- [67] HashiCorp Authors. *Health checks*. HashiCorp, Inc. URL: <https://developer.hashicorp.com/consul/docs/services/usage/checks> (visited on 04/28/2024).
- [68] HashiCorp Authors. *Sessions and Distributed Locks Overview*. HashiCorp, Inc. URL: <https://developer.hashicorp.com/consul/docs/dynamic-app-config/sessions> (visited on 04/28/2024).

- [69] HashiCorp Authors. *Consul API Overview*. HashiCorp, Inc.
URL: <https://developer.hashicorp.com/consul/api-docs> (visited on 12/26/2023).
- [70] HashiCorp Authors. *Consul Architecture*. HashiCorp, Inc.
URL: <https://developer.hashicorp.com/consul/docs/architecture> (visited on 12/10/2023).
- [71] HashiCorp Authors. *Consul Vocabulary*. HashiCorp, Inc.
URL: <https://developer.hashicorp.com/consul/docs/install/glossary> (visited on 12/11/2023).
- [72] HashiCorp Authors. *Consul Architecture*. HashiCorp, Inc.
URL: <https://developer.hashicorp.com/consul/docs/architecture#cross-datacenter-requests> (visited on 12/10/2023).
- [73] HashiCorp Authors. *Install Consul*. Hashicorp, Inc. URL:
<https://developer.hashicorp.com/consul/docs/install> (visited on 12/25/2023).
- [74] HashiCorp Authors. *Memory requirements*. HashiCorp, Inc.
URL: <https://developer.hashicorp.com/consul/docs/install/performance> (visited on 04/12/2024).
- [75] HashiCorp Authors. *Anti-Entropy Enforcement*. HashiCorp, Inc.
URL: <https://developer.hashicorp.com/consul/docs/architecture/anti-entropy> (visited on 12/26/2023).
- [76] HashiCorp Authors. *Agent caching*. HashiCorp, Inc.
URL: <https://developer.hashicorp.com/consul/api-docs/features/caching> (visited on 12/26/2023).
- [77] Pierre Souchay. *Consul Streaming: What's behind it?* Criteo R&D Blog. Feb. 2, 2021.
URL: <https://medium.com/criteo-engineering/consul-streaming-whats-behind-it-6f44f77a5175> (visited on 12/10/2023).
- [78] PubNub Authors. *What is Long Polling?* PubNub.
URL: <https://www.pubnub.com/guides/long-polling/> (visited on 04/27/2024).
- [79] HashiCorp Authors. *Service configuration with Consul Template*. HashiCorp, Inc.
URL: <https://developer.hashicorp.com/consul/tutorials/developer-configuration/consul-template> (visited on 12/29/2023).
- [80] HashiCorp Authors. *Agents Configuration File Reference*. HashiCorp, Inc.
URL: <https://developer.hashicorp.com/consul/docs/agent/config/config-files> (visited on 12/29/2023).
- [81] HashiCorp Authors. *Agent HTTP API*. HashiCorp, Inc.
URL: <https://developer.hashicorp.com/consul/api-docs/agent#graceful-leave-and-shutdown> (visited on 12/26/2023).
- [82] Prometheus Authors. *HTTP API*.
URL: <https://prometheus.io/docs/prometheus/latest/querying/api/> (visited on 04/16/2024).

Contents of the enclosed media

README.md.....	a brief description of the content for this folder
DemonstrationAttachment.pdf.....	screenshots for demonstration of the prototype
FaultToleranceTest	
README.md.....	a brief description of the content for this folder
testReport.log.....	recorded output from terminal with the fault tolerance testing mentioned in Section 9.4.1
register.json.....	the first payload for the service registration operation
register2.json.....	the second payload for the service registration operation
deregister.json.....	the first payload for the service deregistration operation
deregister2.json.....	the second payload for the service deregistration operation
thesis.zip.....	the source folder for L ^A T _E X
thesis.pdf.....	the text of the thesis in PDF format