HAW HAMBURG

MASTERTHESIS
Daniel Sarnow

Fault-Tolerant Service Discovery for the Internet of Things

FAKULTÄT TECHNIK UND INFORMATIK
Department Informatik

Faculty of Computer Science and Engineering
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Daniel Sarnow

Fault-Tolerant Service Discovery for the Internet of Things

Masterarbeit eingereicht im Rahmen der Masterprüfung
im Studiengang *Master of Science Informatik*
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Martin Becke
Zweitgutachter: Prof. Dr. Jan Sudeikat

Eingereicht am: 09. Juni 2021

Daniel Sarnow

Thema der Arbeit

Fault-Tolerant Service Discovery for the Internet of Things

Stichworte

Verzeichnisdienst, Internet der Dinge, Fehlertoleranz, Fault-Injection

Kurzzusammenfassung

Das Ziel des Internets der Dinge (IoT) ist es eine Vielzahl an heterogenen Geräten über ein globales Netzwerk zu verbinden, so dass jeder mit jedem über jedes Protokoll und Netzwerk miteinander interagieren kann. Essentiell für eine solche Umgebung sind robuste und fehlertolerante Infrastrukturdienste. Einer davon ist *service discovery*. Das verteilte, heterogene und sehr dynamische Verhalten im IoT führt zu neuen Anforderungen und insbesondere auch Herausforderungen für diese Dienste. Diese Arbeit stellt einen Ansatz für einen solchen Verzeichnisdienst vor und evaluiert diesen mit Hilfe von *fault-injection* Methoden.

Daniel Sarnow

Title of Thesis

Fault-Tolerant Service Discovery for the Internet of Things

Keywords

discovery service, Internet of Things, fault-tolerance, fault-injection

Abstract

The Internet of Things (IoT) is envisioned to bring a multitude of heterogeneous devices together; interconnected via a global network infrastructure. In order to provide such an environment, it is crucial to provide IoT applications and devices with robust and fault-tolerant infrastructure services that minimize the need for human intervention. Such an essential infrastructure service is *discovery*. The distributed, dynamic and heterogeneous nature of the IoT however brings new challenges and requirements to the design and development of discovery services. These challenges have to be addressed by IoT implementations. Therefore, this thesis evaluates the challenges that discovery mechanisms face in the context of the IoT, proposes a fault-tolerant discovery service approach, evaluates this approach and provides a framework that can help to integrate fault-injection experiments in the development cycle.

Contents

List of Figures

List of Tables

Listings

Acronyms

CaDDS Context-aware Distributed Discovery Service.

DHT Distributed Hash Table.

ICN information-centric networking.

IoT Internet of Things.

JSON JavaScript Object Notation.

M2M machine-to-machine.

NDN named data networking.

NVP n-version programming.

OWL Web Ontology Language.

OWL-S Web Ontology Language for Web Services.

RDF Resource Description Framework.

RDFS Resource Description Framework Schema.

SPOC single point of communication.

SPOF single point of failure.

TCP Transmission Control Protocol.

XML eXtensible Markup Language.

1 Introduction

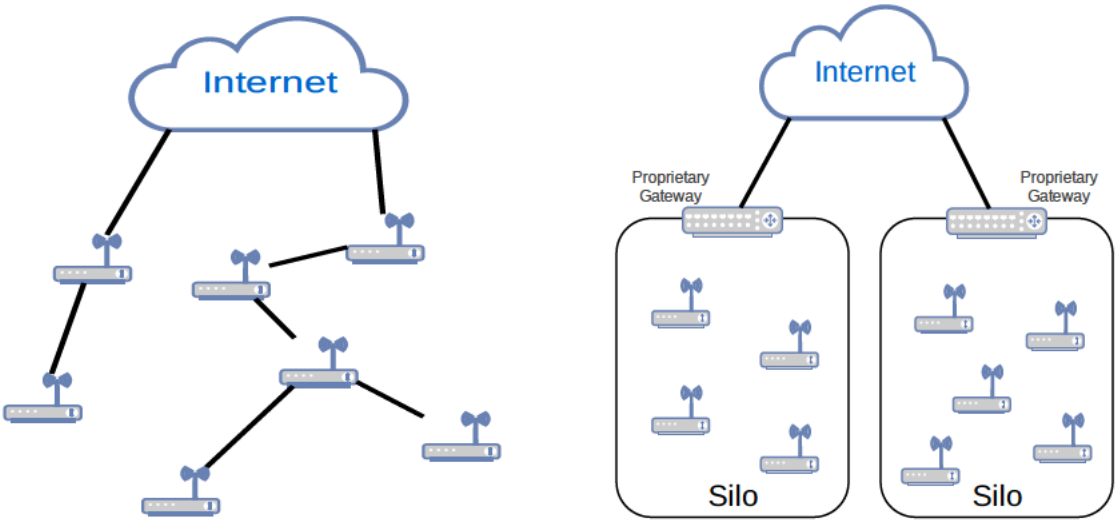
This chapter outlines the motivation of this thesis, shorty illustrates the challenges that come with *Service Discovery* in an *Internet of Things* environment, defines the goals of this work and finally gives a short overview of the thesis' structure.

1.1 Motivation

The Internet of Things (IoT) is envisioned to bring a multitude of heterogeneous devices together; interconnected via a global network infrastructure as illustrated in Figure 1.1a. However, currently the IoT is made up of manufacturer-oriented silos that connect proprietary devices via a proprietary gateway to the internet as illustrated in Figure 1.1b. From this it follows, that only devices from the same manufacturer can really communicate and interact with each other. This goes against the idea of a open and interoperable network that can connect heterogeneous devices that can interact with each other.

In order to provide an open and interoperable global network infrastructure, it is crucial to provide IoT applications and devices with robust and fault-tolerant infrastructure services that minimize the need for human intervention. Such an essential infrastructure service is *discovery*. It is generally accepted that effective service discovery is of great importance for the interoperability and openness of the future Internet of Things [38].

The distributed, dynamic and heterogeneous nature of the IoT however brings new challenges and requirements to the design and development of discovery services. To illustrate; such challenges include IoT devices that operate in Low-Power and Lossy Networks (IoT). Low-Power and Lossy Network are typically composed of many embedded devices with limited power, memory, and processing resources interconnected by a variety of links, such as IEEE 802.15.4 or low-power Wi-Fi [46]. These links are usually characterized by high loss rates, low bandwidth, and instability. One can easily imagine that use cases such as industrial monitoring, building automation, smart homes, health care,



(a) Idea (b) Reality

Figure 1.1: Internet of Things

environmental monitoring, urban sensor networks, energy management, assets tracking, and refrigeration are subject to LLNs [46].

These challenges have to be addressed by IoT implementations. This applies to a particular high degree to IoT infrastructure services, such as discovery services, as they provide fundamental services to the whole environment. IoT infrastructure services are expected to continue normal operation despite the presence of hardware or software faults. Thus, *robustness* and *fault tolerance* are essential design goals for all IoT infrastructure services. However, most proposed discovery mechanisms do not take *robustness* or *fault tolerance* into account. Therefore, this thesis evaluates the challenges that discovery mechanisms face in the context of the IoT, proposes a fault-tolerant discovery service approach, evaluates this approach and provides a framework that can help to integrate fault-injection experiments in the development cycle.

1.2 Goals

This master thesis proposes a distributed fault-tolerant discovery service – called *Context-aware Distributed Discovery Service (CaDDS)* – for the Internet of Things (IoT). The approach enables consumers to discover services on a local and global scale. The proposed approach is evaluated using different degrees of fault-tolerance measures in order to gain a

better understanding about the value of different fault-tolerance measures in the context of the IoT.

Furthermore, the thesis proposes a test framework for measuring the degree of fault-tolerance of a given IoT service. The test framework is specifically tailored to simulate the characteristics of the IoT.

1.3 Organization

The thesis is organized as follows:

Chapter 2 – Challenges and Characteristics – provides an overview of the topics: *Internet of Things*, *Discovery Services*, and *Fault-Tolerance*.

Chapter 3 – Related Work – provides a short overview of the scientific work related to the development of *discovery services* for the *Internet of Things*.

Chapter 4 – Context-Aware Distributed Discovery Service – presents the proposed context-aware and distributed service discovery approach for the Internet of Things.

Chapter 5 – Experiment Setup – describes the experiment environment and procedures that are applied to the measurements performed in this thesis. Furthermore, network measurement and analysis tools are presented that were used to help identify the level of fault-tolerance.

Chapter 6 – Evaluation – discusses the challenges of fault-tolerance in the context of the Internet of Things. Evaluates the proposed approach.

Chapter 7 – Conclusion – summarizes the findings of this thesis and provides an outlook for future work.

2 Challenges and Characteristics

This chapter provides an overview of the topics *Internet of Things*, *discovery services*, *context* and *fault tolerance*. The topics are introduced by characterizing the major concepts and challenges that need to be taken into account for the design of infrastructure services . This forms the basis for the evaluation of related work in chapter 3 and needs to be taken into account for the design of the proposed approach in chapter 4.

2.1 Internet of Things

The Internet of Things (IoT) is visualized to seamlessly bring a multitude of heterogeneous devices together that interact with each other via a global network infrastructure [3].

Even though the IoT is a widely used term a definitive commonly-accepted definition can not be provided. The European Research Cluster on IoT (IERC) defines the IoT in [41] as: The Internet of Things allows people and things to be connected anytime, anyplace, with anything and anyone, ideally using any path/network, and any service. IERC's understanding of the Internet of Things is illustrated in Figure 2.1.

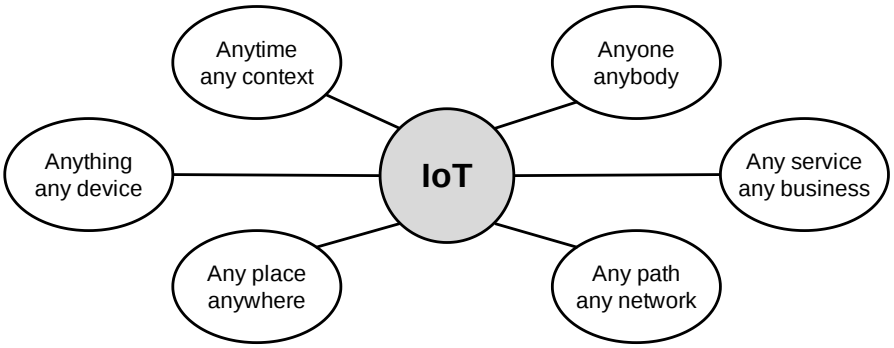


Figure 2.1: Definition of the Internet of Things according to IERC [41]

Atzori et al. evaluate in [3] numerous definitions of IoT and come to the following conclusion: The Internet of Things is a conceptual framework that leverages on the availability of heterogeneous devices and interconnection solutions, as well as augmented physical objects providing a shared information base on global scale, to support the design of applications involving at the same virtual level both people and representations of objects. [3]

The IoT is characterized by *heterogeneous* and often *constrained* devices that are part of a ultra-large scale global network infrastructure with large numbers of events and spontaneous interactions in a dynamically changing environment [38]. Based on the proposed requirements for middleware in the IoT from [38], this thesis proposes a set of requirements that are crucial for IoT discovery services:

- Discovery services need to be highly scalable in order to accommodate growth in connected devices and their accommodating services.
- In order to support many different use cases, discovery services should allow discovery on a different scales. The discovery scope can reach from a few feet, to local, corporate and also global scales.
- Many IoT use cases not only rely on service information at the location of the requester, but also at remote locations. The discovery mechanisms should support such use cases.
- IoT applications also often depend not only on logical correctness but also on timeliness as many IoT services and applications provide a near real-time state of the physical world. Thus, discovery services need to provide real-time services to IoT applications.
- Reliability and availability is vital for discovery services. Therefore, it is crucial for a discovery mechanism to be fault-tolerant.
- Security and privacy also need to be taken into account when designing discovery services.

Architecturally discovery services need to be distributed in order to fulfill nonfunctional requirements, like scalability and availability.

- In order to control complexity, self-governance (autonomy) and self-management (autonomicity) need to be achieved.

- Moreover, it needs to be taken into account that the level of heterogeneity of technologies in the IoT domain is high. Thus, interoperability is a crucial requirement. In order to support interoperability and to overcome the challenges of heterogeneous devices, a discovery service might need to be semantics-aware if no common vocabulary is used.
- The importance of context-awareness in the IoT is discussed in subsection 2.3.

2.2 Discovery Services

Service discovery describes the ability to locate services that comply with a set of requirements without prior knowledge about services. Ververidis et al. characterizes service discovery in [47] as follows: Service Discovery is a process that enables connected devices to advertise their services, query services provided by other entities, select the most appropriately matched services and invoke services. MacKenzie et al. [29] defines a service as follows: A service is a mechanism to enable access to one or more capabilities via a prescribed interface and is exercised consistent with constraints and policies as specified by the service description. This paper adopts the proposed definition of service discovery from [47] and the definition of a service as proposed in [29].

Service discovery protocols can be partitioned in Non-IP-based, such as ZigBee and Bluetooth SDP, and IP-based approaches. This work only focuses on IP-based approaches, because these are more significant in IoT scenarios than Non-IP-based approaches [3].

2.2.1 Architecture

Discovery service architectures can be categorized into three basic types:

- **Directory-less** approaches do not use a directory to store service information. Service information dissemination relies on multicast, broadcast or some form of flooding [47].
- **Directory-based** approaches use a directory to store service information.
- **Hybrid** approaches also use a directory to store service information. If a service cannot locate a directory server, a directory-less mechanism can be used to advertise a service [47].

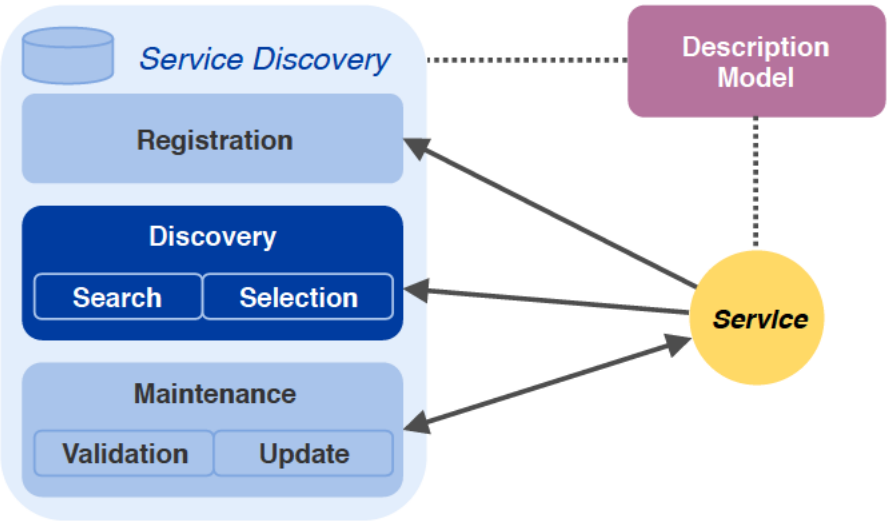


Figure 2.2: General Directory-Based Discovery Service Architecture

This thesis proposes a general architecture for discovery services as depicted in Figure 2.2. Figure 2.2 illustrates the differences between the different architectural types. In contrast to directory-less approaches, directory-based approaches also have to implement mechanisms for registration and maintenance (highlighted in light blue in Figure 2.2). Every Discovery Service is based on a common description model (highlighted in violet in Figure 2.2). Description models can be implemented by one of the following data types: continuous text, key-value, markup-scheme, graphical, object-based, logic-based or ontology-based. All approaches have to implement a discovery mechanism (highlighted in dark blue in Figure 2.2). The discovery procedure can be divided into two parts: *search* and *selection*.

Directory-based Directory-based architectures use a directory to store service descriptions. Ideally the directory's state represents a near real-time view of the services' states. However, achieving this goal might come at a great price. In order to keep the directory up-to-date maintenance routines have to be run periodically. The maintenance routines have to be run by the discovery service, the service providers or by both parties. The freshness of the service state information depends on the update interval. The ideal interval value for efficient updates is not a trivial task. The ideal update interval can differ for service providers of different domains. An update interval that too low can overwhelm a service provider as most resources will be used to keep the directory up-to-date as opposed to handling consumer requests.

Directory-based architectures can be further categorized as overlay-based and overlay-less approaches. Overlay-based approaches can be used to control the multicast of discovery queries. A controlled multicast restricts and can greatly reduce the network traffic. Furthermore, overlay networks in the form of distributed hash tables can be used as a distributed directory. However, overlay networks require nodes to periodically perform routines to form and maintain the overlay network topology. Algorithms to form and maintain the overlay topology are both resource demanding tasks. Thus, this approach might not be suitable for constrained nodes.

Directory-less In directory-less architectures there are no service directory nodes that mediate the communication between service consumers and service providers. Such an approach might seem less complex in regards to the functionality that has to be provided. Functions related to registration an maintenance, including validation and updates, do not have to be implemented. However new challenges come with the lack of a directory: A directory-less approach has to determine the frequency of service provider advertisements, in order to reduce the network load and redundant transmissions. Directory-less approaches can either use a global flooding mechanisms or some form of controlled flooding mechanism for service provider advertisements or service consumer requests. Instead of broadcasting advertisements or requests, multicasting can be used. Another alternative for selective forwarding can be leveraged by name-based routing. Named data networking (NDN) might be a promising approach that belongs to the class of information-centric networking (ICN) that tries to move from a host-centric to an information-centric paradigm. In NDN routing is based on names instead of host identifiers, such as IP addresses.

2.2.2 Discovery

Search The *search* procedure is responsible for searching for suitable services by given filter options in the search message. The techniques used by the *search* procedure depends on the chosen architectural type. In case of directory-based approaches the *search* procedure can either be *active* or *passive*. If service registration is stored in the local cache of the contacted service discovery node, the search procedure forwards a search message to all other nodes if no suitable services are registered in its local cache. This thesis defines this method as *active search*. If every service discovery node store all service registrations, then a search in its local cache is sufficient. This thesis defines this method as *passive search*. Passive search will be faster than active search, because

a lookup in the local cache is sufficient. However, changes in the environment have to propagate to every service discovery node in the cluster. From this is follows that state changes can lead to inconsistent caches and depending on the contacted nodes query result might differ until equilibrium is reached again. Another approach is selectively forwarding search message to other service discovery nodes by using some form of rules or other reasoning technique. Directory-less approaches rely on some form of flooding, multicast or some form of overlay networking technique for search message forwarding.

Selection The query result from the search request can lead to a set of suitable services, that all match the patterns provided in the search message filter options. The *selection* procedure is an essential part of the discovery process, because it is responsible for choosing the most suitable service that is finally invoked. Selection can either be manually performed by the end-user or done automatically by a criteria-based algorithm. The automatic selection procedure is especially important in machine-to-machine (M2M) communication. Since, M2M communication can be viewed as an important part of the IoT, an automatic selection procedure is essential for a discovery service for the IoT. The criteria used can be for example network metrics, such as lowest hop count, round-trip-time, packet loss rate, or maximal bandwidth. Other criteria might also be service provider specific, such as radio-cycle, current load or energy level.

Search and selection procedures are often integrated, but responsibilities can shift. In directory-less approaches, search and selection is typically performed by the nodes that needs to discover a suitable service. In directory-based approaches search and selection can both be performed by the service discovery node. However, the responsibility to select the suitable service that will be invoked can also be handed off to the requesting node. The requesting node can either forward the set of suitable service to the user to make the choice manually or implement a selection algorithm.

2.3 Context and Context-Awareness

Context is any information that can be used to characterize the situation of an entity as defined in [1]. In [49] the decisive role of context in the Internet of Things is illustrated. They argue that humans as well as things provide context information. These trigger events which invoke IoT services. Thus, context plays a significant role for many IoT use cases. Most IoT use cases are driven by context information of users and things, such as

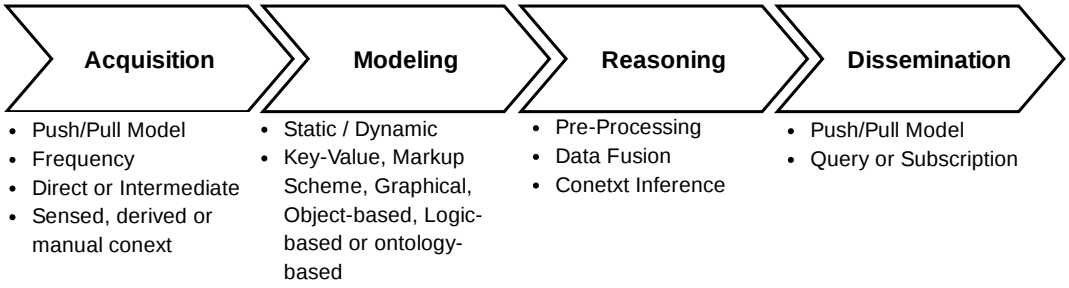


Figure 2.3: Context Life Cycle

geographical location. From this it follows that service discovery in IoT also has to be context-aware.

With the rapidly growing number of connected devices the amount of available services also increases greatly. In order to effectively enable provision of adequate services to users and other services based on their surrounding environment, context-awareness is believed to be crucial. Perera et al. argue in [35] that context-aware computing has already played an important role in previous paradigms, therefore it is likely to successfully overcome the described challenges.

This thesis adopts the commonly recognized definition of context and context-awareness as proposed in [1].

> Any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and the application themselves.

Furthermore, context-awareness is defined in [1] as follows:

> A system is context-aware if it uses context to provide relevant information or services to the user, where relevancy depends in the user's task.

This thesis adopts the context life cycle proposed in [34] to design a context-aware IoT discovery service approach. According to [34] the context life cycle consists of four stages as depicted in Figure 2.3.

2.3.1 Context Acquisition

A push or pull model can be primarily used for context acquisition. In the push model the context source pushes context data to the context sink. The component that provides

the context, e.g. a physical or virtual sensor in the IoT context, is called context source. The context sink is the component that is responsible for acquiring context data. In the pull model the context sink poses a request, e.g. query, to the context source to ask for context data.

The pull model has the advantage that the context sink can request new context information from context sources, when these are needed for context reasoning. It provides the context sink with a near real-time view of the environments state. Thus, context-reasoning can be performed on more up-to-date context information. However, there are also some disadvantages that come with the pull model. The context sink can overwhelm a context source with requests asking for context state. The context source might not be able to process the requests fast enough, or work energy efficient. The context sink also has no information on the frequency of context state changes of context sources. Hence, requests might be unnecessarily sent during a period of infrequent context state changes. This not only unnecessarily congests the network infrastructure, but also forces the context source to process requests even though no context state change has occurred. It is also possible that context requests arrive at an inconvenient point in time, due to a context source radio cycle, or low energy levels. Since low processing power, unreliable and small-bandwidth networks and constrained access to energy are core characteristics of the IoT, a pull model might not be the most efficient approach.

The push model has the advantage that the responsibility of sending and updating context state lies with the context source. Thus, this model prevents by design that the context source can be overwhelmed by context update requests or is contacted by the context sink at inconvenient times. Since the context source is responsible for sending and updating context state the constrained resources of processing power, energy and networking capabilities can be more efficiently utilized by the context source. Once the context state changes the context source can either immediately or periodically send the update to the context sink. The update frequency can therefore be set by the context source and depend on the current work load, energy levels and network infrastructure state. A disadvantage of the push model comes with the inability of the context sink to contact or force the context source to send an update. The context reasoning process might work with old and even false context information, leading to false context reasoning results. The context sink can neither be sure about the freshness [11] of the context information nor if the context source is even still alive.

2.3.2 Context Modeling

The term context modeling is also widely refereed to as context representation. It is the second stage in the context life cycle. Context Modeling defines the process of collecting data.

Context models can be either static or dynamic. Static context models have a predefined set of context data points that are collected or distributed. [51] The requirements that need to be taken into consideration when modelling context information are identified and explained in [9] as heterogeneity, mobility, relationships, dependencies, timeliness – i.e. freshness –, imperfections, reasoning, and efficient context provisioning. The parameters that are considered for context modeling are very subjective and vary from one use case to another. In [34] Perera et. al. provide high-level guidance towards modeling context in different use cases, but there is no standard to specify what type of information needs to be considered in context modeling.

Popular techniques for modeling context are: key-value, markup-scheme, graphical, object-based, logic-based, and ontology-based. Each of the techniques has its own advantages and disadvantages depending on the specific use case. Key-value modeling uses key-value-pairs to model context information. It is the simplest form of context modeling compared to other techniques. It uses different representations such as text files or binary formats. Key-value modeling is easy to manage and process, but is not very scalable. It is not a suitable approach to store complex data structures. Hierarchical structures of relationships cannot be easily modeled.

Markup Scheme Modeling, also called tagged encoding, models data using tags. Popular markup languages such as eXtensible Markup Language (XML) provide tools for sophisticated validation checks. Range checking or regular expressions are also to some degree possible in schema definitions. However, because of its lack of design specifications, context modeling, retrieval, interoperability, and re-usability over different markup schemes can be difficult. The lightweight, text-based, language-independent data interchange format JavaScript Object Notation (JSON) [12] also allows data to be hierarchically structured. Validation by schema definitions is also possible with JSON. Wright, Andrews and Hutton propose the JSON-Schema validation method in [50]. Tuples can also be used as a tag-based modeling approach as described in [51].

Object-based modeling is similar to the graph-based approach as it also use relationships and hierarchical structures. This approach promotes re-usability and encapsulation. It

can be easily integrated with many programming languages as those are object oriented by design. However, object-based approaches are commonly programming language specific which decreases interoperability. Thus, this approach is rather used in non-shared internal context modeling scenarios.

Logic-based Modeling uses facts, expressions and rules to represent context. Rules are primarily used to express policies, constraints, and preferences. Logic-based Modeling supports logic-based reasoning. In ontology-based modeling context is organized into ontologies using semantic techniques, commonly known standards are the RDF, the RDFS, the OWL and the OWL-S. Ontology-based modeling supports semantic reasoning. However, context representation can be quite complex and context retrieval can be computationally intensive and time consuming when the amount of data increases.

2.3.3 Context Reasoning

The third stage of the context life cycle is context reasoning. Context reasoning can be defined as a method of deducing new knowledge, i.e. high-level context deductions, based on the available set of contexts as stated by Bikakis et. al. in [10].

Perera et. al. classify in [34] six categories of context reasoning techniques: supervised learning, unsupervised learning, rules, fuzzy logic, ontological reasoning, and probabilistic reasoning.

Rules are a simple and straightforward method for context reasoning. Rules usually follow the structure of `if-then-else`. Rules are simple to define, are relatively easy to extend and are less resources intensive according to storage and processing power compared to other reasoning models. Rules are expected to play a significant role in the IoT, where they are the easiest and simplest way to model human thinking and reasoning in machines [34].

Ontology-based reasoning has the advantage that it integrates well with ontology-based modeling. Ontology-based modeling is discussed in section 2.3.2. Ontological reasoning has the disadvantage that it is not capable of finding missing values or ambiguous information, where statistical reasoning techniques are quite good at. Rules can be used to minimize this weakness by generating new context information based on available low-level context. Missing values can also be tackled by having rules that enable missing values to be replaced with suitable predefined values. [33]

Lim and Dey evaluate in [28] the popularity of conext reasoning models of 109 context-aware applications as well as 50 recognition applications. They come to the conclusion that over half of the applications use rule-based techniques, followed by about 15% supervised learning (decision tree) techniques, and about 13% probabilistic logic-based techniques.

2.3.4 Context Dissemination

The last stage of the context life cycle is context dissemination. It provides methods to deliver context to the consumers. From the consumer perspective this task can be called context acquisition. It is the counterpart to the first phase in the context life cycle and the discussion in section 2.3.1 is completely applicable. Two methods that are commonly used in context distribution are query-based approaches and subscription-based approaches.

2.4 Fault-Tolerance

Partial failure is an omnipresent characteristic of the IoT environment. From this it follows that fault tolerance is not just an important goal in the design of distributed systems, but essential in the case of the IoT.

2.4.1 Basics

Fault tolerance is defined by the IEEE Standards Committee in [19] as the ability of a system or component to continue normal operation despite the presence of hardware or software faults. As concluded in [45] the distributed system has to be constructed in a way that it can automatically recover from partial failures without seriously affecting the overall performance.

Pullum states in [36] that the requirements for fault tolerant systems are strongly related to the requirements for dependable systems. Kopetz defines in [22] availability, reliability, safety and maintainability as key requirements for dependable systems. Availability refers to the probability that a system is working correctly at any given moment in time. Thus, availability can be defined as a function of time that returns the average fraction of time

that the system has been working correctly [36]. Reliability is defined as the ability of a system to perform its required functions under stated condition for a specified period of time [19]. It is important to notice that reliability is defined in terms of a time interval instead of an instant in time. A system that functions incorrectly for just a second every day has an outstanding availability, but rather poor reliability. Safety is the systems' ability to prevent the coinsurance of catastrophic events in the presence of critical failure modes (*malign*), e.g. fatal errors [23]. Maintainability is defined as the measure of time interval required to repair a system after the occurrence of a non-critical failure modes (*benign*) [23].

The fault tolerance discipline distinguishes between a human action (a mistake), its manifestation (a hardware or software fault), the result of the fault (a failure), and the amount by which the result is incorrect (the error). A failure is defined as the inability of a system to perform its required functions within specified performance requirements. Error tolerance describes the ability of a system or component to continue normal operation despite the presence of erroneous inputs. Robustness is the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions. [19]

2.4.2 Failure Models

Failures and Faults have been categorized by [45, 13, 4] and adopted in this thesis as follows:

Crash Failure A crash failure brings a system to a premature halt. A typical crash failure can be observed when an operating system crashes and comes to a complete halt. In this case the only solution seems to be to perform a cold restart.

Omission Failure An omission failure occurs when a system fails to respond to a request. Omission failures can be further categorized as follows:

Receive omission failure In the case of receive omission failures the system never receives the request. This failure can even occur if the connection was correctly established, because e.g. no thread is actually processing incoming connections.

Process omission failure A failure during processing, e.g. due to infinite loops or faulty memory management. In contrast to receive omission failures the state

of the system might have been partly changed or corrupted during the parts of processing that ran correctly.

Send omission failure Send omission failures occur if the system receives the request, processes the request, but fails to send the response. In contrast to receive omission failures the state of the system might have been changed after processing.

Timing Failure Failures related to timing. These types of failure occur if the response lies outside of a specified time interval. Timing failures can seriously impact the user experience and is e.g. in the case of streaming platforms an important failure to mitigate. Furthermore, timing failures might also lead to critical failure modes, if results are e.g. needed in hard real-time applications such as many process control systems.

Response Failure The returned answer by the system is incorrect. Two kind of failures may occur. Either the system simply returns a wrong answer. This is called a **value failure**. On the other hand the system might receive a request that it processed incorrectly, because the implementation is not designed to handle such an edge case. As a result a state-transition failure occurs. Thus, this type of failure is called **state-transition failure**.

Arbitraty Failure These types of failure are also called **byzantine failure**. This failure may happen if a server randomly produces incorrect response at different points in time. A possible reason for byzantine failures might be malfunctioning components that give conflicting information to different parts of the system. A possible solution to byzantine failures are multiple *different* processes that all compute a result to the same input and perform a majority vote to determine a possibly correct result as proposed by [25].

3 Related Work

As argued in section 2 context-awareness and service composition are crucial features of service discovery in the context of IoT. Therefore, state-of-the-art approaches [33, 15, 27, 48, 37, 52, 26, 21, 17, 42] from recent years are evaluated according to their architectural decisions, discovery mechanisms, service descriptions models, context-awareness and service composition.

3.1 Architecture

Service discovery architectures can be divided into three basic categories: directory-based, directory-less, and hybrid architectures. All evaluated approaches use either directory-based or hybrid architectures.

Directory-less approaches do not need to maintain a service description directory. This can be advantages in high mobility networks, because a drawback of an directory-based approach is that the directory needs to be kept up-to-date. In a high mobility network this might become a challenge. In directory-less approaches service advertisements and requests are based on flooding [47]. However, flooding limit the discovery scope to the local network. This excludes many IoT use cases that rely on information from a remote location. Moreover, flooding has potentially a very high overhead, because service advertisements and requests are delivered to many nodes that are not interested in the information. Therefore, directory-less approaches are not suitable in a broader perspective on IoT.

Since almost all approaches offer discovery of services on a global scale a directory-based or hybrid approach is more suitable. Compared to previous surveys [38] and [5] the number of approaches that utilize P2P architectures – mainly distributed hash tables (DHT) – has greatly increased as depicted in Table 3.1. Other directory-based approaches are based on DNS implementations or agent-based architectures.

Table 3.1: Summary of related service discovery approaches

Approach	**Architecture**	**Discovery**				**FT**	**Context**		
		Scope	**Mechanism**	**SDL**	**SC**		**CA**	**Modeling**	**Reasoning**
[33]	*n.i.*	global	SPARQL Query	*n.i.*	✗	*n.i.*	✓	ontology	*n.i.*
[15]	Res. Directory, P2P/DHT	global	CoAP lookup, location-based lookup	JSON-WSP, CORE Link Format	✗	✗	✗		
[27]	P2P/DHT	global	matching ontology concepts	⟨OWL-S⟩	✗	⟨✓⟩	✓	ontology	logic, semantics
[48]	/, epcGlobal Arch. Framework	spatial proximity	ONS lookup	PML	✗	✗	✓	markup scheme	rules
[37]	Agent-based	local network	Response Threshold Model (RTM)	Tuples	✓	✗	✗		
[52]	P2P	global	matching ontology concepts	OWL-S	✗	⟨✓⟩	✓	ontology	*n.i.*
[26]	P2P/DHT	global	matching ontology concepts	⟨OWL-S⟩	✗	⟨✓⟩	✓	ontology	rules, policies
[21]	DNS-based	global	DNS lookup	DNS records	✗	(✓)	✗		
[17]	Hierarchical Blockchain	broadcast domain	broadcast, lookup	*n.i.*	✗	*n.i.*	(✓)		
[42]	CORE Res. Dir., P2P/DHT	global	DHT lookup	CORE Link Format	✗	✗	✗		

Legend								
	n.i.	*no information*	✓	*supported*	(x)	*partially x*	SDL	*Service Description Language*
			✗	*not supported*	$\langle x \rangle$	*presumably x*	SC	*Service Composition*
							CA	*Context Awareness*
							FT	*Fault Tolerant*

In [21] an approach that utilizes DNS-SD [14] for discovery and CoAP [39] to control IoT nodes is proposed. This design allows service discovery on a global scale. However, the discovery of services is a multi-step process. First, a list of IoT devices of a specific DNS domain is retrieved. Then, the services offered by these IoT devices have to be queried individually. This procedure is rather ineffective if only specific services need to be discovered. Furthermore, DNS record propagation is known to be a rather slow process in magnitude of days rather than seconds or minutes. One advantage of utilizing well established and globally used technology, like DNS, is that implementation becomes easier because an existing infrastructure only has to be extended. DNS-based approaches are only suitable in the IoT context, if the IoT device to be used is already know *a priori*. Furthermore, such an approach cannot effectively implement context awareness and service composition. The reason for this is that the relevant information are only available at the service level and are limited to the restrictions of the `TXT` record.

An agent-based approach is proposed in [37]. In this approach the IoT network consists of high-level devices, such as smartphones, tablets, etc., and low-level devices like sensor nodes. Every high-level device is an agent. Each agent holds a local service registry.

The advantage of this approach is that data is decentralized and no global control is needed. Each agent in the multi-agent system is an autonomous decision-making entity that only has an incomplete view. In most cases multiple agents have to work together to satisfy a request.

In [38] it is stated that these autonomously acting agents can lead to unpredictability and they are susceptible to message loss. Moreover, the time needed to discover the requested services can vary depending on the agent that first receives the request. It is possible that the number of agents needed to satisfy the request greatly varies. And since the agents work sequentially the overall discovery time also varies.

DHT-based architectures are utilized in the approaches [15, 27, 26] and [42]. DHT-based architectures are often used because they implement a distributed data structure. These are based on hash tables that can be used as a distributed directory. Furthermore, they can be deployed instantly as they are not dependent on infrastructural provisions. Most implementations have good scalability capabilities in regards to complexity of communication and storage, and are reliable. However, a drawback of most DHTs is that they cannot control where data is stored. Moreover, most DHT implementation also introduce a communication overhead as compared to the shortest path in the underlying network, also known as delay stretch. Generally data is stored at the node that is

responsible for the hash range that the generated hash of data to be saved belongs to. Li et al. overcome this by using the SkipNet overlay network that allows explicit control content availability and placement [27].

A hybrid approach based on blockchain technology is proposed in [17]. Essentially a blockchain is a distributed database of records as stated in [16].

Trust, traceability, transparency and security are part of the strengths of blockchain technology. However, these strengths come at a price; a lack in performance as compared to other databases is a major drawback of blockchains. Transactions are more complex in blockchains as signatures have to be verified, consensus mechanisms have to be applied, and transactions have to be processed by every node individually [32].

Because a blockchain keeps a record of all transactions back to the first block, scalability is also an concerning issue. A single global blockchain for service discovery is therefore not feasible. Moreover, it is stated in [17] that naming, discovery as well as privacy are further challenges in blockchain-based discovery architectures.

Blockchain-based approaches are strong when it comes to trust, traceability, transparency and security. These are properties that are often neglected by many proposed approaches. The only approach that addresses these challenges is proposed in [17]. However, in the case of blockchain they come at a great price: A decline in scalability. Hence, blockchain might not be a suitable technology for service discovery in IoT.

Fault tolerance is a crucial requirement for infrastructure services such as service discovery mechanisms. However, none of the proposed approaches specifically addresses this crucial requirement. Thus, an assessment of the fault-tolerant capabilities of the proposed approaches can only be derived from the architectural choices, design decisions and used technology. Table 3.1 illustrates that none of the proposed approaches uses a comprehensive approach to fault-tolerance. The approaches [33, 15, 48, 37, 17, 42] do not use any mechanism to increase the systems' fault-tolerance capabilities and do not uses a technology that provides some fault-tolerance capabilities. The approaches [27, 52, 26, 21] also do not implement specific fault-tolerant mechanisms, but make use of technologies that might provide some level of fault-tolerance. Kim et. al propose in [21] an approach that is based on the existing domain name system. The domain name system is based on a inverted tree architectural style that uses redundancy to provide basic fault-tolerance capabilities. The remaining approaches are partly or fully based on DHT-based architectures. Distributed hash tables (DHT) distribute a key-space over a number of

nodes. Each node is responsible for a subset of the key-space. If one node fails only a subset of the key-space might not be available, depending on the DHT implementation's approach towards redundancy. DHT implementations can provide different levels of fault-tolerance.

3.2 Discovery Mechanism

As illustrated in Figure 2.2 discovery consists of *search* and *selection* mechanisms. *Search* is either performed on a directory of registered services or by flooding if no directory is used. In case of directory-less or hybrid approaches service description distribution can either be flooded periodically by service providers or triggered by service request messages. *Selection* of suitable services can either be performed by the requester after a list of discovered services is received or automatically by the discovery service based in the requester's query. If selection is done by the discovery service the queries have to be able to express the requester's requirements.

Many approaches only support simple search mechanisms based on some kind of numerical or string-based identifier. Selection therefore has to be done by the requester. By design DHT-based approaches promote simple lookup schemes. The reason for this is that data is organized in key-value-pairs. The DHT-based approaches [27] and [26] do not overcome the mentioned design limitation. Even though context information for services is provided in information repositories, the data can only be used after the resource or service is discovered via *a priori* known unique identifier. This goes against the idea of discovering a set of suitable services according to a set of specified requirements. In [21] a DNS-based discovery approach is proposed. Service discovery is based on a DNS lookup and returns published services of an IoT node. However, the DNS name of the IoT node has to be known by the requester prior to the lookup. The work presented in [42] is also limited to discovering resources or services by a unique identifier. This goes against the definition of service discovery provided in section 2.2 which states that no knowledge about the service to be discovered should be needed a priori.

To overcome this drawback of DHTs [15] proposes a DHT-based overlay architecture consisting of two overlay networks. The first DHT maps geographical locations to CoAP Gateways. The second DHT maps CoAP Gateways to a list of offered services by these gateways. This approach overcomes the lookup limitations described in the last paragraph, but introduces a larger communication overhead because discovery is now a

multi-step process. First a requester gets a list of gateways in a geographical area and then has to iterate over that list to retrieve the services provided by the IoT gateways. Furthermore, the requester has to use more resources, such as storage, computing power and energy, for service discovery. In case of constrained devices this approach can be disadvantageous.

An agent-based discovery mechanism is proposed in [37]. This approach is based on a set of input-output-pairs that is passed to an agent. The agent then searches and selects appropriate services based on the request and returns a list of services to the requester. The advantage of the agent-based approach is that search and selection is performed by agents and the requester does not have to spend resources in selection. On the other hand the effectiveness of a multi-agent-based approach strongly depends on the coordination mechanism. Agent-based approaches might reduce the complexity of designing a system, but at the risk of unpredictability because of the autonomous nature of agents [38].

In [17] an approach based on blockchain technology is proposed. Blockchain-based approaches often have the same weaknesses as DNS-based and DHT-based approaches. They try to overcome this weakness by restricting the scope of discovery to the broadcast domain of the requester. The requester broadcasts a generic message in order to receive blockchain addresses of service providers from IoT devices. Thus, this approach only supports specific use cases. For example use cases that rely on service information from spatial proximity can not be undoubtedly supported.

An alternative is attribute-based naming. The approaches [33, 27, 52] and [26] are ontology-based and use concept matching for discovery. Thus, in these approaches it is not necessary to know the name of a resource or service a priori. Depending on the specific technique context reasoning can be quite computationally intensive.

3.3 Service Description & Context-Awareness

As illustrated in Table 3.1 only half of the evaluated approaches are context-aware. The majority of approaches use ontology-based context modeling. The approach [48] uses a markup scheme based approach for context modeling.

Presumably all ontology-based approaches use OWL-S [30] for context modeling. Because of its service grounding concept OWL-S is not only restricted to WSDL. Thus, it can be used as an description model for IoT use cases. However as a traditional web service

technology, OWL-S offers rather rich and heavyweight descriptions of service's functional and non-functional properties. From this it follows that communication and computation can be quite intensive. Therefore, OWL-S might not be the best candidate for the IoT.

A balance between a lightweight service description scheme that is also suitable for context- and QoS-aware service discovery is crucial. Some ontology approaches already exists that are specifically designed for the IoT, e.g. IoT-Lite [8].

This ontology reduces the complexity by focusing on key concepts of IoT that allow interoperability and discovery. However, IoT-lite can also be extended by different models to increase its expressiveness. [8]

It can be observed that context reasoning is mostly based on logic, policies and rules. The approach proposed in [27] also applies semantics reasoning. The context reasoning methods are chosen by approaches to find a balance between good reasoning capabilities, limited computation capabilities and timeliness.

Other approaches [15] and [42] use the CoRE Link Format [39] for service descriptions. However, it is a rather simple description model based on key-value-pairs that might be too restrictive for context-aware discovery. In [48] the Physical Markup Language (PML), a specification designed specifically for describing physical objects in a physical environment, is used. However since the IoT has evolved from being mostly based on RFID tags PML's scope is rather limited. Other approaches, [21] and [37], use DNS records or tuples to describe services. These models are also rather limited in describing services and only allow for simple lookup mechanisms.

3.4 Service Composition

Service composition describes the process of selecting a set of services, so that the resulting composed service satisfies the user's functional and non-functional requirements. As illustrated in Table 3.1 only one approach supports service composition.

The approach that offers service composition is a bio-inspired agent-based service discovery architecture proposed in [37]. However, the approach offers only very basic service composition capabilities, because of the simple service description model based on 2-tuples. This approach can only satisfy functional requirements, such as input and output parameter. It does not include non-functional requirements in the selection of services

for composition. Context-aware service composition is therefore not possible. However, we believe that context-aware service composition is a crucial requirement for discovery services in IoT because of the dynamic, context-sensitive and heterogeneous nature of the IoT.

In [18] service composition techniques for the IoT are surveyed. They come to the conclusion that the IoT brings new challenges to service composition that are not yet overcome by any existing work. According to [18] many approaches improve execution time, scalability, and cost. However, availability and reliability is rarely considered.

4 Context-Aware Distributed Discovery Service

This thesis proposes a fault tolerant discovery mechanism that enables consumers, e.g. end-users, to locate IoT services that comply with a set of requirements without prior knowledge about such services. The proposed approach is called *Context-aware Distributed Discovery Service (CaDDS)*. The approach enables consumers to search and select suitable IoT services on a local, as well as on a global, scale.

4.1 Architecture

The proposed discovery service *CaDDS* follows a handle-driven, context-aware, and fault-tolerant directory-based approach. The major components of *CaDDS*'s architecture design are illustrated in Figure 4.1. *CaDDS* uses a short-term distributed cache *QueryCache* as a directory. In order to keep the directory up-to-date *CaDDS* utilizes a distributed publish-subscribe architecture. The component that embodies the publish-subscribe architecture is called *QueryDistribution* as depicted in Figure 4.1.

Generally, directory-based approaches are by design less suitable to be used in service discovery for the IoT. First, if a directory just holds immutable IoT service provider data, e.g. metadata, a request cannot be satisfied by a single directory lookup. It takes several iterations by following multiple handles until suitable service providers are selected. This extensively increases communication overhead and the overall discovery time. Second, if a directory holds mutable IoT service provider data, i.e. state, constant record update routines either by pull or push methods are essential in order to keep the directory up-to-date and portrait a near perfect image of the real-world state. Thus, a directory-based approach can come at great communication cost due to the update mechanism. Furthermore, a great deviation between directory state and real-world IoT state can lead to a great number of false-positive results. This might cause even greater delays

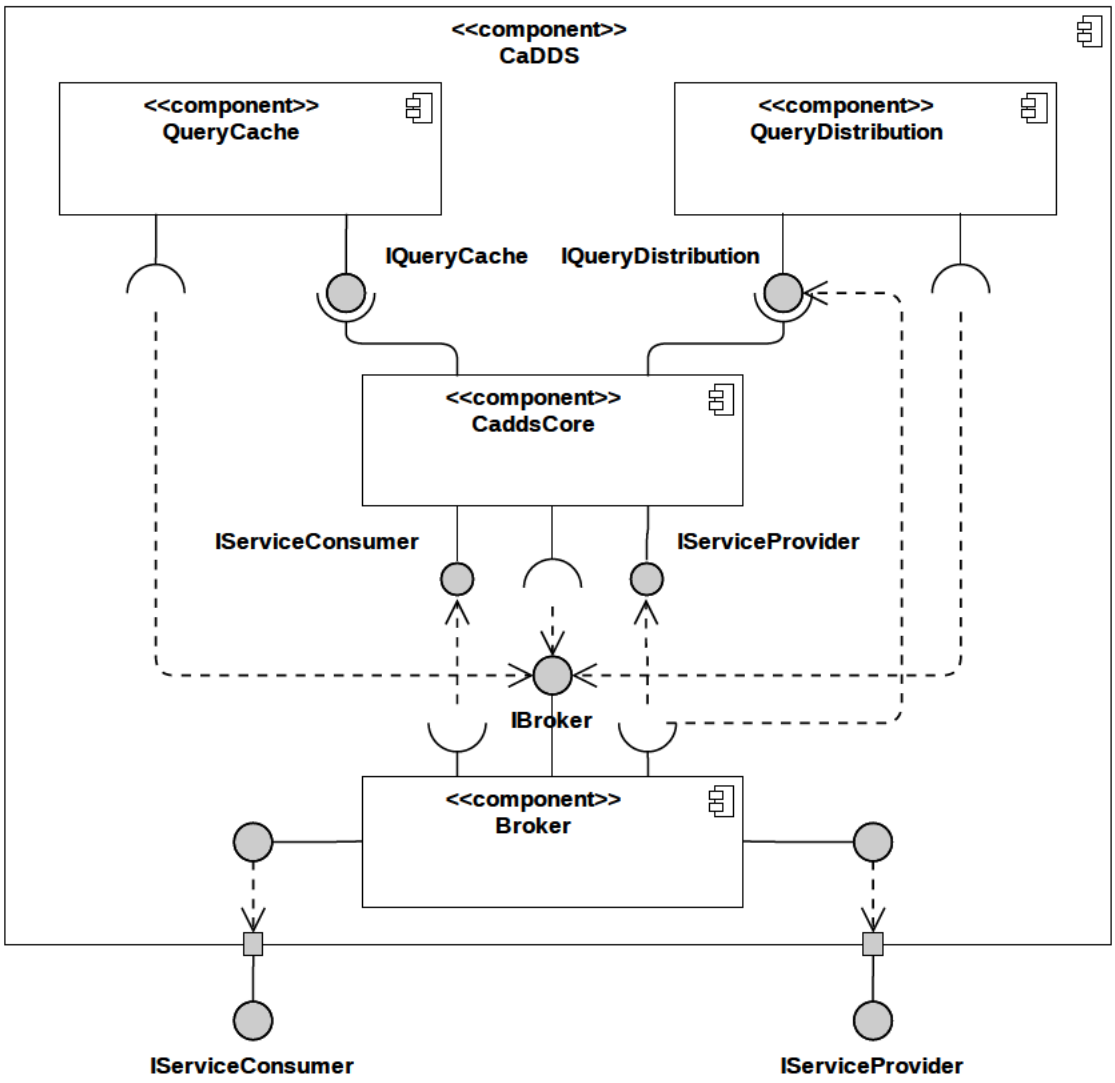


Figure 4.1: Context-aware Distributed Discovery Service (CaDDS) Architecture

in applications, because the false-positive service provider handles have to be recognized as faulty and the discovery process might even has to be repeated. Furthermore, is can harm the user-experience and will not be adopted. Thus, an approach that is based on a directory has to cope with these challenges.

The proposed approach tries to mitigate the discussed challenges by using a short-term cache rather than a long-term directory storage. Furthermore, there is no registration routine for service providers in order to publicize their services. *CaDDS* rather uses a novel demand-based approach to build the directory. The demand-based approach is build on a publish-subscribe architecture. Service providers subscribe to suitable topics and once demand arises the service consumer query is disseminated by publishing. The strength of this approach is that each IoT service provider only needs to update its own state internally, which it has to do anyway, and subscribe to suitable topics for consumer query requests. Once a service consumer request is received the service provider decides whether it can satisfy the query requirements or not. The service provider offers are then distributed by the *QueryCache* component. The publish-subscribe architecture not only supports the demand-based approach, but also allows flexibility in membership changes. By design it is possible for service providers to join and leave *CaDDS* at any time without great administrative expense. Thus, this approach might be more suitable for constantly changing environments and varying state of IoT services, because it does not rely on registration and update routines.

As this approach is based on a short-term cache to store service provider information, an update routine has to be implemented. Section 2.2 highlights that a unsuitable update frequency not only unnecessarily uses network resources, but can also use up vital service provider resources. If the service provider spend a significant amount of time in the update routine it cannot handle consumer requests. In order to mitigate this challenge the publish-subscribe architecture uses a pull model. Thus, the service provider can pull consumer query requests at a convenient time. Caching is then used to stretch the update cycle. Furthermore, this work assumes that consumer query distribution will follow a long-tail distribution: A small amount of query requests generates the great majority of traffic and the majority of query will only generate some traffic. From, this it follows that caching plays a crucial role in discovery services. The expiration time can be specifically set for different *need*s. This is reasonable as different *need*s require different update cycles.

Furthermore, *CaDDS* is a general-purpose design approach that can potentially be used in many different domains and for various use cases. Besides public implementations, that provide access to publicly available IoT services, private and corporate implementations with different use cases and characteristics are also possible. Hence, the proposed approach can be tailored to different discovery scopes.

Efficiency and scalability are also important requirements for discovery mechanisms, especially in the context of the IoT. Unnecessary request distribution is minimized by two aspects: First, *CaDDS* is designed to be partitioned into multiple disjoint clusters. Partitioning by service area, i.e. geographical region, is only one of many possible metrics. Cluster selection needs to be made by the type of partition. Second, each service provider is categorized by a *need* selector. The *need* selector describes the need that is satisfied by the service. Each service provider subscribes to the topic associated with its *need* selector at the cluster responsible for its service area.

Making the assumption that the great majority of service consumer requests are location-dependent, a partition by geographical region is proposed for a discovery service in a public context. In case of publicly available IoT services, a smart-city or -country solution can be realized by the following partition scheme: `<rc>.<cc>.cadds.net` for addressing a *CaDDS* clusters. The country code `<cc>` follows the two-letter codes defined in ISO 3166-1:2013 [20]. A possible country code is `de` for Germany. The regional code `<rc>` is derived from the postal code and is therefore country-dependent. As an example: In order to find suitable services, e.g. in the center of Hamburg near the University of Applied Sciences, a service consumer would therefore connect to *CaDDS* by querying the address *20xxx.de.cadds.net*, because the service areas postal code is 20099 Hamburg, Germany. This example uses a wildcard character x to specify that the specified service area is any german postal code starting with *20*. In metropolitan areas it might be necessary to partition even further by using less wildcard characters. The component *Core* implements the interfaces for service consumers, -providers and the core functionality of *CaDDS*. The public interfaces are based on HTTP as it provides a well-known application protocol that is suitable for a resource-based approach. The HTTP-based approach that uses JSON as a data exchange format leverages the openness and interoperability of the approach.

The publish-subscribe architecture is based on *Apache Kafka*. *Apache Kafka* is described in [44] as a project that aims to provide a unified, high-throughput, low-latency distributed platform for handling real-time data streams. Furthermore, it allows to be used in a

message queuing or a publish-subscribe fashion. This allows to mitigate the weaknesses of each messaging model. Each Apache Kafka cluster is limited by the number of topics it can manage, but not by the number of subscribers. Moreover, it possible to group subscribers in consumer groups to follow the message queuing model.

The *CacheQuery* component uses the distributed *ignite* cache. *Ignite* is described in [43] as a horizontally scalable, fault-tolerant distributed in-memory computing platform for building real-time applications that can process terabytes of data with in-memory speed. Ignite is a ACID-compliant key-value store that can be horizontally scaled in a shared nothing architecture. Hence, it is suitable to be used as the query cache in the *CaDDS* architecture.

4.2 Fault Tolerance

Based on [36] designing and implementing fault tolerant systems can be divided into three main parts: failure prevention, failure removal and failure masking.

4.2.1 Fault Tolerance Techniques

Figure 4.2 provides an overview of techniques used to achieve fault tolerance. These techniques are discussed in more detail in the following subsections.

Failure Prevention

Failure Prevention is an essential part in the software engineering process for fault tolerant systems, by reducing the number of faults introduced during software construction. Failure Prevention techniques contribute to the system's dependability through rigorous specification of system requirements, use of structured design and programming methods, formal methods and software reusability. The reuse of software is very attractive for numerous reasons: Software reusability as it reduces the development costs. But more important in the case of fault-tolerant systems, it possibly increases the dependability. Software that has been well exercised is less likely to fail, because many faults have already been reported and fixed.

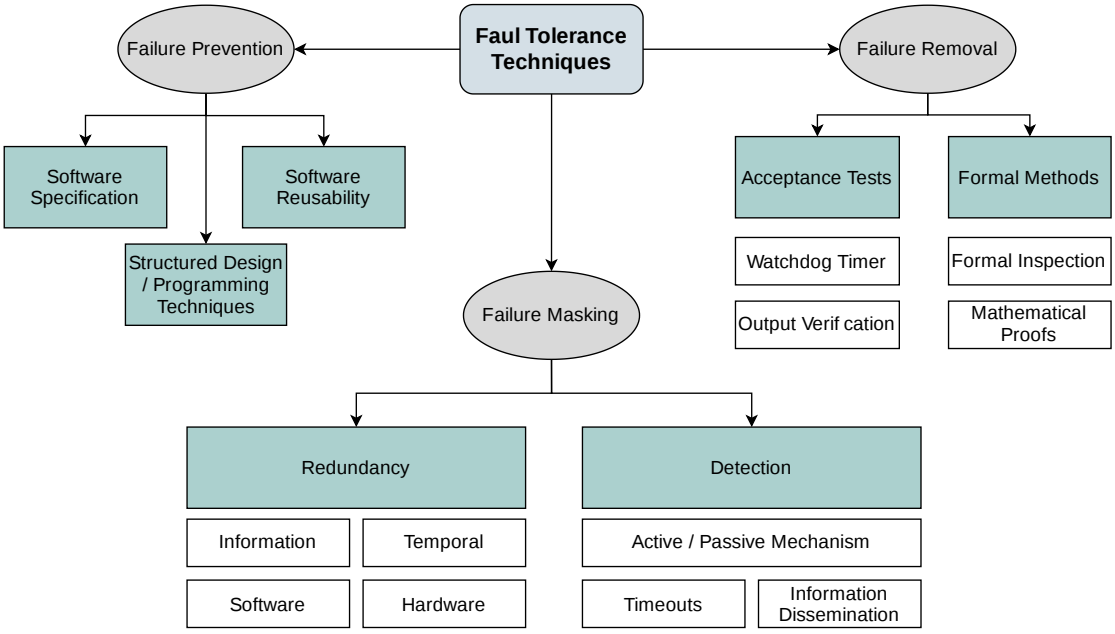


Figure 4.2: Overview of Fault Tolerance Techniques

Failure Removal

These techniques are employed during software verification and validation to enhance the dependability of a software system. One of the most common fault removal techniques involves acceptance tests. Acceptance tests mostly are used in wrappers and recovery blocks. They are used to test for reasonableness of inputs or outputs.

Most acceptance tests fall into one of the following categories: (1) Timing checks are used if a rough idea of the execution time of a piece of code exists. A so called watchdog timer can be set accordingly. If the timer triggers, the system can assume that a failure has occurred. (2) Usually it is much easier to check if the output is correct than incorrect. In this case acceptance tests to output verification can be used. Probabilistic tests cannot detect every erroneous event, but most. Sometimes this is sufficient and saves time. (3) However, the correctness of the output cannot always be exactly calculated and checked. In this case, range checks are used to specify a range of correct possible values. Values outside these bounds are considered erroneous. [24]

Other techniques include formal inspection and formal design proofs. Formal inspection focuses on examining source code by comparing it to the software specification in order

to locate faults, correct them and verify the correctness. Formal design proofs try to achieve a mathematical proof of the correctness of a software system.

Failure Masking

According to [45] the key technique for masking the occurrence of failures from other processes is to use redundancy. Redundancy is the property of having more of a resource, e.g. process, than is minimal necessary to perform the task. As failure occurs, redundancy is exploited to mask or otherwise work around these failures. Thus, maintaining the desired level of functionality. Redundancy can take several forms: hardware, software, information, and time [24].

Information Redundancy In order to tolerate errors in data that may occur when data is transmitted over a noisy channel, such as the internet, or even if it is stored on disk, information redundancy is introduced. The most common form of information redundancy is *coding*. Error-detecting or error-correcting algorithms are used to protect data. Coding, such as parity-bits or checksums, add bits to data that allow the transmitted or stored data to be verified for correctness, and in some cases even correct the erroneous data bits. Storage virtualization is another example for information redundancy. A well-known example is the Redundant Array of Independent Disks (RAID). There are various RAID configurations that allow different levels of availability and reliability. Moreover, data replication in distributed systems may also help with data accessibility. Keeping a copy of data on just a single node will cause this node to become single point of failure (SPOF) and a single point of communication (SPOC). The node might become a performance bottleneck and leave the data vulnerable to the failure of the node. A possible solution would be to keep identical copies of data on multiple nodes.

Temporal Redundancy Nodes can also exploit temporal redundancy through the re-execution of the same programm or function on the same hardware [24]. Temporal redundancy is mainly effective against transient faults. Especially in networking scenarios temporal redundancy can be effective, because network faults, such as packet loss due to buffer overflows or hardware malfunctions, are mostly transient faults. A well-known example of temporal redundancy are retransmissions of data segments at the transport layer. Temporal redundancy can also be used when other means of detecting errors is

in place and the system is able to recover from the effects of the fault by repeating the computation (re-execution) [24].

Hardware Redundancy This type of redundancy is provided by incorporating extra hardware in the design. The additional hardware can either detect or override the effects of a failed component. The main objective of static hardware redundancy is to immediately mask a failure. In the case of three processors the majority of the output can be used to override a wrong output. On the other hand, dynamic hardware redundancy uses spare components that are activated upon failure of the currently active component.

Software Redundancy Every large piece of software that has ever been produced has contained faults, i.e. bugs, as stated in [24]. One way to exploit software redundancy is to have multiple *different* versions of a component, also called n-version programming (NVP). In NVP, N independent development teams design and implement components to the same software specifications, resulting in N components. There are two main ways to use these different components: (1) Different development teams implement a component to the same software specification and a consensus algorithm is used to vote for the correct input. (2) The versions that are implemented differ in complexity. The primary version uses the component that uses the most effective/accurate, but maybe rather complex, algorithm. Secondary versions use less effective/accurate, but also less complex, algorithms to solve the same problem. Secondary versions are used upon the failure of the primary version. Thus, multiple versions of a component can either be executed concurrently or sequentially upon failure detection. It should be noted that, concurrent execution requires hardware redundancy and sequential execution require time redundancy.

4.2.2 Fault Tolerance Approach

The proposed approach uses structured design and programming approaches such as wrappers to validate queries and offer based on JSON schemas. However, even if building robust software in order to mitigate the number of possible failures is important, this section focuses more on the fault tolerance techniques that need to be applied in a distributed context.

As described in section 2.4, the reuse of software components and systems can be beneficial when building fault-tolerant systems. Such components are under constant testing through usage in production systems and therefore are less likely to fail as compared to custom solutions. Hence, the proposed approach makes use of the *Apache Kafka* distributed publish-subscribe architecture. As stated in [44] more than 80% of all fortune 100 companies use *Kafka* in their production systems. Furthermore, the proposed approach uses *Ignite* as a distributed caching component. According to [43], *Ignite* is used at many major companies, such as Microsoft, Netflix, Apple and Paypal, in their production systems.

As partial failure and changing network characteristics play a major role in the IoT, infrastructure services such as discovery mechanisms have to cope with such challenges. According to [45] a key technique for failure masking is redundancy. The concept of redundancy is used in *CaDDS* on multiple levels:

Redundancy can be used on a informational-, temporal-, software-, and hardware level. Information, such as query requests and service provider offers can be distributed either when they are stored in the query cache or the queries are published via *kafka*. Both *kafka* and *ignite* replicate the received data. In the proposed system, *kafka* makes sure that a received messages is replicated at $n-1$ nodes, n being the number of replicas. While *ignite* replicates messages to a all nodes. Thus, information redundancy can ensure that data is accessible even in case of disk failure. Furthermore, the *borker* and *core* component, i.e. service, can be replicated. The broker component is used as a handle driven broker, in case of the scenario without further fault-tolerance measured, that is used internally. And in the fault-tolerant scenario the broker has to become a forwarding broker that also supports a load balancing function.

The *core* component offers a stateless service to consumers and providers. Therefore, the *core* component can be easily scaled by replication. As the *core* component offers a stateless service, all replicas can be active and process consumer request queries or provider offers. However, as the *core* component interacts with *kafka* and *ignite*, fault detection algorithms need to be implemented. A failure detection algorithm needs to take the following cases into account: node failure, network failure or degradation and node stress. Such a mechanism can either use an active or passive approach. In an active approach the component that needs to detect failure sends heartbeats to the components that need to be observed. In a passive approach the observed components regularly send messages to prove that they are alive. An example of a passive approach can by achieved

by gossip protocols. The proposed approach will implement both strategies and compare them. In case of the active approach the *core* component uses heartbeats to monitor the state of the *QueryCache* and *QueryDistribution* services. In case of the passive approach the components *QueryCache* and *QueryDistribution* disseminate their status to the *core* components.

```
def probe_service_tcp(service_handle, remaining, failed, successful) do
    if remaining == 0 or failed >= 3 do
      if failed >= 3 do
        %{:service => service_handle, :state => :unavailable}
      else
        if failed == 0 and successful >= 5 do
          %{:service => service_handle, :state => :available}
        end
      end
    else
      {_name, addr, port, _probe, update_frequency} = service_handle
      {status, socket} = :gen_tcp.connect(addr, port, [:binary])
      if status == :ok do
        :gen_tcp.close(socket)
        :timer.sleep(update_frequency)
        probe_service_tcp(service_handle, remaining-1, failed, successful+1)
      else
        if status == :error do
          :timer.sleep(update_frequency)
          probe_service_tcp(service_handle, remaining-1, failed+1, successful
    )
        end
      end
    end
  end

probe_service_tcp(
  %{:service => "CaddsCore", :addr => "cadds_core_1", :port => 4000,
    :update_frequency => 1000, :probe => "/ping" :state => :available}, 5, 0,
     0})
```

Listing 4.1: Failure Detector based on TCP Probe

The forwarding broker supports two failure detection algorithms that are evaluated in the experiments. The first algorithm approach uses TCP probes to judge about the availability of the service under test. A successful TCP connection establishment is

used as a probe. Once the connection is established the service under test is marked as available. If three consecutive tests fail, the service is marked down and becomes unavailable. Listing 4.1 illustrates the TCP-based failure detection algorithm.

The second failure detection algorithm generates a HTTP dialog and evaluates the response according to the status code that is returned and the overall response time. Other than that the structure of the HTTP-based failure detector is similar to the TCP-based mechanism. Based on the metrics collected via the HTTP probes the load balancing mechanism can favor services that provide good metrics and postpone services that seem to struggle. An alternative would be to evaluate the forwarded traffic in order to gain information about the state of a service, but this follows a reactive rather than a proactive approach. In order to prevent the occurrence of failed responses an proactive approach is favored.

The proposed approach is run in the containerized environment docker. A containerized environment brings multiple advantages. From a fault-tolerance point of view, in such an environment services can be easily scaled and monitored. Policies on crash failures can be implemented. An example of such a policy can be to start a new replicated container of a given service if the number of replicas does not match the specified goal. A container environment can only mask crash failures by replication. Other types of failures need to be engaged in the distributed application.

In order to ensure that data is transferred complete, in-order and to ensure the integrity of the transferred data the transport protocol TCP is used for communication between *CaDDS* components. Hardware redundancy is not used in the proposed proof-of-concept approach, but could easily be implemented in order to mitigate the risk of node failure in case of a failed power supply or corrupt disk.

4.3 Discovery Mechanism

The proposed approach implements a handle-driven and context-aware discovery mechanism. As described in section 2.2, the discovery process consists of two parts: *search* and *selection*. Search is implicitly dealt with by the *need* selector based publish-subscribe architecture. Service Providers subscribed to the specific topic process request query messages. If they can satisfy the requirements a service offer is generated. Thus, part of the selection process is already been performed by the service providers. The final

selection is done by the service consumer, once the service offers have been collected. Thus, the selection process is split between service providers and the consumer. Compared to other approaches evaluated in section 3 – that shift the selection process to the consumer – the proposed approach reduces the computational cost for consumers in the selection process. Thus, this approach is especially beneficial for battery-powered devices, such as smartphone.

In order to find suitable service providers, in a certain geographical region, the service consumer connects to the administrative *CaDDS* cluster and posts a query request. In case of a geographical partition scheme the address of the administrative cluster can be derived from the postal code that is specified in the request query. To find suitable service, e.g near the location of the University of Applied Sciences Hamburg, a service consumer needs to connect e.g. to the administrative *CaDDS* cluster at *20xxx.de.cadds.net*. Once the request is submitted to the cluster and the query is already cached, service provider offers from the cache are returned. If the query is not not in cache, it is published to the topic that is responsible for the *need* selector specified in the request. To collect offers, the service consumer retrieves the service provider offers from the response cache by using the returned query hash.

Service Providers connect, according to their service area, to one of more administrative *CaDDS* clusters and subscribe to the topic that is responsible to the *need* selector they satisfy. It is quite possible that some service provider will connect to multiple *CaDDS* clusters: Either the service area of the service provider stretches over multiple regions or the service area crosses one or more regional borders. This can certainly put more load on the service provider, because they might need to process requests from multiple queues. The difference between service area and area subscribed to might be proportional to the number of discarded messages. Messages are discarded because the service provider cannot satisfy the requirements of the request. The service provider periodically pulls new messages, i.e. service consumer query requests, from the *CaDDS* cluster.

4.4 Service Description Model

The service description model uses a hybrid naming scheme, consisting of a structured and an attribute-based description model. Service Providers describe their service by

attribute-based naming and make their service available by subscribing to certain geographical regions (structured naming). The proposed implementation uses JSON [12] as an string-based encoding method.

This service description model uses a schema-less, i.e. implicit schema. An implicit schema has the disadvantage that the application has to know which keys might exist and check for their existence before accessing the data. The reason for this is that no schema is enforced by the database. However, an implicit schema is more flexible and thus more suitable in scenarios where data is heterogeneous. Furthermore, data that belongs together usually resides in a file. This can be an advantage because no complex joins have to be performed. Thus, an implicit schema is the better choice for consumer queries and provider offers in the context of the IoT.

Listing 4.2 exemplifies a service consumer request message.

```
{
  "api_version":"0.1",
  "query": {
    "query_version": "0.1",
    "need":"parking",
    "location": {
      "country_code": "DE",
      "postal_code": "20099",
      "street": "Berliner Tor"
    },
    "spec_params": {
      "vehicle": {"type": "van"},
      "diabled_parking": false,
      "electric_charging": true,
      "toll_free": true
    }
  }
}
```

Listing 4.2: Description Model Request Example

Each request and response message consists of a set of common fields and *need*-dependent fields. Common fields make up the smallest common denominator across all services. As the proposed approach focuses on partition by geographical region, mandatory keys are *need* and *location*. Through this method, interoperability and the heterogeneity of IoT

devices is better supported. A request requires common fields, such as *version*, *location*, and a *need* selector. Additional parameters can be added in the specific parameters section *spec_params*. The specific parameters section can accept values according to the specified *need* selector.

The response, as illustrated in Figure 4.3, consists of the original *query*, a *hash*, that can be used when querying the cache directly, and a list of service provider *offers*.

```
{
    "query": {},
    "offers": [],
    "hash": "c61f64a130d6d945..9b210"
}
```

Listing 4.3: Description Model Response Example

For this approach to work it is necessary to define a vocabulary for *needs* and need-specific parameters and value ranges. Due to the *version* tag, as part of the common fields, the vocabulary and associated values can evolve in an iterative process. This is clearly a drawback to other approaches that support semantic similarity matching. However, these approaches are computationally rather intensive and thus might not be the best option for an IoT setting.

Through this service description model it is possible to bring a new level of context-awareness to discovery services that is not limited to the location of consumers / providers and is not heavy-weight. Ontology-based approaches allow for a very detailed description of context. However, they are heavy-weight and computational expensive. JSON on the other hand is rather light-weight. It is not just a container for key-value pairs, but allows to describe simple objects. Thus, it can also cope with more complex context descriptions. Context reasoning is based on rules and policies. Furthermore, it is possible for service providers to act as service consumers an find sub-services, e.g. transportation services, to satisfy consumer requirements. Thus, a certain level of service composition can be achieved by the proposed approach.

Private and Corporate setups can benefit from authentication and authorization. User management or API keys can be used in conjunction with HTTPS to restrict access to the *CaDDS* cluster. Furthermore, Apache Kafka supports ACL to restrict the access to certain topics. However, this requires some administrative work for user management

and ACL configuration. Moreover, private and corporate setups could be restricted to be only accessible by clients from the home or corporate network.

In the context of public setups the encryption of data in transit between the service consumer/provider applications and *CaDDS* is possible via HTTPS. However, if there is no registration or validation process of service providers user requests could be collected from multiple topics and used improperly. Thus, user behavior could be tracked. To mitigate this weakness, location data should not be too explicit. To achieve this, house numbers can be omitted in the initial request or slightly altered.

Furthermore, being not too specific when it comes to location data in the initial query is also beneficial when it comes to caching of queries. Query distribution will most likely follow a *long tail* distribution. This means that some queries will be very popular and requested by the majority of consumers and some queries are only of interest to a few consumers. From this it follows that caching will be crucial for discovery mechanisms.

5 Experiment Setup

The verification of distributed systems is quite more complex than the verification of non-distributed systems. A distributed systems differs fundamentally from a non-distributed system as follows: It lacks global state, a global time frame and shows non-deterministic behavior [7]. This is the reason that makes it quite hard to verify the correctness of a distributed system.

5.1 Verification Methods

Figure 5.1 provides an overview of possible techniques that can be used to verify the correctness of a distributed system or at least gain confidence in it. The proposed techniques can be grouped into three categories: Proactive methods can be used before the system is deployed and used productively, reactive method can only be used effectively once a failure has occurred and hybrid approaches can either be used during development and in production.

Software Testing through e.g. unit or integration tests are a techniques that is widely used in non-distributed systems. And even if the scope of tests that can be performed in a distributed context might be limited. [53] demonstrates that such tests are quite useful. Yuan et al. shows in [53] that three or fewer nodes are sufficient to reproduce most failures. Furthermore, [53] evaluates that testing error-handling code can prevent the majority of catastrophic failures. In addition, incorrect error handling of non fetal errors is the cause of most catastrophic failures. Yuan et al. demand in [53] that at minimum error handling code should be verified through the use of unit and integration tests.

Formal verification is another important proactive verification method. Formal verification systems allow developers to verify the correctness of a system by proof. A important technique to proof the correctness of a system is propositional logic using formal languages.

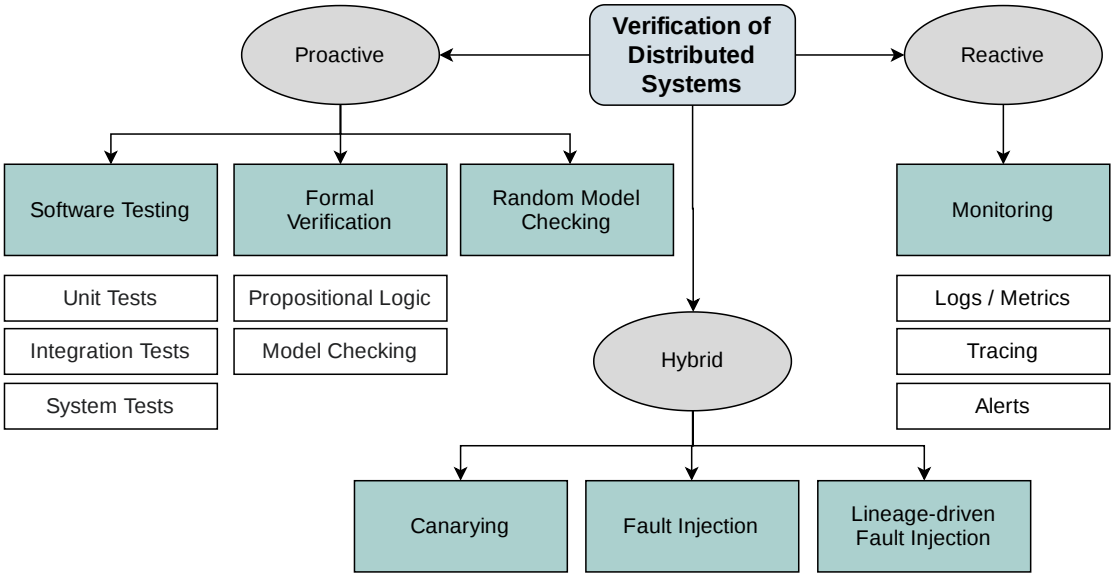


Figure 5.1: Overview of Verification Methods of Distributed Systems

For every system that needs to be verified a definition of the program, a list of failure classes, and a specification of the guarantees needs to be specified. These specifications are then used to verify the correctness of the system under test by proof. Amazon used formal verification to proof the correctness of core components of the Simple Storage Service as described in [31]. Another formal method, model checking, can also determine if a system is provably correct. Model checkers use state-space exploration to prove the correctness of a system. All formal verification techniques can become incredibly time and resource consuming given a multitude of inputs and failure modes a system. Furthermore, one cannot conclude that the correctness of a system can be derived from the correctness of the individual components [6].

Random model checkers cannot provable declare a system to be correct, but they can help to build confidence in the system under test. Random model checker generate random inputs and tests to a given specification of properties. If the properties hold for all inputs that were generated the system passes the test run. If some properties could not hold a counterexample is provided. Random model checkers try to mitigate the time and resource challenges by exploring only part of the system's state space.

A reactive approach is to deploy the system and use monitoring to be able to act on failure. Monitoring through logs, tracing and alters can be used for human intervention in order to fix a problem before a failure occurs or as a basis for *post-mortem* analysis.

Hybrid approaches on the other hand are quite promising methods to engage verification. Canarying is a deployment-pattern that gradually introduces new code into the production system. Instead of replacing all nodes of a service with the new code, only a few nodes are upgraded to the new code. Then, monitoring can be used to compare important metrics of the new nodes with the statistics of the nodes that still operate on the older code base. If the metrics show that the nodes with the new code operate better, more nodes can be upgraded to the new code. On the other hand, if metrics show that the newly introduced code performs worse, the upgrade can be rolled back. This approach can greatly decrease the risk of major failure, but the conclusion that can be drawn from canarying are only as good as the diversity in the test period. If during the test period network characteristics were perfect and no important nodes failed the conclusions might be limited. Thus, the duration and environmental challenges need to be acceptable in order to get useful results. Canarying cannot verify a systems correctness or its fault tolerance, it can only make a statement about the changed behavior in comparison to other versions.

Methods based on fault injection deliberately cause or introduce a fault in the system. In distributed systems a fault can range from a dropped message to the loss of an entire region. The benefit of fault injection is that is can be used from the very early stages of the development cycle to the production system. The fault injection technique forces failures to occur which allows system engineers to observe and measure the effects and implications of such a failure. As with all the other techniques, monitoring is also a crucial part of the fault injection method as it allows system engineers to analyze the system's behavior.

Ongoing research focuses on a method called lineage-driven fault injection. This technique tries to greatly reduce the state space. It starts with correct outputs and works its way back to uncover possible failures, as described in [2].

From all discussed verification techniques only formal methods can provable test system correctness. All other methods can only increase the confidence in the system. However, using formal methods becomes rather impossible if the number of services increases in a distributed system. Thus, formal methods are only suitable to evaluate a few core components, as also discussed in [31]. The most promising technique is fault injection as it can be adopted at every stage of the development cycle from early prototypes to production systems. This thesis will focus on a fault injection-based approach to evaluate the proposed discovery service *CaDDS*.

5.2 Experiment Design

Traditionally, metrics to measure fault-tolerance depend on those designed for availability and reliability. All these metrics are defined as a function of time. An example for such a metric is Mean-Time-to-Failure (MTTF). MTTF is defined as the average time until a component fails [45]. A major drawback of these metrics is their definition as a function of time. Especially in high-available and high-reliable systems a huge time period, e.g. perhaps the last year, has to be taken into account for accurate predictions. This also means that changes to the system that improve fault-tolerance are not visible instantaneously. Furthermore, the validity of measurements is greater if data comes from the production system.

However, this method is neither suitable during development nor for prototype implementations. A modern approach, that is suitable for prototype implementations – as it is often the case in scientific papers, relies on the concept of chaos engineering. Chaos Engineering is the discipline of experimenting on a distributed system in order to build confidence in its capability to withstand turbulent conditions in production [6]. The advantage of this approach is that it can be used in development and for prototypes. Moreover, implementation changes can be evaluated and rated instantaneously.

Therefore, chaos engineering can be used at different abstraction levels and scopes for evaluation. The proposed fault injection-based experiment setup uses chaos engineering to evaluate a systems' fault-tolerance capabilities by running multiple repeatable fault-injection experiments in a virtualized environment (e.g. Docker) as black box tests. A fault is injected into the system and for a set period of time the behavior of the system under test is recorded and evaluated against baseline metrics. The process of an experiment run is illustrated in Figure 5.2.

The proposed system is evaluated to the following criteria:

- The system under test is expected to return a correct response. A correct response to be expected to return a HTTP status code that lies between 100 and 499. Responses that differ from this specification will be considered a response failure.
- The system under test is expected that the upper quantile for the recorded response time is below 100ms.

In order to be able to fault-inject an instance of a service and to establish an experiment environment that is controllable and repeatable container virtualization is used. Docker is

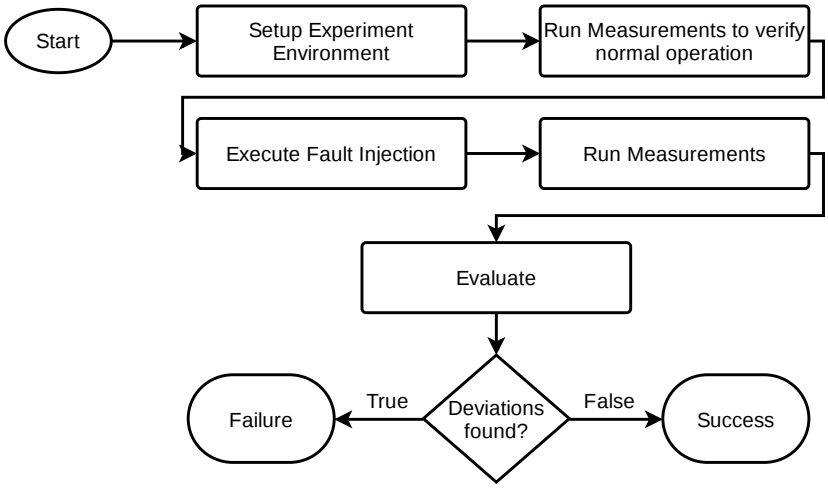


Figure 5.2: Sequence of an Experiment Run

a platform as a service product that uses OS-virtualization to run software in standardized units called containers. Containers are isolated from each other and contain everything that is needed to run a specific software. This includes libraries, system tools, code and a runtime environment. Services can be defined in a configuration file and easily scaled. Moreover, docker also provides a load balancing function to present multiple replicas of a service as a single instance to the host machine. Another interesting feature of docker is its ability to restart a container on failure and ensure that a specified number of replicas run a every point in time. Furthermore, the host can interact with running container via the docker CLI. The CLI can not only be used to start and stop containers, but also to invoke commands on containers. This is especially useful in an experiment setup.

Injecting node failure and restarting nodes is rather straight forward achieved with docker commands `stop` and `start`. In order to inject node stress the linux command-line tool `stress-ng` is used. `stress-ng` is able to stress test a system by exercising various physical subsystems of a computer as well as the various operating system kernel interfaces. Manipulation of *engress* or *ingress* traffic of specific containers can be achieved using the linux command-line tool `tc`.

Through virtualization it is possible to establish a stable initial state of the system-under-test before running a given experiment. This makes experiments repeatable and more conclusive. Additionally, containers run in a dedicated virtual network that can be easily controlled and manipulated during an experiment. Furthermore, limitations

Table 5.1: Overview of Experiments

C	**Category**	**Exp.No.**	**Description**
1	Node Failure	1.a	1-3 container of $g \in G$, FD_{TCP}
		1.b	1-3 container of $g \in G$, FD_{HTTP}
		1.c	1-3 container of $g \in G$, temporary failure
2	Node Stress	2.a	1-3 container of $g \in G$, FD_{TCP}, CPU stress
		2.b	1-3 container of $g \in G$, FD_{HTTP}, CPU stress
		2.c	1-3 container of $g \in G$, FD_{TCP}, Disk stress
		2.d	1-3 container of $g \in G$, FD_{HTTP}, Disk stress
3	Network Manipulation	3.a	1-3 container of $g \in G$, FD_{TCP}, delay: 1000ms
		3.b	1-3 container of $g \in G$, FD_{HTTP}, delay: 1000ms
		3.c	1-3 container of $g \in G$, FD_{TCP}, loss: 10%
		3.d	1-3 container of $g \in G$, FD_{HTTP}, loss: 10%

on scalability can be neglected, because most of the catastrophic errors in distributed systems can be reproduced with three or fewer nodes, as stated in [53].

The proposed experiment approach has two main advantages: First, it offers a uniform experiment environment, which makes it possible to compare different approaches and implementations. Second, projects can incorporate this fault-injection based approach early in the development process, in order to create IoT services with a greater degree of fault-tolerance.

Service Groups are defined as $G = Core, Broker, QueryCache, QueryDist$. Failure Detection types are FD_{TCP} for the TCP-based mode and FD_{HTTP} for the HTTP-based mode. The fault-tolerant test framework focuses on the types of faults stated in Table 5.1.

6 Evaluation

This chapter discusses the measurement results from the experiment runs and the applied fault-tolerance techniques are evaluated in different failure scenarios. The section *Zero-Redundancy* focuses on the experiment runs that were conducted on the proposed approach without any fault-tolerance techniques applied. The section *Tripple-Redundancy* evaluates the experiment runs that were conducted on the proposed approach with a total of three replicas for each component. An experiment is considered successful if the HTTP status code that is returned is ranged between 200 and 499 and the upper percentile of the response time is lower than 100 ms, as specified in chapter 5.

6.1 Zero-Redundancy

In order to better understand the systems behavior, the proposed approach is evaluated without redundancy and failure detection techniques applied. Only the measures that increase the resilience to erroneous inputs and network error, such as a failed socket connection, are implemented. Thus, the system will not fail if it receives malformed requests or fails to connect to other components.

All following plots are composed in the same way (e.g. see Figure 6.1): The upper plot displays the round-trip-time of a request, labeled execution time. The lower plot shows the number of successful requests, i.e. responses that return a HTTP status code between 200 and 499, and failed request that return a http status code between 500 and 599. The first measurement is always the baseline. This data series is taken before any faults are injected into the system. It is used as a guide to visualize the degree of deviation.

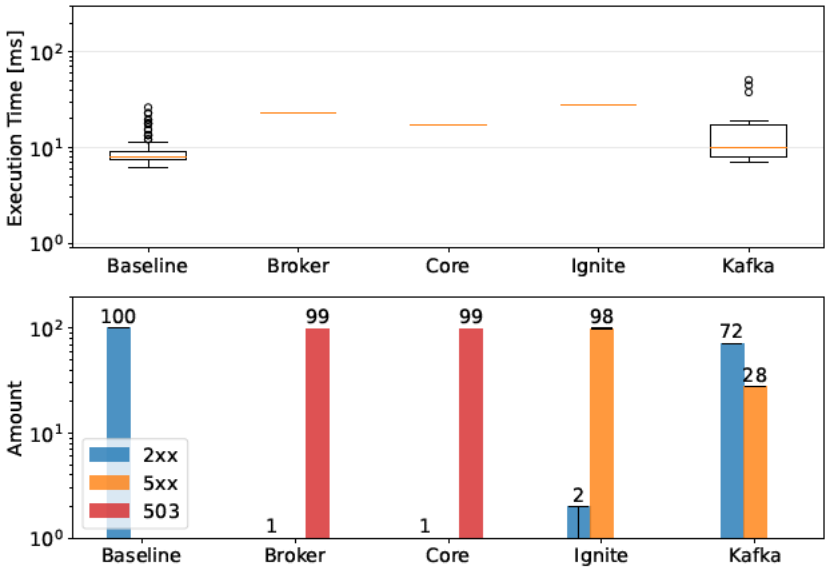


Figure 6.1: Zero Redundancy Failure Experiments

6.1.1 Node Failure

Figure 6.1 shows the measurement series of fault-injecting a crash failure in the proposed approach that does not use physical redundancy. It can be observed that all remaining components are still running, but that the *CaDDS* system cannot continue normal operation in the presence of failure.

If either the component *broker*, or *ignite* fails the service is still available, but cannot provide an adequate response to a consumer query request. The service is still available, but cannot fulfill its specification. Hence, the HTTP status code 500 is returned. However if the *core* component fails, then the service becomes unavailable. This is documented by the returned HTTP status code 503.

The only crash failure that is partly tolerated by the system under test is the *QueryDistribution* component. The reason for this is that as long as query requests and provider offers to a given *need* selector are available in the cache, the system only queries the cache and does not publish the consumer query. The failures that are reported originate either from the expiration of a cache entry or from a query request that has not been published before and thus has not been added to the cache.

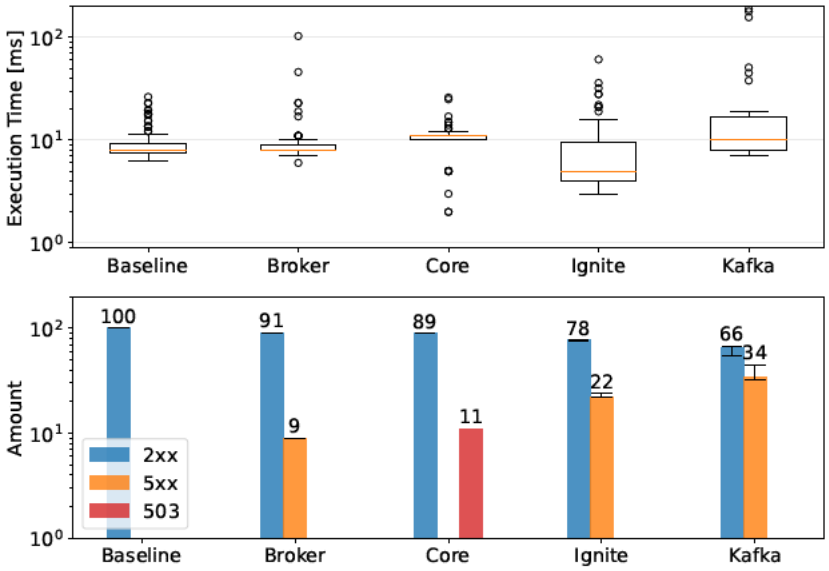


Figure 6.2: Zero Redundancy Restart Experiments

Figure 6.2 depicts the behavior of the system under test if a component is restarted. In every experiment run the system under test is able to recover from the temporary loss of a component. However, there are periods of time were the system under test is either unavailable in case of the temporary loss of the *core* component or is not able to produce a suitable response, as marked by the HTTP status code 500.

Furthermore, it is interesting to notice that the amount ot time each component requires to become operational again from about ten lost requests to around 34. Which is in temporal terms between ten and 40 seconds.

6.1.2 Network Manipulation

It does not come at a surprise that the zero redundancy approach cannot tolerate changes in network characteristics well. If a great delay is inflicted on a given service than the system as a whole will pass that delay to the consumer or provider, as depicted in Figure 6.3. The same is true for a high loss rate that is inflicted on a given service. The requests can be processed but the retransmissions that occur due to the higher loss rate decrease the performance of the transport protocol. Hence, this also leads to response and timing failures, as demonstrated in Figure 6.4.

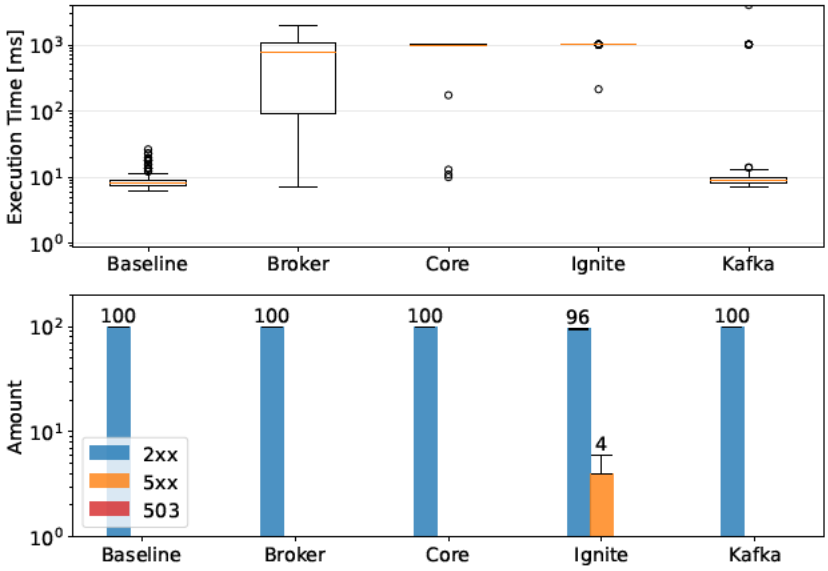


Figure 6.3: Zero Redundancy Delay Experiments

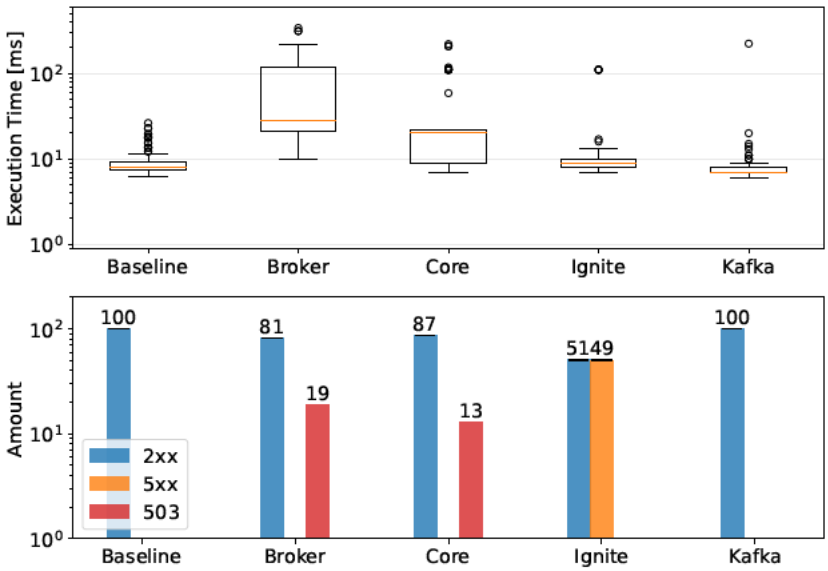


Figure 6.4: Zero Redundancy Loss Experiments

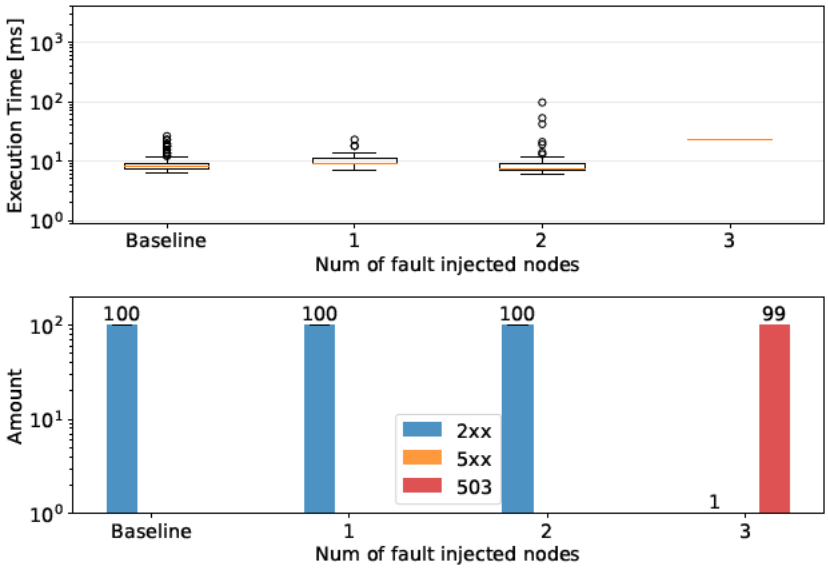


Figure 6.5: Triple Redundancy Experiment - Broker Fail

6.2 Tripple-Redundancy

In this section the experiment runs that were conducted on the system under test that utilizes fault-tolerance techniques in order to mask failures. The following subsection will focus on node failure and restarts, node stress and network manipulation experiments. The measurements that show the most interesting results are discussed.

6.2.1 Node Failure

Figure 6.5 depicts three experiments. The first measurement of every figure is always the baseline that was recorded when no fault was injected and the system ran under normal conditions. In the next three data series one to three replicas fail to run due to a crash failure injection. As Figure 6.5 illustrates, the impact of the broker failing can become quite severe, as it is implemented as a forwarding broker. A total system failure occurs if all three instances of the broker service experience a crash failure. If only one or two instances fail the system can continue normal operation. It can be observed that some outliers occur in the case that two of three instances fail.

Figure 6.6 displays the crash failure experiments of the core service. The core service can tolerate the crash failure of up to 2 instances without severely affecting normal operation.

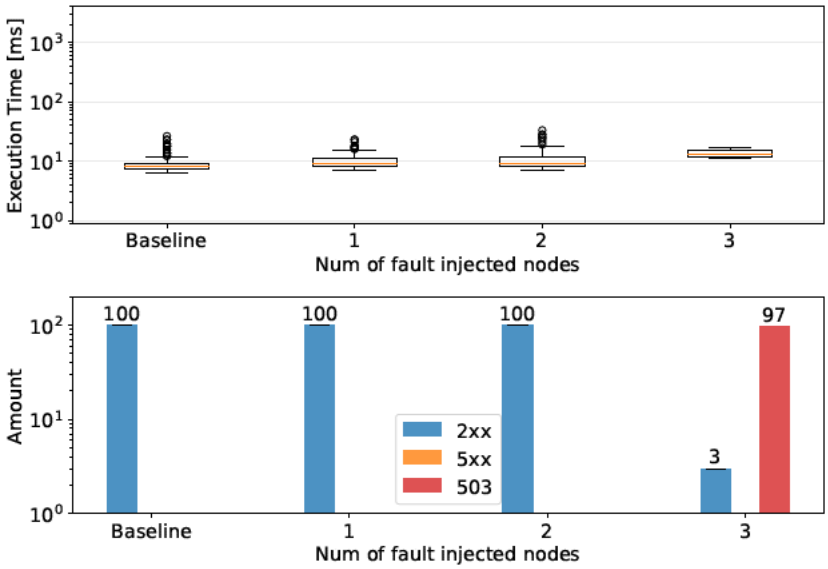


Figure 6.6: Triple Redundancy Experiment - Core Fail

As the replicas are implemented as stateless services, the failure of one instance does not affect other instances or the system as a whole. If the third instance also crashes the whole system fails to respond to requests and becomes unavailable. This is depicted in Figure 6.6 by the status code 503 that is returned by the client application. The reason for this is that the *Core* component provides the public interface and ties everything together. However, service provider are still able to pull consumer query requests from the *QueryDistribution* component, but service providers are not able to publish their offer if the query specification can be satisfied.

Figure 6.7 displays the crash failure experiments of the *QueryCache* service. The *QueryCache* bahaves similar to the *Core* service in terms of the returned status code. But it can be observed that the the execution times significantly increase. In case of a single crash failure the average response time is at least double of the baseline. If two replicas fail the number of outliers also increases. The reason for this might be that the *QueryCache* is based on a distributed hash table and needs to execute maintenance mechanisms, such as updating the finger table and replicating state. State replication is required, because the *QueryCache* is configured to replicate every key on all other nodes. A failure of all three replicas of the *QueryCache* service greatly affects the overall performance of the system.

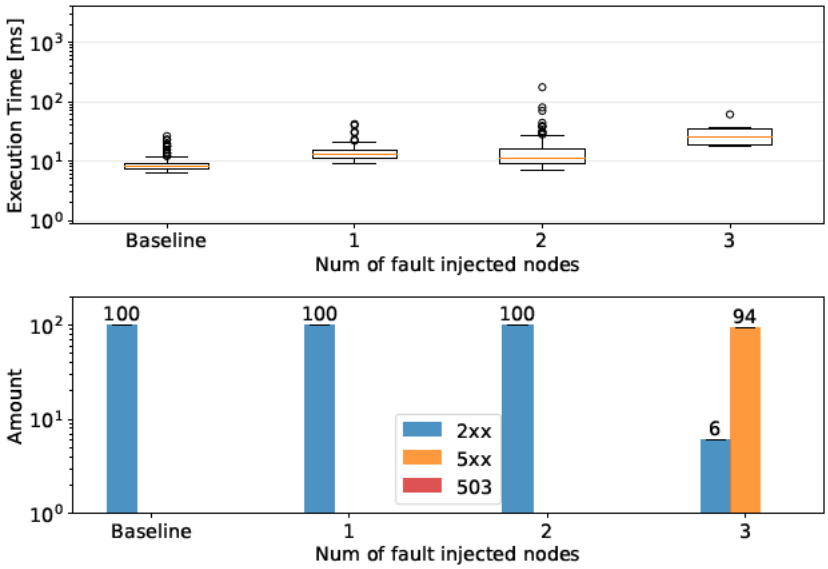


Figure 6.7: Triple Redundancy Experiment - QueryCache Fail

The Failure of the *QueryDistribution* service behaves similar to the failure characteristics of the *Core* and *QueryCache* service. However, if all three replicas of the *QueryDistribution* serivce fails the impact might not be as severe compared to the other mentioned components. On the one hand, service providers are not able to pull consumer query requests from the topics they subscribed to, but they are still able to process query specifications that were already collected and post their offer. Furthermore, service consumers are also able to query the cache. The stored queries and associated offers are available as long as the cache entry has not expired.

In summary, the proposed discovery service can tolerate simultaneous crash failure of two of three instances of a service. Furthermore, experiment runs on the restart behavior of nodes was also conduced. The figures are not presented here because the systems' behavior is similar to the crash failure experiments. Not surprisingly, it can be noted that both failure detection algorithms performed equally.

6.2.2 Node Stress

Another important type of fault is node stress. Stress can be injected into a given instance of a service by various command-line tools. The following evaluation will focus

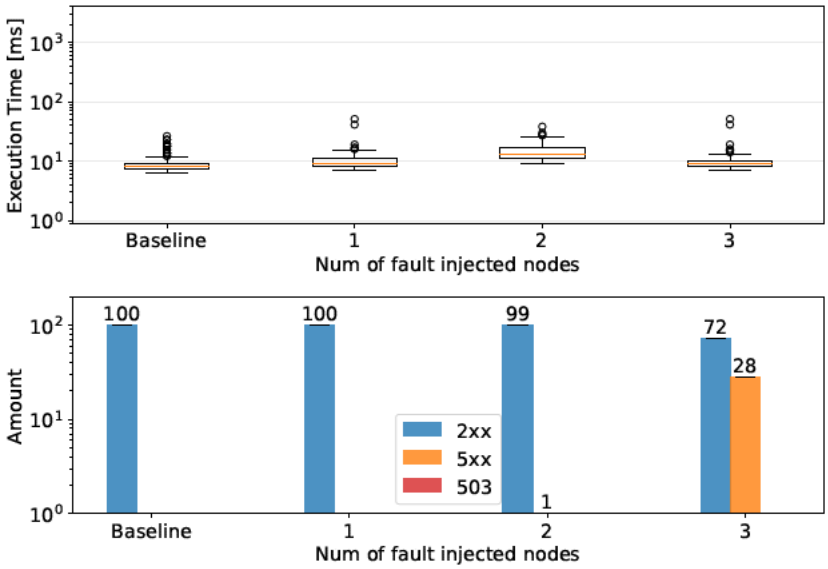


Figure 6.8: Triple Redundancy Experiment - QueryDistribution Fail

on node stress introduced by CPU load as those experiment runs have provided the most interesting results.

As discussed before the broker is a vital part of the system as it forwards the incoming requests to the core service and implements a load balancing mechanism. Thus, a failure of all three instances leads to the failure of the whole system as depicted in Figure 6.9. It can be observed that requests are correctly processed, but that response times greatly increase.

Putting stress on the *Core* service will definitely affect the systems performance as it is the backbone of the system. As displayed in Figure 6.10 the TCP-based failure detector fails to detect that a replicated core service is under stress. This results in the load balancing algorithm wrongly scheduling the stressed service as it was functioning normally. A simple probe is not enough to detect that a service is under stress. This is further illustrated by the case of two stressed services. If two services are under heavy load, the remaining service could take on more requests to relieve the other services. However, this is not the case. If all three replicated services are under stress there is not much that can be done.

Figure 6.11 displays the measurement results of the stressed *Core* services, but this time making use of the HTTP-based failure detector that uses a HTTP probe to gather

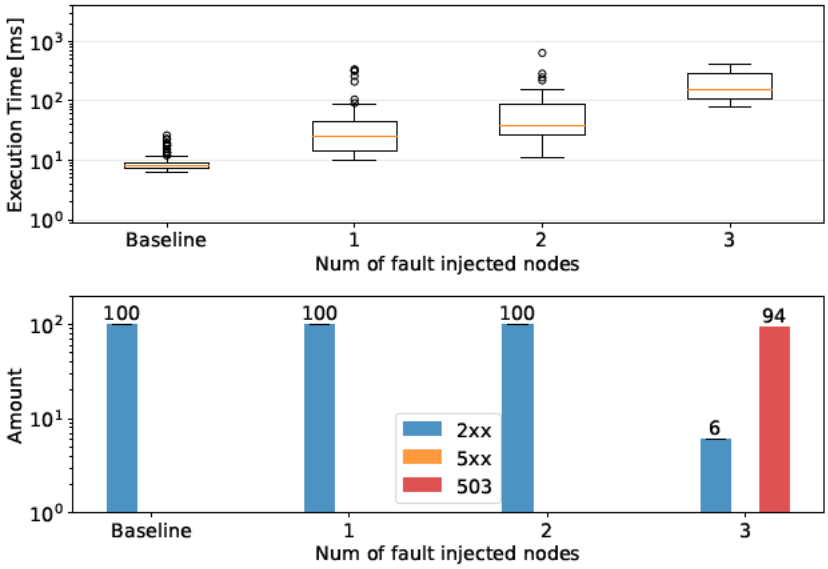


Figure 6.9: Triple Redundancy Experiment - Broker Stress

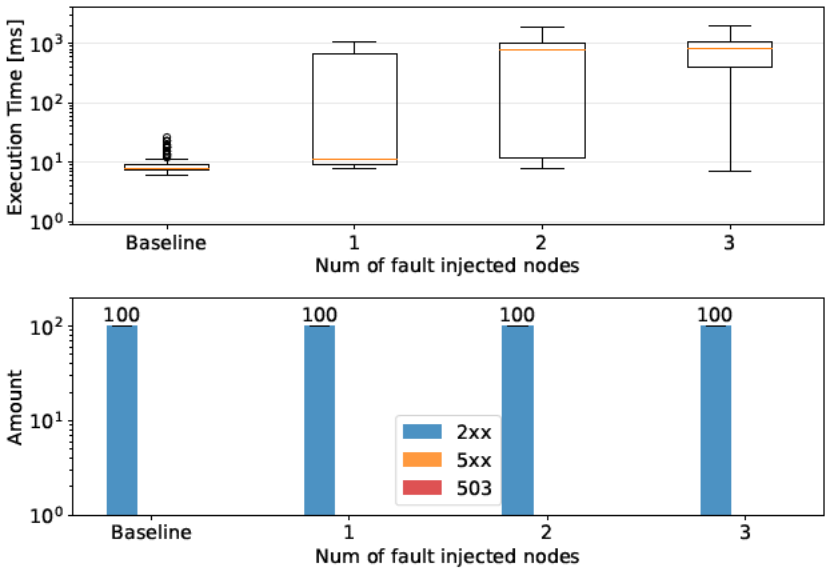


Figure 6.10: Triple Redundancy Experiment - Core Stress FD_{TCP}

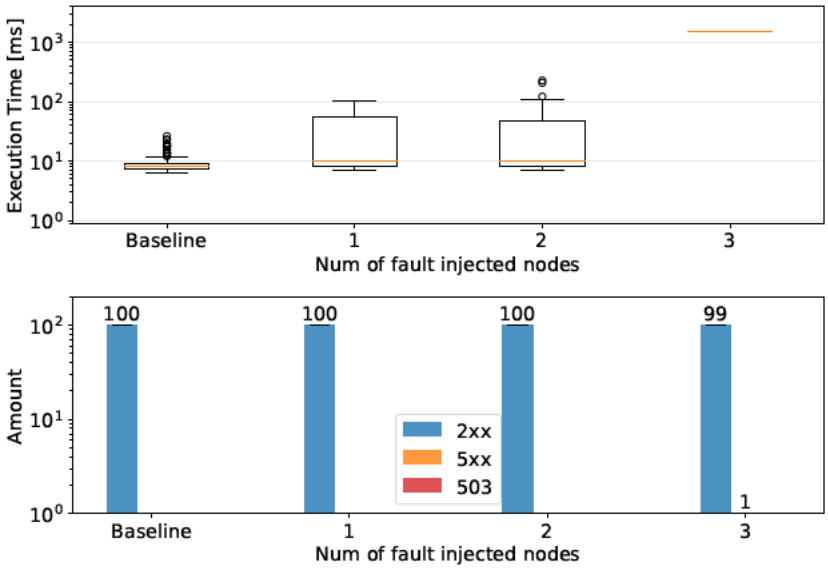


Figure 6.11: Triple Redundancy Experiment - Core Stress FD_{HTTP}

information about the services' state. It can be observed that the more sophisticated failure detector can detect that some instances are struggling and can therefore temporarily remove them from the list of healthy nodes. As stress is not as homogeneous as a crash failure the detection is not as accurate as it could be. But it performs far better than the trivial approach. Instead of using HTTP probes to access the state of an instance it might be more effective to observe the ongoing communication, as this might produce more realistic data than the HTTP probes.

Figure A.1 displays the experiment results associated with the introduction of CPU load at one or more replicated *QueryCache* services unding the TCP-based failure detector. In the case of the *QueryCache* the effect of the advanced failure detector is not as pronounced. The experiment result is provided in Figure A.1. Furthermore, it is important to notice that almost all queries, that are posted during the experiment of three stressed services, return a status code from the 500 range. This means that the *Core* service is unable to connect or interact with the cache service. Further examination revealed that a system-critical thread could not execute its task in the appropriate time which lead to the nodes leaving the cluster topology. The nodes then tried to build the topology again, but failed during that process with a crash failure. At this point the docker restart policy gets activated and restarts the failed services. After the successful restart the ring topology is formed again and the caches can be accessed again. However,

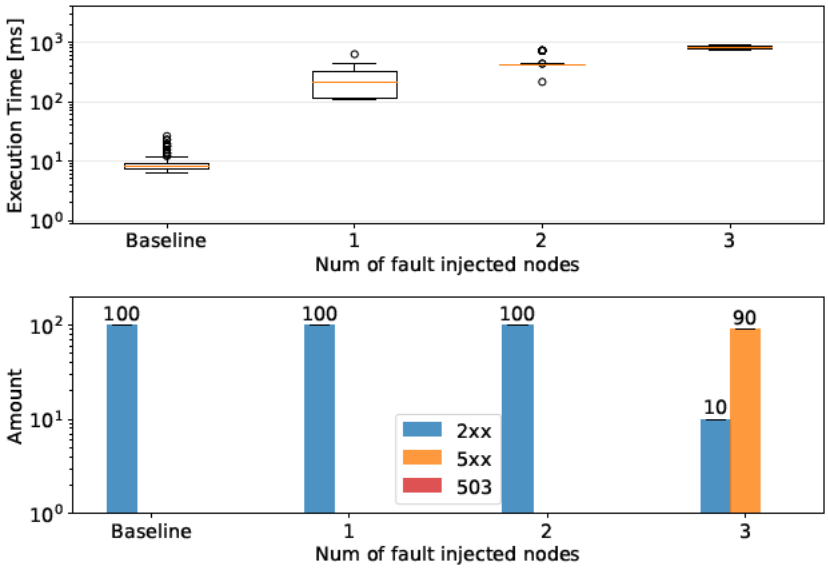


Figure 6.12: Triple Redundancy - *QueryCache* Stress FD_{TCP}

obviously the cache is empty and needs to be build up again. But this is rather a minor issues compared to a crashed cache cluster that just is not accessible.

The *QueryDistribution* service is also affected by the CPU load that is injected into the replicated services as depicted in Figure A.2. Especially the response times are affected by the fault. But in contrast to the cache cluster, the publish subscribe architecture continues to be available. Only the response times are a lot longer. As consumers do not directly interact with the publish-subscribe architecture the introduced delay will not greatly decrease the user experience. Only if a new or expired query is posed, the time span until offers arrive is longer.

In the case of the *QueryDistribution* service neither the trivial nor the more advances failure detector are able to detect and mitigate the stress injection. The experiment result is provided in Figure A.2. The reason for this is possibly linked to the replication algorithms that are affected by the stressed instances of the *QueryDistribution* services.

In summary, the proposed approach is able to tolerate stress to a certain degree. The TCP-based Failure Detector fails to detect the stressed instances. The HTTP-based Failure Detector is able to mitigate the impact of stress to the whole system if a few instances are affected. Some outliers are recorded that still lead to timing failures, but

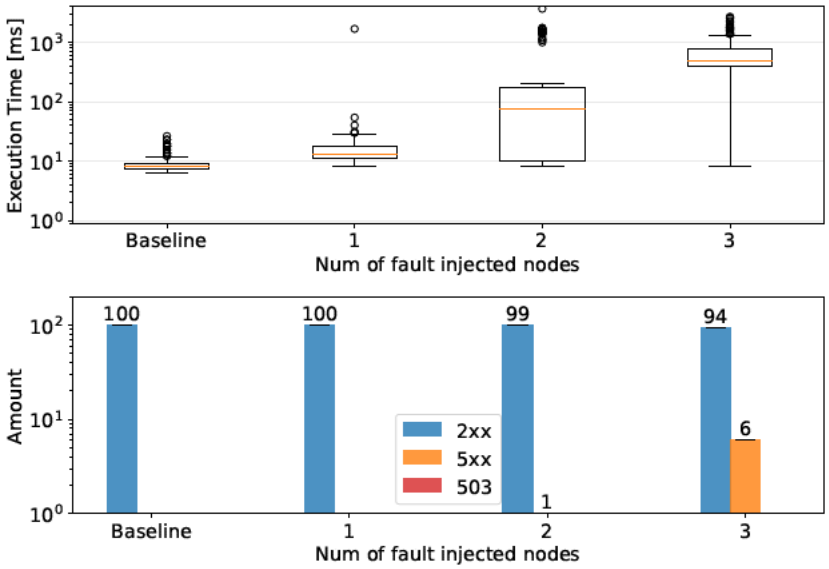


Figure 6.13: Triple Redundancy - *QueryDistribution* Stress FD_{TCP}

the great majority of requests can be successfully mask if the *Core* or *QueryDistribution* services are affected.

6.2.3 Network Manipulation

Changes in the network and link characteristics are another important source of faults that might develop into failure. In this subsection the most interesting measurements for the experiment set are evaluated.

Introduction of Delay

The following experiment results depict the systems behavior if some nodes experience a greater delay on the network link than others.

Figure 6.14 and 6.15 present the affects of introducing egress delay on the *Core* component. It can be observed that both failure detectors can detect that one or more instances are affected. In case of the TCP-based failure detector some outliers can be observed. From this can be concluded that the TCP-based failure detector is not consistent in detecting the affected instance. The HTTP-based failure detector on the other hand is able to overcome this challenge and reliably detect the affected instances. The system behaves

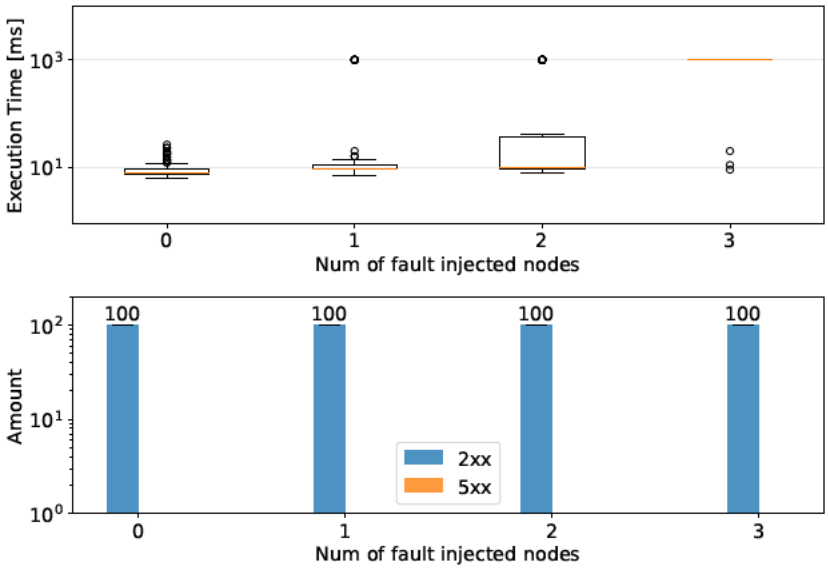


Figure 6.14: Triple Redundancy Experiment - Core Delay FD_{TCP}

nearly perfect to the scenario of normal operation if one or two instances are affected. If all instances of the core service are affected the failure detectors can detect the failure, but nothing can be done to mitigate it.

The consequences of the introduction of delay in one or more instances of the *QueryCache* service are displayed in Figure 6.16 and 6.17. It is interesting to notice that the introduced delay not only increases the response times, but also generates many new cache entries. Some new cache entries are expected but most of them should be looked up, as they already exist. Normally, the core service should be able to retrieve a key that was already stored, but the instances of the cache service seem to be disrupted by the delay and therefore part in two disjoint clusters. Thus, the core service generates a new cache entry and returns the status code 201. Looking at response times the HTTP-based failure detector is able to detect the affected instances more consistently than the TCP-based approach. However, the disjoint cluster is still a failure that cannot be mitigated by better failure detection.

The impact of delay on the *QueryDistribution* service is not as severe as for the other services. One reason is that the publish subscribe architecture is not as frequently used as the other components. However, it is interesting to note that the *QueryDistribution* service still continues to operate quite normally. Only a few outliers are reported. The reason for this is that replication is only performed on $n-1$ instances (n being the total

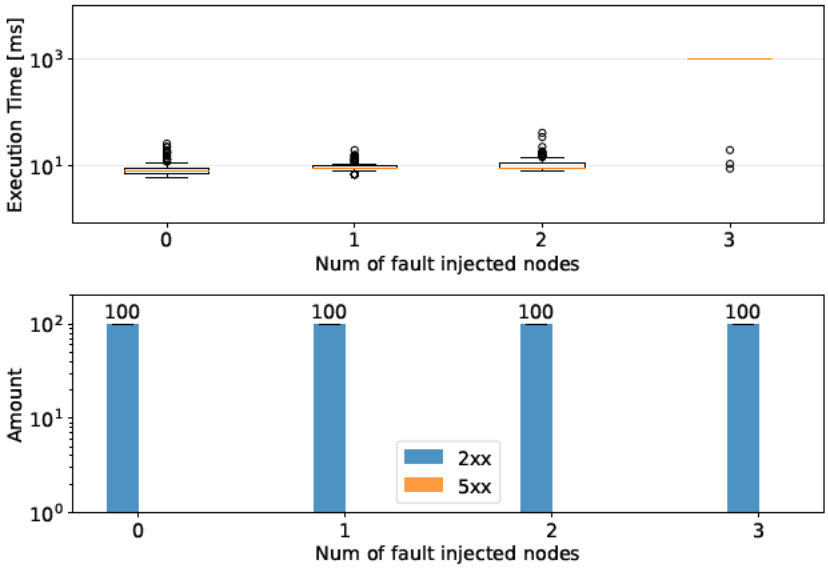


Figure 6.15: Triple Redundancy Experiment - Core Delay FD_{HTTP}

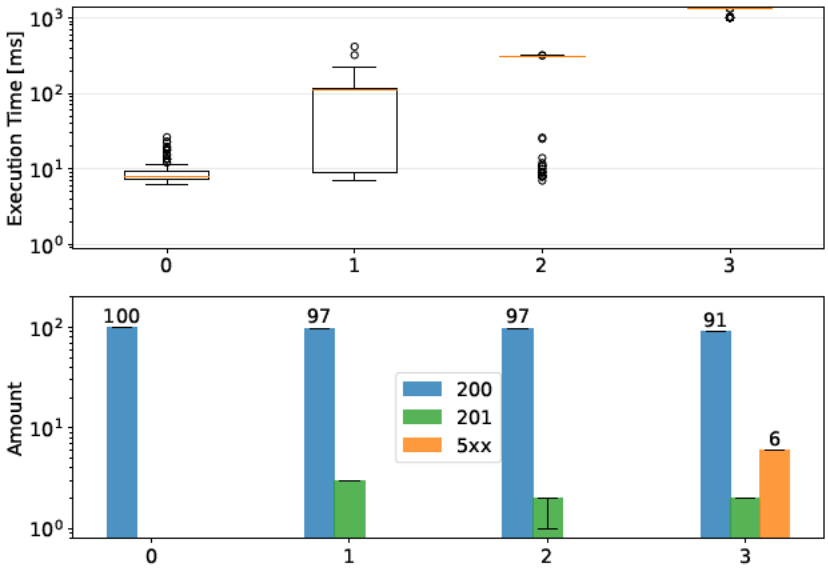


Figure 6.16: Triple Redundancy Experiment - *QueryCache* Delay FD_{TCP}

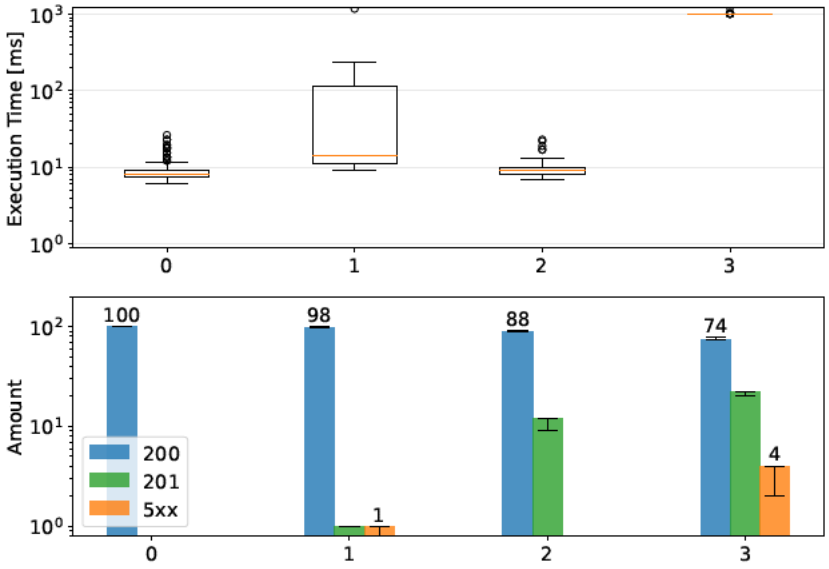


Figure 6.17: Triple Redundancy Experiment - *QueryCache* Delay FD_{HTTP}

number of instances of the service). Thus, the communication overhead is not as great as in case of the *QueryCache* component.

In summary, the TCP-based failure detector was not able to recognize the affected instances. The HTTP-based failure detector on the other hand was able to significantly reduce the response times for up to $n-1$ affected instances of a given service. However, it has to be noted that the HTTP-based approach did not consistently recognize all affected instances all the time, as Figure 6.17 demonstrates. The upper quantile is slightly above 100ms and thus leads to some timing failures as it was defined in chapter 5.

Introduction of Loss

The following experiment results depict the systems behavior if some nodes experience a greater packet loss rate on the network link than others.

Figure 6.19 depicts the experiment results for the introduction of a loss rate of ten percent for one or more instances of the core component. Figure 6.19 will be used as a representative for the other components as the results are very similar. The missing figures are presented in the appendix A. In the scenario that all instances of a given service experienced a higher loss rate some request are answered by a HTTP status code of 500. This is not acceptable, as the repetition of the action could have probably resolved the

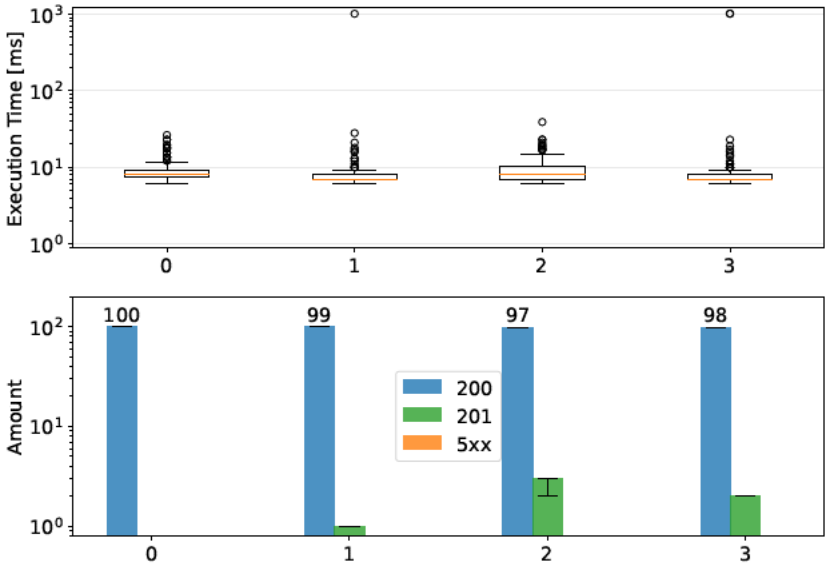


Figure 6.18: Triple Redundancy Experiment - *QueryDistribution* Delay

issue. In case of one or two affected services quite a few outliers are recorded. Neither of the two failure detection algorithms are able to detect this fault. As both failure detectors rely on probes to evaluate and measure the destinations state the packets that are miss are only encountered from time to time. Thus, they have not really an effect on the failure detection algorithms.

6.3 Concluding Remarks

The experiments conducted on the discovery service approach (A), that does not utilize the fault-tolerant techniques of process replication and failure detection, demonstrate that it is not sufficient to rely on robustness and reliable client-server communication in a distributed environment. The approach A is able to tolerate faults due to erroneous inputs and unreliable network environments, but struggles with the loss of services and service restarts. Crash Failure and restarts lead to the whole system becoming unavailable. This is not acceptable for a infrastructure service. However, once a service is restarted the system can operate normally again. Thus, it is possible to restart single components without crashing other services in the process.

The discovery service approach (B), that implements process replication and failure detection, tries to mitigate the shortcomings of the approach A. The observations of

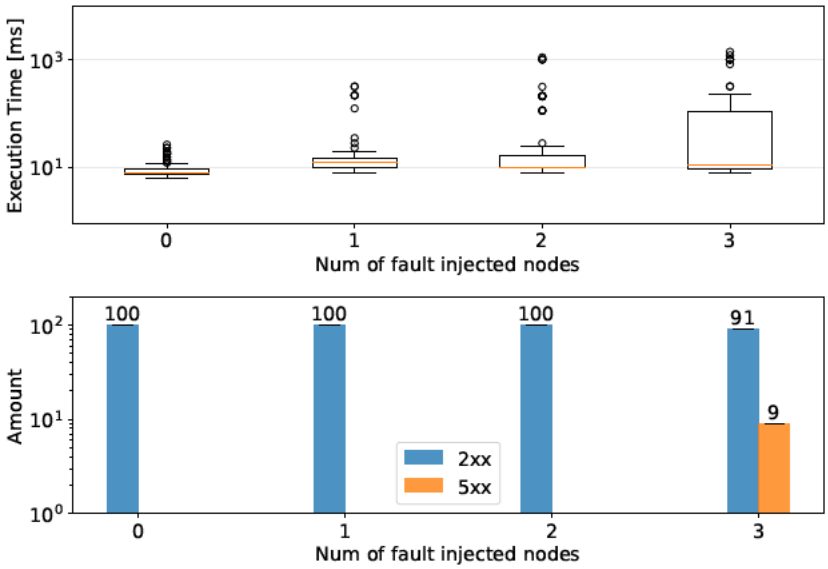


Figure 6.19: Triple Redundancy Experiment - Core Loss

the experiment runs show that crash failure can be masked up to $n-1$ instances of a service with a total of n replicas. Both failure detection algorithms were able to detect the the affected instances. The approach B is able to tolerate node stress with respect to correctly processing requests up to $n-1$ failed instances. However, response times increased. Therefore, timing failures still occur. In this case the HTTP-based failure detection approach is able to detect the affected instances and mitigate the fault. The TCP-based failure detection approach fails to detect the affected instances. A summary is presented in Table 6.1.

Table 6.1: Failure Detection Evaluation for $n-1$ affected instances of a service

FD	**Fault-Injection Method**			
	Crash	*Stress*	*Delay*	*Loss*
TCP-based	✓	✗	(✓)	✗
HTTP-based	✓	(✓)	(✓)	✗

Legend		
	✓	*successfully masks the injected failure*
	(✓)	*partially masking the injected failure*
	✗	*fails to mask the injected failure*

It is important to note that the *QueryCache* service that is based on *ignite* is greatly affected by node stress. If the stress is too great or enough nodes are affected by stress the whole cluster will fail. As Ignite is a distributed cache that is based on a *Distributed Hash*

Table (DHT) this behavior is especially interesting as many other research approaches also incorporate or solely depend on DHTs.

Moreover, Ignite is also vulnerable to changes in link characteristics due to delay. Figure A.5 illustrates that nodes of the Ignite cluster can even crash if another node experiences a great delay. This is a rare incident, as it could only be reproduced two times in several experiment runs, but it was observed. Furthermore, in case of an introduced delay the number of multiple creations of new query cache entries could be related to a cluster that became disjoint. This is a great example that every component of a system and the system as a whole has to be evaluate using experiments to exploit weaknesses. A possible way to mitigate the failure of the cache cluster would be to introduce a local cache for each client and core service containing the most popular consumer queries and provider offers.

Other services, such as the core service, handled the introduced delay rather well. Up to $n - 1$ instances the failure could be mitigated by the HTTP-based failure detector. The TCP-based approach did not perform well in this scenario. In case of $n - 1$ affected instances the HTTP-based approach was able to completely mask the fault.

Furthermore, as illustrated in chapter 4, the proposed discovery approach is designed to be deployed in multiple disjoint clusters according to the partitioning scheme. Thus, a failure of a whole cluster would indeed make the cluster become unavailable, but other clusters can continue to work. Since the clusters are disjoint a partial failure is not supposed to have an impact on other clusters of the *CaDDS*.

In order to further examine the system more experiments, especially with more dynamic behavior, would be interesting to conduct. Furthermore, it would be quite interesting to to implement passive failure detection algorithms in order to evaluate their strengths and weaknesses as compared to active approaches. Using caching on multiple levels and consensus algorithms to determine the state of another service, e.g. in order to detect disjoint clusters, would be also be interesting.

7 Conclusion

The main focus of this thesis is to demonstrate and evaluate the importance of fault-tolerant systems in the context of infrastructure services, such as discovery mechanisms, especially for the IoT. This thesis proposes a discovery service that complies with the requirements of the IoT, as discussed in chapter 2, and promotes the openness and interoperability of the future IoT. Currently, the IoT is still a collection of silos with devices that use proprietary software and application protocols connected via proprietary gateways. In order to transition to an open and interoperable environment in which anyone can communicate with every service through any network and over any protocol, robust and fault-tolerant infrastructure services are essential.

This thesis offers an open and interoperable approach for service discovery in the IoT. Furthermore, the proposed approach is evaluated based on different fault-tolerance levels. The evaluation compares a system with no process redundancy to an approach that uses process redundancy to mask failures of different kinds. The two approaches are evaluated using *fault-injection* methods. The experiments are able to demonstrate that robust software can withstand some faults, but faults such as crash failures, changes in network characteristics or stress cannot be masked or only to a certain extend. Proposing two different failure detection mechanism, this thesis is able to demonstrate the areas in which they perform effectively and their shortcomings. It can be concluded that effective failure detection is crucial for masking failures.

Moreover, this thesis shows that the DHT-based cache application *ignite* reacts rather sensible to stress and delay, leading to a disjoint cluster topology and even to crash failure of the whole cluster. This is an interesting observation as many other research approaches on discovery services for the IoT rely partly or fully on similar DHT-based architectures. This exemplifies the importance of evaluating all components of a system, also the "off the shelf" components, such as distributed hash table components, that are incorporated into the system.

Further evaluations using more dynamic experiments, passive fault detection mechanisms via information dissemination, would be quite interesting to conduct on the proposed approach. Moreover the effects of different layers of caching could also be of interest. The caveat of further caching could be that caches at different levels might represent different states of the IoT environment and might increase the amount of false-positive results.

Bibliography

[1] Abowd, Gregory D. ; Dey, Anind K. ; Brown, Peter J. ; Davies, Nigel ; Smith, Mark ; Steggles, Pete: Towards a better understanding of context and context-awareness. In: *International symposium on handheld and ubiquitous computing* Springer (Veranst.), 1999, S. 304–307

[2] Alvaro, Peter ; Rosen, Joshua ; Hellerstein, Joseph M.: Lineage-driven fault injection. In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, 2015, S. 331–346

[3] Atzori, Luigi ; Iera, Antonio ; Morabito, Giacomo: Understanding the Internet of Things: definition, potentials, and societal role of a fast evolving paradigm. In: *Ad Hoc Networks* 56 (2017), S. 122–140

[4] Avizienis, Algirdas ; Laprie, J-C ; Randell, Brian ; Landwehr, Carl: Basic concepts and taxonomy of dependable and secure computing. In: *IEEE transactions on dependable and secure computing* 1 (2004), Nr. 1, S. 11–33

[5] Aziez, Meriem ; Benharzallah, Saber ; Bennoui, Hammadi: Service discovery for the Internet of Things: Comparison study of the approaches. In: *2017 4th International Conference on Control, Decision and Information Technologies (CoDIT)* IEEE (Veranst.), 2017, S. 0599–0604

[6] Basiri, Ali ; Behnam, Niosha ; De Rooij, Ruud ; Hochstein, Lorin ; Kosewski, Luke ; Reynolds, Justin ; Rosenthal, Casey: Chaos Engineering. In: *IEEE Software* 33 (2016), Nr. 3, S. 35–41

[7] Bengel, Günther: *Grundkurs Verteilte Systeme - Grundlagen und Praxis des Client-Server und Distributed Computing*. Springer Vieweg, 2014

[8] Bermudez-Edo, Maria ; Elsaleh, Tarek ; Barnaghi, Payam ; Taylor, Kerry: IoT-Lite Ontology / W3C. URL https://www.w3.org/Submission/2015/SUBM-iot-lite-20151126/, November 2015. – Forschungsbericht

[9] Bettini, Claudio ; Brdiczka, Oliver ; Henricksen, Karen ; Indulska, Jadwiga ; Nicklas, Daniela ; Ranganathan, Anand ; Riboni, Daniele: A survey of context modelling and reasoning techniques. In: *Pervasive and mobile computing* 6 (2010), Nr. 2, S. 161–180

[10] Bikakis, Antonis ; Patkos, Theodore ; Antoniou, Grigoris ; Plexousakis, Dimitris: A survey of semantics-based approaches for context reasoning in ambient intelligence. In: *European Conference on Ambient Intelligence* Springer (Veranst.), 2007, S. 14–23

[11] Bouzeghoub, Mokrane: A framework for analysis of data freshness. In: *Proceedings of the 2004 international workshop on Information quality in information systems*, 2004, S. 59–67

[12] Bray, T.: The JavaScript Object Notation (JSON) Data Interchange Format / RFC Editor. RFC Editor, December 2017 (90). – STD. – ISSN 2070-1721

[13] Cachin, Christian ; Guerraoui, Rachid ; Rodrigues, Luís: *Introduction to reliable and secure distributed programming.* Springer Science & Business Media, 2011

[14] Cheshire, S. ; Krochmal, M.: DNS-Based Service Discovery / RFC Editor. RFC Editor, February 2013 (6763). – RFC. – URL http://www.rfc-editor.org/rfc/rfc6763.txt. – ISSN 2070-1721

[15] Cirani, Simone ; Davoli, Luca ; Ferrari, Gianluigi ; Léone, Rémy ; Medagliani, Paolo ; Picone, Marco ; Veltri, Luca: A scalable and self-configuring architecture for service discovery in the internet of things. In: *IEEE Internet of Things Journal* 1 (2014), Nr. 5, S. 508–521

[16] Crosby, Michael ; Pattanayak, Pradan ; Verma, Sanjeev ; Kalyanaraman, Vignesh u. a.: Blockchain technology: Beyond bitcoin. In: *Applied Innovation* 2 (2016), Nr. 6-10, S. 71

[17] Daza, Vanesa ; Di Pietro, Roberto ; Klimek, Ivan ; Signorini, Matteo: CONNECT: CONtextual NamE disCovery for blockchain-based services in the IoT. In: *2017 IEEE International Conference on Communications (ICC)* IEEE (Veranst.), 2017, S. 1–6

[18] Hamzei, Marzieh ; Navimipour, Nima J.: Toward efficient service composition techniques in the internet of things. In: *IEEE Internet of Things Journal* 5 (2018), Nr. 5, S. 3774–3787

[19] IEEE Standards Committee u. a.: IEEE Std. 610.12-1990 IEEE Standard Glossary of Software Engineering Terminology. In: *IEEE* (1990)

[20] ISO 3166-1:2013 Country Codes / International Organization for Standardization. Geneva, CH, Januar 2013. – Standard

[21] Kim, Seokhwa ; Lee, Keuntae ; Jeong, Jaehoon P.: DNS naming services for service discovery and remote control for Internet-of-Things devices. In: *2017 International Conference on Information and Communication Technology Convergence (ICTC)* IEEE (Veranst.), 2017, S. 1156–1161

[22] Kopetz, Hermann: *Real-time systems: design principles for distributed embedded applications*. Springer Science & Business Media, 2011

[23] Kopetz, Hermann ; Verissimo, Paulo: Real time and dependability concepts. In: *Distributed systems (2nd Ed.)*. 1993, S. 411–446

[24] Koren, Israel ; Krishna, C M.: *Fault-Tolerant Systems*. Morgan Kaufmann Publishers, 2020. – ISBN 978-0-12-088525-1

[25] Lamport, Leslie ; Shostak, Robert ; Pease, Marshall: The Byzantine generals problem. In: *Concurrency: the Works of Leslie Lamport*. 2019, S. 203–226

[26] Li, Juan ; Bai, Yan ; Zaman, Nazia ; Leung, Victor C.: A decentralized trustworthy context and QoS-aware service discovery framework for the internet of things. In: *IEEE Access* 5 (2017), S. 19154–19166

[27] Li, Juan ; Zaman, Nazia ; Li, Honghui: A decentralized locality-preserving context-aware service discovery framework for internet of things. In: *2015 IEEE International Conference on Services Computing* IEEE (Veranst.), 2015, S. 317–323

[28] Lim, Brian Y. ; Dey, Anind K.: Toolkit to support intelligibility in context-aware applications. In: *Proceedings of the 12th ACM international conference on Ubiquitous computing*, 2010, S. 13–22

[29] MacKenzie, C M. ; Laskey, Ken ; McCabe, Francis ; Brown, Peter F. ; Metz, Rebekah ; Hamilton, Booz A.: Reference model for service oriented architecture 1.0. In: *OASIS standard* 12 (2006), Nr. S 18

[30] Martin, David ; Burstein, Mark ; Hobbs, Jerry ; Lassila, Ora ; McDermott, Drew ; McIlraith, Sheila ; Narayanan, Srini ; Paolucci, Massimo ; Parsia, Bijan ; Payne, Terry u. a.: OWL-S: Semantic markup for web services. In: *W3C member submission* 22 (2004), Nr. 4

[31] Newcombe, Chris ; Rath, Tim ; Zhang, Fan ; Munteanu, Bogdan ; Brooker, Marc ; Deardeuff, Michael: How Amazon web services uses formal methods. In: *Communications of the ACM* 58 (2015), Nr. 4, S. 66–73

[32] Niranjanamurthy, M. ; Nithya, B. ; Jagannatha, S.: Analysis of Blockchain technology: pros, cons and SWOT. In: *Cluster Computing* (2018), 03

[33] Perera, Charith ; Zaslavsky, Arkady ; Christen, Peter ; Compton, Michael ; Georgakopoulos, Dimitrios: Context-aware sensor search, selection and ranking model for internet of things middleware. In: *2013 IEEE 14th international conference on mobile data management* Bd. 1 IEEE (Veranst.), 2013, S. 314–322

[34] Perera, Charith ; Zaslavsky, Arkady ; Christen, Peter ; Georgakopoulos, Dimitrios: Context aware computing for the internet of things: A survey. In: *IEEE communications surveys & tutorials* 16 (2013), Nr. 1, S. 414–454

[35] Perera, Charith ; Zaslavsky, Arkady ; Christen, Peter ; Georgakopoulos, Dimitrios: Context aware computing for the internet of things: A survey. In: *IEEE communications surveys & tutorials* 16 (2014), Nr. 1, S. 414–454

[36] Pullum, Laura L.: *Software Fault Tolerance – Techniques and Implementation.* Artech House Publishers, 2001. – ISBN 978-1-58053-137-5

[37] Rapti, Elli ; Houstis, Catherine ; Houstis, Elias ; Karageorgos, Anthony: A bio-inspired service discovery and selection approach for IoT applications. In: *2016 IEEE International Conference on Services Computing (SCC)* IEEE (Veranst.), 2016, S. 868–871

[38] Razzaque, Mohammad A. ; Milojevic-Jevric, Marija ; Palade, Andrei ; Clarke, Siobhán: Middleware for internet of things: a survey. In: *IEEE Internet of things journal* 3 (2015), Nr. 1, S. 70–95

[39] Shelby, Z.: Constrained RESTful Environments (CoRE) Link Format / RFC Editor. RFC Editor, August 2012 (6690). – RFC. – URL http://www.rfc-editor.org/rfc/rfc6690.txt. – ISSN 2070-1721

[40] Somani, Nisha A. ; Patel, Yask: Zigbee: A low power wireless technology for industrial applications. In: *International Journal of Control Theory and Computer Modelling (IJCTCM)* 2 (2012), Nr. 3, S. 27–33

[41] Sundmaeker, Harald ; Guillemin, Patrick ; Friess, Peter ; Woelfflé, Sylvie: Vision and challenges for realising the Internet of Things. In: *Cluster of European Research Projects on the Internet of Things, European Commision* 3 (2010), Nr. 3, S. 34–36

[42] Tanganelli, Giacomo ; Vallati, Carlo ; Mingozzi, Enzo: Edge-centric distributed discovery and access in the internet of things. In: *IEEE Internet of Things Journal* 5 (2017), Nr. 1, S. 425–438

[43] The Apache Software Foundation: *Apache Ignite*. 2020. – URL https://ignite.apache.org/. – Zugriffsdatum: 2020-06-30

[44] The Apache Software Foundation: *Apache Kafka*. 2020. – URL https://kafka.apache.org/. – Zugriffsdatum: 2020-06-30

[45] Van Steen, Maarten ; Tanenbaum, Andrew S.: *Distributed Systems*. Maarten van Steen Leiden, The Netherlands, 2017. – ISBN 978-15-430573-8-6

[46] Vasseur, JP: *Terms Used in Routing for Low-Power and Lossy Networks*. RFC 7102. Januar 2014. – URL https://rfc-editor.org/rfc/rfc7102.txt

[47] Ververidis, Christopher N. ; Polyzos, George C.: Service discovery for mobile ad hoc networks: a survey of issues and techniques. In: *IEEE Communications Surveys & Tutorials* 10 (2008), Nr. 3

[48] Wang, Edward ; Chow, Richard: What can i do here? IoT service discovery in smart cities. In: *2016 IEEE International Conference on Pervasive Computing and Communication Workshops (PerCom Workshops)* IEEE (Veranst.), 2016, S. 1–6

[49] Wei, Qiang ; Jin, Zhi: Service discovery for internet of things: a context-awareness perspective. In: *Proceedings of the Fourth Asia-Pacific Symposium on Internetware* ACM (Veranst.), 2012, S. 25

[50] Wright, Austin ; Andrews, Henry ; Hutton, Ben ; Dennis, Greg: JSON Schema: A Media Type for Describing JSON Documents / IETF Secretariat. URL https://www.ietf.org/archive/id/draft-handrews-json-schema-02.txt, September 2019 (draft-handrews-json-schema-02). –

Internet-Draft. https://www.ietf.org/archive/id/draft-handrews-json-schema-02.txt

[51] Yanwei, Song ; Guangzhou, Zeng ; Haitao, Pu: Research on the context model of intelligent interaction system in the internet of things. In: *2011 IEEE International Symposium on IT in Medicine and Education* Bd. 2 IEEE (Veranst.), 2011, S. 379–382

[52] Ying, Pan ; Guan-Yu, Li ; Zhong-Jun, Lu: Community Division-Based SWoT Ontology Directory Service. In: *2016 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC)* IEEE (Veranst.), 2016, S. 123–127

[53] Yuan, Ding ; Luo, Yu ; Zhuang, Xin ; Rodrigues, Guilherme R. ; Zhao, Xu ; Zhang, Yongle ; Jain, Pranay U. ; Stumm, Michael: Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In: *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, 2014, S. 249–265

A Appendix

A.1 QueryCache Stress Experiment, HTTP-based FD

A.2 QueryDistribution Stress Experiment, HTTP-based FD

A.3 QueryCache Loss Experiment

A.4 QueryDistribution Loss Experiment

A.5 QueryCache Delay Experiment - Crash Failure

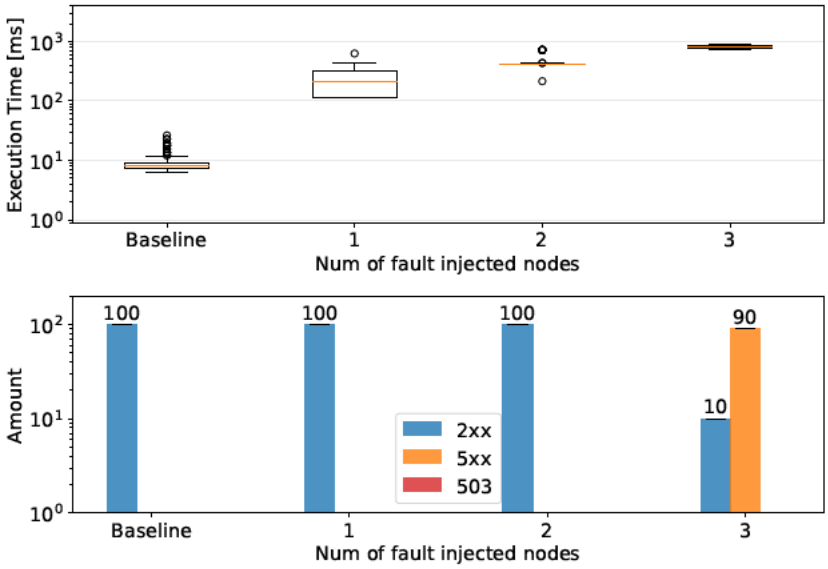


Figure A.1: Triple Redundancy Experiment - `QueryCache` Stress FD_{HTTP}

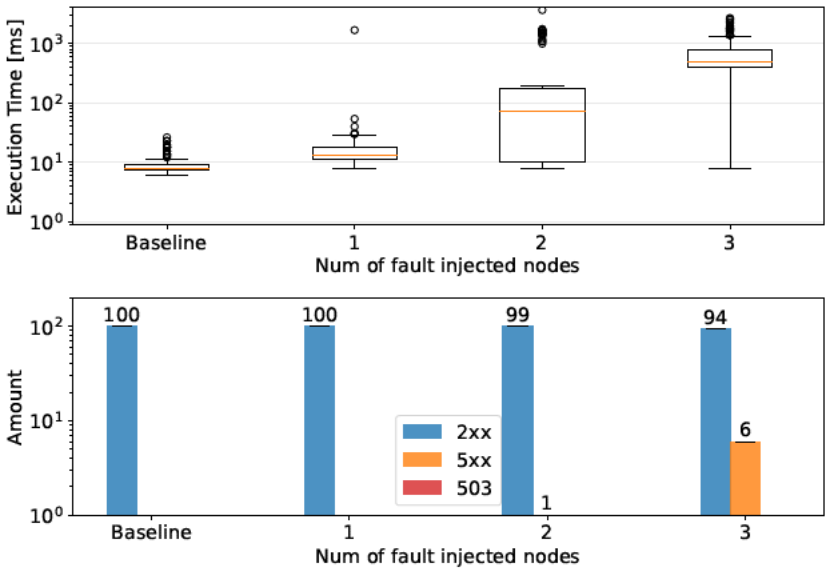


Figure A.2: Triple Redundancy - `QueryDistribution` Stress FD_{HTTP}

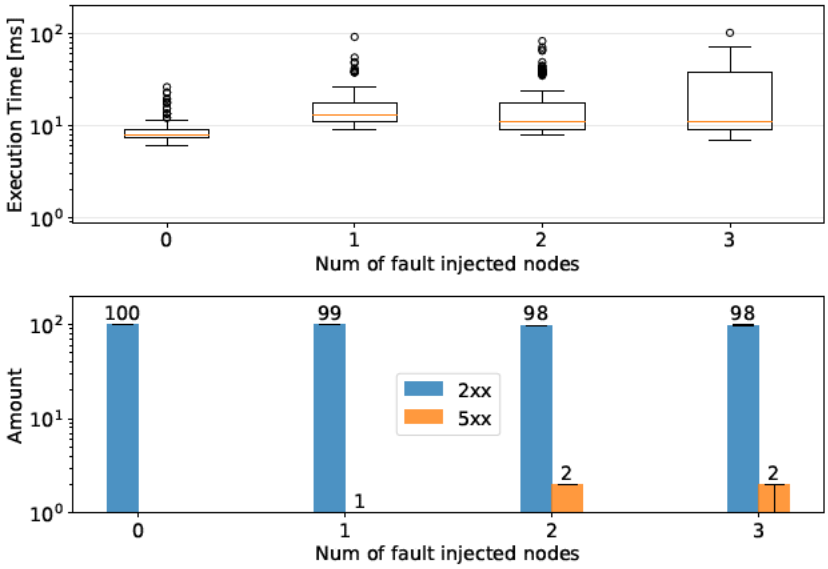


Figure A.3: Triple Redundancy - QueryCache Loss

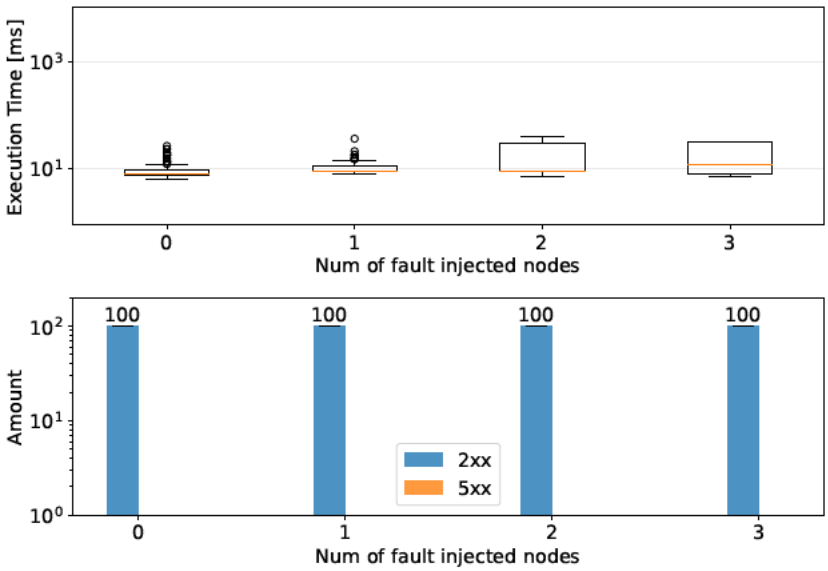


Figure A.4: Triple Redundancy - QueryDistribution Loss

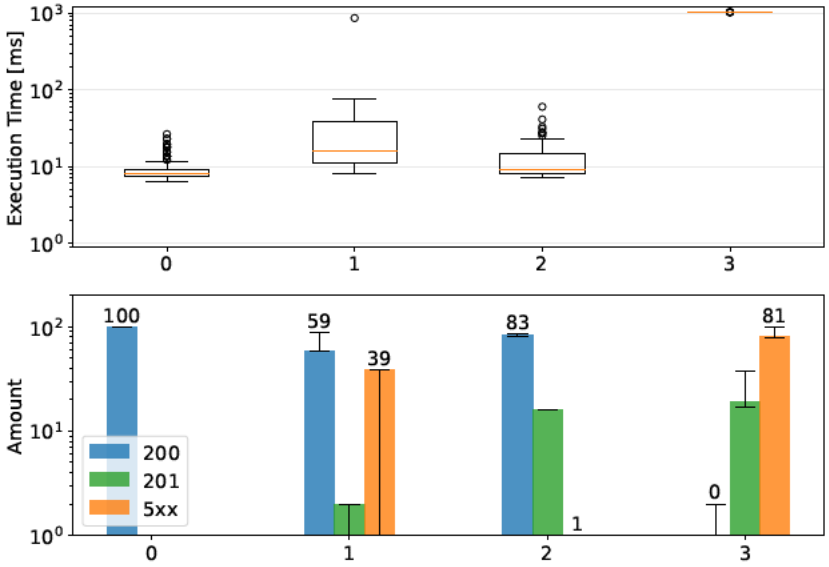


Figure A.5: Triple Redundancy - QueryCache Delay leads to crash failure

Glossary

availability The degree to which a system or component is operational and accessible when required for use.

context Any information that can be used to characterize the situation of an entity.

context acquisition Process of acquiring context from various sources.

context dissemination Process of distributing context information to interested consumers.

context reasoning A method of deducing new knowledge and better understanding, based on available context.

context representation The process of collecting data. Context models can either be static or dynamic.

context sink A component that is responsible for acquiring context data.

context source A component that provides context data, e.g. a physical or virtual sensor in the IoT scenario.

context-awareness Using context to provide relevant information or services to the user, where relevancy depends in the user's task.

error The difference between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition [19]. An error is part of the system's state that may lead to a failure [45].

error tolerance The ability of a system or component to continue normal operation despite the presence of erroneous inputs.

failure The inability of a system or component to perform its required functions within specified performance requirements. Note: The fault tolerance discipline distinguishes between a human action (a mistake), its manifestation (a hardware or software fault), the result of the fault (a failure), and the amount by which the result is incorrect (the error).

failure mode The physical or functional manifestation of a failure. For example, a system in failure mode may be characterized by slow operation, incorrect outputs, or complete termination of execution.

fatal error An error that results in the complete inability of a system or component to function.

fault An incorrect step, process, or data definition in a computer program [19]. A fault is the cause of an error [45].

fault injection Deliberately cause or introduce a fault in a system.

fault tolerance The ability of a system or component to continue normal operation despite the presence of hardware or software faults.

fault tolerant Pertaining to a system or component that is able to continue normal operation despite the presence of faults.

freshness Describes how "old" a piece of information is. Freshness can be distinguished in a currency and timeliness factor [11].

maintainability Measure of the time interval required to repair a system after the occurrence of a non critical failure mode.

mistake A human action that produces an incorrect result.

reliability The ability of a system or component to perform its required functions under stated conditions for a specified period of time.

robustness The degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions.

safety The ability of a system to prevent the occurrence of catastrophic events in the presence of critical failure modes.

ZigBee A low-cost, low-power, wireless mesh network standard targeted at battery-powered devices in wireless control and monitoring applications [40].

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort	Datum	Unterschrift im Original