Container Orchestration in Limited Availability Zones Domains

University of Twente

In collaboration with Thales Nederland B.V.

Author
Saül Abad Copoví
M.Sc. Embedded Systems
University of Twente

Thales Supervisor
Max Riesewijk MSc

University of Twente Supervisors
Prof.dr.ing. Paul J.M. Havinga
Dr.ir. Peter Bosch

Examination Committee
Prof.dr.ing. Paul J.M. Havinga
Dr.ir. Peter Bosch
Prof.dr.ir. R.M. van Rijswijk-Deij

Enschede, The Netherlands
1st May, 2024

UNIVERSITY OF TWENTE.

THALES

***Abstract*—Mission-critical systems are of vital importance and their failure may result in catastrophic outcomes. Therefore, applying cloud-native technologies such as container orchestration to support them is no trivial task; even more so when the environment greatly differs: while usually cloud-native focuses on a geographically disperse scenario, with servers hosted in distant data-centres, and thousands of clients, the mission-critical systems at hand are restricted to only 2 Availability Zones (AZ), as exemplified by mission-critical systems in the naval domain. Specifically, when AZs are limited, the commonly used consensus protocols of the cluster state store of container orchestrators fail to address the requirements of such vital systems: if one node fails, the cluster can never achieve consensus and no further changes can be made on its state.**

This thesis considers a solution to the problem by the use of a publish-subscribe database. A benchmark comprised of four metrics, namely latency, throughput, consistency and partition tolerance, is described. In a prototype system, results are gathered and analysed by themselves and in comparison with proven cluster state stores (i.e. *etcd* and relational databases, specifically PostgreSQL). The final results indicate that a pub-sub data store should be capable of addressing the needs of cluster state stores for mission-critical systems, thereby opening a new avenue of research not thoroughly explored before.

***Index Terms*—Distributed systems, distributed databases, container orchestration, publish-subscribe, cluster state store.**

I. Introduction

Failure in mission-critical systems may result in catastrophic outcomes: destruction of property or even the loss of lives. Take, for instance, the case of Ariane 5, in which the failure of its flight control system resulted in the self destruction of a rocket merely 40 seconds after take-off, causing the loss of approximately 370 million USD [1]. Although this example is one of the most infamous ones, it is not an isolated accident, and others have succeeded it, like the 2019 crash of the Israeli Beresheet, which sadly ended up crashing against the moon due to a reboot induced by the failure of one of its two redundant accelerometers [2]. Despite the two mentioned cases being limited to spacecraft errors, mission-critical systems are present in many other areas, such as the naval domain.

Thales Nederland is a company that develops naval mission-critical systems as part of its core business. Thales considers that the use of cloud-native technologies can greatly benefit its Combat Management Systems (CMS), particularly the use of containerisation, one of the foundational pillars of these technologies [3]. However, failures in such system could easily result in tragedy, and it is a fact that the usual cloud-native environment greatly differs from the naval one. Cloud-native is related to a scenario in which thousands of clients access a service, usually located in multiple, geographically-distributed servers, as noted in Gannon *et al.* [4]. Furthermore, these servers are often not hosted in-house, but at a cloud service provider's servers (such as Amazon, Google or Microsoft) [4]. In Thales' case, due to strict security reasons, servers need to be installed on-premise (i.e. on the vessel itself). Therefore, spatial limitations exist that restrict the number of Availability Zones (AZs). Where in usual container orchestration 3 AZs exist, there are only 2 in the most common naval scenario, in the form of two separate server rooms equipped with several racks and redundant mechanisms to protect the mission-critical systems from failure. Nevertheless, this is not the only difference: while in typical cloud-native situations thousands of geographically distributed clients need to access the system, in the naval domain that number is much lower; in fact, users are limited to the officers using the Multifuctional Operator Consoles (MOCs) located in the ship's Combat Information Centre.

A. Problem Statement

As aforementioned, the differences between the usual cloud-native environment and that of a limited AZs domain (e.g. naval domain) are many. Issues arise when only 2 AZs exist. In order to understand the problem at hand, it is necessary to talk about Kubernetes (K8s); originally developed by Google and maintained by the CNCF as an open source project, Kubernetes is the most widely used container orchestrator [5] and regarded as the most mature. However, K8s can only use the Key-Value Store (KVS) *etcd* as its cluster state store[1]. *etcd* guarantees strong consistency by using the Raft consensus algorithm: in order to write into the store, be it a change of value or to introduce or delete a key, a quorum among *etcd* nodes needs to be reached. Considering any cluster with n nodes, quorum is achieved with a minimum of $\lfloor n/2 \rfloor + 1$ node votes in favour [6]. This means that to achieve fault tolerance of one node, a minimum of three nodes are needed. Then, if only 2 AZs exist ($n = 2$), and one was to fail, the minimum quorum of $\lfloor 2/2 \rfloor + 1$ could never be reached. Effectively, that means that even if one node is still alive, the *etcd* cluster cannot log any writes, thus making the state of the K8s cluster unchangeable. The use of vanilla K8s in limited AZs domains is not a suitable solution, even less when robustness is of the essence, as is the case of the CMS example: if a projectile where to hit a server room, the whole CMS could be rendered inoperative, leaving the ship defenceless.

[1]Database in which K8s stores the state of the cluster. Any change in the state, such as the creation of a node or the allocation of resources, needs to be logged into the cluster state store for it to take effect. Refer to the appendix for a diagram depicting *etcd* in a default K8s cluster.

B. Consensus Protocols

As explained previously in subsection I-A, the problem lies on the Raft algorithm *etcd* uses. These sort of algorithms, usually known as consensus protocols, are of vital importance to ensure consistency in distributed databases (DB), be they a KVS, like *etcd*, or even for a much simpler state machine.

As detailed in Mullender [7], consensus protocols first appeared to keep consistency for distributed state machines. Afterwards, they were implemented on more complex systems, usually using them to define a leader on a cluster of replicas, a model also known as primary-backup systems. With apparently the first protocol defined on 1976 by Alsberg and Day [7], new developments appeared shortly after, such as the 2-Phase Commit protocol in 1979 [8] or the Oral Message protocol in 1982 [9]. It was with the definition of the consensus-based Paxos by L. Lamport in his mythical paper *The Part- Time Parliament* [10] (and its subsequent simpler explanation in [11]) that the scene really changed. With Paxos becoming the state of the art for consensus protocols, many improvements have been published on it, such as Fast Paxos [12] (which reduces message delays compared to the original Paxos), Vertical Paxos [13] (which focuses on improving changes in the cluster configuration considered by the protocol), Leaderless Byzantine Paxos [14] (which improves Paxos' resilience against Byzantine failures), or Multi-Paxos [15] (which allows the use of multiple leaders concurrently), to name some. It is necessary to note that the Paxos protocol has been implemented in systems such as Google Spanner, ZooKeeper (as a consensus algorithm named Zab [16]) and Azure Cosmos [8], albeit with pertinent variants and modifications for each case.

Even after the many years it has been in use, Paxos is still notorious for its complexity. With this as a main trigger and with the purpose of becoming an alternative to Paxos, Raft was created in 2014 [17]. Raft is a consensus-based protocol and equivalent to its predecessor in terms of performance and fault-tolerance, while being simpler to comprehend [18]. A key feature of Raft is its persistent storage with a historic log, that ensures that no data is lost even in the presence of failures, as well as a mechanism for log compaction to avoid storage exhaustion. Another feature which introduces a refinement over Paxos is Raft's dynamic reconfiguration, one of the pillars of the protocol. All these features make Raft capable of ensuring strong consistency even in unfavourable environments, and its advantages have fostered the development of many Raft implementations, with the *etcd*/raft being one of them. In any case, Raft, as well as Paxos, still requires 3 AZs in order to guarantee failure tolerance, which makes them unsuitable for limited AZs domains.

But Paxos and Raft were not created to deal with Byzantine attacks. In fact, one of the assumptions made by Raft is that no Byzantine failures can occur [18]. This kind of failures are typically induced by malicious nodes that intentionally deliver false messages in order to disrupt the system, although they can also be caused by failures in software [7]. To face this problem, the Practical Byzantine Fault Tolerance (PBFT) protocol was created in 1999 [19]. The main objective of this protocol is to provide consensus even in the presence of Byzantine attacks, which previously had been done in an unpractical manner, usually relying on synchrony among nodes.

Finally, with the emergence of cryptocurrency and blockchain, new consensus protocols have appeared which are also Byzantine-tolerant. Some of them are widely known, such as Proof of Work (the consensus mechanism behind Bitcoin), or Proof of Stake, as explained in [8]. However, most of these protocols have been created to deal with public blockchains in highly Byzantine environments, which have and require characteristics widely different to those of a closed network with limited AZs, such the example at hand.

C. *Related Work*

The naval domain is not the only environment in which mission-critical systems are subjected to spatial constraints. The automotive and aviation worlds face a similar problem, with systems sometimes relying on 2 computers. With the aim of overcoming the strict constraints of distributed real-time systems in vehicles, the Time Triggered Protocol (TTP) was defined in [20]. Extensive literature exists on TT architectures, with current real-life implementations and uses, such as in the case of flight control systems on airplanes [21] [22]. Although it can be seen that a TT Architecture could be applied in the subject treated in this thesis, the cluster state store of a container orchestrator system is neither tied to strong real-time needs nor is considered to be a large real-time application comprised of several nearly autonomous clusters and nodes, as opposed to the situation defined in [22].

The fact that K8s can only use *etcd* as its cluster state store has prompted some research on its replacement. Specially, *etcd*'s performance has been questioned in regard to the use of the Raft algorithm, as it requires more communication and confirmation from other nodes to achieve a quorum. As explored in [23], latency increases together with the number of *etcd* nodes, even more so when a write operation is performed. Similarly, bigger clusters are affected by a degradation of throughput. Research analysing the impact *etcd* has in performance has also been published, such as in [24] and [25], which also comment on *etcd*'s scalability limitations. Although both [23] and [25] propose solutions to said issues, they are aimed towards edge architectures in the typical cloud-native environment.

Still considering the criticism of K8s dependency on *etcd*, it is necessary to take into account the existence of the lightweight Kubernetes distribution K3s[2]. Said distribution actually tries to decouple *etcd* from Kubernetes, by offering a shim capable of translating *etcd* calls to SQL queries, thus allowing the use of relational databases as cluster state store. In spite of this, the main K3s architecture does not directly consider the use of a distributed relational database, which requires the use of additional mechanisms.

Other databases that could be potentially used as cluster state store for K8s exist, although they offer no native support. There is interest in KVS in the research community, as well as on other types of distributed databases, for which several designs have been published. Notwithstanding, according to the PACELC theorem[3] [26] not every desired property can be achieved: apart from having to choose between availability and consistency in the case of network partitions, one must choose between latency and consistency during normal working conditions.

Some of the designs examined focus on cloud environments in which they must serve a high amount of petitions in the lowest time possible, thus choosing to improve availability and latency while sacrificing strong consistency. For instance, this is the case of Riak, which was built based on [27] (although it offers strong consistency, for which consensus is required [28]), as well as that of Anna [29] (which focuses on allowing scalability in order to meet the needs of rapidly growing cloud systems).

[2]More details on K3s are provided in subsubsection II-E1.

[3]Extension of the CAP theorem that also considers the properties of the network under normal working conditions, when no partitions occur.

Other stores were built with consistency in mind. One of them is CockroachDB [30], which implements a KVS on top of which an SQL layer is used for user interfacing purposes. But CockroachDB also uses the Raft algorithm for consistent replication over its nodes. Another example of a consistent database is Google's Spanner [31], which started as a KVS but eventually evolved into a distributed relational DB capable of using SQL syntax [32]. Spanner backs Google's mission-critical systems and uses Paxos for consensus, but it is restricted to hosting on Google's infrastructure. Even though all these mentioned databases are diverse, there is one characteristic common for all of them: they have been designed with a globally distributed cloud environment and data centres in mind, with many of them using Raft in order to ensure consistency.

As seen in [33], the implementation of *etcd* in a two-node cluster has been considered before, explicitly proposing that *etcd*'s mirror functionality could be used to replicate data from a master node into a slave node. Nonetheless, apart from requiring the development of a custom loadbalancer, this functionality cannot offer ordering guarantees and the order of updates should not be regarded as reliable [34]. Furthermore, also in [33], the developer team indicates that there is no interest in implementing any solution that could introduce inconsistency in the store and explicitly discards a 2-node implementation.

For these reasons seen above, it has been decided to focus the research on replacing the default cluster state store of Kubernetes, *etcd*, with a different store.

D. Preliminary Considerations

Considering everything mentioned in subsection I-B and subsection I-C, a series of considerations have been made. First, it is apparent that no consensus protocol exists that directly addresses the needs of container orchestration in limited AZs domains. Secondly, it has been seen that ensuring strong consistency is often achieved using Raft or Paxos, which are not applicable when only 2 AZs exist. Thus, it has been decided to explore the use of eventually consistent stores as cluster state store for Kubernetes and, more precisely, the use of Publish-Subscribe architectures.

E. Publish-Subscribe Systems

Publish-Subscribe (Pub/Sub) is a common communication schema for distributed systems, a step ahead from the earlier message queue paradigm. Even though many approaches exist, the basis of a Pub/Sub system is common: a set of publishers send messages that are then received or retrieved by those subscribers interested in them, which need not have explicit knowledge of each other [35]. They were originally developed as a middleware in order to provide support to large-scale settings, decoupling time, space and synchronisation among publishers and subscribers [36]. By using common transport layer protocols (e.g. TCP), this sort of systems are usually flexible and easy to deploy in any network, provided firewalls and other security measures have been configured accordingly [37]. Additionally, although not originally intended with this purpose, many implementations of Pub/Sub provide the possibility of keeping persistent logs and can subsequently be used as a data stores. Because of this, considering the constricted environment of limited AZs domains, it has been deemed relevant to explore these systems to overcome the difficulties introduced by the need of consensus of the *etcd* store.

Taking into account that Pub/Sub systems have been a topic of interest for distributed architectures since the nineties, a number of different implementations of the model exist. Among them, one of the most widely known is Apache Kafka. Originally developed by LinkedIn employees before being open sourced, the main purpose of Kafka was log processing while combining the capabilities of log aggregators and Pub/Sub message systems, based on a distributed log [38]. Nowadays, the functionalities Kafka offers are wider: apart from being a log aggregator, it is also used as a message broker, for metrics collection and for data streams processing, to name some of its uses [39]. A versatile system, Kafka is designed to be horizontally scalable, durable and fault tolerant, which makes it ideal to act as a central data hub for organisations [40]. It has been shown to be suitable for the management of high amounts of data with high throughput and low latency, for applications such as fast data analytics and big data [41][42]. Everything considered, it would be theoretically possible to use Kafka as cluster state store for K8s in a limited AZs domain. Although Kafka provides high throughput with low latency, the latter is in the order of milliseconds, which could compromise meeting the requirements of the system[4] [43]. As such, it has been disesteemed as an option for the case at hand.

Other systems usually capable of implementing the Pub/Sub schema are Message Oriented Middleware (MOM) systems. In fact, Kafka itself can be classified as a MOM. RabbitMQ is usually mentioned as the preferred open source alternative to Kafka and it offers similar functionalities, albeit with different architecture, as shown in [44]: RabbitMQ relies on a network of exchanges where routing of messages to a specific queue is decided. It is from the queues that the messages eventually reach the subscribers, whereas Kafka focuses on a topic-specific,

[4]Refer to subsection III-B, more specifically to requirement R5.

indexed log, from which consumers can retrieve messages. RabbitMQ has been shown to be particularly useful in situations where complex routing is needed and it is capable of achieving high scalability [41]. However, as shown in [44], RabbitMQ initially did not provide long term storage capabilities. Newer versions allow for the use of a persistence layer, although it may underperform when a high number of queues need simultaneous disk access [45]. Therefore, considering the persistence limitations, RabbitMQ is considered suboptimal to be used as cluster state store.

Nevertheless, it needs to be highlighted that RabbitMQ is an implementation of the Advanced Message Queuing Protocol (AMQP), one of the MOM standard protocols. Other protocols exist, such as the older Java Messaging Service (JMS, which only supports Java), the eXtensible Messaging and Presence Protocol (XMPP), MQ Telemetry Transport (MQTT), and the Data Distribution Service (DDS). Each of these protocols has different purposes: AMQP is used by industries that require high reliability, such as finance [44]; MQTT is used on low-power devices with small message payloads [46]; XMPP is oriented towards instant messaging [47]; and DDS targets real-time, dependable systems [48]. Not all protocols, then, can meet the needs of a cluster state store, where persistent storage, consistency and reliability are key.

While MQTT could be an option, it has been shown in [49] that most implementations favour availability and performance over communication integrity, and ensuring resiliency is a challenge. Because of this, with AMQP having been explored with RabbitMQ, JMS being dependant on Java and XMPP's focus on instant messaging purposes, DDS seems to be the best suited choice[5].

[5]Refer to subsection I-F for further elaboration on the decision.

Finally, apart from the systems mentioned above, major cloud providers also offer Pub/Sub solutions, such as Google's Pub/Sub and Pub/Sub Lite [50], Amazon's Simple Notification Service (SNS) [51], or IBM's MQ [52]. However, all these Pub/Sub implementations are not open source and need to run on their respective vendors' infrastructure, as well as being oriented towards the most common cloud-native environment, thus being unsuited for the requirements of limited AZs domains.

F. Solutions Considered

Considering the results of the literature research, the use of a pub/sub DDS-based cluster state store is considered as a possible solution to implement the cluster state store for K8s in domains with limited AZs. First of all, the DDS standard has been used in demanding fields such as defence, automotive, finance and medical [53], and has a wide presence in mission-critical distributed real-time systems, such as air traffic control systems [54]. Also, according to [55], the US Navy sees DDS as the choice for cyber-secure and cohesive communications. In addition, in [56], it has also been demonstrated that DDS works well in small wired networks, a scenario that closely resembles the naval domain example. Even though DDS provides eventual consistency, as opposed to *etcd*'s strong consistency, the fact that it is a system oriented towards real-time applications promotes that the distribution of the data is as fast as possible, thus limiting the inconsistency time to a minimum during normal operation conditions. Finally, DDS has been shown capable of meeting strict requirements of mission-critical systems' stores, as it is used in Thales Nederlands' current system. Because these reasons, DDS seems to be a suitable solution for container orchestration in domains with limited AZs.

Nonetheless, in order to determine whether the use of a DDS-based cluster state store is feasible, it is necessary to test it. In order to do so, it is first needed to create a prototype as explained in subsection II-G, and then determine how it compares against other proven cluster state stores. Among those, the most evident is *etcd*, which needs to be taken into account as the base for comparing any cluster state store different from the default (leaving aside the fact that this system requires an auxiliary witness node capable of voting to reach consensus, which makes it unfeasible as a solution in the context of limited AZs). Finally, as briefly mentioned in subsection I-C, another possibility is the one provided by the Kubernetes distribution K3s; it allows using an external relational database as cluster state store out of the box, thanks to its Kine shim. Notwithstanding, additional mechanisms are needed to distribute the DB among the 2 AZs[6].

G. The Data Distribution Service

In this section, a general understanding of the Data Distribution Service (DDS) is provided, with the aim to better introduce the reader to the inner workings of the system and to the terms used in subsequent sections.

As detailed in [48], DDS is a MOM standard maintained by the Object Management Group. The DDS model is build around a fully distributed *Global Data Space*, accessible by the publishers and subscribers. *DomainParticipants* are used by the processes to connect to a particular communication plane, the so called *domain*. In order for a process to become a *publisher*, it needs at least one *DataWriter* object, while *subscribers* require at least one *DataReader*. *DataWriters* and *DataReaders* are linked to a particular *topic* they are interested in. *Topics* in DDS consist of a triad of a unique name, a data type and a series of Quality of Service (QoS) policies. These QoS are used to specify the behaviour of any DDS entity: for example, the behaviour of a *publisher* is defined by the QoS of the *topic* it is associated to, together with the QoS of the *topic*'s *DataWriter* and the QoS of the *domain participant* itself.

DDS uses a specific protocol: the DDS Interoperability Wire Protocol or Real-Time Publish-Subscribe Protocol (DDSI-RTPS or RTPS for short), specified in [57]. Said protocol is strongly regulated by the QoS of the system and was created in order to meet the requirements of DDS applications. Out of the four modules composing RTSP, the most important aspects of the Structure module are presented first, followed by an overview of the Behaviour module, as they are most relevant for this paper[7].

Within RTSP, *publishers* and *subscribers* are considered *endpoints* of the system, which map to corresponding DDS entities (e.g. RTSP's *Participants* map to DDS' *DomainParticipants*). In RTSP, both *publishers* and *subscribers* maintain a *HistoryCache*, the element responsible of forming the interface between DDS and RTSP. The content of *HistoryCaches* differs for *writers* and *readers*: for *writers* they contain the partial history of changes to data-objects needed to service existing and future matched *readers*, whereas for the *readers* they keep a partial superposition of changes to data-objects made by their matching *writers*. Therefore, *writers* first log changes in their *HistoryCache*, which can then be published together with a

[6]The distributed SQL solution is out of the scope of this thesis.

[7]The other two modules are the Message module, which defines the structure of RTSP messages, and Discovery, which deals with how RTSP is capable of discovering relevant *participants* and their *endpoints*. In the appendix, a diagram showing such structure is included. Do refer to the complete specification [57] for more detail.

heartbeat. Whenever a corresponding *reader* receives the message, it then logs the change in its *HistoryCache* and then responds with an *AckNack* message indicating that the change has been placed in the *HistoryCache*[8]. Thus, retrieving previously received messages is an operation that can be done independently by *readers* from the data contained in their *HistoryCache*, which corresponds with the data kept by *writers*.

II. Methodology and Approach

A. Research Questions

In order to determine how the problem explained in subsection I-A can be solved, the following main research question has been defined:

How can the cluster state store in container orchestration technologies meet the resilience and reliability needs of mission-critical systems when limited by having only two availability zones?

To aid in answering the main research question, several sub-questions have been abstracted from it:

RQ1. What are the four most important properties that must be considered when evaluating the cluster state store of a container orchestrator for a mission-critical system with only 2 availability zones?

RQ2. Which method is suitable in order to quantify latency, throughput, consistency and partition tolerance for a cluster state store of a container orchestrator system within the context of a mission-critical system with only 2 AZs?

[8]Refer to Figure 2 for a simplified sequence diagram of a DDS write.

RQ3. How does a DDS-based cluster state store compare to an SQL cluster state store and to an *etcd* cluster state store in terms of latency, throughput, consistency and partition tolerance?

B. Hypothesis

From the research questions, the following hypothesis has been formulated:

A DDS-based cluster state store for container orchestration is a viable alternative to an etcd and to an SQL cluster state store in terms of latency, throughput, consistency and partition tolerance.

C. Contributions

This work aims to shed some light on the use of cloud-native technologies, namely container orchestration, in constrained environments. The first contribution of this work is the definition of metrics that enable the comparison of a cluster state store for container orchestrators when applied in mission-critical systems in a limited AZs domains. Such comparison benchmark can be used in further research on the field. Secondly, a prototype system has been built[9] in order to allow the use of a DDS-based DB as cluster state store for Kubernetes, which has finally been evaluated according to the benchmark. This system can be used as a first prototype to kickstart the study on pub/sub standards for container orchestration purposes, an area in which research is currently scarce.

D. Methodology

As per [58] and considering that one of the necessary parts of this work involves the creation of an artefact, a design science methodology is considered to be the

[9]For more details refer to subsection II-G.

most adequate choice. Many design science methodologies exist, ranging from detailed ones, such as [59] or [60], as well as more concise options, like [61]. The methodology defined in [62], is oriented towards industry-academia collaborations, which is the case of this research. Moreover, it originates as the synthesis of previous frameworks, all while being less restrictive and extensive than other options.

Following the research process proposed by Offermann *et al.* [62], the first part of the project focuses on literature research and interviews with experts within Thales Nederland in order to identify the most relevant properties to consider when evaluating cluster state stores for mission-critical systems, thus focusing on answering RQ1. After this step, a method suitable to measure the properties considered is decided upon, thus answering RQ2. Afterwards, an artefact is created to allow the usage of a DDS-based database as cluster state store for K8s, during the solution design phase. Finally, once the necessary cycles have been completed, during the evaluation phase, the solutions are compared according to the previously defined benchmark, giving answer to RQ3 and to the main research question.

E. Design Decisions

1) Kubernetes Distribution: Even though vanilla K8s is the state of the art in container orchestration, it has been modified and several distributions exist: as of April 2024, CNCF certifies over 90 K8s distributions, several of which are open source [63]. One of those is K3s, a lightweight distribution which allows the usage of relational databases (SQLite, MySQL, PostgreSQL and MariaDB) as cluster state data store, apart from *etcd*. It requires less storage when compared to vanilla K8s as studied in [64], thanks to the removal of certain components which are also not necessary in this context. Note that this distribution is not only highly valued by the cloud-native community, but it is also comparable to vanilla Kubernetes in terms of performance and resource consumption, as shown in [65], [66], [67] and [68]. Furthermore, K3s implements a standalone shim named Kine. It emulates the behaviour of *etcd*, making the chosen relational database implement intrinsic *etcd* mechanics and translating K8s' *etcd* calls into SQL queries. As Kine is open source as well and can be adapted, K3s can be used to analyse both DDS and SQL, thus improving the comparability of measurements. K3s has been chosen as the Kubernetes distribution used in this project.

2) DDS Implementation: Different implementations of the DDS standard exist, such as RTI's Connext, eProsima's Fast-DDS and ZettaScale's[10] OpenSplice and CycloneDDS, with the latter backed by the Eclipse Foundation. Although literature on them is scarce, there is research that compares the implementations, mostly related to the use of the DDS standard as the de facto communication layer in ROS2 (which officially supports CycloneDDS, Fast-DDS, RTI Connext and the commercial GurumDDS [69]). In [70], three distributions are compared in the context of ROS2, namely Connext, CycloneDDS and Fast-DDS. Even though the focus of the paper is on security measures, latency is also measured, with the results showing that CycloneDDS performs slightly better than the other two distributions. However, when security options are enabled, the difference between CycloneDDS and Fast-DDS narrows and no recommendation can be made. The same three distributions are also compared within ROS2 in [71], with results showing that CycloneDDS has a lower latency than Connext and Fast-DDS, and that Connext

[10]Part of ADLINK, before it became a spin out.

performs sub-optimally when more nodes are added. Results in [72] also coincide on the fact that the latency of CycloneDDS is lesser to that of Fast-DDS, but also under the context of ROS2.

If the licenses of the different distributions are taken into account, Connext only provides commercial and research licenses; however, CycloneDDS is subject to Eclipse Public License v2.0 and Fast-DDS to Apache 2 [69], which are permissive free software licenses. Everything considered, the possible choices are narrowed down to CycloneDDS and Fast-DDS. The following points have been taken into account to make the final decision:

- In terms of maturity, Fast-DDS has an advantage; even more so when considering its curated documentation, more extensive and detailed than CycloneDDS'. In spite of this, Fast-DDS lacks *QueryConditions*, a critical functionality of the standard indispensable for the implementation of a database. Without this, it is not possible to efficiently search through the messages in the system for specific characteristics, such as a particular name or an id greater than a certain number, simple queries extensively used by Kine. Furthermore, this vital functionality is not included in Fast-DDS' roadmap [73]. On the other hand, *QueryConditions* do exist in CycloneDDS[11], as well as the other required functionalities.
- The current system used by Thales Nederland uses OpenSplice. CycloneDDS is the direct successor of OpenSplice, which makes it a more meaningful choice. There is extensive knowledge in Thales Nederland about OpenSplice, which can be easily extrapolated into CycloneDDS, as opposed to Fast-DDS.
- If the communities behind both implementations are examined, Fast-DDS' is larger than CycloneDDS'[12]. However, CycloneDDS offers more direct communication channels apart from GitHub, such as Discord, thus providing easier approach to the developers, which greatly improves support enquiries.

All the previous items evaluated, with special attention to the first point, and coupled with the slightly better performance of CycloneDDS, have laid the foundation for the selection of CycloneDDS over Fast-DDS. Notwithstanding, it is crucial to understand that all of the products evaluated are still in development and not all features specified by the standard are yet fully implemented.

F. Experimental Setup

The test setup consists on 2 Virtual Machines (VM) running on VMware Player. The host runs Fedora 37 Workstation and has the following specifications:

- CPU: Intel Core i5-8365U[13].
- Memory: 16 GB DDR4.
- Disk: Samsung Electronics NVMe SSD Controller SM981/PM981/PM983 with 476 GB.

The specifications of the VMs, which run Ubuntu Server 22.04, are:

- Memory: 6044 MB.
- Disk: 50 GB.
- Processors: 4.

The versions of the software used are listed below:

- CycloneDDS 0.10.3 (Lettres Dansantes).
- K3s version v1.29.0, with Metric Server and Traefik deactivated.

[11]Although they are not yet directly wrapped by the C++ API and without support for SQL-like syntax, as of June 2023.

[12]As shown by the amount of GitHub contributors and stars [74] [75]

[13]Full CPU specifications available at [76].

- Kine version 0.10.3.
- PostgreSQL version 14.2 (latest version supported by K3s [77]).
- Nginx 1.14.2, as pod application (following the examples from K8s documentation [78]).

G. *Prototype*

Given that no DDS-based store is available to be used as cluster state store for any Kubernetes distribution, it has been necessary to develop a prototype in order to conduct the measurements. Because of the similarity between SQL and a DDS-based DB[14], K3s' Kine's source code (available in [79]) has been modified so that *etcd* calls are translated into DDS queries. This prototype has enabled the use of DDS as cluster state store for K8s and has made it possible to obtain the results presented in this paper.

As Kine is written in Go and the CycloneDDS-based DB in C++, it is not possible to directly request the Queries to the DB. To allow that functionality, gRPC has been used to establish a communication channel between Kine and the Database. With the DB acting as a server, Kine can request all necessary queries to DDS. In Figure 1, the prototype system with 2 AZs is shown, which represents the scenario with limited AZs.

III. Requirements, Assumptions and Metrics

To determine the content of the following subsections, interviews have been conducted with key personnel within Thales Nederland, namely system architects and cloud-native experts. Consequently, the information collected mainly refers to the specific naval context with which Thales deals. Nevertheless, most aspects have been generalised in order to refer to mission-critical systems in domains limited by having only 2 AZs.

[14]In fact, the DDS specification lists a subset of SQL-syntax DDS queries should be able to use.

A. *Requirements*

R1. The system has to be able to operate on only 2 AZs.
R2. The system has to be capable of continuing operation upon failure of one of the two AZs.
R3. The system only modifies the Kubernetes' or Kubernetes distribution's cluster state store.
R4. The system has to supports Kubernetes clusters containing up to 100 pods.
R5. The cluster state store replicas have to achieve consistency under 0.5 seconds.

B. *Assumptions*

Considering the specific environment faced by Thales, a series of assumptions have been made. Those are used in order to frame the tests and measurements conducted on the system, and are the following:

A1. No network partitions can happen[15].
A2. The system works in a completely secure network and failures caused by network attacks cannot happen.
A3. The system performance is not affected by the resources consumed by the applications running on the containers.
A4. Only one application runs per pod and all applications running on the pods are the same.
A5. The system never runs out of resources.
A6. Loss of messages in the gRPC communication are not considered system failures.

[15]This assumption stems from the example use case of Thales Nederland, where networks are susceptible to physical attacks and strong redundancy measures are used to counter failures.

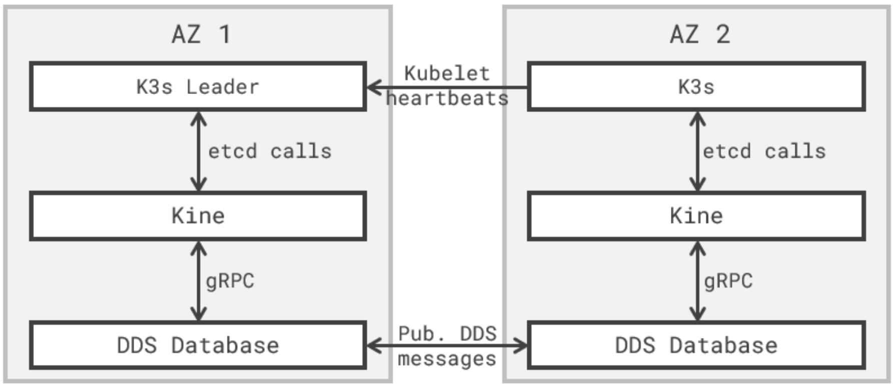

Figure 1: Overview of the Prototype System with 2 AZs.

C. Failures

Given that the system is distributed over different AZs, it is possible that failures exist that make communication impossible. In Figure 2, the simplified sequence diagram of a DDS reliable write is shown, taking into account the particularities of the RTPS protocol. With said diagram as reference, the following main failures are identified:

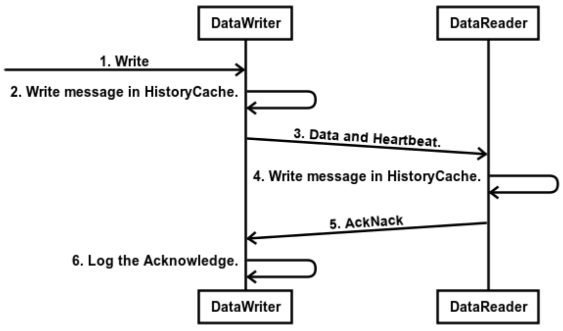

Figure 2: Simplified sequence diagram of a DDS write [57][80].

F1. In step 3, the data and heartbeat fails to reach the DataReader and the message is not received by the subscriber.

F2. In step 4, the DataReader is forced to drop the message instead of logging it into its HistoryCache (e.g. due to a lack of resources).

F3. In step 5, the AckNack[16] fails to reach the DataWriter.

F4. The node is rendered completely inoperative for any reason at any step.

Note that both failure F1 and failure F3, when caused by network infrastructure issues, are not considered system failures due to assumption A1. Furthermore, in the event of a failure F1, messages would be automatically re-sent when the failure is detected, as specified by the DDS RELIABLE QoS. Similarly, failure F2 can be prevented by an adequate configuration of QoS. Finally, failure F4 depicts the scenario explained in subsection I-A, when one of the two nodes dies. It is this failure that establishes the foundation for this research.

Finally, Figure 3 shows the simplified sequence diagram of the execution of a query considering the whole prototype system (detailed in subsection II-G). As per assumption A6, note that a failure in the gRPC service is not considered a system failure. Therefore, the only failures considered in the whole system have been listed before (failures F1, F2, F3 and F4).

[16]An AckNack is an acknowledge message which can signify either a positive or a negative acknowledge.

D. Metrics

Metrics have been selected through the previously mentioned interviews with Thales experts. Additionally, existing literature has been considered. First of all, during interviews, the CAP theorem has been used as reference to determine which of the three characteristics are more relevant, while rejecting the remaining one. Due to the fact that losing one of the AZs is one of the possible outcomes of combat, considering the specific Thales case, partition tolerance is a the most relevant property to consider. Furthermore, the K8s Scheduler uses the data available in the cluster state store in order to distribute workloads among nodes, so inconsistencies within the store could lead to unfavourable situations (e.g. deploying applications on resource-starved nodes). This leads to make Consistency a critical property. Other metrics mentioned multiple times during the interviews have been latency and throughput.

On [81], several methods of comparing databases are explored. Out of 9 different methods, 7 consider latency as a relevant metric, while 3 consider the throughput. Looking into other papers consulted during the literature research ([44], [41], [70] and [30], for instance), latency and throughput are used as common metrics for comparison. Because of this, and their mention during interviews, both metrics are considered relevant. Other metrics are mentioned multiple times in literature, namely scalability and availability. However, none are relevant in this case, as only 2 AZs exist and partition tolerance and consistency take priority over availability.

Therefore, answering the first research question, the metrics taken into account are the following:

M1. *Latency*: latency can be defined as the time between making a request and beginning to see a result [82]. Considering the whole prototype system, latency is the time elapsed from step 1 until step 7 has finalised, following Figure 3; additionally, writes also consider latency from step 1 to 4 (where 4 is the entirety of steps shown in Figure 2), thus measuring the latency since the start of a publish query until the database logs the message.

M2. *Throughput*: throughput is defined as the items processed per time unit and is inversely related to the latency [82].

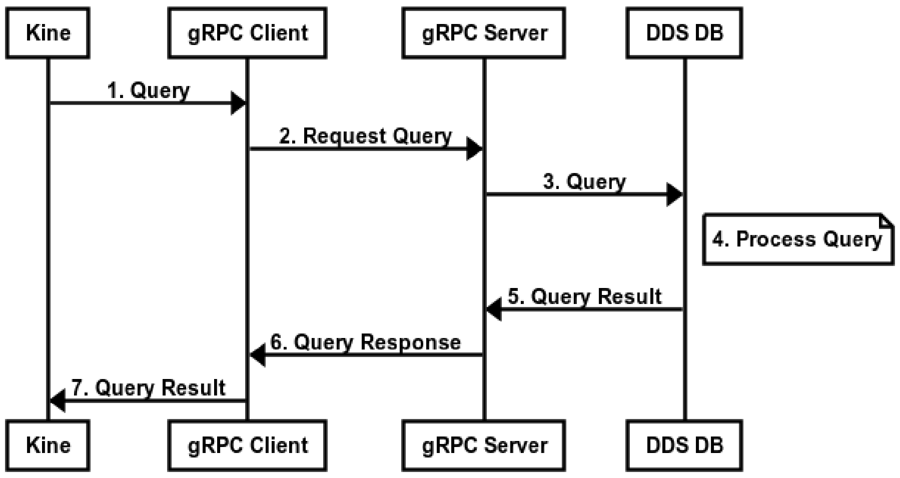

Figure 3: Simplified execution of a query in the whole prototype system.

In this case, throughput is defined as the number of queries requested per second. Considering the communication as depicted in Figure 3, one single query represents the execution of the entire process, from step 1 to step 7. Throughput is differentiated according to the type of request: read requests and write requests (i.e. publications).

M3. *Consistency*: given that DDS is an eventually consistent system, this metric focuses on determining whether a message published is logged by all subscribers in a time lesser than or equal to 0.5 seconds. If the message is logged after 0.5s have elapsed, it is considered an inconsistency. The time is determined by requirement R5, with 0.5s considered as a safe margin during which inconsistencies in the cluster state store should not affect the K8s scheduler. The result of this measurement is binary: the system is either consistent or not.

M4. *Partition Tolerance*: given that the system runs only on 2 AZs, this metric considers whether the system operation can continue after the total failure of one of the two AZs (i.e. when a failure F4 occurs), as specified by requirement R2. Therefore, the system is considered partition tolerant if entities running on the defunct AZ (e.g. pods and deployments) are restarted in the healthy AZ and if new entities can be added to the resulting 1-node cluster. This metric is binary, the system is partition tolerant or it is not.

IV. System Evaluation

In the previous section the metrics were defined. As per the second research question, it is necessary to define the measurement method. This is included in this section, as well as the results gathered, necessary to answer the third research question.

A. Preliminary Considerations

Before moving on to the results of the measurements, there are a few key points that need to be mentioned:

1) Given that the two nodes run as guests on the same host, as explained in subsection II-F, time needs to be synchronised among the VMs. The underlying Time Stamp Counter (TSC) of the host's CPU[17] has been used for this purpose. Even though reading the real TSC of the host can introduce overhead caused by trapping mechanisms used by the VM manager [83], this method has been considered the most accurate to measure time consistently in the experimental setup. Another alternative considered was the use of the Network Time Protocol (NTP) with reduced synchronisation time and higher adjustment frequency. However, given that VMs are susceptible to time drifts, this NTP method was dismissed.
2) In the DDS DB, two different topics have been used. The Main topic, which consists of the data equivalent to *etcd* content (in a KVS, the key and all its information, value included), and the

[17]Pseudoperformance counters have been employed. In practise, the *rdpmc 0x10000* instruction has been employed, as specified in [83]. For DDS, a plain inline assembly instruction has been used due to the database being written in C++. In the case of Kine, which is written in Go, extra assembly instructions have been needed in order to move the result into the stack so that it can be retrieved and used.

MaxId topic, which has been specifically used to track the maximum id in the DB (i.e. the last key created). Whenever possible, measurements have been performed separately for each topic. While the size of the MaxId topic is fixed, that of the Main topic is variable[18].

3) Read queries usually translate into several reads on the DDS DB, spanning the two topics. Therefore, a system read or write does not necessarily translate into one single DDS DB read or write[19].
4) Due to the high multithreading introduced by gRPC, measurements are subjected to overhead caused by multithreading-safe mechanisms. In essence, that means that there are critical sections delimited by locks which may block certain queries for a period of time, which makes measurements susceptible to the state of the system.
5) Finally, the frequency of the processor needs to be used in order to offer a TSC to time conversion (necessary in subsubsection IV-C2). Given that the frequency of modern processors is not fixed and can greatly vary according to the workload, it has been calculated on the VMs and established to be 2.24GHz[20].

B. Limitations

The results presented in the next subsection are limited by the following factors:

1) The first limitation to consider derives from assumption A1: the prototype system cannot tolerate network partitions.
2) Following the last point, and according to assumption A2, no network attacks can occur. Because of this, the system does not implement any mechanisms that would make it resistant to such attacks.
3) As aforementioned, the available DDS distributions are not yet completely mature. This has caused the necessity of using certain workarounds to ensure the persistency of data, as well as precluding the use of SQL-like syntax for queries, for instance. Further development of the DDS distribution would improve the overall quality of the prototype system.
4) The use of Kine as a base shim, instead of developing a shim specifically tailored for the system, has limited the approach to the implementation of the cluster state store to a particular way of translating calls to *etcd*.
5) Finally, the limitations of the experimental setup itself need to be taken into account. Using VMs in order to simulate a 2 AZs environment is not as realistic as using a physical environment with two separate nodes. The use of a real scenario would not only allow for a more accurate data collection, but would also introduce other factors (e.g. network latency) which have been disregarded in this work.

C. Latency

Latency has been measured on the 2-node setup shown in Figure 1, during 40 5-minute runs of the system. Note that the initial bootstrapping of the system is not considered for the evaluation of the latency, as it is only performed at the setting up of the system and has no meaningful effect during normal working conditions. Additionally, the throughput during that period is unusually high, as shown in subsection IV-D, and would possibly increase the overall latency of the results.

The results on the measurements are shown in subsubsection IV-C1. In order to compare the results with PostgreSQL and *etcd*,

[18]More details on this are explained in the appendix.
[19]Refer to the appendix for a detailed explanation.
[20]Refer to the appendix for an in-depth explanation.

latency has also been measured for those data stores. These results are presented in subsubsection IV-C2.

1) DDS System Results: Given that Kine communicates with the cluster state store in in its same node, measurements for the query roundtrip (i.e. from steps 1 to 7 in Figure 3, roundtrip latency from now on) only consider the one node. These are shown in Table I, distinguishing reads from writes. Additionally, in Table II, the measurements for the latency considering steps 1 to 4 in Figure 3 are shown (subscriber latency from now on). In this case, a distinction on nodes is made. Take into account that, due to the inner workings of Kine, queries need to read and write on the two topics[21].

R/W	Metric	Thousands of ticks
Read	Median	19704
	Mean	21011
	std	7611
Write	Median	3242
	Mean	3654
	std	2163

Table I: Numerical analysis of the read and write roundtrip latency results for the whole system.

AZ	Metric	Thousands of ticks
Leader	Median	1889
	Mean	2219
	std	1768
Other	Median	2364
	Mean	2824
	std	6386

Table II: Numerical analysis of the write subscriber latency results for the whole system.

In Figure 4, results for the roundtrip latency are shown graphically, as a distribution and as a boxplot. In Figure 5, the same is depicted for the subscriber latency.

[21]Refer to Preliminary Considerations for an introduction on the topics. The multiple access is better clarified in subsection IV-D, when exploring the throughput.

The results have been obtained by logging the TSC timestamp in Kine just before issuing each query, and again immediately after receiving a response. In the case of the subscriber latency for writes, the second timestamp has been recorded in the DDS DB upon reception. The difference of ticks has resulted in the latency.

2) DDS compared with PostgreSQL and etcd: To put the results presented previously in perspective, measurements have also been taken on a PostgreSQL-based K3s and on an *etcd*-based cluster. Only results considering the roundtrip have been examined, as measuring the latency within those DBs is not part of the scope of this research, and literature already exists on the topic.

PostgreSQL measurements have followed the same method for the same number of runs as the DDS-based system, adding TSC timestamps that log whenever a query is launched until the response is returned. When using *etcd* as cluster state store, K3s does not use Kine. Therefore, a different method has been used in order to quantify *etcd*'s read and write latency. Although *etcd* provides a metrics service, it does not log said metrics. For this reason, it has been decided to use the *etcd* benchmark tool [84], performing a total of 100000 measurements 40 times and calculating their average[22]. In order to simulate the use of an auxiliary node as explained in subsection I-F, 3 nodes have been used. Details on the PostgreSQL and *etcd* experiment setups can be found in the appendix.

The roundtrip latency results for PostgreSQL an *etcd* are shown in Table III, together with the DDS results shown in the previous section. Finally, the calculated time in microseconds is included for DDS and PostgreSQL, calculated using the aforementioned frequency of 2.24GHz.

[22]Configuration used for the measurements is explained in the appendix.

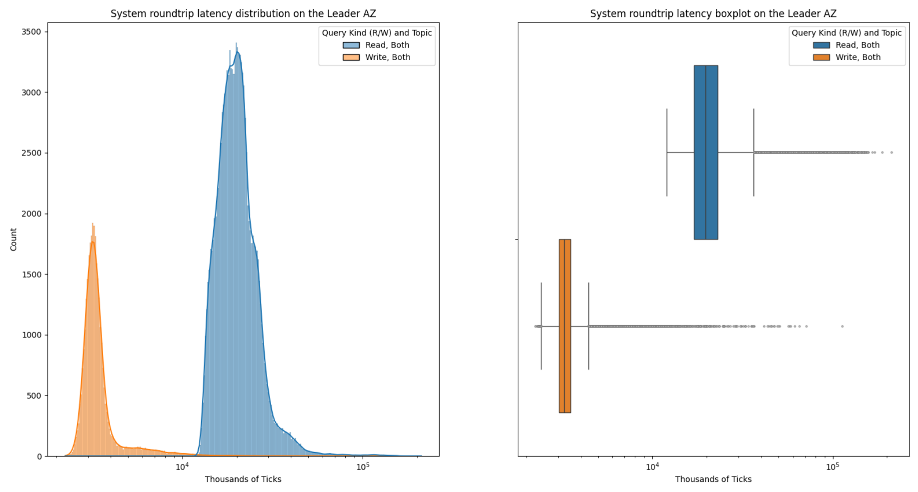

Figure 4: Distribution and boxplot of the roundtrip latency for system reads and writes in the publishing node, with ticks in logarithmic scale. Note the higher latency for reads as compared to writes, as well as the proportionally higher deviation for writes.

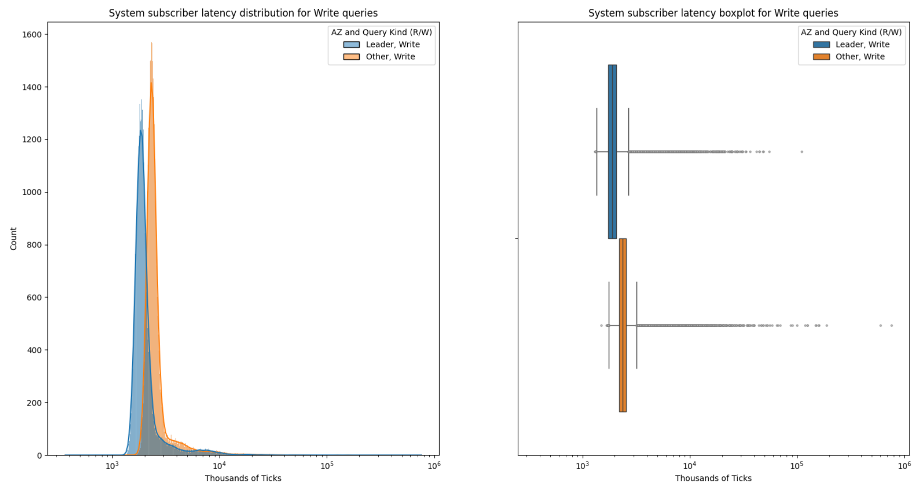

Figure 5: Distribution and boxplot of the system write queries, considering the subscriber latency, with ticks in logarithmic scale. Note the increase of latency and deviation added by the network communication on the other node.

Rear/Write	Metric	PostgreSQL		DDS		*etcd*
		TSC Ticks	µs	TSC Ticks	µs	µs
Read	Mean	1,843,000	820	21,011,000	9380	2810
	std	778,000	350	7,611,000	3400	1610
Write	Mean	1,627,000	730	3,654,000	1630	5651
	std	459,000	200	2,163,000	970	3034

Table III: Numerical analysis of the read and write roundtrip latency results for PostgreSQL and *etcd*, contrasted with the DDS results. Compare the low latency shown by the PostgreSQL system, evidence of using an SQL-tailored shim. Although slower in reads than the other two cluster state stores, DDS outperforms *etcd* on writes, while showing less difference on reads than with the SQL counterpart.

D. *Throughput*

Measurements for the throughput consider the whole system, given that the DDS DB does not make any requests by itself; in fact, all requests are generated by Kine. Given that a system using PostgreSQL would also have the same throughput, as it also uses Kine, and *etcd* implements all the mechanisms emulated by Kine internally, what has been analysed for this metric is how the throughput changes according to the number of nodes and pods. The conditions of the measurements have been decided according to:

- The number of nodes contemplated follows the problem statement. A maximum of 2 nodes and a minimum of 1 is used.
- To observe how the metric changes when introducing more pods, deployments with 1, 5 and 10 pods have been launched in different runs, experiment that stems from requirement R4.

Measurements of the throughput have been performed taking into account the kind of query (i.e. whether it reads or writes) and the topic, as well as the number of nodes and pods. In Figure 6 and Figure 7, the number of queries of each kind against time is plotted, starting with the bootstrapping of the cluster. Furthermore, considering that queries within the DDS DB usually execute several reads and writes, the DDS DB throughput has been considered too[23].

To get the throughput measurements, a TSC timestamp has been logged every time a query has been launched. Note that the results represent the average throughput of reads and writes per 1 billion (10^9) TSC ticks. The DDS results have been obtained by mapping the internal DDS reads and writes per specific Kine query.

As distinct phases can be observed in Figures 6 and 7, the throughput analysis focuses on the 3 parts separately: 1) the stable phase, 2) the addition of a new node to the cluster and 3) when new pods are launched. The reason behind this choice is: for 1), under a normal production environment the state of the cluster would stay stable; for 2), the main failure considered is failure F4 (the disappearance of one of the two AZs), therefore the throughput of adding an AZ to the cluster is measured; and finally, for 3), if new applications need to be added to the system, new pods have to be created. The result of the measurements is shown in Table IV. Even though the count of queries plotted in the figures considers both AZs, results in the table are only measured in the Leader AZ, to consider the worst case scenario, given its higher throughput.

[23]Refer to the appendix for a detailed explanation on the DDS throughput.

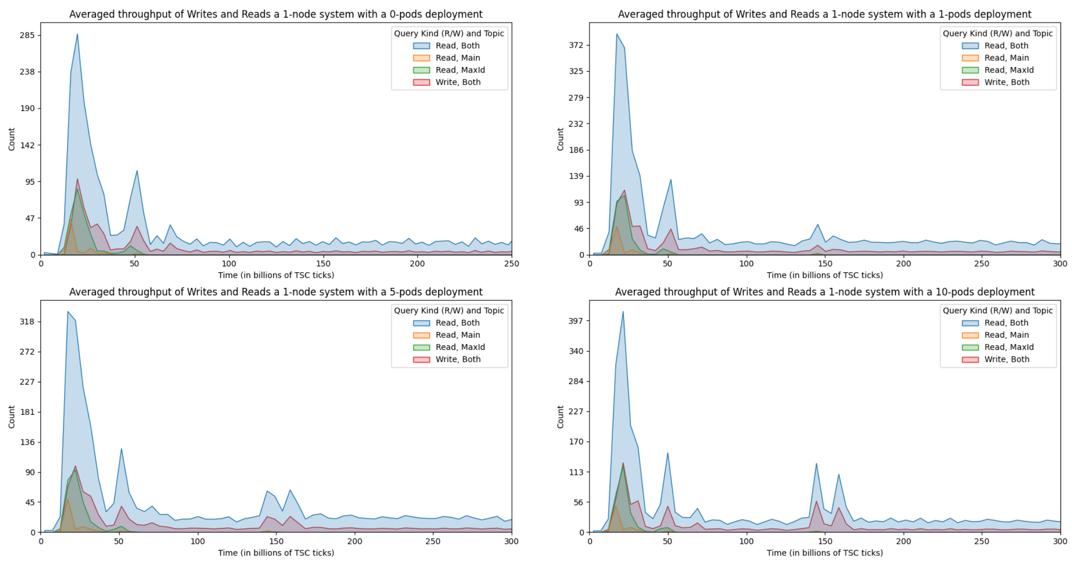

Figure 6: Average number of queries in time of a 1-node system. Notice the initial peak in throughput caused by the bootstrapping of the system, followed by a stable phase only disrupted by the creation of pods. Additionally, compare the higher amount of reads as opposed to writes.

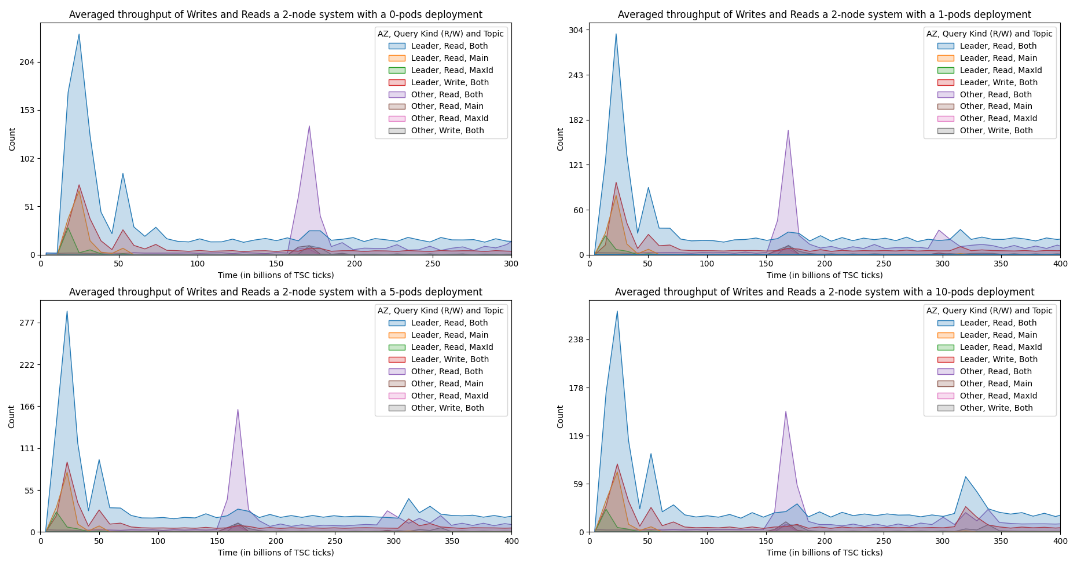

Figure 7: Average number of queries in time of a 2-node system. Although similar in base to Figure 6, especially during the bootstrapping, note how adding a node increases throughput mainly on the node added, and how adding pots balances the thoughput between the 2 nodes in the cluster.

Nodes	Pods	R/W	Topic	Average queries			DDS[a]		
				Stable	+ AZ	+ pods	Stable	+ AZ	+ pods
1 node	0 pods	Read	Both	4.68	-	-	-	-	-
		Read	Main	-	-	-	8.55	-	-
		Read	MaxId	-	-	-	5.50	-	-
		Write	Both	1.21	-	-	1.21	-	-
	1 pod	Read	Both	4.57	-	6.84	-	-	-
		Read	Main	-	-	0.09	8.55	-	12.95
		Read	MaxId	-	-	0.13	5.49	-	8.61
		Write	Both	1.21	-	2.05	1.21	-	2.05
	5 pods	Read	Both	4.55	-	11.19	-	-	-
		Read	Main	-	-	0.09	8.52	-	21.62
		Read	MaxId	-	-	0.13	5.47	-	15.02
		Write	Both	1.21	-	4.13	1.21	-	4.13
	10 pods	Read	Both	4.56	-	16.25	-	-	-
		Read	Main	-	-	0.09	8.54	-	31.81
		Read	MaxId	-	-	0.13	5.48	-	22.65
		Write	Both	1.22	-	6.66	1.22	-	6.66
2 nodes	0 pods	Read	Both	4.72	5.38	-	-	-	-
		Read	Main	-	-	-	8.77	17.66	-
		Read	MaxId	-	-	-	5.61	11.28	-
		Write	Both	1.22	1.42	-	1.22	2.45	-
	1 pod	Read	Both	4.62	5.39	6.24	-	-	-
		Read	Main	-	-	0.08	8.61	17.38	12.05
		Read	MaxId	-	-	0.13	5.51	11.13	7.95
		Write	Both	1.21	1.42	1.84	1.21	2.43	1.84
	5 pods	Read	Both	4.62	5.38	8.79	-	-	-
		Read	Main	-	-	0.08	8.60	17.33	17.15
		Read	MaxId	-	-	0.13	5.50	11.10	11.87
		Write	Both	1.21	1.43	3.21	1.21	2.43	3.21
	10 pods	Read	Both	4.63	5.37	12.2	-	-	-
		Read	Main	-	-	0.08	8.61	17.39	23.92
		Read	MaxId	-	-	0.13	5.51	11.14	17.08
		Write	Both	1.21	1.42	5.03	1.21	2.44	5.03

[a] Throughput of writes in DDS is the same for Main and MaxId topics. That is, 1 Main write also implies 1 MaxId write.

Table IV: Throughput of reads and writes per topic per billions of TSC ticks during the Stable phase and when an AZ or pods are added to the cluster. Note the similarity of throughput during the stable phase for the 1 and 2-nodes system, as well as the low increase in the Leader node when another node is added. Additionally, consider the balancing of throughput when new pods are added in the 2-node system, as shown by the higher throughput in the 1-node counterpart.

It is necessary to mention that occasional gRPC message loss has been observed during periods of abnormally high throughput. Such events have not been seen during the measurements presented in this section, but in tests with deployments with more than 50 pods. Due to Assumption A6, these message losses are not considered system failures but they do affect requirement R4, in the sense that launching 100 pods at once has no guaranteed success.

E. Consistency

Consistency is measured using the whole system spanning over two nodes, as shown in Figure 1. Two different periods have been measured: when queries are at its peak (i.e. during the bootstrapping phase) and when the cluster stabilises. Consistency measurements during the moment of maximum throughput span the creation of the cluster in the leader node and the addition of the other node, followed by 2 more minutes afterwards. These measurements run during 3 minutes and 15 seconds, with variations of at most 7 seconds depending on how long the setup takes. Measurements during the stable phase are taken 1 minute after the second node is added to the cluster and span 1 minute.

In order to assert the consistency of the system among nodes, the process is: 1) first, a message is sent from the leader; 2) when the subscriber in the leader AZ receives said message, it logs all the contents of the database with an id lesser or equal to the message; and finally 3) when the subscriber on the other AZ receives the message, it waits 0.45s and annotates all messages in the database up to the message id. Then, the content of the leader database is compared with the other node's. The waiting time has been set at 450ms to take into account the latency. Note that only the Main topic has been considered, as it is the one that constitutes the database.

The results of the worst case scenario (i.e. bootstrapping process), as well as the stable phase, show that the nodes are consistent within the safe bound of 0.5s. Therefore, it can be said that the system is consistent as per the given definition of the metric.

F. Partition Tolerance

Partition tolerance has been measured starting on the 2-node setup, as shown in Figure 1, on which the failure of one node (failure F4) has been simulated, thus ending with only one functional node.

The simulation of the failure has been done by uninstalling K3s altogether, as well as killing the Kine and the DDS DB processes[24]. As that CycloneDDS does not implement the TRANSIENT durability QoS as of January 2024, TRANSIENT LOCAL durability has been used instead. Such QoS causes the invalidation of messages whose DataWriter is not alive, which results in entries from the DB not being resent when the defunct node reconnects. In order to circumvent such behaviour, once the failure of a DB node is detected by the reception of invalid messages, the remaining node is tasked with the republication of all messages issued by the recently defunct node[25]. Once the TRANSIENT durability is implemented, this work-around, which adds additional overhead, will not be necessary.

To determine whether the system is partition tolerant, a simple experiment has been used: with the system running normally with 2 pods deployed (one on each node), the system in one of the nodes is killed; after 6 minutes, the information of the nodes and pods is retrieved. Then, the system in the defunct node is restarted and the cluster state is logged 4 minutes afterwards. To asses both the death of the leader node, as well as the non-leader, 40 independent runs have been used for each situation.

This results of the test show that the system is completely capable of withstanding the death of an entire AZ, with the entities that ran on the defunct node being restarted on the healthy node in every run. Furthermore, entries in the cluster state store are not lost when a node dies and, when it reconnects, they are all resent again, maintaining consistency in the system.

[24]This method has been chosen instead of directly powering off the VM as it makes automatising of the measurements smoother.

[25]In practise, that means messages whose instance state is NOT_ALIVE_NO_WRITERS. More details are provided in the appendix.

V. Discussion

In this section, the results presented in the previous section are evaluated for each metric.

A. Latency

1) DDS System Latency: Looking into the results for the DDS system latency, it can be seen how the latency for reads is considerably higher to that of writes. One of the main reasons for this is the fact that queries requested from kine are designed for relational databases instead. This causes inefficient searches in the store. Furthermore, this needs to be coupled with the need of using critical zones in order to make the system multi-threading safe. In the case of writes, comparing the roundtrip with the subscriber latency show how the extra communication layer introduced by gRPC impacts the latency negatively.

2) DDS Compared with PostgreSQL and etcd: By comparing the results achieved on PostgreSQL with those of DDS, is evident that Kine is a shim specifically designed to translate *etcd* queries into SQL. DDS shows a clear increase on read latency, which is more than 10 times higher than Postgres'. Whereas PostgreSQL exploits efficiency mechanisms available to relational databases, the DDS system simply implements the same queries on a non-SQL architecture. In the case of writes, however, it can be seen how the difference is much lower, with DDS latency only being slightly higher than twice Postgres'. If the subscriber latency is to be considered, that difference is reduced even further. As with the reads, this increased latency is caused again by Kine: where SQL is capable of seamlessly increasing the id of new rows added to the database, DDS needs to do so using another topic that requires retrieval of the value before applying the increment.

For the comparison between PostgreSQL and DDS, even though measurements have only been conducted on the leader AZ in both cases, the SQL system lacks any distribution or synchronisation mechanism, as the DB only resides in the leader. That is not the case for DDS, thus having its load increased. It also needs to be mentioned that, while SQL counts with a dedicated scheduler for queries, the DDS-based system does not, which implements multi-threading safety with much simpler critical zones, thus incrementing latency.

Finally, comparing the DDS latency with *etcd*'s, it can be seen how the communication among nodes of the Raft consensus protocol increases write latency as well as deviation among measurements, with DDS' results being more than 3 times lower than *etcd*'s. *etcd* reads are shown to be faster than DDS but the difference is lesser than that of SQL, slightly above 3 times higher.

Seeing the results, it can be said that the DDS performance is suitable to meet the needs of the cluster state store for container orchestration for mission-critical systems, presenting a low difference with the default *etcd* database and even outperforming it on write queries. However, the creation of a new, DDS-oriented shim is a possibility that needs to be explored, as shown by the difference with PostgreSQL performance, as well as the prototype's higher read latency compared to both other systems. Finally, the use of a distributed PostgreSQL cluster state store appears to be an option worth considering, albeit out of scope in this research.

B. Throughput

If the phases are to be analysed separately, the following is observed:

- During the stable phase, which can be considered as a representation of the system during normal working conditions, the throughput is lower than in any other phase. It also shows little variation with the addition of nodes and no difference is observed by the addition of pods. Therefore, it can meet the need of deploying bigger clusters suffering a minimal impact on the overall performance of the system.

- The addition of an extra AZ does not entail a significant increase in write throughput, and it is minimal for reads, on the leader node. As depicted in Figure 7, the increase of throughput is most significant on the node that is added. In the event of the failure of an AZ, the consequences of adding it again would be low on the healthy node, on which the system depends completely until load-balancing mechanisms come into place. This is due to the fact that the increase on requests on the leader during the addition of the AZ compared to the stable phase is lesser than 1 per million TSC ticks for reads and roughly 0.2 for writes, as shown in Table IV.
- Adding multiple pods at the same time, causes a considerable increase in the system's throughput, for reads as well as for writes. Although, when the cluster consists of 2 AZs, the throughput is balanced among the nodes. Due to the use of gRPC, launching a high amount of pods causes high spikes in throughput, which may cause missing gRPC deadlines. Although such occurrences are non-existent with the number of pods applied in the measurements, applying bigger deployments at once should be done with caution, as the higher throughput may cause said losses.

From the results shown in subsection IV-D, it can be seen that the system is read intensive in every of the three phases during which the throughput has been evaluated. However, looking at DDS's latency in Table I, which is considerably higher for reads than for writes, an approach less dependent on reads would be more suitable for a DDS-based cluster state store. Nevertheless, the DDS cluster state store has shown to be capable of withstanding the higher read throughput.

C. Consistency

First of all, note that comparison with PostgreSQL and *etcd* for this metric is not meaningful, due to the different approach to consistency. PostgreSQL and *etcd* enforce strong consistency, be it by using the ACID properties or by requiring quorum to accept any write in the database, while DDS is eventually consistent, thus allowing for a period in which inconsistencies can occur.

Notwithstanding, the results show that the database is consistent, as per the given definition of the metric. This holds during the initial bootstrapping process, when the throughput is at is highest, as well as during the stable phase, which represents normal work conditions. To sum up, it can be said that a DDS-based cluster state store is capable of satisfying the consistency needs of mission-critical systems when limited to 2 AZs.

D. Partition Tolerance

Partition tolerance is arguably the most relevant property the proposed system tries to satisfy. As explained in subsection I-A, *etcd* cannot withstand the failure of 1 node in 2-node systems. In the case of SQL, it should be theoretically possible to achieve partition tolerance by using a distributed SQL mechanism; nonetheless, this is out of scope of this paper and would require further research.

Focusing on the results achieved by the system, it has been seen that it is capable of continuing normal operation after the death of an entire AZ, as well as detecting the failure and restarting the entities that were running on the defunct node. However, as pointed out, the DDS database cannot achieve partition tolerance by itself due to the lack of the TRANSIENT durability QoS. Nonetheless, with the implemented resending mechanism, the desired behaviour has been replicated and it can be said that a DDS-cluster state store meets the criteria to be considered partition tolerant.

VI. Conclusions and Future Work

The primary objective of this thesis has been to determine how the cluster state store in container orchestrators can meet the needs of mission-critical systems when they are limited to 2 availability zones. It has been shown that a pub/sub cluster state store, using the DDS standard in specific, is capable of sustaining the failure of 1 AZ while still allowing changes on the cluster state, as opposed to the default *etcd* store. Furthermore, it has also been shown that such DDS-based system achieves the required consistency to ensure the Kubernetes scheduler is not affected by the lack of strong consistency. Additionally, analysis of the throughput of the prototype indicates that the system could support the required number of pods[26] even when only 1 AZ exists, albeit with limitations to the number of deployments launched at once. Launching a high number of pods needs to be managed to limit punctual throughput spikes that can cause gRPC package loss. Finally, regarding the latency, the prototype system has shown worse read and write performance than that of an SQL-based system, but better write performance than an *etcd* one. Despite of this, the latency is still acceptable for the purpose of the system.

While K8s and DDS are a suitable solution for the naval domain, taken from Thales Nederland's example, mission-critical systems have specific requirements tied to their own domains and may require a more detailed exploration of other options. K8s is a wide purpose container orchestrator and, as such, it includes many functionalities that are possibly not required in most mission-critical systems and K8s' performance may not be adequate[27]. The DDS-based cluster state store, on the other hand, has shown favourable results and studying its behaviour under other container orchestrators could prove to be an interesting avenue of research. Apart from this, the following items have been considered relevant for future research:

- As stated in subsection IV-B, the prototype system has been based on a shim designed for an SQL system. Even though DDS is capable to execute SQL-like queries and would eventually be able to use SQL-like syntax with that objective, it has been shown that its performance has been hampered due to Kine. Therefore, a creation of a DDS-specific shim would better explore the full potential of the solution; it could exploit the advantages of DDS' writes while attenuating the impact of reads, which are more abundant despite their higher latency. Moreover, particularities of the DDS standard could be implemented (e.g. easiness of data overwriting or the possibility to use multiple topics as one would indexes in SQL).
- In line with the previous point, the creation of a full system that circumvented the need for additional communication through gRPC would give more meaningful measurements, unhindered by an unnecessary layer.
- Running the experiments in a realistic environment (i.e. in a truly distributed system, without simulating AZs with

[26]See the requirements.

[27]For example, if a system requires restarting a container under 1s, K8s would not be the most suitable solution.

VMs on a same machine) would result in better insight to the capabilities of the solution, even though it would require a revised consistent time measurement.

- Additionally, this work has mainly focused on the exploration of pub/sub systems, while other kinds of distributed architectures could be suitable for the same purpose. Research on this path could result in beneficial findings. Finally, as the results of PostgreSQL have shown, it would be relevant to explore the possibility of using it as cluster state store, with the pertinent distribution mechanisms.

Acknowledgements

I would like to sincerely thank my supervisor at Thales, Max Riesewijk, for his guidance, help and understanding throughout the thesis, as well as to my University of Twente supervisors: Peter Bosch, for his most helpful insight and invaluable debugging tips, and Paul Havinga, for his feedback and advise, which proved key to kickstart this project.

Finally, I would like to express my most heartfelt gratitude to my family, without whom I would have never been able to do this; to Marta, for her patience and heartwarming support in the most difficult of times; and to my friends, for those much-needed moments of respite.

References

[1] M. Dowson, "The ariane 5 software failure," *SIGSOFT Softw. Eng. Notes*, vol. 22, no. 2, p. 84, Mar. 1997, ISSN: 0163-5948. DOI: 10 . 1145 / 251880 . 251992. [Online]. Available: https://doi.org/10.1145/251880.251992.

[2] E. Nevo, *What Happened to Beresheet*, https:https://davidson.weizmann.ac.il/en/online/sciencepanorama/what-happened-beresheet, Last accessed: March, 2024, Feb. 2020.

[3] Microsoft, *What is Cloud Native?* https://learn.microsoft.com/en-us/dotnet/architecture/cloud-native/definition, Last accessed: February, 2023, 2022.

[4] D. Gannon, R. Barga, and N. Sundaresan, "Cloud-Native Applications," *IEEE Cloud Computing*, vol. 4, no. 5, pp. 16–21, 2017. DOI: 10.1109/MCC.2017.4250939.

[5] Cloud Native Computing Foundation, *CNCF Cloud Native Interactive Landscape*, https://landscape.cncf.io/, Last accessed: March, 2024, 2024.

[6] etcd authors, *etcd FAQ*, https://etcd.io/docs/v3.5/faq/, Last accessed: February, 2023, 2022.

[7] S. Mullender, Ed., *Distributed Systems (2nd edition)*, English, 2nd. Addison-Wesley, 1993, ISBN: 978-0201624274.

[8] I. Bashir, *Blockchain Consensus: An Introduction to Classical, Blockchain, and Quantum Consensus Protocols*. Aug. 2022, ISBN: 978-1-4842-8178-9. DOI: 10.1007/978-1-4842-8179-6.

[9] R. Shostak, M. Pease, and L. Lamport, "The byzantine generals problem," *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 3, pp. 382–401, 1982.

[10] L. Lamport, "The part-time parliament," *ACM Trans. Comput. Syst.*, vol. 16, no. 2, pp. 133–169, May 1998, ISSN: 0734-2071. DOI: 10.1145/279227.279229. [Online]. Available: https://doi.org/10.1145/279227.279229.

[11] L. Lamport, "Paxos made simple," *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pp. 51–58, Dec. 2001.

[12] L. Lamport, "Fast paxos," *Distributed Computing*, vol. 19, pp. 79–103, Oct. 2006.

[13] L. Lamport, D. Malkhi, and L. Zhou, "Vertical paxos and primary-backup replication," in *Proceedings of the 28th ACM Symposium on Principles of Distributed Computing*, ser. PODC '09, Calgary, AB, Canada: Association for Computing Machinery, 2009, pp. 312–313, ISBN: 9781605583969. DOI: 10.1145/1582716.1582783. [Online]. Available: https://doi.org/10.1145/1582716.1582783.

[14] L. Lamport, "Leaderless byzantine paxos," *Distributed Computing: 25th International Symposium: DISC 2011, David Peleg, editor.*, pp. 141–142, Dec. 2011.

[15] H. Du and D. J. S. Hilaire, "Multi-paxos: An implementation and evaluation," *Department of Computer Science and Engineering, University of Washington, Tech. Rep. UW-CSE-09-09-02*, 2009.

[16] F. P. Junqueira, B. C. Reed, and M. Serafini, "Zab: High-performance broadcast for primary-backup systems," in *2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks (DSN)*, 2011, pp. 245–256. DOI: 10.1109/DSN.2011.5958223.

[17] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIX ATC'14, Philadelphia, PA: USENIX Association, 2014, pp. 305–320, ISBN: 9781931971102.

[18] Raft Authors, *The Raft Consensus Algorithm*, https://raft.github.io/, Last accessed: March, 2023, 2023.

[19] M. Castro, B. Liskov, *et al.*, "Practical Byzantine Fault Tolerance," in *OsDI*, vol. 99, 1999, pp. 173–186.

[20] H. Kopetz and G. Grunsteidl, "Ttp - a time-triggered protocol for fault-tolerant real-time systems," pp. 524–533, 1993. DOI: 10.1109/FTCS.1993.627355.

[21] H. Kopetz and W. Steiner, *Real-Time Systems: Design Principles for Distributed Embedded Applications*, 3rd. Springer Cham, 2022, ISBN: 9783031119910.

[22] H. Kopetz and G. Bauer, "The time-triggered architecture," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 112–126, 2003. DOI: 10.1109/JPROC.2002.805821.

[23] A. Jeffery, H. Howard, and R. Mortier, "Rearchitecting Kubernetes for the Edge," in *Proceedings of the 4th International Workshop on Edge Systems, Analytics and Networking*, ser. EdgeSys '21, Online, United Kingdom: Association for Computing Machinery, 2021, pp. 7–12, ISBN: 9781450382915. DOI: 10.1145/3434770.3459730. [Online]. Available: https://doi.org/10.1145/3434770.3459730.

[24] L. Larsson, W. Tärneberg, C. Klein, E. Elmroth, and M. Kihl, "Impact of etcd Deployment on Kubernetes, Istio, and Application Performance," *CoRR*, vol. abs/2004.00372, 2020. arXiv: 2004.00372. [Online]. Available: https://arxiv.org/abs/2004.00372.

[25] S. Sagkriotis and D. Pezaros, "Scalable Data Plane Caching for Kubernetes," in *2022 18th International Conference on Network and Service Management (CNSM)*, 2022, pp. 345–351. DOI: 10.23919/CNSM55787.2022.9964497.

[26] D. Abadi, “Consistency Tradeoffs in Modern Distributed Database System Design: CAP is Only Part of the Story,” *Computer*, vol. 45, no. 2, pp. 37–42, 2012. DOI: 10.1109/MC.2012.33.

[27] G. DeCandia, D. Hastorun, M. Jampani, *et al.*, “Dynamo: Amazon’s highly available key-value store,” in *ACM Symposium on Operating System Principles*, 2007. [Online]. Available: https://www.amazon.science/publications/dynamo-amazons-highly-available-key-value-store.

[28] Riak authors, *Riak documentation*, https://docs.riak.com/riak/kv/latest/configuring/strong-consistency/index.html, Last accessed: February, 2023, 2023.

[29] C. Wu, J. Faleiro, Y. Lin, and J. Hellerstein, “Anna: A kvs for any scale,” in *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, 2018, pp. 401–412. DOI: 10.1109/ICDE.2018.00044.

[30] R. Taft, I. Sharif, A. Matei, *et al.*, “CockroachDB: The Resilient Geo-Distributed SQL Database,” in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD ’20, Portland, OR, USA: Association for Computing Machinery, 2020, pp. 1493–1509, ISBN: 9781450367356. DOI: 10.1145/3318464.3386134. [Online]. Available: https://doi.org/10.1145/3318464.3386134.

[31] J. C. Corbett, J. Dean, M. Epstein, *et al.*, “Spanner: Google’s Globally Distributed Database,” *ACM Trans. Comput. Syst.*, vol. 31, no. 3, Aug. 2013, ISSN: 0734-2071. DOI: 10.1145/2491245. [Online]. Available: https://doi.org/10.1145/2491245.

[32] D. F. Bacon, N. Bales, N. Bruno, *et al.*, “Spanner: Becoming a sql system,” in *Proceedings of the 2017 ACM International Conference on Management of Data*, ser. SIGMOD ’17, Chicago, Illinois, USA: Association for Computing Machinery, 2017, pp. 331–343, ISBN: 9781450341974. DOI: 10.1145/3035918.3056103. [Online]. Available: https://doi.org/10.1145/3035918.3056103.

[33] X. Li, *Running ETCD as master/slave model with two node "cluster"*, https://groups.google.com/g/etcd-dev/c/XTxwO6mDlkI?pli=1, Last accessed: April, 2023, 2017.

[34] ETCD authors, *Mirror Maker*, https://github.com/etcd-io/etcd/blob/main/etcdctl/doc/mirror_maker.md, Last accessed: April, 2023, 2018.

[35] M. Van Steen and A. S. Tanenbaum, *Distributed systems*. Maarten van Steen, Leiden, The Netherlands, 2017.

[36] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, “The many faces of publish/subscribe,” *ACM Comput. Surv.*, vol. 35, no. 2, pp. 114–131, Jun. 2003, ISSN: 0360-0300. DOI: 10.1145/857076.857078. [Online]. Available: https://doi.org/10.1145/857076.857078.

[37] R. Baldoni and A. Virgillito, “Distributed event routing in publish/subscribe communication systems: A survey,” *DIS, Universita di Roma La Sapienza, Tech. Rep*, vol. 5, 2005.

[38] J. Kreps, N. Narkhede, J. Rao, *et al.*, “Kafka: A distributed messaging system for log processing,” in *Proceedings of the NetDB*, Athens, Greece, vol. 11, 2011, pp. 1–7.

[39] Apache Kafka Authors, *Use Cases*, https://kafka.apache.org/uses, Last accessed: April, 2023, 2023.

[40] S. Sakr and A. Zomaya, *Encyclopedia of Big Data Technologies*, 1st. Springer Publishing Company, Incorporated, 2019, ISBN: 331977526X.

[41] S. T and S. N. K, "A study on Modern Messaging Systems - Kafka, RabbitMQ and NATS Streaming," 2019. arXiv: 1912.03715 [cs.DC].

[42] R. Shree, T. Choudhury, S. C. Gupta, and P. Kumar, "Kafka: The modern platform for data management and analysis in big data domain," in *2017 2nd International Conference on Telecommunication and Networks (TEL-NET)*, 2017, pp. 1–5. DOI: 10.1109/TEL-NET.2017.8343593.

[43] G. van Dongen and D. Van den Poel, "Evaluation of stream processing frameworks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 8, pp. 1845–1858, 2020. DOI: 10.1109/TPDS.2020.2978480.

[44] P. Dobbelaere and K. S. Esmaili, "Kafka versus RabbitMQ: A Comparative Study of Two Industry Reference Publish/Subscribe Implementations: Industry Paper," in *Proceedings of the 11th ACM International Conference on Distributed and Event-Based Systems*, ser. DEBS '17, Barcelona, Spain: Association for Computing Machinery, 2017, pp. 227–238, ISBN: 9781450350655. DOI: 10.1145/3093742.3093908.

[45] VMware, *Persistence Configuration*, https://www.rabbitmq.com/persistence-conf.html, Last accessed: April, 2023, 2023.

[46] J. Yongguo, L. Qiang, Q. Changshuai, S. Jian, and L. Qianqian, "Message-oriented middleware: A review," in *2019 5th International Conference on Big Data Computing and Communications (BIGCOM)*, 2019, pp. 88–97. DOI: 10.1109/BIGCOM.2019.00023.

[47] P. Saint-Andre, K. Smith, and R. Tronçon, *XMPP: the definitive guide.* " O'Reilly Media, Inc.", 2009.

[48] Object Management Group, Objective Interface Systems, Inc., Real-Time Innovations, Inc., and THALES, "Data distribution service for real-time systems version 1.4," *Object Management Group (OMG)*, Mar. 2015. [Online]. Available: https://www.omg.org/spec/DDS/1.4/PDF.

[49] S. Gruener, H. Koziolek, and J. Rückert, "Towards resilient iot messaging: An experience report analyzing mqtt brokers," in *2021 IEEE 18th International Conference on Software Architecture (ICSA)*, 2021, pp. 69–79. DOI: 10.1109/ICSA51549.2021.00015.

[50] Google, *What is Pub/Sub?* https://cloud.google.com/pubsub/docs/overview, Last accessed: April, 2023, 2023.

[51] Amazon, *Amazon Simple Notification Service*, https://en.wikipedia.org/wiki/Amazon_Simple_Notification_Service, Last accessed: April, 2023, 2023.

[52] IBM, *IBM MQ*, https://www.ibm.com/products/mq, Last accessed: April, 2023, 2023.

[53] Ö. Köksal and B. Tekinerdogan, "Obstacles in data distribution service middleware: A systematic review," *Future Generation Computer Systems*, vol. 68, pp. 191–210, 2017, ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2016.09.020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X1630351X.

[54] M. J. Michaud, T. Dean, and S. P. Leblanc, “Attacking omg data distribution service (dds) based real-time mission critical distributed systems,” in *2018 13th International Conference on Malicious and Unwanted Software (MALWARE)*, 2018, pp. 68–77. DOI: 10.1109/MALWARE.2018.8659368.

[55] P. Thulasiraman, Y. K. D. Cheng, and B. Allen, “Evaluation of the Data Distribution Service for a Lossy Autonomous Hybrid System,” in *2022 IEEE International Systems Conference (SysCon)*, 2022, pp. 1–8. DOI: 10.1109/SysCon53536.2022.9773896.

[56] J. Fernandez, B. Allen, P. Thulasiraman, and B. Bingham, “Performance study of the robot operating system 2 with qos and cyber security settings,” in *2020 IEEE International Systems Conference (SysCon)*, 2020, pp. 1–6. DOI: 10.1109/SysCon47679.2020.9275872.

[57] Object Management Group, Real-Time Innovations, Inc., THALES, *et al.*, “The real-time publish-subscribe protocol dds interoperability wire protocol (ddsi-rtps) specification,” version 2.5, *Object Management Group (OMG)*, Apr. 2022. [Online]. Available: https://www.omg.org/spec/DDSI-RTPS/2.5/PDF.

[58] C. Wohlin and P. Runeson, “Guiding the selection of research methodology in industry–academia collaboration in software engineering,” *Information and Software Technology*, vol. 140, p. 106 678, 2021, ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2021.106678. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584921001361.

[59] R. J. Wieringa, *Design science methodology for information systems and software engineering*, English. Netherlands: Springer, 2014, ISBN: 978-3-662-43838-1. DOI: 10.1007/978-3-662-43839-8.

[60] K. Peffers, T. Tuunanen, M. A. Rothenberger, and S. Chatterjee, “A Design Science Research Methodology for Information Systems Research,” *Journal of Management Information Systems*, vol. 24, no. 3, pp. 45–77, 2007. DOI: 10.2753/MIS0742-1222240302. eprint: https://doi.org/10.2753/MIS0742-1222240302. [Online]. Available: https://doi.org/10.2753/MIS0742-1222240302.

[61] A. R. Hevner, “A three cycle view of design science research,” *Scandinavian journal of information systems*, vol. 19, no. 2, p. 4, 2007.

[62] P. Offermann, O. Levina, M. Schönherr, and U. Bub, “Outline of a Design Science Research Process,” in *Proceedings of the 4th International Conference on Design Science Research in Information Systems and Technology*, ser. DESRIST ’09, Philadelphia, Pennsylvania: Association for Computing Machinery, 2009, ISBN: 9781605584089. DOI: 10.1145/1555619.1555629. [Online]. Available: https://doi.org/10.1145/1555619.1555629.

[63] T. L. Foundation, *Certified Kubernetes Software Conformance*, https://www.cncf.io/training/certification/software-conformance/, Last accessed: May, 2024, 2024.

[64] T. Goethals, F. De Turck, and B. Volckaert, “FLEDGE: Kubernetes Compatible Container Orchestration on Low-Resource Edge Devices,” in *Internet of Vehicles. Technologies and Services*

Toward Smart Cities, C.-H. Hsu, S. Kallel, K.-C. Lan, and Z. Zheng, Eds., Cham: Springer International Publishing, 2020, pp. 174–189, ISBN: 978-3-030-38651-1.

[65] M. Fogli, T. Kudla, B. Musters, *et al.*, "Performance evaluation of kubernetes distributions (k8s, k3s, kubeedge) in an adaptive and federated cloud infrastructure for disadvantaged tactical networks," pp. 1–7, 2021. DOI: 10.1109/ICMCIS52405.2021.9486396.

[66] S. Telenyk, O. Sopov, E. Zharikov, and G. Nowakowski, "A comparison of kubernetes and kubernetes-compatible platforms," vol. 1, pp. 313–317, 2021. DOI: 10.1109/IDAACS53288.2021.9660392.

[67] S. Böhm and G. Wirtz, "Profiling lightweight container platforms: Microk8s and k3s in comparison to kubernetes," 2021.

[68] V. Kjorveziroski and S. Filiposka, "Kubernetes distributions for the edge: Serverless performance evaluation," *The Journal of Supercomputing*, vol. 78, no. 11, pp. 13 728–13 755, 2022. DOI: https://doi.org/10.1007/s11227-022-04430-6.

[69] Open Robotics Authors, *About different ROS 2 DDS/RTPS vendors*, https://docs.ros.org/en/humble/Concepts/About-Different-Middleware-Vendors.html, Last accessed: March, 2023, 2023.

[70] M. Aartsen, K. Banga, K. Talko, *et al.*, "Analyzing interoperability and security overhead of ros2 dds middleware," in *2022 30th Mediterranean Conference on Control and Automation (MED)*, 2022, pp. 976–981. DOI: 10.1109/MED54222.2022.9837282.

[71] T. Kronauer, J. Pohlmann, M. Matthe, T. Smejkal, and G. Fettweis, "Latency Overhead of ROS2 for Modular Time-Critical Systems," Jan. 2021.

[72] Y. Yang and T. Azumi, "Exploring Real-Time Executor on ROS 2," in *2020 IEEE International Conference on Embedded Software and Systems (ICESS)*, 2020, pp. 1–8. DOI: 10.1109/ICESS49830.2020.9301530.

[73] eProsima team, *Fast DDS Discussions, Roadmap question: QueryCondition implementation*, https://github.com/eProsima/Fast-DDS/discussions/3644, Last accessed: June, 2023, 2023.

[74] eProsima team, *Fast DDS Github*, https://github.com/eProsima/Fast-DDS, Last accessed: September 2023, 2023.

[75] CycloneDDS authors, *CycloneDDS Github*, https://github.com/eclipse-cyclonedds/cyclonedds, Last accessed: December, 2023, 2023.

[76] Intel Corporation, *Intel® Core™ i5-8365U Processor*, https://ark.intel.com/content/www/us/en/ark/products/193555/intel-core-i5-8365u-processor-6m-cache-up-to-4-10-ghz.html, Last accessed: January, 2024.

[77] K3s Project Authors, *Cluster Datastore*, https://docs.k3s.io/datastore, Last accessed: February, 2024.

[78] The Kubernetes Authors, *Deployments*, https://kubernetes.io/docs/concepts/workloads/controllers/deployment/, Last accessed: September, 2023, Sep. 2023.

[79] Rancher, *K3s Kine Github*, https://github.com/k3s-io/kine, Last accessed: February, 2024, 2023.

[80] RTI, *Overview of the Reliable Protocol*, https://community.rti.com/static/documentation/connext-dds/5.2.3/doc/manuals/connext_dds/html_files/RTI_ConnextDDS_CoreLibraries_UsersManual/Content/UsersManual/

Overview_of_the_Reliable_Protocol.htm, Last accessed: July, 2023, 2016.

[81] K. Kolonko, “Performance comparison of the most popular relational and non-relational database management systems,” M.S. thesis, Blekinge Institute of Technology, Department of Software Engineering, 2018, p. 66.

[82] P. Killelea, *Web Performance Tuning: Speeding Up the Web* (O’Reilly Series). O’Reilly Media, Incorporated, 2002, ISBN: 9780596001728. [Online]. Available: https://books.google.nl/books?id=sX60mAi0eQUC.

[83] VMware, “Timekeeping in vmware virtual machines,” Dec. 2011.

[84] etcd Authors, *Performance*, https://etcd.io/docs/v3.5/op-guide/performance, Last accessed: February, 2024.

[85] K. Authors, *Kubernetes Components*, https://kubernetes.io/docs/concepts/overview/components/, Last accessed: April, 2024, Jan. 2024.

[86] Cpp Reference Community, *Fixed width integer types*, https://en.cppreference.com/w/cpp/types/integer, Last accessed: December, 2023, Oct. 2023.

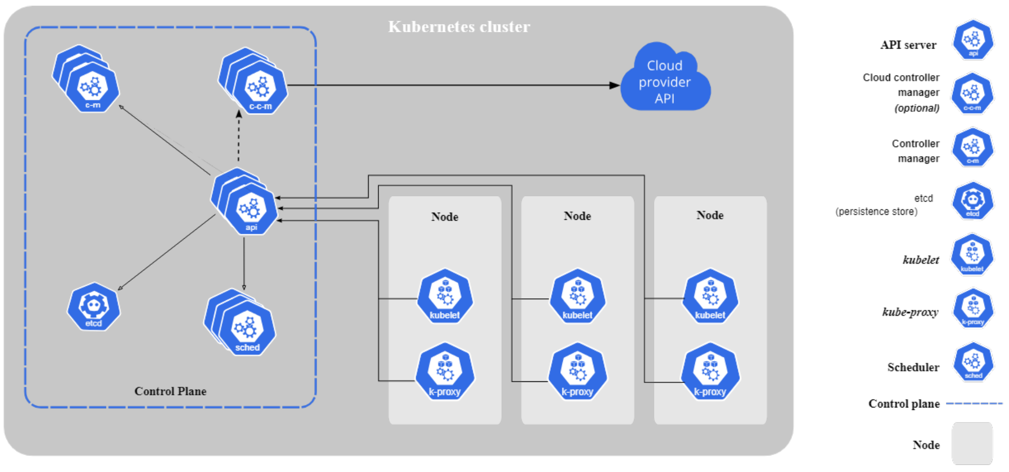

Figure A.1: Components of a Kubernetes cluster. Reproduced from [85].

Appendix A
Kubernetes Components

Figure A.1 depicts the components of a default Kubernetes cluster. Note the API communication channel with *etcd* in the Control Plane.

Appendix B
RTPS Module Structure

In Figure B.2, a diagram showing the four modules composing RTPS is depicted.

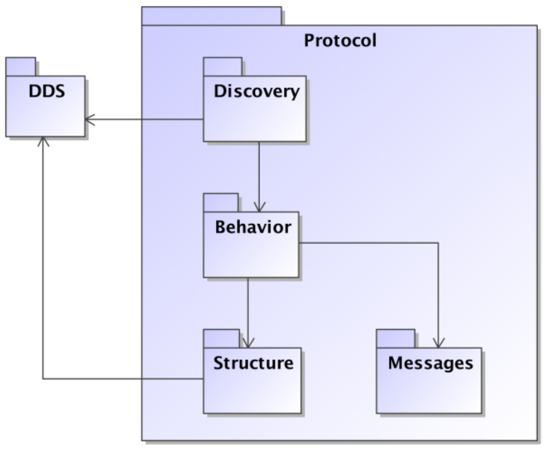

Figure B.2: RTPS modules. Reproduced from [57].

Appendix C
Instance State Chart Flow

In Figure C.3, a simplified fragment of the instance state flow of a DDS message is shown, when using the TRANSIENT LOCAL QoS in the context of the prototype system. Note how, once no live writers for the message remain, its state transitions from ALIVE to NOT_ALIVE_NO_WRITERS. With the workaround implemented, when a writer dies, the change of state in the messages is detected and they are republished by the remaining writer, thus effectively keeping them ALIVE and ensuring persistency in the DDS database.

Appendix D
Frequency Calculation

The frequency of the processor used, whose specifications are explained in subsection II-F, has a base frequency of 1.60 GHz and a maximum turbo frequency of 4.10 GHz [76]. However, during the execution of the system, the real frequency can vary within that range and it is even less predictable within the VMs. Therefore, in order to offer an acceptably accurate estimate, the TSC has

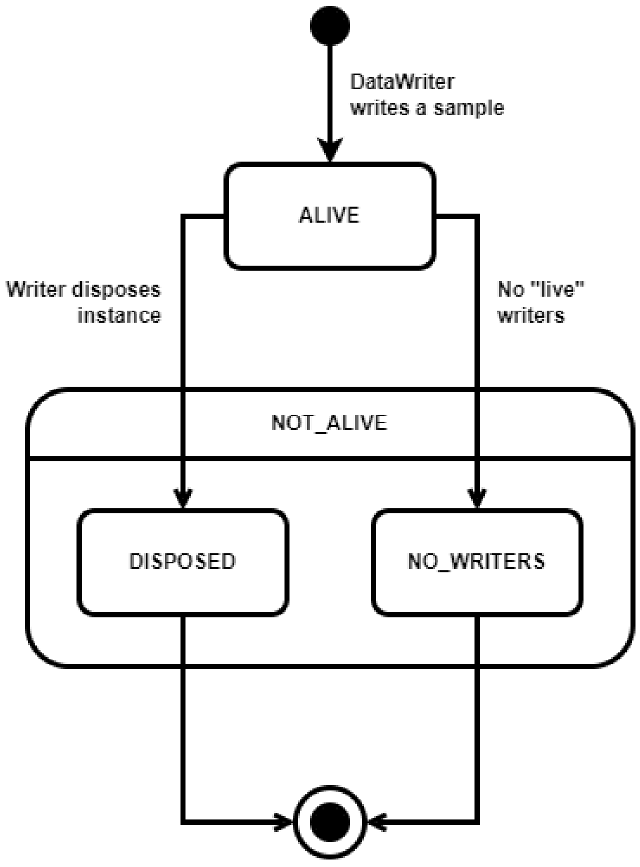

Figure C.3: Simplified instance state chart. Adapted from [48].

been used to determine the frequency. In order to do so, while the system is running, the TSC value has been logged at the beginning and end of a fixed period of 1000µs. With the values gathered in such way, the following formula has been used to calculate the frequency:

$$f = \frac{TSC_F - TSC_0}{1000\text{µs}}$$

A total of 100000 runs have been performed, which has given a mean frequency of 2.24GHz.

Appendix E
etcd Benchmark Tool Configuration

In order to simulate the conditions of the DDS prototype system on *etcd* and the conditions of using *etcd* with a witness node, the following configuration has been used:

1) Global configuration
   a) Clients has been set at 3, one per node.
   b) Connections set at 3, one per node.
2) Write-specific configuration
   a) Key size set at 4B (size of a uint32_t in C++).
   b) Value size set at 953, average size of messages calculated for 40 2-minute runs of the system, ignoring Kine specific messages without value.
   c) Rate of writes should have been set at 3 requests per second (rps), as per the measured stable throughput. However, such low throughput requires a con-

siderable amount of time to complete and, after observing very little difference in results using 3 rps and others using 100 rps, the rate has been set at 100.

3) Read-specific configuration
   a) Rate of reads set at 10 reads per second, extracted from the stable throughput. As with write measurements, 10 rps results in long measurements with little to no difference to the latency measured. Rate has also been set at 100 reads per second.

Appendix F
PostgreSQL and *etcd* Experimental Setups

The testing setup used for PostgreSQL is the same 2-node setup as for DDS, as explained in subsection II-F. However, there is one main difference: the PosgreSQL database is limited to the leader AZ, with no replica on the secondary node.

In the case of *etcd*, an auxiliary node has been added to the 2-node setup. This node has the same characteristics as the other 2 nodes aforementioned, with the difference that its RAM memory has been reduced to 3072 MB due to the limitations of the host's hardware. Given that the measurements have been taken with the *etcd* benchmark tool, as explained in Table III, the *etcd* latency has been measured without the creation of a K3s cluster.

Appendix G
DDS Topic Data

In Table G.1, the data structures of both topic messages are shown. Note the difference of size of messages of both topics: the Main Topic is already over six times bigger than the other when considering empty *name*, *value* and *old_value*. Furthermore, the Maximum Id topic size is fixed given that a C++ type *uint32_t* has always the same size of 32bit [86], while the Main topic has fields of variable size, namely those using *std::string* and *std::vector<uint8_t>*.

Topic	Field name	C++ Data Type
Main	id	uint32_t
	name	std::string
	created	int32_t
	deleted	int32_t
	create_revision	int32_t
	prev_revision	int32_t
	lease	int32_t
	value	std::vector<uint8_t>
	old_value	std::vector<uint8_t>
MaxId	maxid	uint32_t

Table G.1: Field names and data types of the DDS topics.

Appendix H
DDS Queries

In order to create the prototype, Kine queries have been translated into DDS. In essence, that means that SQL queries have been translated into C++. In Table H.2, all DDS queries accessed by Kine are listed. In the two last columns, the equivalent of DDS reads and writes on each topic is specified. As it can be seen, most Read queries need to read in both topics, except for the last 2 of them, which indicates that a better method should be devised to query the DDS database. Finally, in the case of Write queries, note how they also require to perform a read, in the first case to retrieve the MaxId to create a message with a consecutive Id and in the second case to find the message that needs to be disposed through its id. Nevertheless, since they are the only queries that perform a write, they are considered only as Write queries.

Query	Read/Write	DDS Main Topic	DDS MaxId Topic
Query Generic 01	Read	2	1
Query Generic 04	Read	2	1
Query Generic 09	Read	3	1
Query Generic 11	Read	2	1
Query Three	Read	2	1
Query Compact Revision	Read	1	-
Query Max Id	Read	-	1
Publish	Write	1	1
	Read	-	1
Dispose	Write	1	-
	Read	1	-

Table H.2: Queries and their corresponding DDS reads and writes per topic.

THALES

Approval Internship report/Thesis of:

Saül Abad Copoví

Title: Container Orchestration in Limited Availability Zones Domains

Educational institution: University of Twente

Internship/Graduation period: February 2023 to March 2024

Location/Department: Hengelo / Platform

Thales Supervisor: Max Riesewijk

This report (both the paper and electronic version) has been read and commented on by the supervisor of Thales Netherlands B.V. In doing so, the supervisor has reviewed the contents and considering their sensitivity, also information included therein such as floor plans, technical specifications, commercial confidential information and organizational charts that contain names. Based on this, the supervisor has decided the following:

- [✓] This report is **publicly available (Open).** Any defence may take place publicly and the report may be included in public libraries and/or published in knowledge bases.

- ○ This report and/or a summary thereof is **publicly available to a limited extent (Thales Group Internal)**.
  It will be read and reviewed exclusively by teachers and if necessary by members of the examination board or review committee. The content will be kept confidential and not disseminated through publication or inclusion in public libraries and/or knowledge bases. Digital files are deleted from personal IT resources immediately following graduation, unless the student has obtained explicit permission to keep these files (in part or in full). Any defence of the thesis may take place **in public to a limited extent**. Only relatives to the first degree and teachers of the ....department <*name department* > may be present at the defence.

- ○ This report and/or a summary thereof, is **not publicly available (Thales Group Confidential)**. It will be reviewed and assessed exclusively by the supervisors within the university/college, possibly by a second reviewer and if necessary by members of the examination board or review committee. The contents shall be kept confidential and not disseminated in any manner whatsoever. The report shall not be published or included in public libraries and/or published in knowledge bases. Digital files shall be deleted from personal IT resources immediately following graduation. Any defence of the thesis must take place **in a closed session** that is, only in the presence of the intern, supervisor(s) and assessors. Where appropriate, an adapted version of report must be prepared for the educational institution.

THALES

Approved:	Approved:
Max Riesewijk	
(Thales Supervisor)	(Educational institution)
Hengelo 27-03-2024	
(city/date)	

(copy security)

THALES