Partitionable Decentralized Topic Key Management

Bryan van Wijk

October 2019

Partitionable Decentralized Topic Key Management

by

Bryan van Wijk

to obtain the degree of Master of Science in Computer Science
Data Science and Technology Track
with a specialisation in Cyber Security
at the Delft University of Technology
to be defended publicly on Thursday November 14, 2019 at 10:00 AM.

Student number:	4363329	
Thesis committee:	Dr. Ir. J.C.A. van der Lubbe,	TU Delft, supervisor
	Dr. Ir. J.H. Weber,	TU Delft
	Dr. T.B. Quillinan,	Thales Research and Technology
	Dr. M. Bjorkqvist,	Thales Research and Technology

TU Delft

Abstract

In a military environment, tactical networks enable information sharing between all the different entities in the field. In this environment, multiple groups of people from different organizations, and with different goals and policies have to share information. The information has to be shared without the risk of leaking information to unauthorized entities. Cryptography algorithms are used to encrypt information with a key to remain in control of when, where and to whom it is shared. All information is encrypted based on the concept of content-based encryption. In this unreliable environment, the cryptographic keys used to secure the data have to be available to continue collecting and processing information.

A key management architecture should be in place, to facilitate the generation and distribution of these keys. The purpose of this key management architecture is to provide the entities in the field with specific keys such that information access policies can be enforced. The challenge here is that in tactical networks, network partitionings are expected to happen. Therefore, the same keys have to be redundantly available at multiple locations to prevent a single point of failure. In a connected network, the keys can constantly be synchronized between these locations. However, the problem of key de-synchronization occurs if the network is split for some time, keys are changed on both sides, and then the network is recombined. This leads to possible conflicting keys because synchronization was temporarily not possible. The key management architecture must be able to handle such conflicts and reintegrate them as necessary.

In this thesis, we present a decentralized key management architecture with a solution for the key de-synchronization problem. We propose to use Conflict-free replicated data types [22], to store the keys at multiple locations and prevent conflicts. Conflict-free replicated data types is a concept to store and replicate data across multiple instances. This data type is characterized by the possibility to update the data in all instances independently, and concurrently, without coordination between the instances. Additionally, three approaches for the coordination of key creation are proposed with different levels of consistency and availability. The architecture and the three approaches are compared in experiments to evaluate the differences and prove the feasibility of the designs.

Preface

Before you lies my master thesis report submitted to obtain the degree of Master of Science (MSc) in Computer Science. It concludes the master's degree in Computer Science with a specialisation in Cyber Security at the Delft University of Technology.

I would like to express my gratitude to a number of people, without whom this project could not have been completed successfully. First of all, Jan van der Lubbe for his supervision as university supervisor. Your meetings helped me to look from another perspective at my work. I also want to thank Thomas Quillinan and Mathias Bjorkqvist from Thales for their guidance and feedback during the project. Finally, I want to thank my family and friends for their support in the last months. A special thanks to my sister, for proofreading parts of this thesis report.

Bryan van Wijk
Delft, October 2019

Contents

1 Introduction

In a military environment, tactical networks enable information sharing between all the different entities, such as the sensors, monitoring interfaces, ships, and other components in the field. Communication using tactical networks leads to multiple challenges such as link disruption, network partitioning, and limited resources. Another complicating factor is that the network resources are shared between multiple groups of people that belong to different organizations, and with different goals and policies. This multi-domain context requires the information to be shared without the risk of leaking information to unauthorized entities. Reliable information sharing is, however, important in a military environment to provide essential intel during missions.

Cryptography algorithms are used to encrypt information with a key to remain in control of when, where and to whom it is shared. All information is encrypted as individual objects directly by the users who produced it such as sensors. Once the information is encrypted it can be stored in a public location or shared over a public network, without compromising confidentiality. This approach of directly securing the information instead of the connection is called content-based encryption [27]. Access policies per group or organization can now be enforced by only providing the keys that give access to information to authorized users. This requires the keys to be generated, stored and distributed by a central key management system independent of the users. In this manner, organizations retain control over access to the information.

Different group key management architectures have been introduced in the literature. These existing solutions are however not designed for tactical networks with their unreliable and dynamic characteristics, in combination with shared resources. The problem is that if the keys are distributed from a single location this leads to an easy single point of failure due to network partitioning. Existing decentralized solutions either do not consider multi-domain contexts with content-based encryption that requires keys to be managed in a central system or did not consider the problem of key de-synchronization in a partitioned network. The key de-synchronization problem arises if the network is split for some time, keys are changed on both sides, and then the network is recombined. The key management architecture must be able to handle such conflicts and reintegrate them as necessary.

To the best of our knowledge, the first work that addresses the problem of key de-synchronization is [15]. In this work, a distributed group key management service (GKMS) architecture is proposed. They propose tackling the network partitioning problem and single point of failure by connecting multiple GKMS instances. The connected instances form a network and keep keys synchronized across all GKMS instances. The key de-synchronization problem is solved by just creating a new version of all conflicting keys when the network is restored. However, this is not ideal in terms of efficiency, because each new key has to be distributed again to all users. The work also does not contain any experimentation of the proposed solution, this would be necessary to evaluate the benefits and drawbacks, and to compare this solution with other options. There are some parts of the architecture left open for interpretation, such as: how to perform the synchronization of keys between instances?; How a master is elected? Why was a single master architecture chosen and whether there are other possible designs? In this thesis, we extend the ideas from the work done in [15] by proposing solutions for these architectural decisions and proposing alternative designs.

In this thesis, we aim to design a decentralized key management architecture that is able to handle conflicts between replicated key managers. Next to this, two different approaches for the coordination of key creation compared to the single master architecture are proposed. The implementation of this architecture and the different approaches is evaluated and discussed. In the next section, a concrete problem use case is presented where such an architecture would be useful.

1.1. Problem Use Case

We base our research on a military setting where information must be encrypted before it is shared and stored. In this setting, network resources and data storage are shared between multiple groups from different organizations. Encryption is therefore used to control who has access to what information in a central system to be easily manageable and consistent. Due to the challenging environment with frequent link failures and network partitioning, a single key manager can expose a single point of failure. To give a concrete example: some navies require that critical systems, including C4I (Command, Control, Communications, Computer, and Intelligence) activities be supported in at least two physical locations onboard a ship. This means that all critical activities must be available whenever there is a disconnect between the compute services and sensors/actuators in different locations onboard. This means for key management systems that they have to be distributed and replicated in more than one physical location onboard. Network outage in this use case could result from accidental or kinetic breakages. The expected duration of the outage in this use case ranges from minutes to days. During this period the two disconnected partitions should be able to work independently but also allow consolidation after the network is restored.

1.2. Research Question

The problem use case explains the need for a key management architecture that can perform in a multi-domain environment. The key management architecture should be designed without a single point of failure while maintaining a central system to ensure the manageability of the keys. Additionally, the architecture needs to take into account the key de-synchronization conflict.

Research Question: "How can a decentralized key management architecture for content-based encryption be designed in a conflict-free way to tackle the key de-synchronization problem that facilitates the needs of a multi-domain environment?"

This research question raises the following sub-questions:

- "How to design a decentralized key management architecture for content-based encryption in a multi-domain environment?"
- "How to prevent conflicts in a decentralized key management architecture to tackle the key de-synchronization problem?"
- "How can the performance of a decentralized key management architecture in handling network partitioning be measured?"

1.2.1. Requirements

Based on the research question, a set of requirements has been created regarding the functionality of the key management architecture:

1. The keys should be stored in n locations and can be retrieved by users at all these locations.
2. The architecture should allow the delivery and generation of keys during a network partitioning and recombine the keys as soon as the network is restored
3. The key management should be independent of the users of the keys.
4. Key rollovers should be requestable in all key managers.
5. The key de-synchronization between key managers should be taken into account and resolved automatically.
6. Keys should be persistent and never get lost.

1.3. Contributions

Our contributions are as follows:

1. We introduce a design to store keys in a decentralized way where conflicts are prevented to solve the key de-synchronization problem.
2. We present three decentralized key management approaches providing different levels of consistency and availability.
3. We provide an experimental comparison of the three approaches in terms of performance to evaluate the differences and prove the feasibility of the designs.

1.4. Report Outline

The structure of this thesis report is composed in the following way. The next chapter discusses the background concepts that are used for the design of our solution. Chapter 3 focuses on existing solutions and related literature. In Chapter 4, the decentralized topic key management architecture is presented. The basic idea is stated, followed by a detailed description of the different components. Chapter 5 provides an analysis of the proof-of-concept implementation details and the simulation results of different test scenarios. Finally, in Chapter 6 we discuss the research questions and provide an outlook for future work.

2 Background

In order to understand the decentralized key management architecture presented in this thesis, it is necessary to present some background knowledge. In this chapter, background knowledge on the used concepts and models of the key management system are provided. Figure 2.1 presents a schematic representation of the important concepts in an example environment of a single organization, four users and four topics.

2.1. Key Management System

The Key Management System (KMS) is responsible for storing, generating and renewing the encryption keys. Users ask the KMS for the keys they need after which the KMS sends the required keys if the user can prove it has the right permissions. Organizations remain in charge of the data; that means they are in charge of when, how long, to what, and which users have access to the data. This is accomplished by selectively handing out permissions to users for periods in which they are allowed to access certain data. The goal of the KMS is to make sure all authorized users can retrieve the latest keys. Users are, for example, sensors producing data or monitoring interfaces consuming the data so they are not necessarily actual people.

2.2. Topic Concept

Not all data is encrypted with the same key because this would mean a user could have access to all data or not at all. The other extreme is using a different key for each data object providing a finer granularity for access control, but it is not very scalable. Therefore, the data users produce are separated into categories of related data called topics. In Figure 2.1, the topics are indicated with a, b, c and d. A topic consists of related data about a subject with the same access constraints. An example of two topics are the medicines patients use and the address details of these patients. It is important information that should be shared between a doctor and a pharmacy, but for example, the medicine delivery person should only have access to the address details of a patient. All data belonging to the same topic is encrypted with the same key called the Topic Key (TK). This means if a user has permission for a certain period to a topic and he retrieves the key he can decrypt all data of this topic created during this period. The shared key of a topic also changes from time to time. There are multiple reasons why a TK changes. The key needs to change when a new user is given permission to a topic or the permission for a user expires. This is done to ensure a user is not able to use the key to decrypt data from before it had access to the topic or after it has no permission anymore. Keys are also changed in a regular interval to limit the damage when a key is compromised. This means data from the same topic in the shared data space can be encrypted with different versions of the TK. Therefore all old TKs are also stored in the case authorized users need to access older data. All encrypted data is labeled with the version of the TK used to encrypt it. This way the users can ask the right key from the KMS.

2.3. Domain Concept

A KMS hands out keys for only one domain, a domain is a superset of all topics. Every topic belongs to exactly one domain. This means the keys for a topic can only be retrieved from one KMS. As an example, in Figure 2.1 the KMS of Domain 1 can only hand out keys for Topic a and Topic b. Keys for Topic c and Topic d can only be retrieved from the KMS of Domain 2.

The circles with numbers in Figure 2.1 indicate a typical order of events to demonstrate the concepts. User C can do a key request for Topic a and Topic b to KMS 1 (1). In this request User C has to provide proof that it is allowed to obtain the keys of these Topics from the KMS of this domain. User C can now use the key of Topic a to encrypt some data and publish it in the shared data space (2). Another possibility is to retrieve a data object from the shared data space that is encrypted with a Topic b key and use the key of Topic b to decrypt it (3). User D can retrieve the just stored encrypted data from the shared data space (5). For the decryption, User D has to obtain the correct key first from KMS 1 (4). The KMS must have a high-availability, the keys regularly change, so the KMS should also be available at all times. It must be resilient against disconnected networks/network partitioning. The context of this research will be referred to as topic key management from now on.

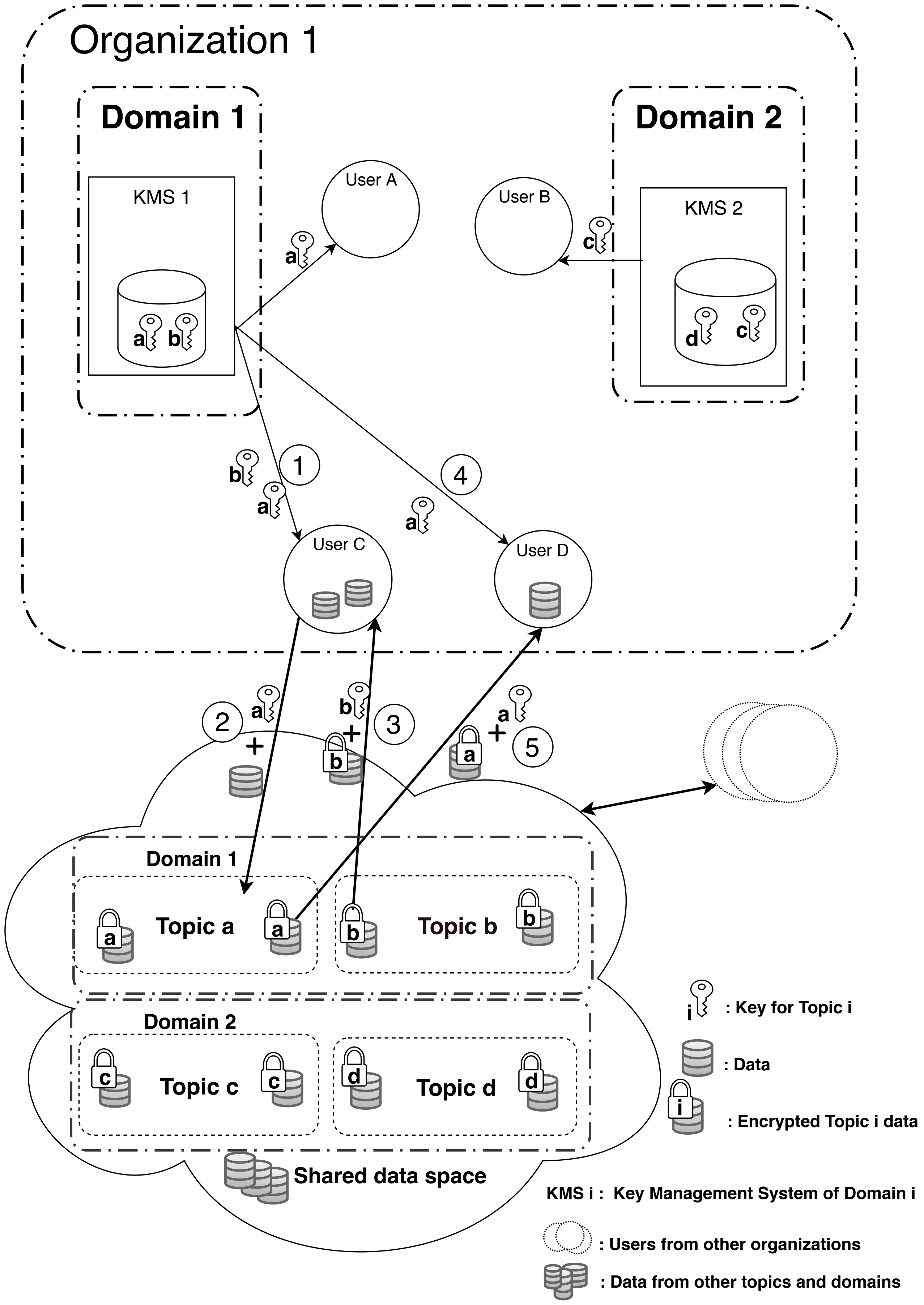


Figure 2.1: Example illustration of the topic concept.

2.4. Data Replication

Making this more abstract, key management can be seen as a service requiring a high availability of the data where the data are the keys. Providing services in a highly available manner with high performance is a difficult problem which is well studied in different areas. An approach to achieve these requirements is by using replication techniques. Decentralized key management requires eventually storing the keys in such a way they are accessible from multiple instances. Copies of the same data (the keys) have to be kept in sync with each other. In this section, data replication, in general, is discussed which provides insights into the key storage system for a decentralized key management system.

Highly available data storage systems can broadly be categorized in a master-replica architecture and multi master architecture [24]. In the traditional master-replica architecture writes can only be processed by one instance: the master. Changes in the master are synchronously or asynchronously applied to all replicas. The replicas are only used for load-balanced data reading or as hot spares. There are no guarantees about the delay between applying changes on the master and the propagation to the replicas. The replicas are always behind and changes might be lost or incomplete when the master crashes. An automatic fail-over mechanism can be used to promote a replica to a master when the master fails or is unreachable due to a network partitioning.
In multi master architectures, all replicas are equal and can all be used for reading and writing. Changes of the data in a node are synchronously or asynchronously replicated across all nodes. Implementation of this approach is more difficult and could decrease performance. Changes cannot be replicated during network connection problems which could lead to conflicting changes in replicas. A possible solution to prevent conflicts is to only allow updates if a majority of other replicas can be reached. An example of such a quorum based approach is [12]. A disadvantage of a quorum based approach is the reduced availability during a network partitioning as only the replicas in a majority part can make progress.
Optimistic replication techniques can be used to improve availability. Changes are allowed to propagate to the other replicas in the background and disconnected replicas can make progress on their own. The downside of this approach is the possible introduction of conflicts between replicas and the need to solve these conflicts. Important design considerations of a replicated data storage are how and when the conflicts are resolved [7]. The best way to solve conflicts depends on the meaning of data and is thus application-specific.

2.4.1. CAP Theorem

There are generally three properties that are desired when designing distributed services: consistency, availability, and partition tolerance. According to the CAP theorem [10], not all three of the properties can be guaranteed simultaneously. This means if you have two nodes on opposites sides of a partition, allowing one node to update its state will make the two nodes inconsistent violating the consistency property. Preserving consistency means the nodes cannot individually make progress so during a partitioning they are unavailable violating the availability property. Communication is necessary to preserve both consistency and availability, but this means partitions cannot happen. Only two of the three properties can be guaranteed simultaneously. This does not mean the third property cannot be guaranteed at all [3]. There can also a trade-off be made between the properties so there is a wide range of different configurations of levels of the properties. During the design of the key management system, the specific requirements of the system should be taken into account to determine and optimize the three properties.

In most real-world systems avoiding network partitions and disconnections is impossible. Therefore a trade-off should be made between availability and consistency during a network partitioning. Temporarily inconsistencies must be tolerated if availability cannot be sacrificed. Allowing inconsistencies will improve the availability, but requires replicas to deal with weaker notions of consistency. During a network partitioning, nodes on both sides keep fulfilling queries by reading and writing data while possibly introducing inconsistencies.

2.4.2. Consistency Levels

The conflict between consistency, availability, and partition tolerance has led to a lot of research to weaker levels of consistency. Three levels of consistency can be distinguished: strong consistency, weak consistency, and eventual consistency [26]. Strong consistency requires an update to be replicated to all replicas before it completes. This ensures any subsequent read from any replica will return the updated value. Weak consistency means that subsequent access is not guaranteed to return the updated value. The system thus contains inconsistencies and does not provide any guarantees on when the system will be consistent again. Eventual consistency is a specific form of weak consistency. It guarantees that when no new updates are made eventually all replicas will converge to the same state. A real-world example of an eventually consistent system is the domain name systems (DNS). A domain name update is slowly distributed in a hierarchy and based on cache timeouts. Eventually, all clients will see the update.

The advantage of eventual consistency over strong consistency is that every replica can make progress on its own and does not have to wait when experiencing long network latencies. Eventual consistency, however, makes reasoning about the state and development of the complete system more difficult. A user might not be able to read the value it has just written, because different replicas could be used for the read and write operation. Translating this to the key management problem means information about a just required key renewal is not present even though it was just requested. In [14] and [18], the authors propose a possible solution to make programming with eventual consistency easier and provide more guarantees by using vector logical clocks. The vector logical clocks are used to provide session guarantees that are weaker than causal consistency but stronger than eventual consistency. Causal consistency respects causal dependencies of updates and does not make an update visible if some of the causal dependencies are not yet in the replica. This is a very useful guarantee but it can cause significant performance overhead in partitioned systems.

2.5. Conflict-free Replicated Data Types

The Conflict-free replicated data type (CRDT) [22] is a concept to replicate data across multiple instances with eventual consistency. This data type is characterized by the possibility to update the data in all instances independently, and concurrently, without coordination between the replicas. This could be used to store the keys in key managers in a resilient way, to solve the key de-synchronization problem. The concept of conflict-free replicated data types was first introduced by Shapiro et al. [22] and more formally described in [9]. CRDTs are a data structure that can be replicated in multiple locations and the replicas can be updated independently and concurrently without coordination. In traditional data structures, conflicts between replicas are prevented by blocking updates if guarantees cannot be satisfied or by implementing conflict resolution policies. The disadvantage of these traditional solutions is the increased time overhead and, thus, longer response time for the users. The idea of CRDTs is to optimistically allow all updates to go through, which possibly can lead to inconsistencies between replicas. Consistency between replicas is eventually achieved through merging the replicas. CRDTs ensure they converge to the same value, as long as all updates are applied to each of them no matter their order. The goal is to avoid coordination by ensuring actions taken independently by replicas can not conflict with each other. A simple example of a replicated data type is a replicated counter that converges because its operations incrementing and decrementing commute. The special feature of this data type is the fact that it is self-stabilizing after multiple replicas have diverged. All replicas are guaranteed to eventually converge to a consistent state. This means all replicas could be independently updated without synchronizing between the remote replicas beforehand. This ensures better performance and scalability in distributed systems compared to a system that first needs to synchronize before updates can be applied. Other systems promising eventual consistency require consensus and/or roll-back on updates that conflict because they are executed concurrently at different replicas. CRDTs provably converge to a correct common state.

Although CRDTs provide some promising features it comes with some limitations on the available data types and the operations possible. Next to some trivial data types, such as registers and counters, there are some non-trivial CRDTs defined in literature [23]. Presented CRDTs designs are sets, graphs, and sequences. These can be used to construct more complex data structures. Other data types can be constructed based on sets. The operations add and remove on a set do not commute. Therefore a correct set cannot be a CRDT. A CRDT set, therefore, approximates a set. There are different types of CRDT sets described. One of these is the grow-only set. A Grow-Only Set is a set where the remove operation is not permitted. A refinement of a Grow-Only Set is a Two-Phase Set (2P-Set). A 2P-Set is a CRDT set where an element may be added and removed, however it can never be added again after it is removed. It is constructed by combining two Grow-Only sets, one for adding and the other for removing. Removing is only allowed if the element is in the set. The lookup operation only returns the element if it is not in the removed set. There are more variations possible with CRDTs sets but these examples show the potential of what is possible.

3 Previous Work

Different classes of key management have been researched in the past. The topic of this thesis can best be classified as group key management. Group key management has been divided into three main classes: centralized, decentralized and distributed approaches [17][20]. In Section 3.1 we will discuss to which class the topic of this thesis belongs. In the rest of this chapter, prior art regarding the topic of this thesis is discussed. Section 3.2 presents the key management architecture presented in [15]. Our key management architecture is inspired by this architecture. Section 3.3 discusses two group key management schemes that are designed for a different purpose but still give some useful insights.

3.1. Decentralized Key Management

The architecture presented in this thesis can be regarded as a decentralized group key management scheme. The concept of decentralized schemes is to have multiple key managers and let each key manager handle the key requests of a subgroup of all users [20]. An important problem these schemes try to solve is the scalability of the number of users. Every time a user joins or leaves a group, the group key should be rekeyed and distributed to all users of that group. Some solutions, therefore, propose to divide a group into subgroups and use an independent local group key for every subgroup, as discussed in [5][11]. This approach is called the subgroup key approach, as opposed to the common group key approach, where all group users use the same group key. The subgroup key approach reduces the impact of rekeying to only a subset of all users of the group. The problem is, this solution requires decryption and re-encryption of data transmitted from one subgroup to the others. It affects the data path because the data has to be translated during transmission. This poses a security risk in the environment of this thesis because the network resources are shared.

The decentralized architectures can be further divided into two sub-categories: membership-driven and time-driven [5]. In membership-driven schemes, the group key is rekeyed after each membership change. In time-driven schemes, the keys are updated after each predefined time slot. Time-driven approaches would only allow users to join and leave at fixed points in time or give users access to more data than possibly desirable. In the military context presented earlier keys, however, have to change both after a predefined amount of time and when the access policy changes.

3.2. Distributed Group Key Management Service Architecture

In 2018 *F. Poltronieri et al.* proposed a decentralized approach for group key management services [15]. Their paper introduces a solution to enable secure and efficient information sharing in multi-domain tactical networks.

They propose a decentralized approach for group key management distribution from a high level of abstraction. Multiple key manager instances are connected to form a network and keep the keys synchronized across all instances. The synchronization is managed with one master instance that makes all decisions. The master key manager is elected collectively by the key managers. When a group of key managers is disconnected from the master, a new master is elected among these key managers automatically. This is a key feature to make the system resilient against network partitioning. On a reconnection of two disconnected network partitions with their own master key manager, the keys in the two master instances must be realigned. The two conflicting master key managers perform the re-election of the master. The new master generates and distributes a new version of all keys. All users can contact any key manager to retrieve the keys, but if the key manager is not the master, it delegates key changes to its master.

The authors mention that further research will entail the in-the-field experimentation of the proposed solution to evaluate the benefits and drawbacks and to investigate possible improvements. The ideas can not directly be used because some parts of the architecture are left open and no other design options are considered. It is, for example, unclear why for a single master architecture was chosen and whether other setups are possible. It is also unclear how key manager instances detect if they are connected and disconnected from the other instances. It is important to understand that this solution does not elegantly solve the key de-synchronization problem. After a network partitioning, for all the conflicting keys a new version is created which means this new key has to be distributed to all users from each partition. This is not ideal in terms of efficiency because the generation and distribution of new keys takes time and possibly lead to more different keys instead of less. If during the distribution of the new key the network is partitioned again in another configuration, the new key cannot be distributed to all users and yet another key will be used by some of the users.

3.3. Other Decentralized Group Key Management Schemes

Other group key management architectures are designed to collectively agree on a common group key instead of the key distribution. These schemes focus on specific use cases, such as wireless, ad hoc networks, and IoT [1][6][8][28]. In these papers specific requirements such as computational limitations, mobility, storage, and bandwidth are important. They are less relevant for this research, but can still give some insights on how to handle key management in a decentralized way. There are multiple reasons why these other solutions are not satisfying the requirements of Section 1.2.1. These solutions do not consider the multi-domain military context that is considered in this thesis. In this context, it is required to have a central distribution system managing the keys instead of all users participating in generating a new key together. This is required because the users themselves can not determine the access policy of the keys. Secondly, the key de-synchronization problem is not considered because the keys are used to encrypt the direct communication channel. They are not used to encrypted stored information and decrypt it at a later point in time. This means the keys are only used for a single communication session. Since direct communication is not possible between users from different parts of a network partitioning, keys never conflict. As soon as a network partitioning is restored and users from previously different network partitions start to communicate a new group communication session is set up with new keys.

These other decentralized group key management schemes are thus not directly a solution for the topic of this thesis. The following two subsections describe two of these decentralized key management solutions that use some useful concepts in the rekeying and synchronization process which could be used for the design of our key management architecture.

3.3.1. Hydra: A Decentralised Group Key Management

The first scheme is described in *Hydra: A decentralised group key management* [16] where a decentralized group key management scheme is presented. The key management scheme is referred to as Hydra. Hydra is a scalable decentralised architecture to create and distribute symmetric cryptographic keys to encrypt IP-multicast messages. Instead of a central server handing out these keys to all users the users are divided into smaller subgroups. Each subgroup has its own key manager handing out keys. These key managers work independently from each other after the initialization of a central group manager. Only key managers participating in a group communication session keep the keys synchronized between them. The main idea described in this work is to use a group communication system *SPREAD*, that provides totally ordered delivery of messages between the key managers. This communication system must guarantee that if two key managers receive the same messages, they are delivered in the same order. It also generates notifications when the network is partitioned, and when it is recombined. The protocol to renew the keys is executed when the network is recombined to ensure all members of a group are using the same key again. Each key manager is equal and can start the rekeying process. Conflicts between key updates are prevented because all key managers will receive these updates in the same order.

The problem with the key management scheme Hydra is that it is based on the concept of a group communication session. Keys for a group communication session are only available in a subgroup of all key managers and only while the group members are connected. The key for a session is changed if the members change which means the data encrypted with this key is connected to a session instead of the actual access policies which could change over time. In a multi-domain military environment, this could lead to a problem as data is not always shared during a single communication session. Data is generated and encrypted independently of when and who will receive the encrypted data. This means keys should be available in all key managers in case an authorized user asks for it.

3.3.2. Dynamic Group Key Management Protocol

The second scheme is the Baal protocol described in *Dynamic Group Key Management Protocol* [4]. Keys are handed out by multiple local key managers to a subgroup of all users. This protocol is closely related to the underlying infrastructure as group members are grouped per subgroup network. The key concept is to have one head key manager with the ability to create local key managers. During the initialization, there is a single point of failure in the form of the head key manager. After the initialization, all key managers can generate new keys and distribute them. The new keys are multi-cast to the other key managers. It ensures consistency between key managers by assigning a priority number to every key manager. The changes from a key manager with the highest priority will be applied. The communication of new keys between key managers works in rounds so all key managers will make the decision based on the same information.

The communication between key managers in rounds prevents conflicts, however, it is expected to have an impact on the latency and thus the overall availability of the system. The problem with this protocol is that it like the Hydra scheme is focused on group communication sessions instead of creating keys per topic. This protocol is also specific to the underlying infrastructure and thus does not present a general solution.

4 Decentralized topic key management architecture

In this chapter, we propose our key management architecture, the CRDT-based decentralized key management architecture. The idea of the architecture is based on the scheme introduced in [15], which is discussed in Section 3.2. The architecture is however built around the concept of Conflict-free replicated data types to manage the replication of keys between multiple key manager instances. Next to this, two different approaches for the key creation coordination compared to the single master architecture from [15] are proposed. These approaches are multi master and synchronized key renewal. This chapter is organized as follows. First, the basic concepts of the architecture are discussed. Followed by the key synchronization in Section 4.4 and the key retrieval protocol in Section 4.5. The last Section, Section 4.6 describes the three key creation coordination approaches. First, the single master approach is described. The following two subsections present the two other approaches: multi master and synchronized key creation.

4.1. Context

The key management architecture has to provide users an access point to retrieve symmetric keys for encryption and decryption. The keys are rolled over when the keys are expired and the key is requested. This means the key managers should be contacted regularly by all users to access the keys. Users are in this case the entities that need a key for their operations. The key manager is in charge of verifying the request and send back a valid key. The authentication and authorization of requests and the distribution of this information to all key managers are out of scope for this research. They are assumed to be available because decentralized access control mechanisms have been researched in the past, and can be used for this [2].

At the basis, the architecture consists of multiple key managers located in physically different locations. The stored keys are kept in sync but the key managers work independently of each other. From the perspective of a user, the system looks like a single system. Multiple key managers complicate the process of key rollover compared to a single key manager due to concurrent changes at different locations. Even if not all key managers are connected, the individual key managers keep functioning independently. They provide older keys and generate new ones as required. The advantage is that during a network partitioning when some key managers are not reachable by a user there is the possibility that another key manager is reachable. This improves the availability of the keys and the resilience against network partitioning. However, as we have seen in Section 2.4.1, there is always a trade-off between availability and consistency if a system has to be partition tolerant. This means that keys created during a network partitioning can not be available in all key managers and thus introduces some inconsistencies.

During a stable scenario, all keys stored in every key manager are kept in sync. This ensures that it does not matter which key manager a user contacts, it will be able to get the requested key from each connected key manager. In a connected network, there is no problem since the key managers can just exchange messages to stay in sync with each other. During a network partitioning, however, this is not possible. Users have to keep performing their operations and new keys will have to be generated on both sides of the network partitioning. Keys for the same topic will differentiate from each other on both sides of the network partition and have to be recombined as soon as the network partitioning is healed. Therefore, we propose to use conflict-free replicated data types to store the keys which prevents the keys on both sides of the network partition to conflict. The proposal is designed in such a way that it does not have a single point of failure. The number of concurrently used keys for the same topic is considered and the authorized users are able to retrieve all past keys.

4.2. Basic Concept

The basis of the architecture is simple, it consists of n redundant key managers where $n > 1$:

$$\{KM_1, KM_2, ..., KM_n\}$$

The subscripts indicates the id of the Key Manager (KM). The other entities in this architecture are the m users:

$$\{U_1, U_2, ..., U_m\}$$

Where $m > 1$ is the total number of users. The users are the entities producing or consuming encrypted data. The data is encrypted per topic as introduced in Section 2.2. A topic key is defined as:

$$TK_{TI}^{V} = (TI, V, VP, KEY)$$

TI is the ID of the topic this key belongs to. Each key contains a unique version string V to be able to uniquely identify every key. The version in contrast to the key does not have to be kept private. It is used to find the right key of encrypted data. The topic key consists further of the Validity Period (VP) of the key, and the key itself KEY. To do the encryption, the users ask for the latest key of a topic by sending a message to any of the key managers. For the decryption, the users ask any of the key managers for the specific key used for the encryption. The capabilities of all the key managers are the same. All are assumed to have access to a method to generate new keys:

$$TK_{TI}^{V} \leftarrow generateNewKey(TI)$$

as well as a method to verify if a user is allowed to obtain the requested key:

$$isAuthorized(U, TK)$$

This method returns true if the user proved it has permission to obtain the topic key TK. The key managers communicate with each other to synchronize their state:

$$S = \{TKS_1, TKS_2, ...\}$$

The state consists of a set of Topic Key Sets (TKS). A topic key set is the set with all versions of the keys for a topic and is defined as:

$$TKS_{TI} = \{TK_{TI}^{1}, TK_{TI}^{2}, ...\}$$

In Figure 4.1 an example with $n = 4$ key managers and $m = 5$ users is shown.

4.3. Key Managers Cluster

Key managers have to keep track of which key managers are currently reachable and which are not. To accomplish this the key managers form a cluster with all key managers that are connected. When a partition in the network occurs the unreachable nodes will be automatically detected and marked as unreachable. If they do not come back up within a predefined timeout they will be removed from the cluster. The key managers in the cluster monitor each other by sending heartbeats to detect if a key manager is unreachable. As soon as a network partitioning is healed the different clusters have to be recombined into a single cluster. Therefore we detect if some other key managers get connected again and a new larger cluster can be created. This is done by regularly testing if a missing key manager is reachable again.

4.3.1. Initialization

At startup, a key manager first tries to join an already existing cluster of key managers. It accomplishes this by sending a join request to all other key managers which IP addresses are given to the key manager on startup. If there does not exist a cluster of key managers yet or if they are not reachable a key manager will start a new cluster with initially only itself as a member. Other key managers who are not a member of any cluster and are reachable can now join this node. As long as all known key managers are not in the same cluster there will be a regular check to see if they are reachable. If a key manager detects that another key manager is reachable again it will send a join request containing its id and the number of currently connected key managers. If a key manager receives such a message it will decide deterministically based on this information whether it should leave its own cluster or wait because the other node will leave its cluster and join.

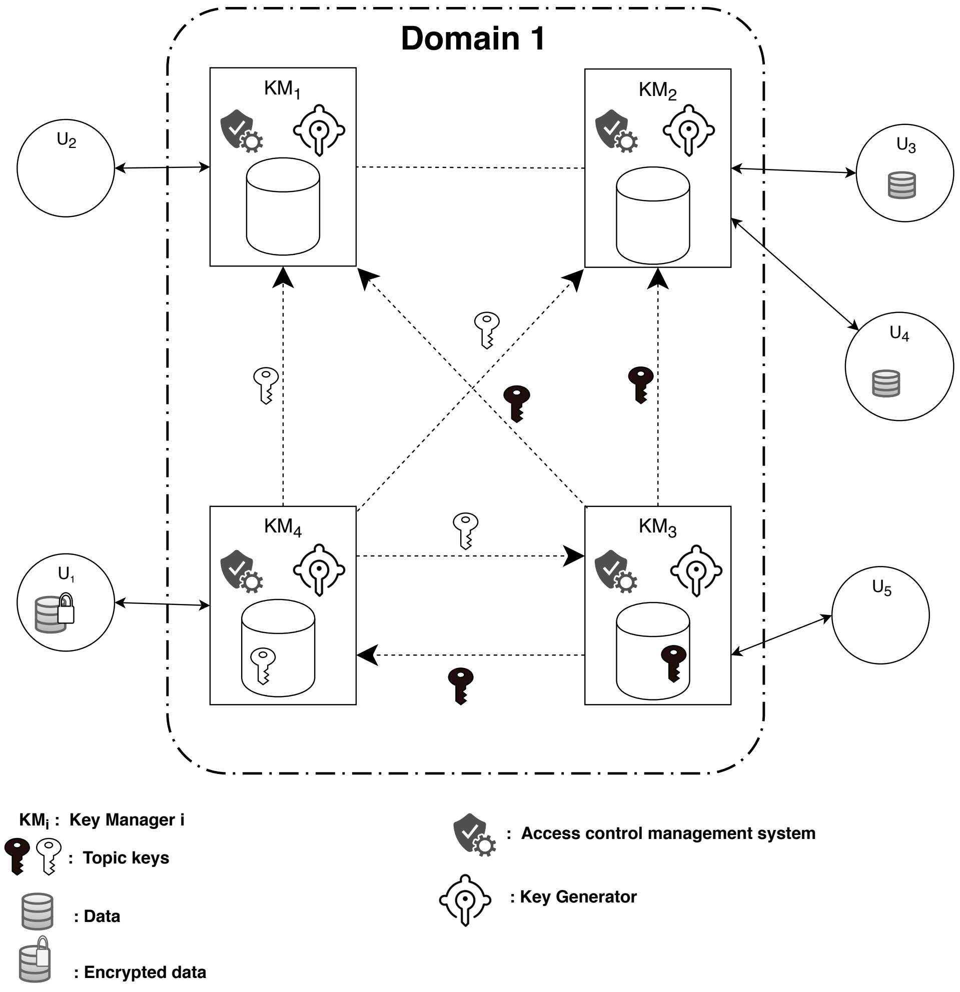


Figure 4.1: Example scenario of all entities involved in the decentralized key management architecture. In this scenario with four key managers and five users.

4.4. Key Synchronization

Within each cluster of key managers, keys are synchronized using the gossip protocol described in Amazon's Dynamo system [7]. The current state of the cluster gossips randomly within each cluster of key managers. Every key manager contacts every predefined timeout another key manager and merge their state with all the keys. Merging is done with the following algorithm:

Algorithm 1 Merging two states

```
1: procedure mergeState($S1, S2$)
2:     $result \leftarrow \{\}$
3:     for each $TKS_i \in S1$ do
4:         for each $TKS_j \in S2$ do
5:             if $i = j$ then
6:                 $result \leftarrow result \cup \{TKS_i \cup TKS_j\}$
7:     return $result$
```

4.4.1. Partition Tolerance

The key feature of the designed architecture is the fact that it is partition tolerant. This means that every key manager instance can keep functioning even if the network connecting the key managers and users is partitioned. During a network partitioning, the synchronization of keys between the key managers from different partitions is not possible. This means that when the network is restored, the state of the key managers from both sides of the partitioning should be merged. If on both sides of the partition a new key is generated for the same topic, the current key for a topic conflicts. Longer periods of a disconnected network means not only the current key conflicts but all keys generated after the network problems started are different.

To guarantee eventual consistency the usual data types require consensus and/or rollback on updates that conflict because they are executed concurrently at different replicas. Conflict-free replicated data types are found to provide partition tolerant storage of data. Conflict-free replicated data types (CRDTs) are specially defined data types preventing these kind of conflicts [9][22]. CRDTs are used for the storage of keys such that changes on different sides of a partitioning do not lead to conflicts when the network is restored.

4.4.2. Conflict-free Replicated Key Set

The set of keys per topic is defined as a CRDT, this means it is possible to add a key without coordination between the different key managers. Every topic has an individual topic key set (TKS) containing all topic keys for this topic. The topic key set is the data type that is replicated in all key managers. A topic key set is a set with all versions of keys for a certain topic. Conflicts are prevented because every topic key is uniquely identifiable. Since the topic key set is defined as a set and all topic keys are unique, concurrently adding topic keys will not lead to conflicts. Removing a topic key from the set is not allowed because removing a key could lead to data loss as this would make decrypting data in the future impossible without the key. The topic key sets will eventually converge to the same state in all key managers as long as all updates are eventually applied to each of them. The current topic key is defined as the latest created key and if there are multiple keys created at the same time the one with the lowest id of the key manager who created it is the current key. This means that as long as all key managers receive all updates they will eventually have the same current key for that topic. When the network is partitioned and the current keys have diverged, the current topic key will converge as soon as all the updates to the topic key set are synchronized. The only requirement, in this case, is all updates eventually are received by all key managers. Every key manager stores all the topic key sets in a persistence data store to make sure all keys persist after a restart and to prevent the loss of keys.

Figure 4.2 show a typical timeline of two replicas of the same topic key sets in two key managers. Both key managers start with only key TK_1^{1-1} in their topic key set of Topic 1. At some point in time, the two topic key sets diverge because both key managers independently generate a new key because TK_1^{1-1} is expired. The topic key set of Key Manager 1 now contains TK_1^{1-1} and TK_1^{1-2}. The topic key set of Key Manager 2 contains TK_1^{1-1} and TK_1^{2-1}. Until both exchange the topic key sets they return a different key for the same topic. As soon as the key managers exchange their state the topic key sets converge again to the same state. The key sets of both key managers now contain three keys TK_1^{1-1}, TK_1^{1-2} and TK_1^{2-1}.

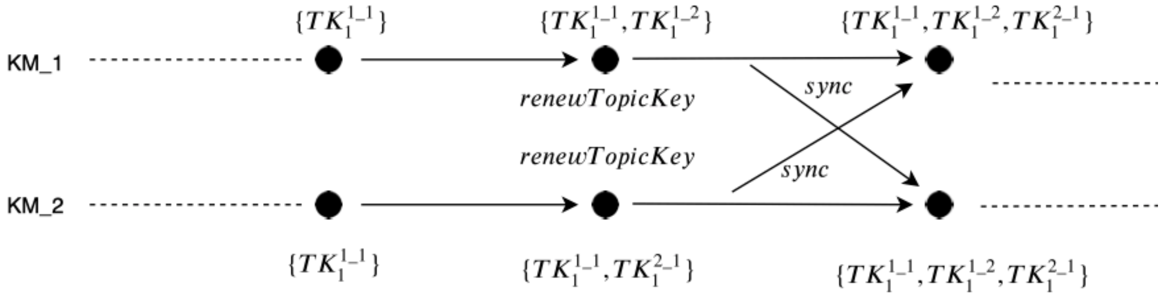

$\{TK_1^{1-1}\}$
$\{TK_1^{1-1}, TK_1^{1-2}\}$
$\{TK_1^{1-1}, TK_1^{1-2}, TK_1^{2-1}\}$
KM_1
$renewTopicKey$
sync
$renewTopicKey$
sync
KM_2
$\{TK_1^{1-1}\}$
$\{TK_1^{1-1}, TK_1^{2-1}\}$
$\{TK_1^{1-1}, TK_1^{1-2}, TK_1^{2-1}\}$

Figure 4.2: Timeline of two replicated topic key sets of Topic 1 in key managers KM_1 and KM_2, illustrating the concept of CRDTs.

4.5. Key Retrieval Protocol

Users send requests for keys to any of the key managers. The key managers process these according to Algorithm 2 to obtain the latest key. On arrival of a key request, the latest key of the requested topic is retrieved from internal memory. If the key is not valid anymore a new key has to be generated. The key manager verifies whether the requesting user is authorized to obtain the requested key before sending the requested key to the user. The protocol $getNewKey(TI)$ to generate new keys is different per approach and these will be explained in Section 4.6.

Algorithm 2 Receiving a get key request from user U_i

```
1: upon receipt of $getTopicKey(TI)$ do
2:     $TK_{TI} \leftarrow getlatestKey(TI)$
3:     if not $isValid(TK_{TI})$ then
4:         $TK_{TI} \leftarrow getNewKey(TI)$
5:     if $isAuthorized(U_i, TK_{TI})$ then
6:         Send $TK_{TI}$ to $U_i$
```

Users can also ask the key management system for keys with a specific version V: Algorithm 3. This is for example used to decrypt data from the past. For older keys, the authorization of the requesting user has to be checked for the time the key was valid. Keys are synchronized between the key managers continuously and not only when a request for a certain topic key is made. This ensures that at the time a key is requested the key can be delivered directly without any contact between the key managers. There are only short gaps in which key managers might not possess the requested key. Just after a new key is generated, the key might still have to be replicated to the other key managers depending on the protocol. Also, after the reconnection of two network partitions when they still have to synchronize to make all keys available everywhere inconsistencies between the available keys can exist. If a requested key is not available the request will timeout and the user has to decide whether it repeats the request or requests another key.

Algorithm 3 Receiving a get specific key version request from user U_i

```
1: upon receipt of $getTopicKey(TI, V)$ do
2:     $TK_{TI}^{V} \leftarrow getKey(TI, V)$
3:     if $isAuthorized(U_i, TK_{TI}^{V})$ then
4:         Send $TK_{TI}^{V}$ to $U_i$
```

4.6. Decentralized Design Approaches for Key Renewal

In the following subsections, three different approaches to design the key renewal protocol in the decentralized topic key management architecture are proposed. The first is the single master approach as described in [15]. The second is the multi master approach which just allows all key managers to generate new keys. The third approach is the synchronized key creation in which the key manager instances first synchronize between them who can create a new key. In the next chapter, the result of an experimental comparison of these approaches is presented.

4.6.1. Single Master Approach

In this approach, new keys are generated by a single master key manager instance per connected cluster. It prevents multiple connected key managers from generating new keys concurrently. The non-master key managers will have to wait for the master to respond with a new key instead of just generating a new key themselves. This requires to elect or determine a master within the cluster. This is defined as the oldest member currently in the cluster. All key managers keep track of the age of the other key managers. As soon as the current master key manager is marked as unreachable all key managers directly know who will be the new master. A non-master key manager will only ask the master for a new key if the current latest key it knows about has expired. When the network is partitioned a new master is defined in the same way in each partition as the oldest member in these new clusters. This means during a network partitioning multiple masters exists as long as some key managers are disconnected from the others. On a reconnection of two disconnect partitions, the oldest master of the two will be the only master again. As soon as a non-master key manager needs a new topic key it sends a topic key request to the master. The master generates a new key or returns a valid key if it already generated a new key for this topic. New keys are written to a majority of other key managers before it is returned to a user to ensure keys are not lost if a single key manager crashes. Being the master key manager is only a role which means that if the key manager is not available another key manager can easily take over. This limits the time in which no new keys can be generated and thus improves the availability of the whole system.

Algorithm 4 Generate new key - Single Master Approach

```
1: procedure getNewKey($TI$)
2:     $Master\_ID \leftarrow getMasterID()$
3:     if $OWN\_ID = Master\_ID$ then
4:         $TK_{TI}^{V} \leftarrow generateNewKey(TI)$
5:     else
6:         $TK_{TI}^{V} \leftarrow askMasterForNewKey(TI)$
7:     Return $TK_{TI}^{V}$
```

In Figure 4.3 a typical scenario is shown for an example with four key managers and five users. KM 3 is asked for the black topic (1) but the current black key it has is expired. KM 3 asks the master KM 1 for the black key (2). KM 1 generates a new key and returns this key to KM 3 (3). In the last step, the new black topic key is returned to the user who requested this key (4).
This approach limits the complexity and theoretically limits the number of keys generated during normal operations in exchange for some reduced availability. The theoretical performance in terms of messages is for this approach constant but the expected delay will be higher compared to a centralized key management solution. The reason for this is the message exchange required when a key is requested from a non-master key manager and it is expired.

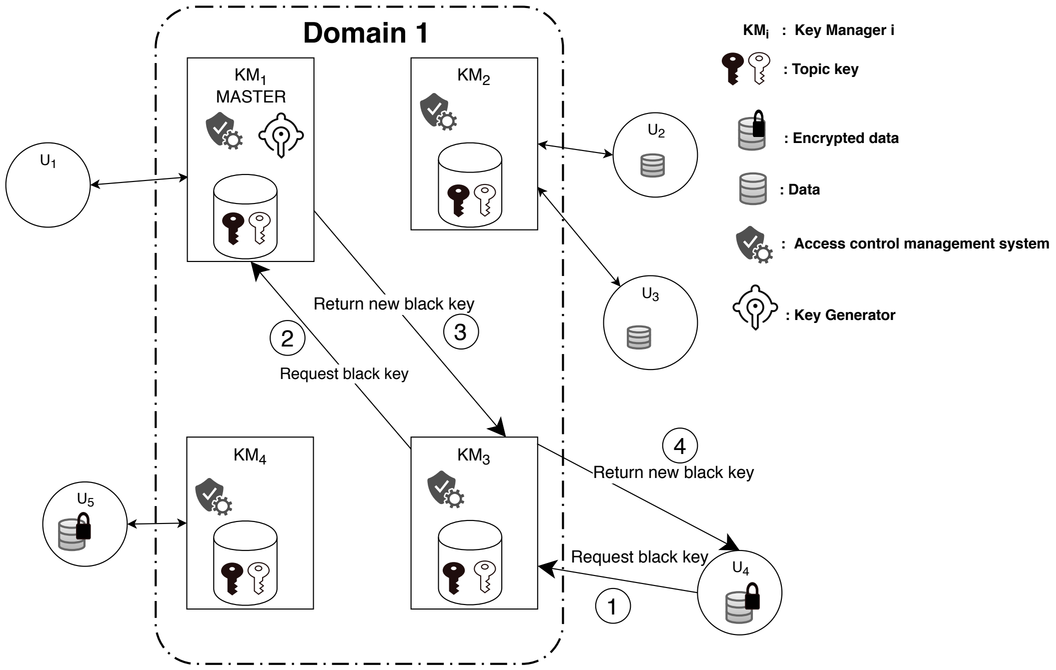


Figure 4.3: Example scenario of the Single master approach.
In this case with four key managers and five users. The current black topic key in KM_3 is expired and a new key has to be requested from the leader KM_1.

4.6.2. Multi Master Approach

In this approach, all key managers are allowed to generate new keys. As soon as a key manager receives a topic key request but the latest key is expired the key manager will generate a new key. This happens independently without first contacting any other key manager. Created keys are written to a majority of other key managers before it is returned to a user to ensure keys are not lost if a single key manager crashes. The new key is communicated to all other key managers with the earlier described gossip protocol after the key request from the user is answered. The potential downside of this approach is that at the same time another key manager could also be generating a new key for the same topic. This leads to a short period in which two different topic keys are used concurrently by different users. This will be solved as soon as the new keys are synchronized to all key managers because of how the topic key set is defined. It however also requires to notify all users about the fact that the current key is changed. The key is not expired yet so they will otherwise keep using the just obtained key (assuming the users cache the keys) leading to more data encrypted with different keys.

Using multiple keys for the same topic will not lead to any conflicts because of how the data types are defined. It could, however, have a negative impact on the overall system. Users who want to read data from the same topic but the data is generated by two different users who used other keys will also need two different keys to decrypt the data. The user has to retrieve two keys and potentially have to wait until both keys are available in all key managers. It also requires extra storage, since as soon a key is used for some data, it has to be kept. Data could otherwise be lost because it could not be decrypted anymore. During network partitionings, it might be impossible to prevent different topic keys to be concurrently used for the same topic. It is impossible to sync the topic keys between two different network partitions. It is however desirable for these reasons to reduce the number of concurrently used keys for the same topic as soon as possible. The advantage of the multi master approach is that it will theoretically have the best performance in terms of latency for the users. There is only a bit added latency due to writing the keys to a majority of other key managers during a key request compared to a centralized key management solution without redundancy.

Algorithm 5 Generate new key - Multi Master Approach

```
1: procedure getNewKey($TI$)
2:     $TK_{TI}^{V} \leftarrow generateNewKey(TI)$
3:     Return $TK_{TI}^{V}$
```

4.6.3. Synchronized Key Creation

Besides the approaches of a single and multi master the third considered approach is synchronized key creation. The idea to synchronize between key managers before performing an update comes from [16]. The authors in [16] use a group communication system that provides totally ordered delivery of messages between key managers. For this thesis, we only use synchronized creation of keys. Key managers first have to notify all other key managers in the connected cluster about the fact that they want to renew a key. Only if the other key managers acknowledge this the key manager will generate a new key. This protocol prevents key managers connected to the same cluster to generate new keys at the same time and thus improves on consistency compared to the multi master approach. It improves on availability compared to the single master approach by not only allowing one key manager per connected cluster but all of them to generate new keys. The disadvantage of this approach is the increased complexity of generating new keys.

New keys can be obtained in any key manager with Algorithm 6, a key manager starts with sending the message $renewKeyRequest$ to all other key managers in the cluster. This message contains the timestamp of the request and the id of the requesting key manager. The other key managers can respond with an acknowledgment ($renewAck$) or wait with replying because it is renewing the key of this topic itself. Only when a key manager has received an acknowledgment from more than half of the key managers in the connected cluster it will create a new key. As soon as a key manager has sent an acknowledgment it will not send a $renewKeyRequest$ itself till it has received a newly created key. This new key will be received as soon as the key manager where the acknowledgment is sent to has generated a new key and this key is distributed. This decision is based on whether it has an outstanding request with an earlier timestamp itself to renew this topic key. If a key manager has an earlier outstanding request it assumes it will be able to renew the topic key itself first so it will not send an acknowledgment. This prevents the other key manager from also creating a new redundant key at the same time. The request to renew a topic key will, in this case, be answered by sending the newly created key to the requesting key manager. This makes any outstanding request to renew a topic key unnecessary and the key manager can answer the users who requested the topic key. Keys are written to a majority of other key managers before they are returned to a user. This ensures keys are not lost if a single key manager crashes.

Algorithm 6 Generate new key - Synchronized Approach

```
1: procedure getNewKey($TI$)
2:     Send $renewKeyRequest$ to all $n$ key managers in the cluster
3:     Wait for $n/2$ $renewAck$ or reception of new key for topic $TI$
4:     if new key for topic $TI$ received then
5:         $TK_{TI}^{V}$ ←received key
6:     if $\#renewAck > n/2$ then
7:         $TK_{TI}^{V} \leftarrow generateNewKey(TI)$
8:     Return $TK_{TI}^{V}$
```

Messages between key managers could get lost or key managers could be disconnected during this key renewal procedure. To prevent the system from getting stuck into waiting for other key managers to create new keys a predefined timeout is used. If the timeout expires any key manager can start the key renewal procedure itself again although they already send an acknowledgment. It will have an impact on the response time to start the key renewal process again but at least the long term availability is not affected. Two key managers could send a $renewKeyRequest$ at the same time to all key managers. There is only one key manager who will send an acknowledgment in this case, the one that received a message with the lowest timestamp or if they are equal the one with the lowest id.

The number of messages that have to be exchanged before a key can be renewed is for this approach the highest. This will have an impact on the response time, especially in a situation with a relatively high network delay. With n key managers the order of messages required is $O(n)$.

Figure 4.4 illustrates the protocol with an example with four key managers and five users. User U_4 requests the black topic key from KM_3. KM_3 has to renew the key for this topic because it is expired. KM_3 is only allowed to renew this key if the other connected key managers have acknowledged this action. KM_3 sends a $renewKeyRequest$ to all other key managers (2). The other key managers respond with a $renewAck$ message (3).

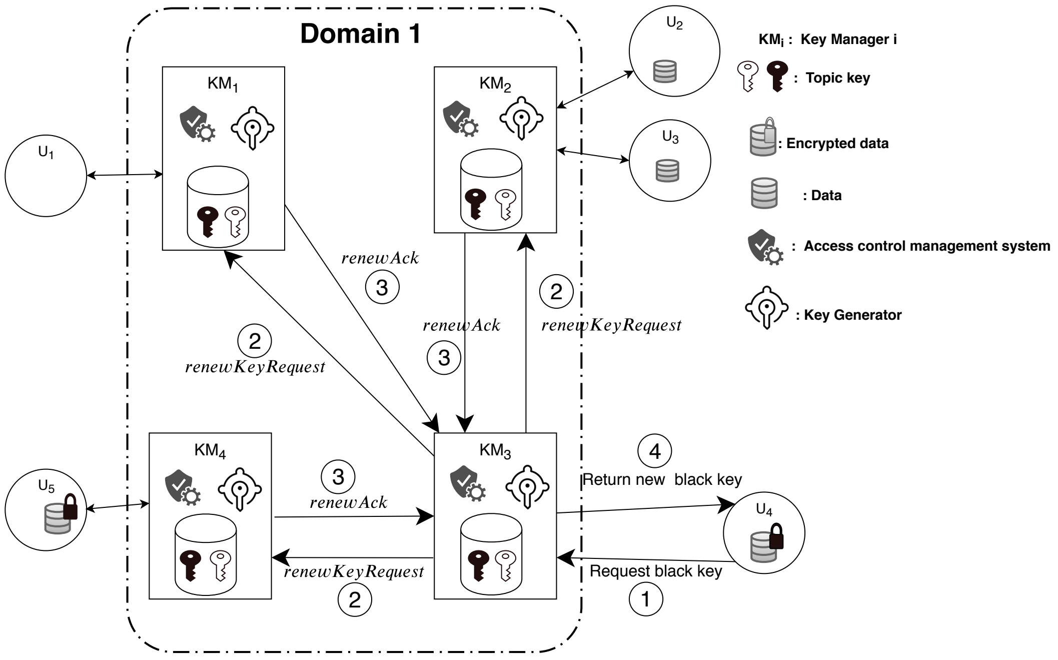


Figure 4.4: Example scenario of the Synchronized key creation.
In this case with four key managers and five users. KM_3 renews a topic key by first notifying the other key managers.

4.7. Confidentiality and Integrity between Key Managers

Keys are important assets for cryptographic solutions, their exposure compromises the complete security of a cryptographic solution [19]. The key managers possess all keys making them an attractive target for attackers. By decentralizing the key management the attack surface of the keys also increases. There are more locations with all the keys and only one of them has to be compromised to obtain all keys. There exist many solutions such as hardware security modules (HSMs) to protect the keys at all these locations. Another risk is the synchronization of keys between the key managers. The communication of keys between key managers has to be secure. The key managers verify each other's identity using pre-installed certificates. Transport Layer Security (TLS) with mutual authentication is used to securely encrypt the communication between the key managers. The only way to compromise the communication is by breaking the encryption algorithm or compromising a valid certificate in any of the key managers. Mutual authentication means that both sides of the connection verify the certificate of the other side. This is necessary to make sure keys are only received from and sent to legitimate other key managers. It prevents an attacker from impersonating key managers and stealing all cryptographic keys.

5 Implementation and Analysis

In Chapter 4, the design of our decentralized key management architecture where users can retrieve topic keys is presented. Additionally, three different approaches for key renewal are proposed. In this chapter, a validation of the requirements set in Section 1.2.1 is provided. We also show the feasibility by creating a proof-of-concept implementation. We look at the performance of the complete system as well as compare the different approaches with each other. The proofs-of-concept are tested on a simplified simulation scenario of the problem use case introduced in the introduction as well as a different more dynamic use case. This chapter starts with the implementation details of the proof-of-concept. Then to be able to compare the different approaches different metrics have to be defined. Finally, we briefly discuss the results of the analysis of the decentralized key management architecture and check whether it fulfills the requirements.

5.1. Implementation Details

We created a proof-of-concept implementation of the relevant elements of the architecture. The focus of the architecture is on partition tolerance and not on the key generation and authentication. Therefore only the relevant parts necessary to compare the different approaches are implemented.
The architecture is implemented in the Java programming language (version 8). Building everything from scratch would be out of scope for this project, therefore, the implementation is based on AKKA[1] a framework to build resilient distributed applications. AKKA is a Java framework to simplify the construction of concurrent and distributed applications. It provides tools to build highly concurrent, distributed and resilient message-driven applications.

For the execution of different test scenarios Docker[2] and Docker Compose[3] are used. Docker is used to virtualize the operating system on which the key managers run. Docker Compose provides some tools to configure and start multiple Docker containers at once and connect

[1]https://akka.io/
[2]https://www.docker.com/
[3]https://docs.docker.com/compose/

them with a virtual network. With Docker Compose we limit the resources to simulate more realistic network delays. The key point of the architecture is to be able to handle situations in which the network is partitioned and there is no communication possible between key managers. To be able to simulate this during the test scenarios we used Blockade[4]. Blockade is a utility to simulate network partitionings and failures in distributed applications. It manipulates the virtual network created by Docker Compose in any possible partitioning.

5.2. Analysis of Implementation Aspects

The architecture uses several concepts that are open for interpretation or assumed to be implemented. These concepts could have an impact on the simulation results presented in the next section. They can also be adjusted to improve the performance but this will depend on external factors such as the network conditions, the expected number of partitions and the maximum acceptable delay. For the experiments, we keep all conditions the same to have a fair comparison.

The gossip interval can, for example, be defined such that keys are synchronized between key managers more quickly. A smaller interval however also means an increased load on the network as more messages have to be exchanged between the instances.

Another variable is the timeout before a key manager is marked as unreachable. A lower timeout might lead to more false positives of key managers who were just slow in responding but are not disconnected. A higher timeout makes the system less responsive to failures and could lead to increased delays.

The results are influenced by the validity period of keys because a short validity time means the keys have to be renewed more often. Renewing a key is slower than just returning the latest key from storage especially with the single master and synchronized approach because communication between the key managers is required. It also influences the total number of keys created during the simulation. For every experiment, we keep the key validity period the same to be able to compare the different approaches. The key validity period is assumed to always be longer than the time it takes from creating the key to delivering it to a user otherwise the delivered key cannot be used.

[4]https://blockade.readthedocs.io/

5.3. Metric

Evaluating the performance and resilience of the key management architecture is important to be able to compare the trade-offs between availability and consistency of the different approaches. The results can also be used to prove the fulfillment of the requirements.
To come up with some metric we looked at how the performance is measured in decentralized distributed data systems. In [21] a framework is presented to compare the availability of different distributed data management systems. The availability is affected by two conditions: Number of requests made by clients concurrently and failures impacting the network connectivity or availability of data items. Input parameters used in [21] to evaluate the availability are single vs multi master, partitioning, number of replicas, requests per second, number of data items and the recovery specification. The metrics to measure the experienced availability are time until the failure is being masked, the amount of failed requests due to data unavailability, the latency of requests and accessibility. This framework is built upon the concepts of [25]. The idea from this work is to improve resilience by increasing the frequency and variety of failures, accomplished by manually introducing failures regularly to simulate real-world failures. The goal is to practice the response and identifying the results. The idea is that by having these failures all the time in the background they blend in without any impact. The concept is to have “monkeys” (autonomous agents) introduce different kinds of failures. They can, for example, introduce network partitions such that no communication is possible between partition, terminate some random instances or delay messages.
In [13] the effectiveness of different data replication solutions is evaluated using the following metrics: (1) the average system throughput, defined as the average number of transactions that are completed per unit of time. (2) the average transaction response time which depends on if the requested data is already available in a replica or it has to wait until it is available. (3) the average storage space.

Based on the ideas described in this section, we can define three metrics to test the differences between different approaches in our decentralized key management architecture:

1. Response time of key requests
2. Number of different keys for the same topic
3. Time until all key managers are synchronized after a network partitioning

The first metric: **Response time of key requests** indicates the availability of the topic keys. It is measured by the time it takes to retrieve a topic key from the perspective of a user. This response time is measured from requesting a key until a response is received. The average response time per key manager is taken as a measurement to have an overall indication of how long users have to wait for a key. When a user is disconnected from any key manager the response time will be at least as long as the duration of the disconnection. Low response time is preferred as this means users have to wait for a shorter period before they receive a key and can continue the encryption or decryption. This should both be measured under different failure conditions and with a varying number of replicated key managers.

The second metric: **Number of different keys for the same topic** measures the consistency. It is tested by counting the number of different keys for the same topic under different conditions. It will be measured through the number of keys for the same topic that are created in the complete system over a certain period. In the ideal case, all key managers have the same view of existing keys all the time. The total number of keys created in period t is then equal to the duration of period t divided by the validity period of keys. There will never be a new key created if there is still a valid key somewhere in the system. This is however in a real system not possible as we have to deal with network partitioning. It is impossible to use the same keys in all key managers when the network is partitioned referring to the CAP Theorem [10]. Ensuring consistency all the time is not possible and some inconsistencies have to be accepted. During a network partitioning on both sides of the partitioning new keys have to be created to continue the encryption of new data.

The third metric: **Time until all key managers are synchronized after a network partitioning** indicates how resilient the architecture is. The time it takes to restore from the network partitioning and time until all keys in every key manager are in sync again after a partitioning is measured. This is an important metric as it will directly impact the availability and consistency if it takes a long time before the keys are synced again to all key managers.

5.4. Use Cases

In Section 1.1, we introduced the problem use case this thesis is based on. It presents a real-world use case where the proposed key management architecture would be useful. We will refer to this use case as the Naval "Two Island" problem. To test whether the proposed architecture presents a general solution that could also be used in other use cases, a second use case is introduced in this section. Both these use cases are used to base the simulation scenarios presented in the next section on.

The second use case is the intra-vehicle information sharing use case. Vehicles for military and civil crisis management are being connected to increase interoperability, information sharing and to enhance situational awareness. More specifically unmanned drones are connected to armored personnel carriers to share information such as video, locations, and other sensor data. In this potentially adversarial situation, information must be kept confidential. Key management is therefore required to be able to encrypt and decrypt the data. The network in this use case is frequently changing what makes the key lifecycle difficult to manage. A central key management system is therefore undesirable. The key management system must ensure keys to remain available for the entire lifetime of the information.

5.5. Simulation Definition

Experimenting with the proposed approaches of the key management architecture in a real environment is not possible. In a real environment, there is a lack of ability to control the environment and obtain reproducible results. It is also difficult to keep track of what events occurred when. Therefore we defined two simplified simulation scenarios based on the two use cases described in the previous section. In the simulations, a virtual network is set up with the key managers. This network is changed in a specific order to simulate network disconnects. During the experiments, users are simulated by constantly doing requests to a specific key manager. This triggers the key managers to create new keys if the keys are expired. If possible, the key managers will synchronize the keys to the other key managers. For the first simulations, we used a setup with four key managers indicated as KM 1, KM 2, KM 3, and KM 4. We choose to test with four key managers although we could have used any number of key managers as long as there are enough resources available to run all docker containers.

The different elapsed times are measured with the Java $System.currentTimeMillis()$ method and extracted from the logs from the Docker instances running the key managers. In the next two subsections, the specifications of the two simulation scenarios are described.

5.5.1. Scenario 1

This scenario is based on the Naval “Two Island” problem use case described in Section 1.1. In this use case, the expected time between network partitionings is in the range of days or weeks and the expected duration ranges from minutes to days. During the simulation scenario, there will be only one network partitioning that takes fifty seconds, this is long enough for the system to stabilize. This means the key managers in both partitions are functioning such that there are two independent key management systems. The keys in the key managers in each partitioning are synced with each other. During the complete simulation, we send every five seconds a key request for the same topic to all key managers. The keys are valid for five seconds, this means new keys have to be generated for every request. Unless another key manager generated a new key and this key already arrived at the other key managers. This simulates the fact that users will request a new key as soon as the key they currently have is expired. We chose five seconds as we determined experimentally that this is long enough for a new key to be generated and arrive at the user when the network is connected. A longer period was not chosen as this would increase the total duration of the experiment and thus the number of repetitions we could do within the duration of this project.
During this simulation, the network partitioning is created after fifty seconds such that there are first some keys created and distributed between the key managers during a connected situation before the partitioning. The network is partitioned such that half of the key managers are in one partitioning and the other half are in the other partition. This could have been any other configuration but this nicely shows how after a network partitioning two independent systems are created. The duration of network partitioning is defined as fifty seconds. This is enough time to stabilize after the partitioning occurred and it shows how some new keys are generated independently on both sides of the partitioning. After another 125 seconds, the simulation is stopped which makes the total duration of the simulation 225 seconds. We will refer to this scenario as Scenario 1.

5.5.2. Scenario 2

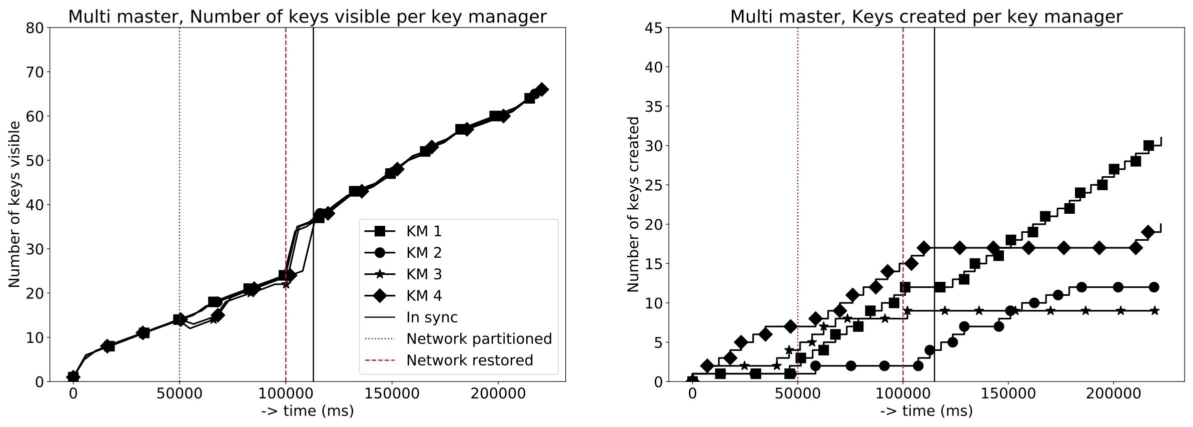
This scenario is based on the Intra-vehicle Information Sharing use case described earlier. In this use case, partitionings occur in the range of minutes and last for seconds or minutes. During the simulation, the network configuration is changed every 15 seconds. We choose 15 seconds as this is in most cases just enough for the system to reestablish a cluster with the still connected key managers. If the frequency of network connection changes would be higher most of the key managers are constantly operating on their own as they do not have enough time to stabilize. At the start, all key managers are connected. After every next 15 seconds, the key managers that are connected with each other are randomly selected. This is done by randomly picking a number of partitions and randomly selecting which key managers are connected both from a uniform distribution. In this way, a highly dynamic environment is simulated where the connections between key managers are constantly changing. The total duration of the complete simulation is the same as in the previous scenario so the simulation is stopped after 225 seconds. This means the network connections are changed 14 times.

The keys are again valid for five seconds, and every five seconds we send a key request for the same topic to all key managers. A uniform random delay of at most thousand milliseconds is added to each request. This is done to make the simulation more realistic and to give the system some time to synchronize the keys. The same network partition configurations are used in every run of the simulations to make a fair comparison between the results. We will refer to this scenario as Scenario 2.

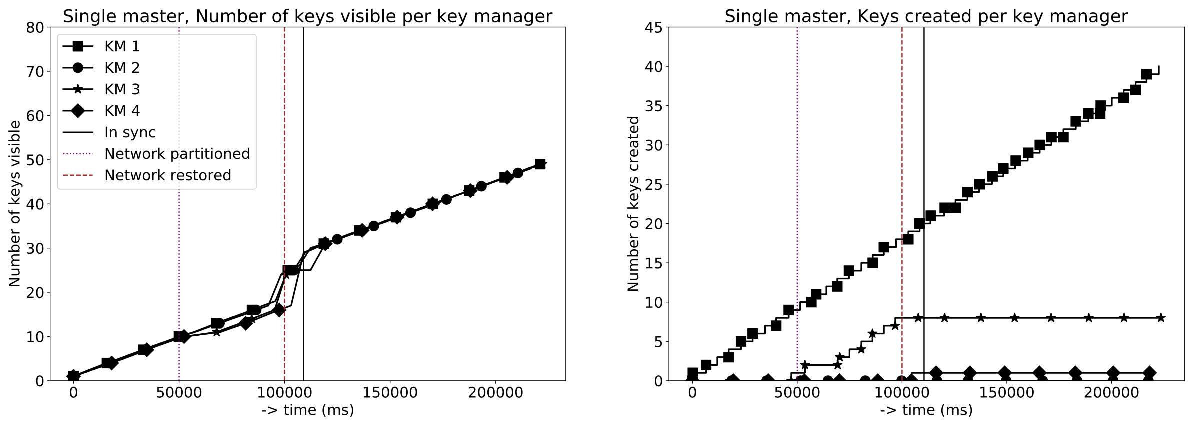
5.6. Simulation Results

Figure 5.1 shows the number of keys created per key manager as well as the total number of keys visible from the perspective of every key manager for one run of Scenario 1. For this experiment, four key managers are used indicated as KM 1, KM 2, KM 3 and KM 4. The three plots of the different approaches verify the correct behavior during the network partitioning and after the network is restored. In the multi master approach, all key managers are creating new keys. The graph shows that some are creating more keys as others what can be explained by the fact that keys are synchronized between them. If one key manager has already created a new key that was distributed to another key manager before a new request came in this key manager is not required to create a new key. For the single master approach, the graph shows that only during the network partitioning two key managers are creating new keys. As soon as all key managers are connected again there is only one of them continuing with creating new keys. The first vertical line indicates the creation of the partitioning of KM 1 and KM 2 in one partition and KM 3 and KM 4 in the other partition. The second vertical line indicates the moment when the network is restored. The last vertical line is the moment when all the keys are synchronized again in all key managers.

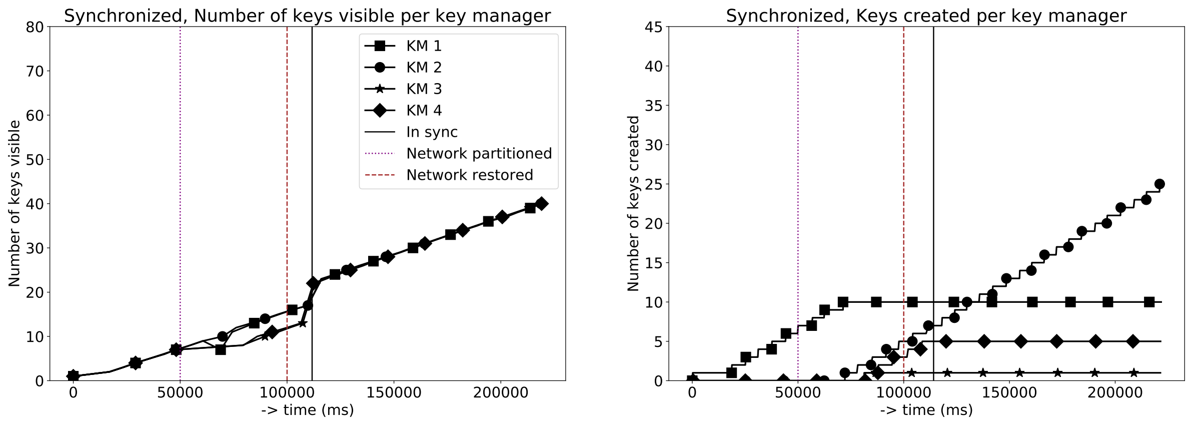
In the period between restoring the network and the moment all keys are available in all key managers, there is a relatively large increase of visible keys in all key managers. This is due to the keys created in the other partition of the network that could not be synchronized before. For the single master approach, we see that after the network is restored there are only new keys created by KM 1 as expected. During the network partitioning, there are only keys created by KM 1 and KM 3, the masters in each partition. In the synchronized approach keys are created in every key manager but not concurrently. It shows nicely that the key managers alternate in creating new keys. Comparing the total number of keys in the three approaches reveals a difference between the multi master approach (69) and the other two approaches, synchronized key creation (42) and single master (50).



(a) Multi Master



(b) Single Master



(c) Synchronized key creation

Figure 5.1: Keys created per node during Scenario 1.

The single master and synchronized approach are, thus, able to reduce the number of created keys significantly. The advantage of fewer keys is greater consistency over the whole system and between the users. There is, however, a trade-off for this consistency: Figure 5.2 shows the average response time for the three approaches during Scenario 1. This response time is measured from requesting a key until a response was received. The test scenario is executed ten times with eight users for every approach. The average response time from the perspective of every user is taken as a measurement. The average response time for the multi master approach is 292 ms, single master 742 ms, and the synchronized approach 1530 ms. Lower response times to retrieve a key are preferred as this means users are blocked for a shorter period before encryption or decryption can continue. The increased response time for the synchronized approach is significant and can be explained by the fact that key managers have to wait on all other key managers to respond before a new key is created. In the single master approach, a key manager only has to wait for one other key manager to create a new key before it can respond to the user. The lowest response time is for the multi master approach as expected since there is no waiting required before new keys can be created.

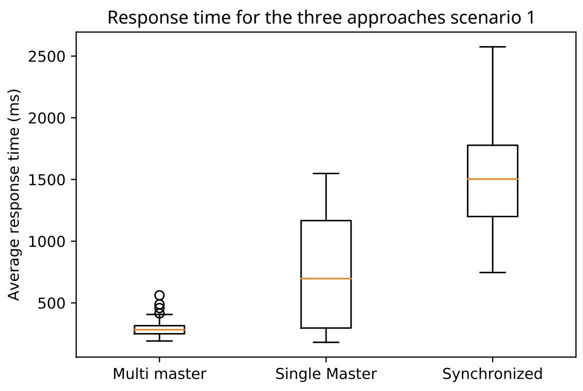


Figure 5.2: Average response time of the three approaches in Scenario 1.

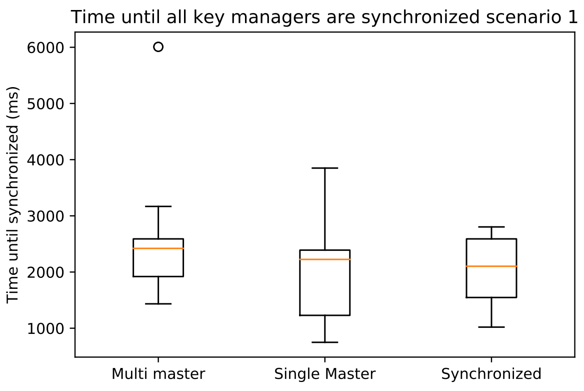


Figure 5.3: Time until all key managers are synchronized in Scenario 1.

In Figure 5.3, the time it took for all key managers to be synchronized after the network was restored in Scenario 1 is plotted. The scenario is executed ten times for every approach. The time it took to synchronize after a network partitioning is determined based on the logs. All keys created before the network is restored are noted. The key managers are assumed to be synchronized when all these keys are visible in every key manager. The difference in results for the different approaches is not significant. This is expected as communication takes some time but the key synchronization settings are the same for all approaches. The multi master approach contains a greater number of different keys and thus more data to synchronize between key managers but since the total duration of the simulation is low this had no impact. For future research, it would be interesting to look at the differences after longer simulation periods.

Up to now, we looked at the results for the different approaches applied to Scenario 1. We hypothesize that the results of the different approaches will differ depending on the use case. In Figure 5.4 the average response time for the three approaches during Scenario 2 are presented. The response time is measured in the same way as for Scenario 1. The average response time for the multi master approach is 484 ms, single master 8272 ms, and the synchronized approach 8187 ms. The interesting part to notice is that the single master approach and synchronized approach have almost the same average response time during Scenario 2. During Scenario 1 the average response time for the synchronized approach was almost twice the average response time for the single master approach. This difference can be explained by the dynamic network behavior in Scenario 2; this will result in a constantly changing master, which leads to higher response times. It takes some time to determine the new leader while for the synchronized approach a key manager can directly start the process to renew a topic key on its own. Also for this scenario, the multi master approach has the best performance in terms of response time.

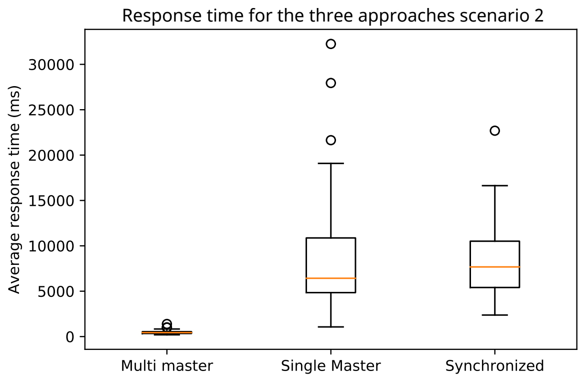


Figure 5.4: Average response time of the three approaches in Scenario 2.

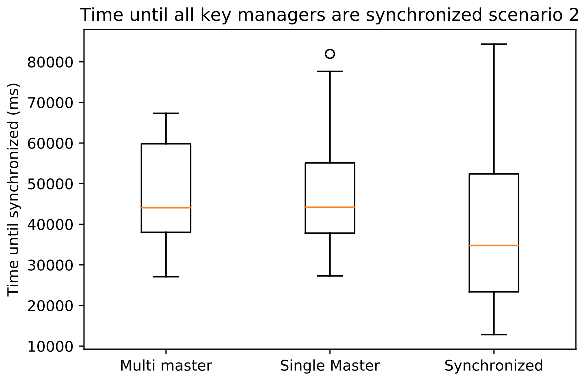


Figure 5.5: Time until all key managers are synchronized in Scenario 2.

Again the average response time is not the only relevant metric when comparing the three approaches. In Figure 5.6 the number of keys used per approach and scenario are plotted. The average number of keys for Scenario 1 are 84, 44 and 46 for multi master, single master and synchronized respectively. For Scenario 2 the average number of keys are 257, 112 and 100 in the same order. The difference between the single master approach and synchronized approach is not very large for both scenarios, but based on the result on the number of keys, the single master approach is the best option for Scenario 1. For Scenario 2 the synchronized approach is the better option with these criteria.
Selecting the best option considering both the average response time as well as the number of keys will depend on the required consistency versus the required availability. The multi master approach provides a lower response time but also introduces more inconsistencies by using more different keys for the same topic. The single master approach is a simple manner to introduce some synchronization between the key managers, but might be too simple for the highly dynamic network behavior from Scenario 2. The synchronized approach is more complex, but might be a better fit for use cases like simulated in Scenario 2. It keeps the number of keys low and does introduce the lowest delay compared to the other approaches.

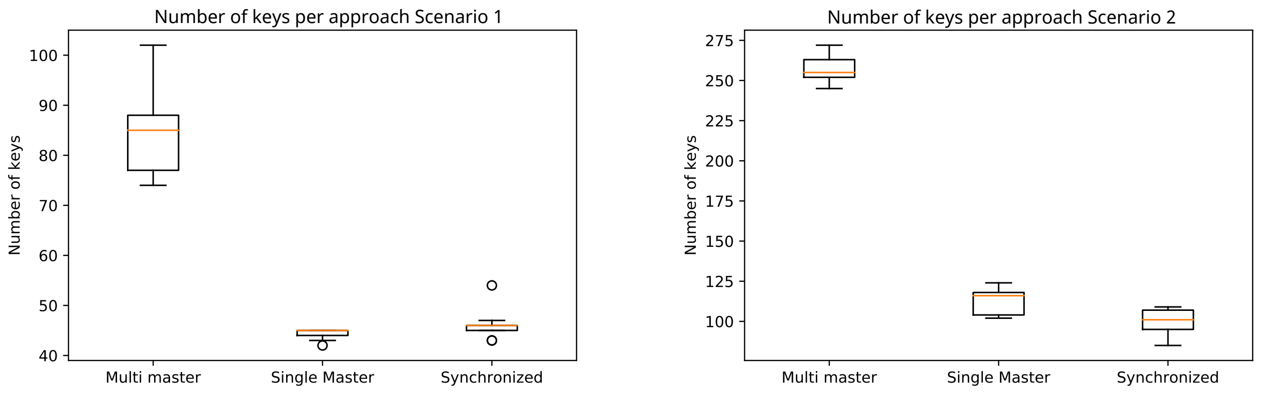


Figure 5.6: The number of keys used per scenario and approach.

We also examined the relationship between the number of key managers deployed to the total number of keys used and the average response time for the two scenarios. In Figure 5.7 the results are plotted. The scenarios are the same as before but this time we have a fixed number of ten users attempting to obtain a new key every five seconds. The users are individual instances in the simulated network. They start by asking a random key manager for keys and switch to the next key manager if that request fails. For Scenario 1, half of the users are connected to half of the available key managers during the network partitioning. This means that when there is only one key manager, half of the users are disconnected from every key manager. This explains the relatively large average response time for the first measurement in the graph of Scenario 1. Half of the users have to wait for the complete duration of the partitioning before they receive a new key. With at least two key managers there is at least one key manager in each partition during Scenario 1. The large peak for the single key manager is not visible in Scenario 2 because the network partitioning took less time, so they had to wait less for new keys.

For Scenario 2, the ten users and a varying number of key managers are partitioned in a random number of network partitions every 15 seconds. This means that during the simulation there could be users disconnected from every key manager, but the chance this happens decreases with the increasing number of key managers. To which users key managers were connected during the simulation was random. This explains the non-decreasing response time for the increased number of key managers. The key managers were not evenly distributed over the partitions with users. In an actual real-world scenario, the key managers would be placed at strategic physical locations, but the network can be partitioned in any way. This also means that in a real situation the key managers will not be evenly distributed over all network partitions.

For both scenarios, the multi master approach has again the largest number of keys. The difference for Scenario 2 is, however, the increase in the number of keys when the number of key managers increases. For Scenario 1, it also increases slightly, but not as much as for Scenario 2. The constantly changing network configuration in Scenario 2 is causing slower synchronization between the key managers. New keys are, thus, not available as quickly in the other key managers as during optimal circumstances. Key managers then have to generate a new key instead of using one that was previously generated by another key manager. This leads to an increase in the total number of keys during Scenario 2, with an increasing number of key managers.

Another interesting result is the increase in the response time for all approaches during Scenario 1 for more than four key managers. Scenario 1 is the scenario with just one partition for a longer period. If the key managers are placed strategically in the network the availability can be guaranteed with just two key managers, one in each partition. The increased response time is caused by the increased overhead of more key managers. All the key managers have to form a cluster and synchronize the keys to all other key managers. The synchronized and single master approach also have an increased delay due to more communication before keys can be created.

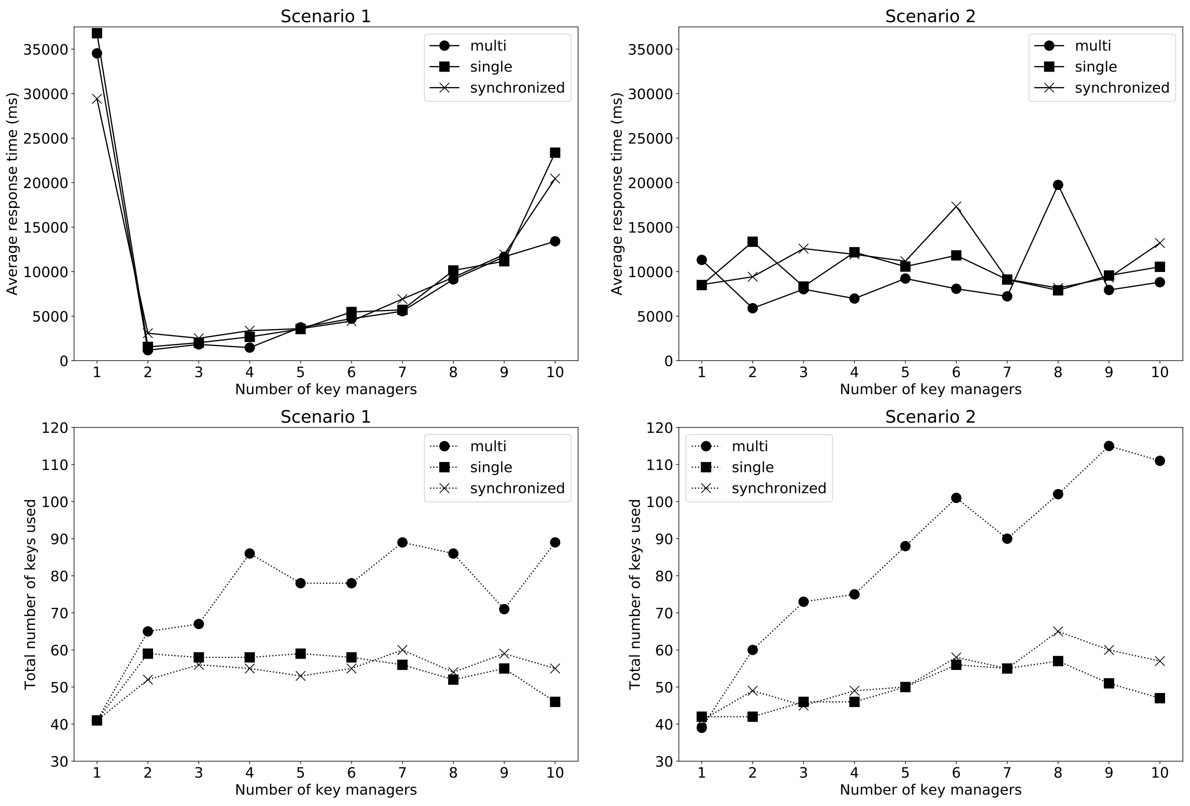


Figure 5.7: Total number of keys and average response time for the two scenarios.

Concluding the analysis of the three approaches, we have shown there is a trade-off between availability and consistency within the three approaches. For partition tolerant key management more consistency means there are fewer different keys used during the same period. The availability is expressed in terms of the response time from the key managers on key requests. A better availability means a lower average response time. The multi master approach has the best availability for both scenarios. It, however, performed the worst in terms of consistency in both scenarios. The single master approach performed worse in terms of availability compared to the multi master approach but offers better consistency. The same holds for the synchronized approach. The difference between the synchronized and single master approach became clear with the second scenario. The combination of the response time and the number of keys used for Scenario 1 is better with the single master approach. During Scenario 2, the performance was better for the synchronized approach. This difference can be explained by the more dynamic behavior of the synchronized approach. During Scenario 2 the network is constantly changing, which means having a static single master is less beneficial. During Scenario 1 there is just one network partitioning which takes a longer period giving the system more time to adapt and continue. The synchronized approach has the advantage that every key manager can create new keys directly if everyone in the connected cluster agrees. Finally, we looked at the performance with a varying number of key managers. We started with a single key manager which mimics the current centralized situation. Especially in Scenario 1, it showed clearly that the long waiting time for users in the other partition as the key manager. These users had to wait for the partitioning to be healed before they could receive a new key. In Scenario 2, this is less clear because the duration of the partitioning is shorter.

5.7. Reflection on Requirements

This section presents how the requirements from Section 1.2.1 are fulfilled by the functionality of the key management architecture.

1. *The keys should be stored in n locations and can be retrieved by users at all these locations*:
   The presented architecture consists of n replicated key managers which all store all keys. They provide all the same functionality to the user. The keys created in one location are synchronized to the other key managers as soon as possible.

2. *The architecture should allow the delivery and generation of keys during a network partitioning and recombine the keys as soon as the network is restored*:
   All key managers can work independently. This means, during a network partitioning keys can be created and delivered even if the individual key managers are not connected. Section 4.4 provides the protocol to recombine the keys when the network is restored.

3. *The key management should be independent of the users of the keys*:
   The key management architecture is designed such that the management of the keys is accomplished by a separate infrastructure independent of the users. The users have no influence on the generation and delivery of the keys. This ensures access policies can easily be enforced and adapted.

4. *Key rollovers should be requestable in all key managers*:
   This requirement is fulfilled by allowing all key managers to generate new keys. In the synchronized and single master approach this is more complicated as it first requires some synchronization but eventually they are all able to generate new keys.

5. *The key de-synchronization between key managers should be taken into account and resolved automatically*:
   The de-synchronization between key managers is solved by using Conflict-free replicated data types. These data types prevent conflicts between the key managers and allow concurrently generation of new keys.

6. *There should be no single point of failure during normal operations*:
   All components of the architecture are available in at least two locations. The keys are stored in n locations and the initialization and formation of the cluster of key managers are accomplished without any central component.

7. *Keys should be persistent and never get lost*:
   The definition of the data type of the topic key set does only allow adding keys and not removing. This automatically ensures keys are not lost. After generating the keys they are stored in persistent storage of the key managers before they are sent to the users.

6 Conclusion

Key management is a critical link in sharing data between all entities in a military environment in a secure way. In tactical networks, key management becomes challenging due to link disruption and network partitioning. In this thesis, we researched the field of key management architectures in this environment. We presented a decentralized key management architecture based on Conflict-free replicated data types. By using Conflict-free replicated data types conflicts between the key managers are prevented and the availability of keys is ensured.
In this chapter, we reflect on the research questions and discuss how the research goal is achieved by answering the research question. Lastly, we discuss improvements and possible future work is identified.

6.1. Reflection on the Research Question

The main research question presented in Section 1.2, is as follows:

"How can a decentralized key management architecture for content-based encryption be designed in a conflict-free way to tackle the key de-synchronization problem that facilitates the needs of a multi-domain environment?"

We broke the research question down into 3 sub-questions:

- "How to design a decentralized key management architecture for content-based encryption in a multi-domain environment?"
- "How to prevent conflicts in a decentralized key management architecture to tackle the key de-synchronization problem?"
- "How can the performance of a decentralized key management architecture in handling network partitioning be measured?"

For the remainder of this subsection, we will answer the research sub-questions which together answer the main research question. To answer the first sub-question, we looked at the different types of key management schemes described in the literature. They are categorized as centralized, decentralized and distributed key management schemes. A decentralized key management scheme is the most appropriate in a multi-domain environment where information must be shared over a tactical network. A decentralized architecture ensures the access policy can be managed by an organization and decouples the key management from the users. To the best of our knowledge, the only work that proposes a key management architecture addressing a multi-domain environment and network partitioning is in the paper of *F. Poltronieri et al.* [15]. The architecture consists of multiple key managers located at physically different locations. The keys stored in the key managers are redundantly stored in every key manager. The key managers can generate and deliver existing keys independently which ensures the resilience against network partitioning. New keys are generated at a single master key manager and when the network is partitioned a new master in each partition is elected. However, the solution from [15] lacked considering any alternative designs and did not perform any experimentation to evaluate the benefits and drawbacks. We extended the work from [15] by proposing the use of Conflict-free replicated data types. Additionally, two different approaches for the synchronization of key creation compared to the single master architecture from [15] are proposed. These approaches are multi master (Section 4.6.2) and synchronized key creation (Section 4.6.3). In the multi master approach, there is not only one key manager allowed to generate new keys but all key managers can generate new keys independently. In the synchronized approach any key manager is allowed to generate new keys but they first have to receive a confirmation from the other key managers before doing so.

The second sub-question is answered in Chapter 4 by defining the storage of keys as CRDT. The topic keys and the topic key set are defined as CRDTs to prevent conflicts when they change in different key managers. The architecture needs to be partition tolerant, this means we have to deal with a trade-off between consistency and availability as we cannot have both. For the decentralized key management architecture, this means the keys used for the same topic cannot always be the same in the complete system. When new keys have to be available at all times during a network partitioning new keys are generated on both sides. The definition of the topic key set makes the storage of keys to be defined as a conflict-free replicated data type. This data type is specifically defined to prevent conflicts when updated simultaneously on different replicas. It avoids the need for coordination by ensuring actions taken independently by replicas can not conflict with each other. All replicas converge to the same value as long as all updates are applied to each of them eventually no matter the order in which they are applied. This ensures that during a network partitioning all key managers can keep creating new keys for every topic. During network reconnection, keys created on both sides which would otherwise conflict can now without any problems be merged such that all key managers are in the same state again. In this way, the key de-synchronization problem is solved.

The last sub-question is answered in Chapter 5, we measured three different metrics during the experiments. The first one is consistency in terms of the total number of keys used. The second is the availability measured by the average response time from a user requesting a key to receiving a new key. This is important as it says something about the availability of the key managers. The last metric we looked at is the time it takes before all keys in every key manager are in sync again after a partitioning. The approaches are tested on a specific scenario based on the use case introduced in the Introduction as well as on a scenario based on a more dynamic use case introduced in Section 5.4.

In the first scenario, one network partitioning occurs between the key managers during the simulation. To evaluate if the presented architecture can also be used in a more dynamic scenario the second scenario is introduced with a constantly changing network environment. Key managers and users are connected and disconnected constantly. The results made clear there is a trade-off between availability and consistency within the three approaches. More consistency means there are fewer different keys used during the same period. The availability is expressed in terms of the response time of key requests. A better availability means a lower average response time. The multi master approach has relatively the best availability for both scenarios. It, however, performed the worst in terms of consistency in both scenarios. The single master approach performed worse in terms of availability compared to the multi master approach but offers better consistency. The same holds for the synchronized approach. There is a difference however between the synchronized and single master approach. The single master approach performed better during Scenario 1 while the synchronized approach performed better during Scenario 2. The results of the experiments provide a basis to start further research on to optimize a decentralized key management solution for a specific use case. The experiments can also be repeated with other solutions to compare them against the solutions proposed in this research.

In conclusion, the CRDT-based decentralized key management architecture is able to perform in a multi-domain environment and solves the key de-synchronization problem. It is shown that our solution answers the research question.

6.2. Discussion

In this thesis, we extended the work done in [15] in which from a more zoomed out level of abstraction an architecture is proposed to do decentralized key management. In this thesis, we proposed to use conflict-free replicated data types to store the keys in a decentralized way. Next to this, two other approaches on the key creation coordination are proposed and we did an experimental comparison of the three approaches. We have shown how conflict-free replicated data types which are designed to provide eventually consistent data storage can be used in a key management architecture.
The architecture proposed in Section 4 is designed as a general solution for different problems. Based on this architecture a proof-of-concept implementation to identify the differences between the different approaches is constructed. There are several optimizations possible for our naive implementations that can improve the presented results. These results, first of all, are depending on the following variables which could be tweaked to improve the performance.

- Gossip interval, the interval with which the new keys are synchronized with the other key managers.
- Write consistency when creating a new key can be varied from writing only to the local storage, to a majority or all key managers.
- The number of key managers used.
- The number of requests for keys made to the key managers.
- The distribution of the key requests.
- The validity period of the keys.

- The timeout before a key manager is marked as unavailable and it is not considered as a member of the cluster anymore.
- Initial delay before creating a new cluster of key managers.
- The timeout before a request times out and a new key manager is tried to obtain a key.

Optimization of these variables was not done during this thesis since optimizing the solution for a specific use case was not the goal of this research. Optimization and selection of the most appropriate approach could be the subject of future research extending the work presented in this research. It would also be interesting to look at a more realistic environment by running the different key managers on physical different devices. This would also mean the network disruptions could be simulated by just unplugging a cable instead of simulating the network partitioning.

6.3. Future Work

It is always important to determine what is feasible to achieve within the available time and define the scope of a thesis. In this section directions interesting to investigate further are discussed.
The first direction might be to look at existing implementations of decentralized database systems such as Cassandra which could provide data synchronization and conflict resolution strategies for the keys. The advantage of using an existing database system is that they are well tested and already used in production environments. It would be interesting to investigate if these solutions provide better or the same performance compared to the proof-of-concept implementation used in this research. For this research is only chosen to look at a simple implementation of CRDTs to keep the protocol simple and easy to test with. In extending work this could, therefore, be a good direction to go in.

Building a synchronized protocol on an asynchronous distributed system is not a trivial task. For this research, we achieved some synchronization in the synchronized key creation approach by using our own implementation. As proof-of-concept implementation only used during the experiments this works well. In the future, it might be worth it to experiment with other well-known synchronization algorithms to achieve even better results. There can also be looked at other forms of synchronization between distributed nodes such as using a token-based algorithm.
The same holds for other algorithms to elect the master in the single master approach. It could be interesting to compare other election algorithms to the one used during this research.

It can be investigated how the IP addresses of the key managers which are static from startup can be made dynamic. These addresses are used by the key managers to contact each other and form a cluster. The advantage of having dynamic addresses of key mangers is that not all key managers have to be redeployed if just one of the addresses of the key managers changes. This would be especially useful in highly dynamic environments where it might be needed to scale up and down the number of key managers.

The scalability is something else that could be investigated further. It should be verified if the proposed protocols also work if there are tens or hundreds of key managers. A larger number of key managers could give problems due to the rapidly increasing number of messages that have to be exchanged between the key managers. A solution could be to build a tree-like structure of key managers. This limits the number of messages that have to be exchanged by all key managers. The key managers would then only send new keys to its direct neighbors who distribute the keys further.

We have seen there are multiple directions in which this research could be extended in the future. Combining different well studied research directions makes decentralized topic key management an exciting topic to work on.

List of Figures

List of Abbreviations

CRDT	Conflict-free replicated data type. 9, 19, 41
GKMS	group key management service. 1
KM	Key Manager. 17
KMS	Key Management System. 5
TK	Topic Key. 5
TKS	Topic Key Sets. 17
VP	Validity Period. 17

Bibliography

[1] Mohammed Riyadh Abdmeziem and François Charoy. Fault-tolerant and scalable key management protocol for iot-based collaborative groups. In Xiaodong Lin, Ali Ghorbani, Kui Ren, Sencun Zhu, and Aiqing Zhang, editors, *Security and Privacy in Communication Networks*, pages 320–338, Cham, 2018. Springer International Publishing. ISBN 978-3-319-78816-6.

[2] J. Arakawa and K. Sasada. A decentralized access control mechanism using authorization certificate for distributed file systems. pages 148–153, 01 2011. ISBN 978-1-4577-0884-8.

[3] Eric Brewer. Cap twelve years later: How the "rules" have changed.(2012). *URL: http://www. infoq. com/articles/cap-twelve-yearslater-how-the-rules-have-changed*, 2012.

[4] Ghassan Chaddoud, Isabelle Chrisment, and André Schaff. Dynamic group key management protocol. In Vladimir I. Gorodetski, Victor A. Skormin, and Leonard J. Popyack, editors, *Information Assurance in Computer Networks*, pages 251–262, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg. ISBN 978-3-540-45116-7.

[5] Y. Challal, A. Bouabdallah, and H. Seba. A taxonomy of group key management protocols: Issues and solutions. In *Proceedings - Wec 05: Fourth World Enformatika Conference*, volume 6, pages 5–17, 2005. Cited By :5.

[6] H. Dahshan and J. Irvine. A robust and redundant key management for mobile ad hoc networks. In *2009 6th International Symposium on Wireless Communication Systems*, pages 433–437, Sep. 2009. doi: 10.1109/ISWCS.2009.5285290.

[7] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, October 2007. ISSN 0163-5980. doi: 10.1145/1323293.1294281. URL `http://doi.acm.org/10.1145/1323293.1294281`.

[8] Orhan Ermi, Erif Bahtiyar, Emin Anarm, and M. Ufuk alayan. A secure and efficient group key agreement approach for mobile ad hoc networks. *Ad Hoc Netw.*, 67(C):24–39, December 2017. ISSN 1570-8705. doi: 10.1016/j.adhoc.2017.10.003. URL `https://doi.org/10.1016/j.adhoc.2017.10.003`.

[9] Fabio Gadducci, Hernán Melgratti, and Christian Roldán. On the semantics and implementation of replicated data types. *Science of Computer Programming*, 167:91 – 113, 2018. ISSN 0167-6423. doi: https://doi.org/10.1016/j.scico.2018.06.003. URL `http://www.sciencedirect.com/science/article/pii/S0167642318302429`.

[10] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002.

ISSN 0163-5700. doi: 10.1145/564585.564601. URL http://doi.acm.org/10.1145/564585.564601.

[11] Junbeom Hur, Youngjoo Shin, and Hyunsoo Yoon. Decentralized group key management for dynamic networks using proxy cryptography. In *Proceedings of the 3rd ACM Workshop on QoS and Security for Wireless and Mobile Networks*, Q2SWinet '07, pages 123–129, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-806-0. doi: 10.1145/1298239.1298261. URL http://doi.acm.org/10.1145/1298239.1298261.

[12] A. Lakshman and P. Malik. Cassandra - a decentralized structured storage system. In *Operating Systems Review (ACM)*, volume 44, pages 35–40, 2010. Cited By :1159.

[13] P. Padmanabhan, L. Gruenwald, A. Vallur, and M. Atiquzzaman. A survey of data replication techniques for mobile ad hoc network databases. *VLDB Journal*, 17(5):1143–1164, 2008. Cited By :74.

[14] K. Petersen, M. Spreitzer, D. Terry, and M. Theimer. Bayou: Replicated database services for world-wide applications. In *Proceedings of the 7th Workshop on ACM SIGOPS European Workshop: Systems Support for Worldwide Applications, EW 1996*, pages 275–280, 1996. Cited By :31.

[15] F. Poltronieri, L. Campioni, R. Lenzi, A. Morelli, N. Suri, and M. Tortonesi. Secure multi-domain information sharing in tactical networks. In *MILCOM 2018 - 2018 IEEE Military Communications Conference (MILCOM)*, pages 1–6, Oct 2018. doi: 10.1109/MILCOM.2018.8599693.

[16] S. Rafaeli and D. Hutchison. Hydra: a decentralised group key management. In *Proceedings. Eleventh IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pages 62–67, June 2002. doi: 10.1109/ENABL.2002.1029990.

[17] Sandro Rafaeli and David Hutchison. A survey of key management for secure group communication. *ACM Comput. Surv.*, 35(3):309–329, September 2003. ISSN 0360-0300. doi: 10.1145/937503.937506. URL http://doi.acm.org/10.1145/937503.937506.

[18] Mohammad Roohitavaf, Jung-Sang Ahn, Woon-Hak Kang, Kun Ren, Gene Zhang, Sami Ben-Romdhane, and Sandeep S. Kulkarni. Session guarantees with raft and hybrid logical clocks. In *Proceedings of the 20th International Conference on Distributed Computing and Networking*, ICDCN '19, pages 100–109, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-6094-4. doi: 10.1145/3288599.3288619. URL http://doi.acm.org/10.1145/3288599.3288619.

[19] Bruce Schneier. *Applied Cryptography (2Nd Ed.): Protocols, Algorithms, and Source Code in C.* John Wiley & Sons, Inc., New York, NY, USA, 1995. ISBN 0-471-11709-9.

[20] R. Seetha and R. Saravanan. A survey on group key management schemes. *Cybern. Inf. Technol.*, 15(3):3–25, September 2015. ISSN 1314-4081. doi: 10.1515/cait-2015-0038. URL https://doi.org/10.1515/cait-2015-0038.

[21] Daniel Seybold, Christopher B. Hauser, Simon Volpert, and Jörg Domaschka. Gibbon: An availability evaluation framework for distributed databases. In Hervé

Panetto, Christophe Debruyne, Walid Gaaloul, Mike Papazoglou, Adrian Paschke, Claudio Agostino Ardagna, and Robert Meersman, editors, *On the Move to Meaningful Internet Systems. OTM 2017 Conferences*, pages 31–49, Cham, 2017. Springer International Publishing. ISBN 978-3-319-69459-7.

[22] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. *Conflict-free replicated data types*, volume 6976 LNCS of *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics).* 2011.

[23] Marc Shapiro, Nuno M. Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of convergent and commutative replicated data types. 2011.

[24] R. Shrestha. High availability & performance of database in the cloud: Traditional master-slave replication versus modern cluster-based solutions. In *CLOSER 2017 - Proceedings of the 7th International Conference on Cloud Computing and Services Science*, pages 385–392, 2017. Cited By :1.

[25] Ariel Tseitlin. The antifragile organization. *Commun. ACM*, 56(8):40–44, 2013.

[26] Werner Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, January 2009. ISSN 0001-0782. doi: 10.1145/1435417.1435432. URL `http://doi.acm.org/10.1145/1435417.1435432`.

[27] K. Wrona and S. Oudkerk. Content-based protection and release architecture for future nato networks. In *MILCOM 2013 - 2013 IEEE Military Communications Conference*, pages 206–213, Nov 2013. doi: 10.1109/MILCOM.2013.44.

[28] Chang N. Zhang and Zheng Li. An efficient group key management scheme for secure multicast with multimedia applications. In Sokratis K. Katsikas, Stefanos Gritzalis, and Javier López, editors, *Public Key Infrastructure*, pages 364–378, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. ISBN 978-3-540-25980-0.