Exploring the viability of a CRDT-based Data Model for a distributed decision support system

Jarl Holme

2020

Abstract

Conflict-free Replicated Data Structures provide strong eventual consistency by ensuring that alterations to the model resolve the same regardless of execution order, conceptually providing a distributed system with both consistency and availability. This report examines the viability of a completely CRDT-based data model for a distributed decision support system to be used by the Norwegian Red Cross under the name *Operativt Beslutningsstøttesystem* (OBS). The domain of search and rescue is an intuitive fit for CRDTs due to the capability of seamless merges with other replicas and guaranteed service regardless of connection.

The product of this report is a proof-of-concept implementation for a data model providing strong eventual consistency with constant availability using Conflict-free Replicated Data Types (CRDTs) for OBS. The solution implements several known CRDTs adapted to fit the hierarchial structure of the model required for the system while ensuring that data is not lost due to concurrent operation and demonstrates that CRDTs are able to provide a sufficient data model while handling a demanding set of requirements.

Contents

1 Introduction

1.1 Situation

The Red Cross is one of several main actors in the Norwegian voluntary rescue service, able to supply all resources and personnel required for search and rescue operations. Although the techniques used in planning and leading these operations are relatively modern the methods implementing them are mostly analog with some simple digital map systems used. In an effort to realize this untapped potential a project is underway to consolidate all aspects of search and rescue operations in a single dedicated system under the name *Operativt Beslutningsstøttesystem* (OBS).

1.2 Motivation

The goal of this report is to explore the viability of Conflict-free Replicated Data Structures to provide the functionality required of OBS. The guaranteed consistency regardless of concurrent operation makes CRDTs an intuitive fit for such a system where functionality must be guaranteed even in situations where connection to a central server would be difficult or impossible.

1.3 Research Questions

In order to conclude on whether OBS can be suitably implemented using CRDTs the following research questions needs to be answered

- **Q1** : Are the properties of a CRDT-based data model suitable for use in the case for OBS?
- **Q2** : If **Q1** holds, are the components required to express the data model for OBS possible to implement using CRDTs?
- **Q3** : If **Q2** holds, can a system providing the required functionality be built on these components?

1.4 Report overview

The introduction chapter introduces the situation and formulates research questions that drives the report. The background chapter covers the relevant theory regarding the search and rescue domain, distributed systems in general and CRDTs in specific. Chapter 3 details the requirements provided for OBS. Chapter 4 proposes a solution for OBS using CRDTs conforming to the requirements detailed while Chapter 5 describes a proof-of-concept implementation of said solution. Chapter 6 validates the implementation against the requirements presented while Chapter 7 addresses the research questions and concludes with the state of the current model as well as future work on this project. This thesis is a further development of the specialization project (TDT4501) and parts of Chapter 1-4 and 7 are based on the report for that subject.

2 Background

2.1 Search and Rescue

Search and rescue (SAR) refers to the field governing locating and rescuing missing persons. This thesis will focus on search operations as they are the most complex in terms of planning and leading. Search operations are anchored in statistics gathered by the International Search and Rescue Incident Database (ISRID) and are based on the concept of searching from the most relevant location (based on intelligence gathering, often the last known location of the missing person or some clue considered relevant) and outwards with an increasing radius [1].

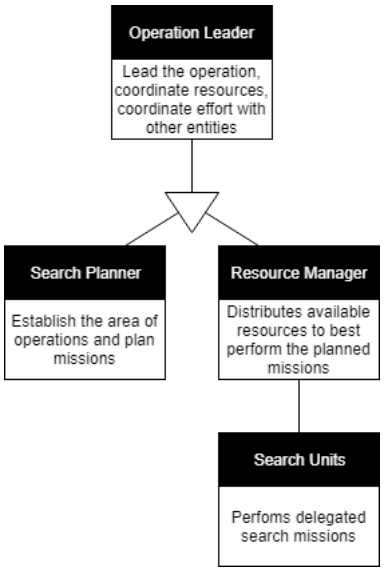


Figure 2.1: Roles and responsibilities in a SAR-operation

A SAR-operation typically consists of a leader element as well as several units conducting the search (see Figure 2.1). The leader element is responsible

for gathering intelligence to define areas where the missing person(s) are most likely to be found as well as planning missions within this area of operation and distributing available resources into units to handle the execution of said missions.

Several different types of units can be involved in any given operation such as foot patrols, ambulance, K9 or snowmobile/all-terrain vehicle units. To facilitate communications each unit and mission in an operation will be given a unique callsign enabling accurate and efficient control over radio communication. These names can be considered human readable ID-values.

As operations are often started on short notice and in unpredictable locations the immediate response tends to be rather ad-hoc as availability of personnel and their arrival time to the area of operations can vary greatly. As such it is not uncommon for the leadership roles to change hands as the operation unfolds.

2.2 Distributed systems

2.2.1 Consistency in distributed systems

According to the CAP-theorem a distributed system is forced to prioritize between three attributes - consistency, availability and partition tolerance [2]. The typical interpretation is that a partitioned/distributed system must prioritize between consistency and availability.

As sacrificing availability, for instance through forcing operations to pass through a centralized server, would present a bottleneck potentially eroding much of the performance advantages of distributing the system in the first place a common approach is to sacrifice consistency through optimistic updates (meaning that updates are accepted if its preconditions exists locally with the assumption that it will resolve with concurrent updates).

When designing a distributed data model the process of consolidating concurrent operations (operations performed in parallel) must be designed according to some philosophy of operation regarding what to prioritize. Seeing how each operation is performed without knowledge of the other and with its preconditions met they can both be viewed as valid operations.

A common practice is to prioritize persisting user input over removing it if the conflict between these operations should arise (for example if an item is concurrently removed and modified then the modify operation should win persisting the item). While this ensures that no user input is lost due to the data model being distributed it does open up the possibility of odd behaviour like the phenomenon in Amazon Dynamo where deleted items in a shopping cart would re-appear [3]. In some cases, such as in those where losing data could be critical to the operation of the system, these cases could be considered the price to pay to ensure that no data is lost.

Eventual consistency

A common way of implementing a distributed system is to sacrifice immediate system-wide consistency by optimistically allowing updates at any node without costly synchronization between other nodes. This approach ensures service availability at multiple partitions executing in parallel with the knowledge that the partitions will converge eventually.

Operational transformation is a common way of achieving eventual consistency, used by several online collaborative frameworks such as Apache Wave [4]. OT handles consistency by transforming operations to maintain their intention as the operations propagate through the set of replicas. This manual handling of cases demands that the framework can handle *any* possible sequence and combination of operations and transform them correctly, forcing much resource usage into design, validation and maintenance of such a structure [5] as well as poorly scaling execution time with larger collection of operations to transform [6].

Strong eventual consistency

Strong eventual consistency is a guarantee in a distributed system, where clients update the data stored independently of each other, that two client will be at exactly the same state the moment they have applied the same set of operations. Strong eventual consistency, if attained, would allow a system to provide both Consistency (eventually all replicas are guaranteed to reach the same state) and Availability (each replica can safely perform any changes with a guarantee that it will eventually converge with the other replica) in a distributed system.

2.3 Conflict-free Replicated Data Types

Conflict-free Replicated Data Types are data types providing strong eventual consistency by design, rather than through application-level logic. This is accomplished by designing data types and operations to result in the same state regardless of the execution order of concurrent operations, meaning that any two replicas will be in equivalent states when they have observed the same changes as illustrated in Figure 2.2. This property allows a CRDT-based distributed system to adopt an optimistic updating approach with the guarantee of an eventually consistent data model.

There are two approaches to implementing CRDTs - state based and operation based. As they can be reduced to each other, implementation-specific factors will dictate which is the better choice [7]. This report will focus on the operation-based implementation as it provides several features desired in the domain-specific application(see Chapter 3.2).

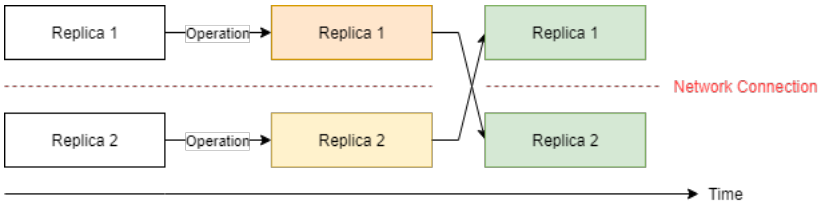


Figure 2.2: *Illustration example for a replicated system using CRDTs. Replicas converge once all operations have been viewed. Colours represent the state of the replica, with equal colours representing equivalent states.*

2.3.1 Operation-based CRDTs

The domain for an operation-based CRDT consists of some initial state common to all replicas and a set of operations that, when applied, alters the state of the replicas. For any domain implementing such a data type to provide strong eventual consistency two factors need to hold [7]

1. Causal delivery must hold for application of operations (an operation will only be applied if its preceding operations have already been applied)
2. Any two operations where its preconditions are met will commute (same state will be reached regardless of execution order)

The payload of communication between replicas will consist of the operations applied to the common initial state. As long as each operation is commutative and no operation will be run unless its preconditions are met (as described above) two replicas receiving the same operations will be in the exact same state regardless of delivery order or concurrently performed work.

2.3.2 Data Types

Much work has already been put into exploring the expressiveness of CRDTs. The data types described below will be used as building blocks for the implementation of the data model.

Observed Remove set [7]

The OR-set is an unordered set where each element has a unique identifier. For each element a set of client ids will also be included listing which client has performed the add-operation on it. Two operations are allowed on the set

- *add(element)* - adds an element to the set and marks it as added by the current client
- *remove(element)* - removes an element from the set by removing all client marks observed at the relevant replica

This data type is a CRDT as sequential operations will resolve without issue (add and remove operations resolves based on order and are idempotent meaning repeated executions have no effect) and concurrent operations will favor *add()* (two separate clients with concurrent add/remove operations will converge on the element remaining in the set as its client list will reflect that the element was added).

A property worth noting is that this structure does not account for sequential add-operations from the same client. This behaviour will be expanded upon in our implementation (see Chapter 4.1.2).

Multi-Value Register [7]

The multi-value register is a map using vector clocks to maintain consistency. MVRs supports a single operation

- *assign(key, value)* - assigns a value to the provided key.

This data type is a CRDT as sequential assigns will result in the last value being assigned, while for concurrent assigns both values will be assigned to the key as a set leaving the user to resolve the conflict.

Replicated Growable Array [8]

The replicated growable array is a data structure that supports a sequence of atoms such as those found in documents. The structure is built using a hash table mapping unique keys to slots of the following structure

```
Slot{
    Object, //contents of the element
    S4Vector, //unique structure resolving collisions
    Key, //the unique key for the element
    NextSlot //pointer to the next slot in the sequence
}
```

Sequence is provided by including a pointer in each slot to the next plot in the sequence. The S4Vector is generated when an element is added to the RGA (derived from the vector clock of the operation) and consists of the following 4 parameters

- SessionId - the current session number (incremented on new session such as when membership changes)
- SiteId - a globally unique identifier for the site running the replica
- Sum - the sum of the vector clock for the operation
- Seq - the value of the vector clock for the replica performing the operation

Globally unique keys for the slots is generated by hashing the S4Vectors. Order is defined between S4Vectors S_a and S_b by checking the following statements in order

1. $S_a[SessionId] < S_b[SessionId] \Rightarrow S_a \prec S_b$
2. $S_a[Sum] < S_b[Sum] \Rightarrow S_a \prec S_b$
3. $S_a[SiteId] < S_b[SiteId] \Rightarrow S_a \prec S_b$

On insertion of an element its final position will be determined by the ordering defined above. The S4Vector orders operations on which session they belong to and the sum of its vector clock (intuitively ordering operations by how many operations that precedes them) with the globally unique client id (SiteId) used as a fallback guaranteeing order. The following operations are implemented on the RGA

- *insert(S4Vector, Object, LocationSlot)* - for the given LocationSlot, traverse its NextSlots until a slot with an S4Vector succeeding the provided S4Vector is found. Place the object between those two slots
- *delete(key)* - marks the provided slot as a tombstone, kept in memory

As it can be shown that S4Vectors are ordered and transitive [8], each replica will reach the same result after the same operations are performed. Tombstones are kept in the structure to allow concurrent operations on deleted slots to be performed.

2.3.3 Garbage Collection

Providing strong eventual consistency comes with some drawbacks. As the state of the data model at any time is generated from the entire history of operations until that point the entire history needs to be preserved. Where a more traditional data model can safely discard deleted or stale data a CRDT data model cannot afford that luxury as some replica somewhere may have branched out concurrently with any other replica at any point.

Deleted entries are commonly referred to as tombstones (a data structure representing the space some piece of data *used* to occupy). Several garbage collection mechanics are proposed using version vectors and client lists utilizing the known participating clients to purge tombstones once all clients have observed operations rendering them obsolete [7][8].

3 Requirements

3.1 Data Model

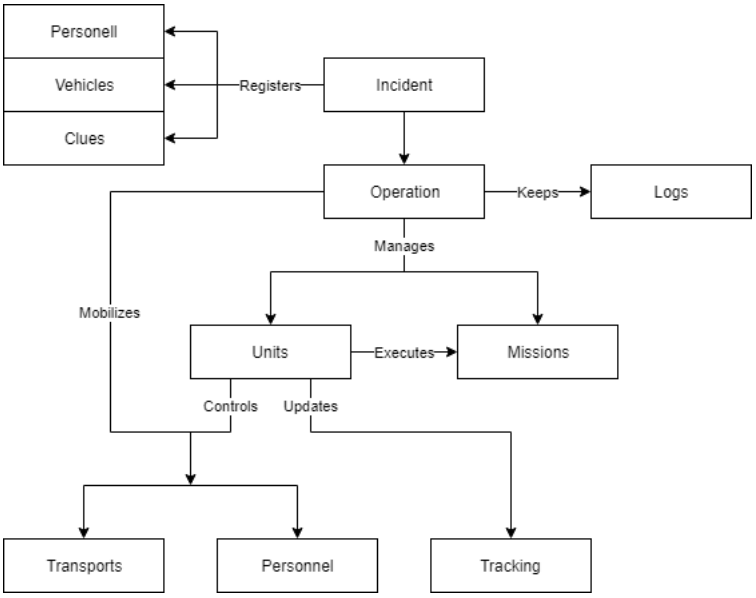


Figure 3.1: Conceptual view of required data structures (boxes represent entities and arrows linking them represent relations between them)

The required data structures for the system are shown in Figure 3.1 (collected from Appendix A). The finished system should also be able to complete the tasks outlined in Appendix B. A model capable of fulfilling the requirements presented will need to represent different types objects linked together in a meaningful way. The base components required for a data model for the system are objects with associated attributes and relations between them.

3.1.1 Objects

Data objects are central to any data model, serving as containers for information. In our context they refer to separate types of containers representing a self-contained object. The minimum required types of data objects are represented as rectangles in Figure 3.1. Each object requires a unique identifier to ensure that we can directly address any object in the system without the risk of collision.

Another requirement of some of the objects (units and missions) is the usage of human readable IDs (e.g. callsigns) as described in Chapter 2.1. The collision or loss of updates on this value could have consequences on communication during the operation and needs to be handled with care to avoid such issues.

3.1.2 Attributes for objects

The data objects defined in the system also needs to be able to store information about themselves. To provide this functionality we need to amass a portfolio of data types that can express the attributes required. The proposed solution must, as an absolute minimum, be able to handle document editing.

3.1.3 Relations

Relations in the system refer to the way data objects are linked to each other. This is represented by the arrows in Figure 3.1. The data model must be able to handle relations between objects.

Another aspect of the relations in the search and rescue domain is (by no means uniquely) the hierarchial nature of relations. An example of this would be a unit executing a mission, both managed by an operation. As long as the unit is executing the mission it is implicitly managed by its operation - severing the link upwards would leave the model inconsistent. A data model must ensure the consistency of the relation hierarchy to avoid such unintended behaviour.

3.2 System Requirements

OBS is required to operate regardless of environment (network availability) at a level that allows parallel operation with no loss of data or functionality as a result. The system principles outlined in Appendix A accounts for how the system as a whole should be implemented. Two of the principles outlined in the document requires the support of the data structure (outlined in **R1** and **R2**). The nature of search and rescue operations as conducted by the Norwegian Red Cross (Chapter 2.1) also makes some functional demands of the system (outlined in **R3** and **R4**). The intended roadmap for development of features for OBS also dictates **R5**.

- **R1 : Offline over Online Data** - The system must continue to operate regardless of connection status
- **R2 : Data Availability over Consistency** - The system must prioritize availability over consistency in an implementation, ensuring that work is allowed to progress at a tempo unimpeded by synchronization processes such as two-phase locks
- **R3 : Variable client participation** - The system must be able to handle a changing set of participating clients
- **R4 : User input must be preserved** - The overall principle on resolution of concurrent operations should be to avoid loss of user input due to concurrent operations
- **R5 : Data model must be easy to build solutions on top of** - As the system is intended to be easy to develop functionality for, working with the data model should not require knowledge of the underlying mechanisms of CRDTs

4 | Proposed data model

4.1 Components

The data model can be generalized into objects and relations. The data objects/database classes (as represented by the boxes in Figure 3.1). Each object will require a set of data and metadata fields. In addition relationships are defined between objects - some of them will also implement qualifications/data to the connection (for example a unit managed by an operations can have a callsign assigned to it).

4.1.1 Objects

ID-values

Any new object regardless of creation time or client of origin in the distributed system must be guaranteed a unique ID. This is the only property the ID must provide as it will refer to unique constructs generated while using the data model. A Lamport clock [9] for each client combined with its client id will guarantee uniqueness as long as each client has a unique ID and for each operation at some client each sequential operation will increment a persistent counter.

Human readable ID-values/callsigns

The core parts of any search and rescue-mission are the units at work and the missions they are tasked to perform. In order to facilitate rapid communication and administrative control of the operation these resources are assigned callsigns (such as unit 2-1 or mission 1). These values are typically issued sequentially and must be accounted for as an identifier since any collision or oversight in these could cause communication difficulties.

Although the system should enforce unique assignment of callsigns the automatic resolution of such collisions has the potential to cause more problems than it solves (as we cannot guarantee complete device distribution a unit can have its callsign altered without its knowledge causing potential communication failure). As such the Multi-Value register was chosen to maintain the assignment of callsigns, leaving the user to resolve any such conflicts.

Attributes

Each object in the data model will have to host a selection of data and metadata to provide the desired functionality. The usage of RGAs will allow concurrent editing of sequential data. The S4Vector ensuring strong eventual consistency for this data type can be expressed by using client ids and vector clocks which the system is perfectly capable of. As each object type can be defined with a set of legal attributes they can be initialized with a map connecting attribute key to its data which can be any appropriate CRDT allowing for sets or sequential data as described above.

Vector clocks

Vector clocks are required to reason about execution order (sequential or concurrent) in RGA, MVRs and the modified OR-set used for relations (see Chapter 4.1.2). For a defined client list a vector clock is a simple thing to implement and reason about. However due to the distributed nature of operation execution we must consider the client list to be variable. As we have unique client ids one approach to this is to initialize an empty map as a clock and assign the pair (client_id, timestamp) on an operation. If we define the value '0' for a client that is not represented in the map it will produce equivalent functionality (with a performance overhead compared to the simpler vector structure).

4.1.2 Relations

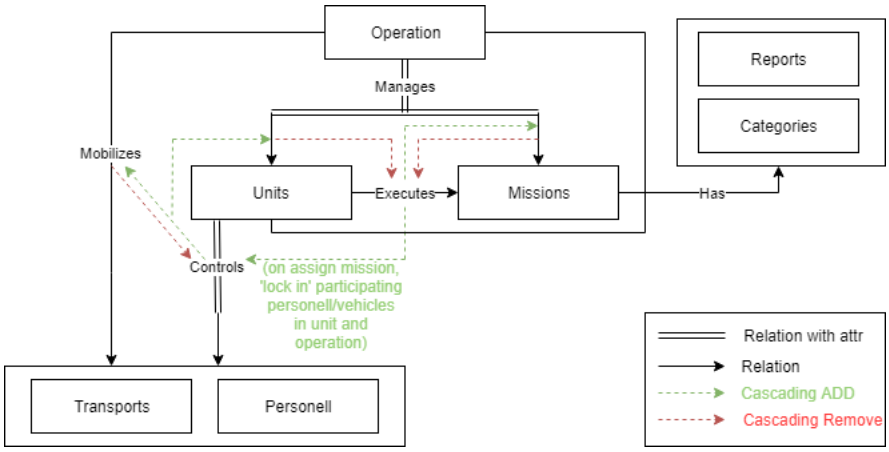


Figure 4.1: Relationship hierarchy

Objects in the data model are organized in a hierarchy (Figure 3.1). This implies that changes in relationships should propagate up and down to keep the total structure consistent (for instance a unit that executes a mission should implicitly be managed by the operation the mission is managed by). In order

to enforce this across replicas we define relations by OR-sets with cascading operations as shown in Figure 4.1. The desired outcome is that alterations higher up in the hierarchy could be reverted if concurrent operations has occurred modifying the relations further down.

Operations on the OR-sets implementing this behaviour consists of *add()* and *remove()*. Each operation is also expanded to include a vector clock as defined in Chapter 4.1.1 to ensure that concurrent operations are detected (in the default implementation of OR-sets sequential *add()*-operations do nothing, we require it to further the vector clock to be able to correctly handle concurrency).

Attributes on relations

In addition to handling the existence of relations some also need to handle attributes for them (units belonging to an operation will need to have the option for a callsign and missions will need a mission name as described in Chapter 2.1).

We accomplish this by using a Multi-Value Register to map callsigns or mission names to the target of the relation. The behaviour of the MVR-structure regarding concurrent, colliding assigns is desirable in this context as described in Chapter 4.1.1. The resulting object will have the following legal operations with target_id representing the id value of the target of the relation and human_readable_name represents the property of the relation such as the callsign for a unit or the name for the mission

- *assign(target_id, human_readable_name)* - runs the following sequence of operations
  - *MVR.assign(human_readable_name, target_id)*
- *remove(target_id)* - runs the following sequence of operations
  - *MVR.assign(human_readable_name, remaining_set)* where *remaining_set* refers to the set of values from *MVR.get(human_readable_name)* minus target_id

As such we end up with two relation types - with and without attributes assigned. Both types supports the cascading behaviour described ensuring that concurrent work will prioritize maintaining user input by maintaining a consistent relation hierarchy.

Garbage collection

Cascading operations as implemented above maintain the structure of the data model but does generate more data as each operation on a relation multiplies based on the number of defined cascading dependencies. To lessen the impact of this we expand the operations on the relation objects to be marked if they are the result of a direct operation or a cascading one. While direct operations are required to provide history the cascading operations are only used to ensure a consistent model. Remember that, for sequential operations for both OR-sets and MVRs, the last operation will 'win'. This leaves us with the following

Proposition 1. *If* $\exists operation_b \Rightarrow (operation_a \prec operation_b)$ *then* $operation_a$ *can be safely discarded without affecting the outcome of the set of operations*

Proof. $(\forall operation \| operation_a \Rightarrow (operation \| operation_b \vee operation \prec operation_b))$, meaning that $operation_b$ will either come after or concurrently resolve with the same set of operations as $operation_a$ leading the payload of $operation_a$ having no effect on the resulting state □

This proposition allows us to discard the payload for all non-concurrent cascading operations (operations that aren't the result of cascading still needs to be maintained due to logging). A tombstone must be left to ensure causal order of operations (Chapter 2.3.1) eating into some of the advantage of this method of garbage collection. The total value of this method and discussion on its inclusion is further elaborated on in Chapter 7.1.1.

4.2 Structure

This section covers architectural definitions regarding how the system should be implemented to allow the system to provide strong eventual consistency along with some performance optimizations. These choices are intended to be maintained through the implementation of an API, as proposed in Chapter 4.3.

4.2.1 Schema

While we are able to ensure uniqueness between objects (by the globally unique lamport clock id value) we are not able to guarantee no collisions between their attributes if the user is allowed to add them at their own discretion. A solution to this is to ensure that new objects are initialized with its legal attribute fields by having it conform to a strict schema. Such a schema would define how an object should be initialized ensuring that concurrent operations on the same objects do not cause duplicate fields or collide unintentionally. A partial implementation for a schema is provided in Chapter 4.2.1, defining id values and relations for a subset of the data model. This could be expanded upon in an implementation of the system.

4.2.2 Read-optimized data

While the basic structures defined in Chapter 4.1.1 provides the functionality required of the data model it is not optimized for all usage patterns. In order to increase efficiency of polling the data model relation data should be duplicated in both ends. As proposed in Chapter 4.1.2 a single structure maintains the actual data for the relationship (for any relation the relation data structure would be stored in one end of the relation leaving the other side with no efficient way of accessing its relations).

To allow for quicker reads this data could be replicated in both objects. As the views proposed can be extrapolated from the data types defining the relation they do not need to be replicated across clients - they can be generated at each replica with identical results given the same observed operations. How these structures could look is provided in Chapter 4.2.1.

4.3 API

Several of the proposed structures in Chapter 4.1, such as the implementation of relation hierarchies using cascading operations (Chapter 4.1.2), rely completely on being updated as a structure rather than updating the components it consists of. As such a data model using these components must implement a layer ensuring that only safe operations are performed on the model. As the data model is also intended to serve as a platform for feature development the model must also not require knowledge of the underlying mechanisms of the data model as outlined in **R4**.

The best way to resolve the requirements listed above is to implement an API handling alterations of the underlying data model. This interface would need to only allow operations fitting their preconditions. Another responsibility would be to maintain the denormalizing data replication structures referred to in Chapter 4.2.2. Having an API handle changes to the data model also simplifies the development process as a middle layer is handling the data model logic allowing the developer to focus on feature implementation rather than wrestling with the mechanics of the data model.

4.4 Logging

Operation based CRDTs consists of an initial state and a set of operations performed. As a consequence of this design choice the sequence of operations can be viewed as a change log for a system implementing them. In the simplest implementation there are some problems with this assumptions however, as the sequence of concurrent operations cannot be determined. Another issue regarding the total sequence of operations is that they are defined this is a relative measurement of time from the systems perspective, with no immediate relation to human time.

These issues should be resolved in order for the system to provide logging, and could be done as simply as appending a local timestamp (milliseconds since epoch) to each operation. This will allow us to reason about order of execution in the aftermath of a SAR-operation.

5 | Data model implementation

In order to determine the viability of a CRDT-based data model in OBS a proof-of-concept data model was implemented. The data model was implemented using Java and the project can be found at

`https://github.com/jrlhlme/master`.

This data model covers a subset of the structures required for a full-feature release (the subset shown in Figure 5.1). This subset was chosen as it covers the functional aspects of the full model, being relations with attributes (as outlined in Chapter 4.1.2) and the cascading behaviour between relations (as outlined in Chapter 4.1.2).

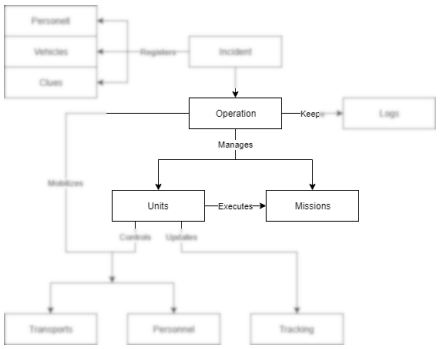


Figure 5.1: Scope for the proof-of-concept model implementation

5.1 Architecture

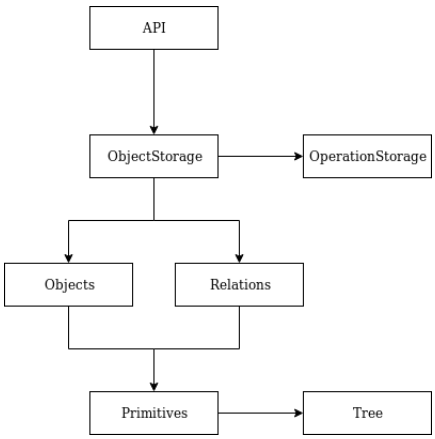


Figure 5.2: Overview of components in the implemented model with arrows indicating usage pattern

The implemented model can be divided into a hierarchy of components with high or low order purpose in the functionality of the data model. The main component types and their relations to each other are shown in Figure 5.2. Each component are presented in detail in the following sections.

5.2 Primitives

Primitives refer to the most basic data structures in the data model and encompass the fundamental data types as well as operations directly on them. The data model implemented uses observed remove sets as well as multi value registers. Both of these share the same operation-handling structure in the form of the tree.

5.2.1 Operation tree

As both implemented primitive data types are operation-based CRDTs some structure keeping track of sequence and concurrency between replicas is needed. This is accomplished through a tree-like structure containing operations and organizing them into child-parent relationships based on the preconditions of the operations. Each operation is assigned a vector clock for its operation as well as a set of vector clocks indicating the preceding operations serving as parents in the tree.

The role of the tree structure is to organize incoming operations and detect concurrency. When concurrency is detected the tree will produce the root of

the concurrency (where two or more replicas diverged in the tree), leaving the resolution of said operations to the implemented primitive data type.

5.2.2 Modified OR-set

The most basic data type in our data model is the OR-set with vector clock modification to allow for sequential adds (as outlined in Chapter 4.1.2). Behaviour is intended to maintain user input, with sequential operations resolving according to their ordering and concurrent operations prioritizing add operations to the point that any single add will win over any number of concurrent remove operations.

5.2.3 Multi-Value Register

The MVR implementation in the data model is built using the OR-set implementation described above, being a register that maps id values to OR-sets determining what value the id corresponds to. Behaviour as defined in Chapter 4.1.2 is implemented when generating operations for the underlying OR-set, enforcing the intended behaviour.

5.3 Components

5.3.1 Objects

Objects in the data model are intended to hold information about their state. Existence of an object in the data model is handled by designated OR-sets holding id values. These id-values map to created objects to allow the developer to access the data as an object.

These objects are intended in a full implementation to hold information derived from operations on the underlying CRDTs allowing read-optimized data for attributes and relations to other objects as proposed in Chapter 4.2.2. Including this feature would also greatly increase the performance of relation operations as this would allow direct references to the relations rather than having to iterate through the list of relations in the worst case.

In the implemented data model objects serve as anchor points for relations. The cascading principles as described in Chapter 4.1.2 are not extended to objects as of this implementation. Implementing this feature would also increase the amount of operations generated. As the state of an object is dictated by the relations it is connected to (a Unit not connected to an Operation is not involved in the execution of said Operation) the delete functionality is disabled on objects in the API (Appendix C) with no real functionality loss. The implementation of this feature is discussed in Chapter 7.3.

5.3.2 Relations

Relations in the data model are defined by three separate maps mapping the source id to a MVR which maps attributes to destination id(s). This structure allows the data model to maintain relations between objects and assign an attribute to any given relation if required.

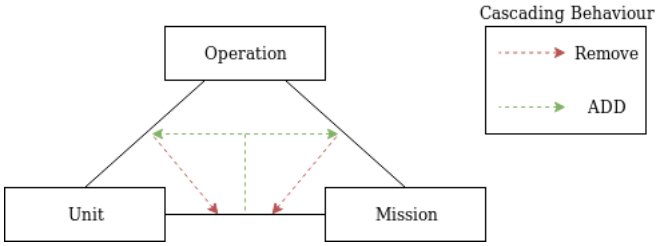


Figure 5.3: Relationship cascading behaviour in the partial implementation

Cascading relationship operations as discussed in Chapter 4.1.2 are implemented in the data model with behaviour as mapped in Figure 5.3. When a relationship is assigned or removed the operation will automatically generate operations for other connected relations. Examples of this behaviour can be found in the Test class in the data model with a full description of the cases included in Chapter 6.1.4.

5.4 Object storage

The Object storage class is a centralized class managing all entities used to store data and contains the components described in Chapter 5.3. Another role of the class is to transform simple functional calls such as creating objects and assigning/removing relations into operations on the underlying primitive data objects while maintaining properties such as cascading. Operations resulting from cascading are also labeled as such so that garbage collection mechanisms such as the one outlined in Chapter 4.1.2.

5.4.1 Operation Storage

The Operation storage class maintains the operations performed on its designated Object storage class as well as keeping track of operations to export when exchanging operations with another Operation storage instance. The stored information of operations performed can serve as a log of changes done to the system as operations are stored sequentially based on execution.

5.4.2 API

The API for the implemented data model consists of the public functions in the Object storage class (specifications can be found in Appendix C). The API is

intended to function as the single contact point for developers working on top of the data model and does not require knowledge of the underlying logic to use.

5.5 Garbage collection

Garbage collection in CRDTs typically require consensus on data structures that are stale and can safely be removed among a known set of participating clients [7]. These constraints are in conflict with the system requirements for our implementation as it requires a lock on a set of data structures (violating the principle of availability over consistency (**R2**)) as well as a known client list (which we are unable to obtain in our domain (**R3**)). These constraints render us unable to completely remove tombstones from stale objects in our data model.

The cascading behaviour described in Chapter 4.1.2 does allow for some garbage collection. As the cascading operations are not needed for logging purposes their contents can be purged once it can be ascertained that they have no impact on the state of the object they were performed on (Chapter 4.1.2). The object storage class that handles cascading behaviour on relation operations also marks operations as cascading or not, thus allowing the above mentioned functionality (some further work on this is required, see Chapter 7.3).

6 Validation

6.1 System Requirements

Several requirements for a data model implementation for OBS were outlined in Chapter 3.2. This chapter considers to what extent the proposed solution is able to fulfill these requests.

6.1.1 R1 - Offline over Online data

To guarantee functionality OBS must operate regardless of connection status. The CRDT-based system proposed has each implementation operating completely independently with the possibility to synchronize with other implementations at any time. All legal operations in the system are guaranteed to resolve as the data types are conflict free. As such the system is guaranteed functionality regardless of connection status as it makes no functional distinction between being connected or not.

6.1.2 R2 - Data availability over Consistency

As each implementation operates independently to each other availability concerns are non-applicable when using the CRDT-based proof of concept data model. Each client has immediate availability of all observed alterations to its implementation.

6.1.3 R3 - Variable client participation

In addition to the connectivity concerns above no assumptions can be made regarding the client list when the data model is in use. The way vector clocks are handled (by defining a zero value for any client not represented in a given vector clock as proposed in Chapter 4.1.1) allows the model to provide functionality regardless of participating clients.

6.1.4 R4 - User input must be preserved

Using CRDTs allows for defining behaviour on concurrent operations performed on the system. In our case operations adding or maintaining data structures are essential to persist when concurrent to remove operations as in this case the alterations made may not be visible to a user removing data and as such the decision to remove was made on an incomplete basis. This behaviour is achieved in the implemented data model by ensuring that operations adding to or editing data takes precedence in concurrent operations. Cascading operations are also implemented to persist a relationship hierarchy regardless of concurrent operation of the system.

To demonstrate the properties described above several test cases were included in the file Test.java in the implemented data model. The results below can be replicated by executing said class and will output the state of each client after each step.

Test Case 1

1. Starting conditions : Operation (id:1-1), Unit (id:1-2) and Mission (id:1-3)
2. Client 1 :
   add relation between Operation (id:1-1) and Unit (id:1-2)
   remove relation between Operation (id:1-1) and Unit (id:1-2)
   Client 2 :
   add relation between Operation (id:1-1) and Unit (id:1-2)
3. Outcome after synchronization : add operation from client 2 is concurrent to the remove operation in client 1, resulting in the relation between the operation and the unit remaining
   State of both clients : Operation (id:1-1), Unit (id:1-2) and Mission (id:1-3) with the following relations : Operation(id:1-1)-Unit(id:1-2)

Test Case 2

1. Starting conditions : Operation (id:1-1), Unit (id:1-2) and Mission (id:1-3) with the following relations : Operation(id:1-1)-Unit(id:1-2), Operation(id:1-1)-Mission(id:1-3)
2. Client 1 :
   remove relation between Operation (id:1-1) and Unit (id:1-2)
   remove relation between Operation (id:1-1) and Mission (id:1-3)
   Client 2 :
   add relation between Unit (id:1-2) and Mission(id:1-3)
3. Outcome after synchronization : cascading adds from add operation in client 2 takes precedence over the two remove operations, persisting all relations from start condition as well as adding the new relation between Unit (id:1-2) and Mission(id:1-3)

State of both clients : Operation (id:1-1), Unit (id:1-2) and Mission (id:1-3) with the following relations : Operation(id:1-1)-Unit(id:1-2), Operation(id:1-1)-Mission(id:1-3) and Unit(id:1-2)-Mission(id:1-3)

Test Case 3

1. Starting conditions : Operation (id:1-1), Unit (id:1-2), Mission (id:1-3) and Mission (id:1-4) with the following relations : Operation(id:1-1)-Unit(id:1-2), Operation(id:1-1)-Mission(id:1-3) and Unit(id:1-2)-Mission(id:1-3)

2. Client 1 :
   remove relation between Operation (id:1-1) and Unit (id:1-2)
   Client 2 :
   add relation between Operation (id:1-1) and Mission(id:1-4)
   add relation between Unit (id:1-2) and Mission(id:1-4)

3. Outcome after synchronization : cascading remove from client 1 will remove the relation between Unit(id:1-2) and Mission(id:1-3), however cascading adds from client 2 will persist the relation that client 1 attempted to remove
   State of both clients : Operation (id:1-1), Unit (id:1-2), Mission (id:1-3) and Mission (id:1-4) with the following relations : Operation(id:1-1)-Unit(id:1-2), Operation(id:1-1)-Mission(id:1-3), Operation(id:1-1)-Mission(id:1-4) and Unit(id:1-2)-Mission(id:1-4)

6.1.5 R5 - Data model must be easy to build solutions on top of

As described in the requirements OBS should fit into a microservice architecture with multiple independent development teams. This naturally extends to any data model implemented and, as such, puts high demands on usability. The implemented proof-of-concept data model relies on the ObjectStorage class to provided a higher-level API for altering the database (as outlined in Appendix C) while handling the data model specific CRDT logic in the background. This allows developers with no knowledge of the internal workings of the data model or CRDTs to implement features using the data model.

7 Discussion

7.1 Research Questions

7.1.1 Q1

Are the properties of a CRDT-based data model suitable for use in the case for OBS?

A CRDT-based data model have several native advantages. The guaranteed convergence property of strong eventual consistency enables full functionality when online, offline, or in cases where the network is intermittent (satisfying **R1** and **R2**). The small payload consisting only of the actual operations also helps in cases where bandwidth is limited, as less data must be exchanged.

Garbage collection is a potential problem as prolonged usage would cause old data (tombstones) to use increasing amounts of space but SAR-operations are not expected to last for long amounts of time lessening this problem somewhat.

While the collision-free nature of a CRDT-based data model sounds like a perfect fit in theory there are some aspects of the SAR-domain that robs us of some of the potential advantages. Operations colliding due to concurrency are both executed on valid preconditions (although at different replicas), and as such should be preserved. As the collision originates from the domain rather than the data model abstraction of it, automatically resolving the collision could lead to loss of data - the conflict in itself is necessary data that needs to be addressed. This does not mean that we get no use of this property however, as on such collisions the proposed relation handling easily rolls back to a consistent state.

In sum CRDTs provide a set of properties that surpasses merely intuitive suitability. In addition to this the downsides of CRDTs are less pronounced in the problem-specific domain, meaning that a CRDT-based data model would be quite suitable for implementing OBS.

7.1.2 Q2

Are the components required to express the data model for OBS possible to implement using CRDTs?

As outlined in Chapter 3.1, the system desired consists of linked unique objects serving as attribute containers. The proposed solution provides generic CRDTs for this situation capable of expressing a set of objects in a hierarchially linked structure. Globally unique IDs ensure no collisions when creating objects concurrently.

Relations are also capable of hosting attributes enabling the use of human readable IDs such as callsigns. The attributes for the objects was not outlined in time for the delivery of this report but the hardest case, sequential data editing, can be provided using RGAs. Globally unique objects, relations and attributes represent the required data model in its entirity, meaning that CRDTs are sufficiently expressive to be used in an implementation of OBS.

7.1.3 Q3

Can a system providing the required functionality be built on these components?

In Chapter 7.1.1 we showed that **R1** and **R2** holds for the system. The modified vector clock structure, while adding complexity to the system, allows us to handle the case of a changing participation list (**R3**). In order to satisfy **R4** much collision resolution is moved from the model to the user (as outlined in Chapter 7.1.1). As such a system constructed on the components proposed could provide the required functionality while ensuring strong eventual consistency. The data model implementation outlined in Chapter 5 is designed based on the requirements outlined and its validation testing described in Chapter 6.1 shows that the functionality required is possible to obtain through a pure CRDT-based data model.

7.2 Conclusion

The goal of this report was to explore the viability of using CRDTs to fulfill the requirements of OBS. The proposed solution has been shown as conceptually viable for usage - being able to utilize the strong attributes of CRDTs such as strong eventual consistency and offline usage while minimizing the impact of the weaknesses such as garbage collection. The proof-of-concept implementation of the data model shows that a CRDT-based data model is capable of handling the system requirements imposed on it by the Search and Rescue domain.

7.3 Future work

As the implemented data model described in Chapter 5 serves as a proof-of-concept several features should be expanded upon if the model should be used in its intended role. Future work divided into components of the system are listed below.

- **Operation Storage** - The provided operation storage is limited to two clients for testing purposes. In order to expand this functionality to serve any number of clients a protocol enabling a client to supply operations regardless of origin should be implemented.
- **Garbage Collection** - A mechanism for removing the unnecessary contents of operations marked by the system as resulting from cascading operations as described in Chapter 4.1.2 should be implemented, reducing the amount of information that needs to be stored.
- **Object functionality**
  - Attributes for objects as described in Chapter 4.1.1 must be implemented. Sequential data in the form of text or similar can be implemented using the RGA data type. Relation information must also propagate into these objects allowing for much better read time as discussed in Chapter 4.2.2.
  - Cascading behaviour for relations could be extended to include the objects they connect allowing users to remove objects. This expansion is not strictly necessary as the state of objects could be maintained by considering its relations, and the implementation would cause further operations to be generated (although the effect of this could be mitigated by implementing the garbage collection functionality discussed above).
- **Collision resolution transparency** - as the system will maintain structures on concurrent operation the user should be informed whenever the synchronization process between clients causes this. This could be handled in the Tree class for each data type implementation as it contains

information on the concurrent operation allowing it to report to the user when concurrent operation has forced the system to prioritize one above another.

Bibliography

[1] Redningstjenesten. *Nasjonal veileder for redningstjenesten - søk etter savnet person på land*. Norwegian. https://www.forf.no/forf/lastned.asp?_page=dokument&_id=227&_subid=154.

[2] Seth Gilbert and Nancy Lynch. *Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services*. https://dl.acm.org/citation.cfm?id=564601.

[3] Giuseppe DeCandia et al. *Dynamo: Amazon's Highly Available Key-value Store*. https://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf.

[4] The Apache Wave Foundation. *Apache Wave (incubating) Protocol Documentation*. https://people.apache.org/~al/wave_docs/ApacheWaveProtocol-0.4.pdf.

[5] Du Li and Rui Li. *An Admissibility-Based Operational Transformation Framework for Collaborative Editing Systems*. https://link.springer.com/article/10.1007%2Fs10606-009-9103-1.

[6] Du Li and Rui Li. *A Performance Study of Group Editing Algorithms*. https://ieeexplore.ieee.org/document/1655675.

[7] Marc Shapiro et al. *A comprehensive study of Convergent and Commutative Replicated Data Types. [Research Report] RR-7506, Inria – Centre ParisRocquencourt; INRIA. 2011, pp.50. ffinria-00555588f*. https://hal.inria.fr/inria-00555588/document.

[8] Hyun-Gul Roh, Myeongjae Jeon, and Joonwon Lee Jin-Soo Kim. *Replicated abstract data types: Building blocks for collaborative applications*. https://www.sciencedirect.com/science/article/pii/S0743731510002716.

[9] Leslie Lamport. *Time, Clocks, and the Ordering of Events in a Distributed System*. https://amturing.acm.org/p558-lamport.pdf.

A | OBS system requirements

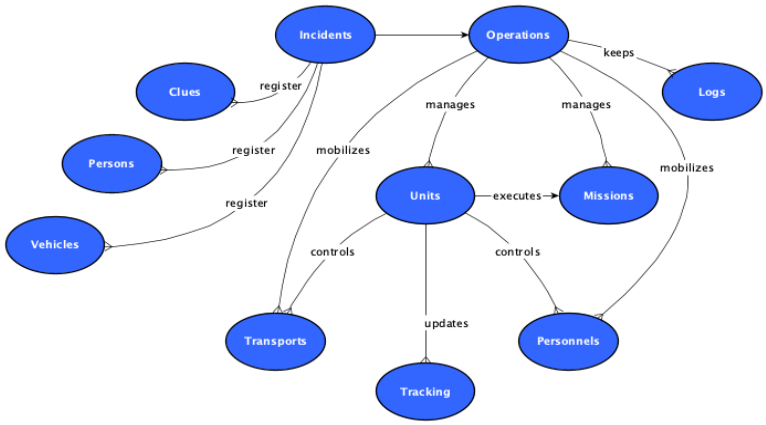
Foundations

Incident response requires resilient and fault-tolerant information systems. Incidents are defined as unplanned situations that require a coordinated response. The response follows a standardized approach to command, control, and coordinate units performing actions required to resolve the situation. Each action is performed asynchronously and decentralized, forming a series of immutable events which combined resolves the situation. This is essentially an eventually consistent append-only log of historical events, which the **Incident Response Management System** (**IRMS**) should model accordingly. The Ubiquitous Language (taxonomy) of **IRMS** and the **Incident Command System** (**ICS**) overlap on key concepts and design only, the hierarchical composition and coordination of responders differs. IRMS is essentially a subset of ICS, by defining a simplified and leaner organizational (compositional) model with only two levels of command, control, and coordination; Incident and Unit (search team, K9, snowmobile, rescue boat).

Domain in Minimal Viable Product

A minimal viable product must implement following entities:

- **Incident**
- **Clue**
- **Person**
- **Vehicle**
- **Operation**
- **Mission**
- **Personnel**
- **Transport**
- **Unit**
- **Tracking**
- **Log**



System Principles

Based on the fundamentals briefly covered above, the implementation of IRMS must abide to the following principles:

- **Offline over Online Data**
- **Data Availability over Consistency**
- **Service Choreography over Orchestration**
- **Microservice over Layered Architecture**
- **High Cohesion and Low Coupling**

Offline over Online Data. Internet connectivity on scene must be assumed to be intermittent and of low bandwidth in general, and non-existing in extraordinary situations. IRMS must therefore be resilient to low bandwidth and intermittent connectivity.

Data Availability over Consistency. Our response to incidents is inherently distributed processes with soft states that are eventually consistent. Each process manages a group of associated state objects which are considered a unity in the context of data changes. In Domain-Driven Design (DDD), this is called an Aggregate. State is shared between these processes regularly as state change events, written or verbally. This gives high degree of freedom, resilience and fault tolerance on an organizational level. When sharing state, the CAP theorem dictates that we have to choose between availability and consistency when network partitions or failures can occur. Traditional database transactions with ACID guarantees would require IRMS to implement a two-phase commit, which is inherently fault-intolerant and hard to implement with high availability in mind. IRMS architecture must instead adhere to BASE semantics (**B**asically **A**vailable, **S**oft state, **E**ventual consistency), when implementing state sharing between processes.

Service Choreography over Orchestration. Since our response is inherently distributed in areas of intermittent Internet connectivity, centralized coordination of concurrent modification of shared states (*Service Orchestration*) using two-phase commit and synchronous operations is violating the principle of *Data Availability over Consistency*. According to this principle, IRMS must implement state sharing that is eventual consistent, which implies asynchronous communication between services using messages (*Service Choreography*). Each message contains a state change, representing the action performed by the process governing the aggregate described above. Messaging must be provided by the Infrastructure Layer (see below) and ensure that same message applied multiple times must not change the result (Idempotence) by supporting Versioning and Optimistic Locking.

High Cohesion and Low Coupling over Provisioning. A single codebase compiled into a monolith could be deployed with all dependencies included. It simplifies configuration management and deploy scripts greatly, compared to a system based on a microservice architecture. Although modular monolith design could have high cohesion and low coupling, this is not enough for this project. Our open source strategy and team sourcing model requires parallel and highly decoupled workflows performed by autonomous teams (separation of concerns). The IRMS architecture must support this.

Microservice over Layered Architecture

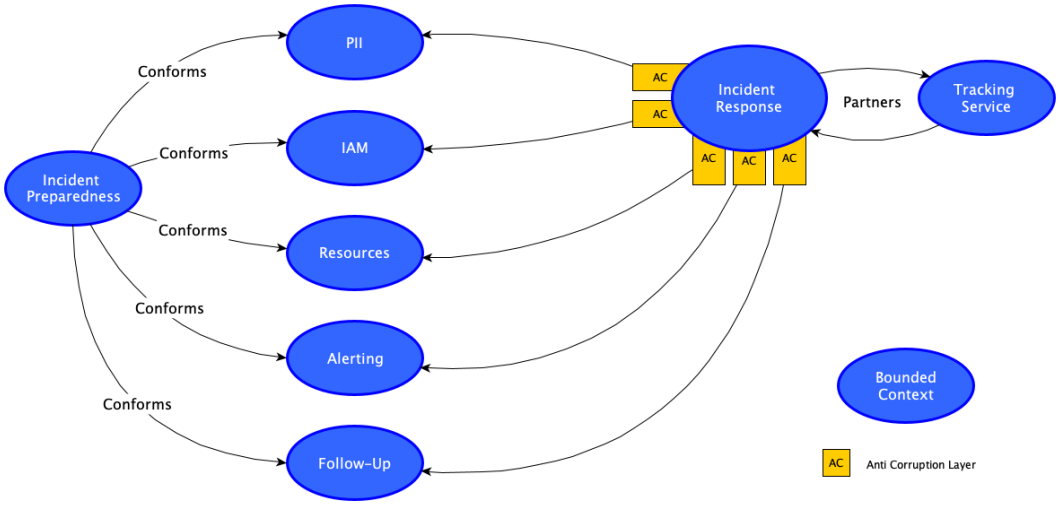
At this point, the IRMS architecture most suitable to fulfil all principles above, is a Microservice Architecture based on Service Choreography using Event Sourcing (ES) and Command Query Responsibility Segregation (CQRS). This results in an architecture that allows for highly cohesive and low coupled services, enabling the use of multiple autonomous development teams. It also allows us to implement eventual consistent sharing of states between services, which favours availability over consistency, a trait shared by the coordinated response that IRMS must support. This however, does not exclude conceptual layers in the microservice architecture, or inside each service. The four conceptual layers of User Interface, Application, Domain and Infrastructure commonly found in DDD, are still viable architectural solutions. This principle only states that when in conflict, the microservice architecture principles take precedence over layered architecture principles.

Proposed Architecture

The proposed IRMS architecture is modelled on DDD principles. Services are grouped in the four conceptual layers commonly used in DDD. The components in the User Interface layers are called **Apps**. All other components are **Services**. For completeness, the **Incident Preparedness Management System** (IPMS) is also modelled in.

Context map

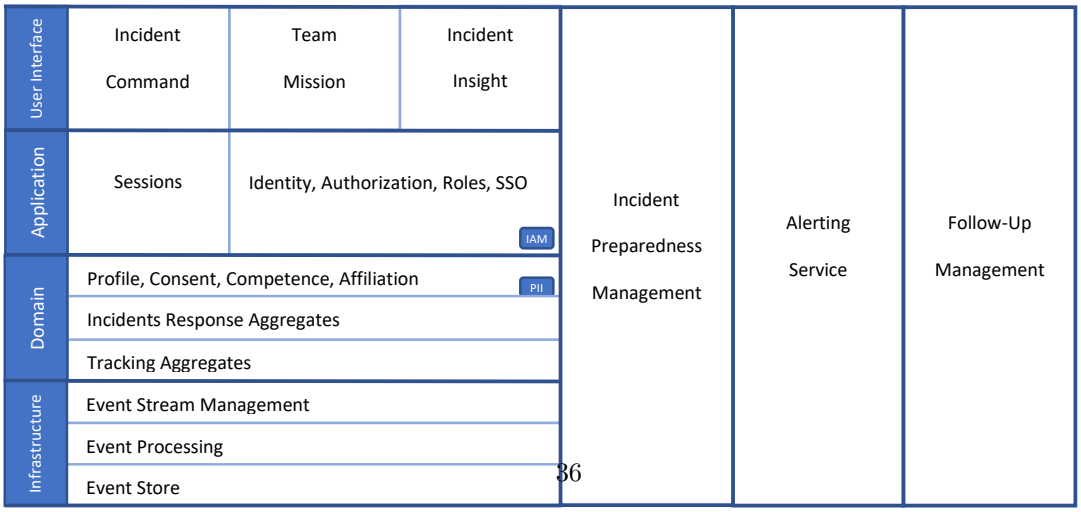
The **Context Map** (DDD) models the relationships between the **Bounded Contexts** (DDD) in IRMS.



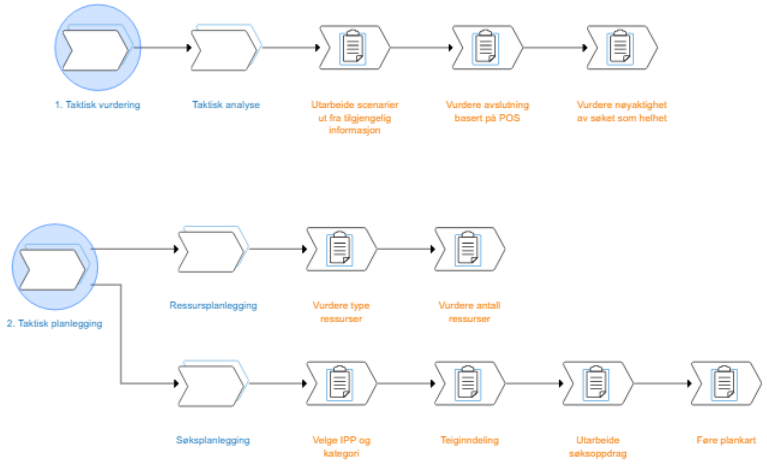
The bounded context **Incident Response** uses **Anti-Corruption Layers** to decouple from specific API implementations for services it consumes. This entails that the bounded context of Incident Response defines stable internal APIs for each external service type it depends on. The anti-corruption layers translate between the internal API and the API of consumed services. If multiple implementations of the same type of service must be supported, an **Open Host Service** must be implemented in the bounded context of Incident Response for each of these types.

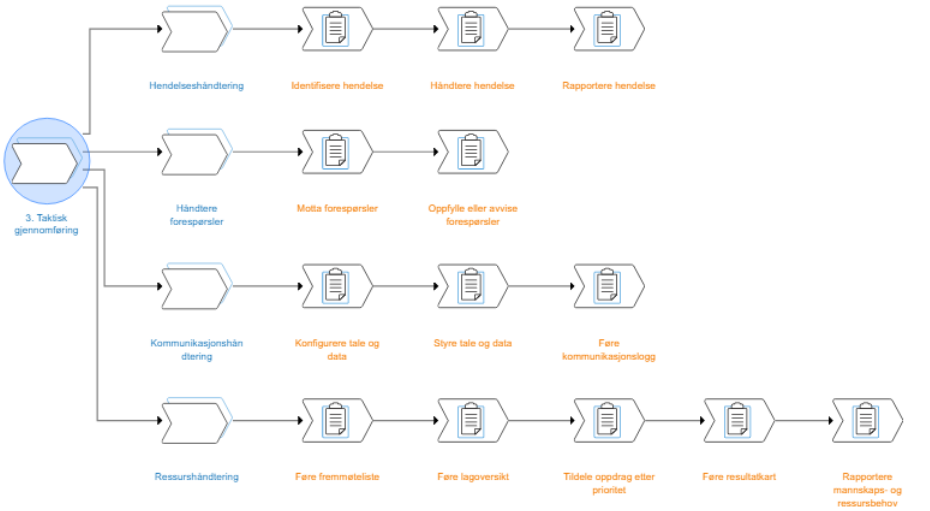
Architecture Layers

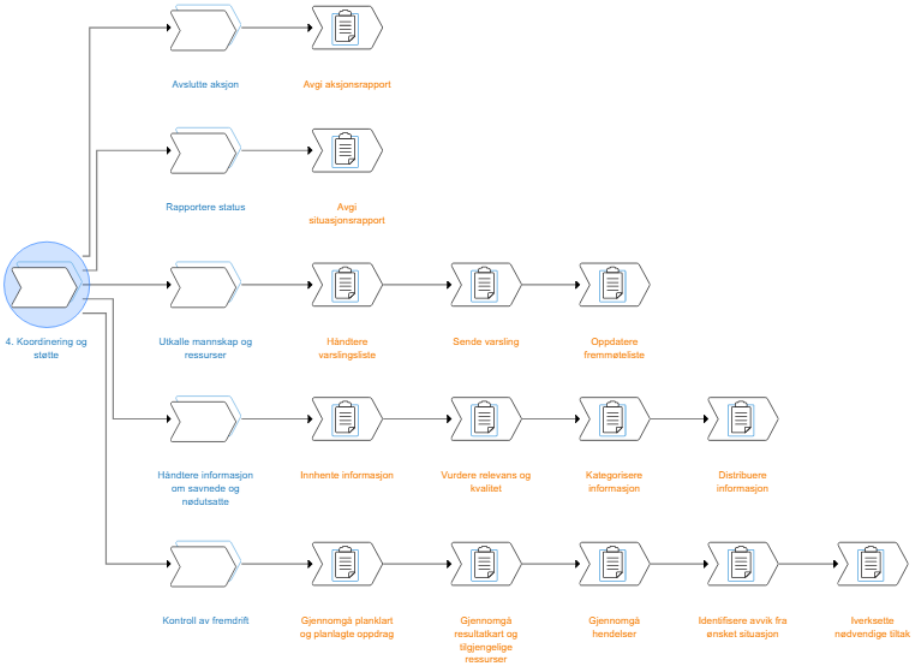
IRMS consists of the following four conceptual layers (IPMS modelled on the right):



B | OBS use cases




3. Taktisk gjennomføring
Hendelseshåndtering
Identifisere hendelse
Håndtere hendelse
Rapportere hendelse
Håndtere forespørsler
Motta forespørsler
Oppfylle eller avvise forespørsler
Kommunikasjonshåndtering
Konfigurere tale og data
Styre tale og data
Føre kommunikasjonslogg
Ressurshåndtering
Føre fremmøteliste
Føre lagoversikt
Tildele oppdrag etter prioritet
Føre resultatkart
Rapportere mannskaps- og ressursbehov


4. Koordinering og støtte
Avslutte aksjon
Avgi aksjonsrapport
Rapportere status
Avgi situasjonsrapport
Utkalle mannskap og ressurser
Håndtere varslingsliste
Sende varsling
Oppdatere fremmøteliste
Håndtere informasjon om savnede og nødutsatte
Innhente informasjon
Vurdere relevans og kvalitet
Kategorisere informasjon
Distribuere informasjon
Kontroll av fremdrift
Gjennomgå planlagt og planlagte oppdrag
Gjennomgå resultatkart og tilgjengelige ressurser
Gjennomgå hendelser
Identifisere avvik fra ønsket situasjon
Iverksette nødvendige tiltak

C | API for data model

```
Operation :
    createOperation()
    getOperation(id)

Unit :
    createOperation()
    getUnit(id)

Mission :
    createOperation()
    getMission(id)


Relations :
    assignOperationUnit(operation, unit, name)
    removeOperationUnit(operation, unit, name)

    assignOperationMission(operation, mission, name)
    removeOperationMission(operation, mission, name)

    assignUnitMission(unit, mission, name)
    removeUnitMission(unit, mission, name)
```