Tampere University

Jami-Petteri Rafael Kimpimäki

APPLYING EVENT SOURCING TO OCCASIONALLY CONNECTED SYSTEMS

Master of Science Thesis
Faculty on Engineering and Natural Sciences
Examiners: Prof. Hannu Koivisto, Prof. Mikko Salmenperä
September 2022

ABSTRACT

Jami-Petteri Rafael Kimpimäki: Applying Event Sourcing to Occasionally Connected Systems
Master of Science Thesis
Tampere University
Master's Programme in Automation Engineering
September 2022

As an information storing technique, event sourcing provides some useful properties over more traditional techniques. There is value for both application developers and end-users to be able to inspect the whole history of the application states.

Event sourcing is usually used in environments with a constant network connection and a centralized database. This thesis aimed to provide necessary means to allow use of event sourcing in occasionally connected systems.

This thesis started off the research by assuming an occasionally connected, event sourced system with a distributed multi-leader database. Problems emerging from this approach were first identified, and then solved on a conceptual level by using methods from existing literature and research. At last, a fictional case study was conducted to produce a system showcasing that the concepts introduced can be applied in practice.

There were a total of three primary problems that were identified. By making event sourcing data model bi-temporal, retroactive sharing of events proved to be achievable without violating the immutability and append-only principle of event sourcing. Conflict and concurrency detection and handling emerging from moving from single leader to multi-leader replication revealed to be a well-known problem in distributed system research around data replication. Last problem was how the system can give guarantees that information it provides to external systems will not change. This proved to be solvable by applying stability properties of distributed systems to the event sourced data model, which allowed to identify a point in the event log dividing the log into stable and unstable parts. These results together provide a foundation for building occasionally connected event sourced systems.

Keywords: Event Sourcing, Distributed Systems, CRDT, Bi-temporal, Occasionally Connected Systems

TIIVISTELMÄ

Jami-Petteri Rafael Kimpimäki: Applying Event Sourcing to Occasionally Connected Systems
Diplomityö
Tampereen yliopisto
Automaatiotekniikan DI-ohjelma
Syyskuu 2022

Tiedon tallennuksen Event Sourcing -tekniikka tarjoaa useita hyödyllisiä ominaisuuksia, joihin perinteisemmät tekniikat eivät kykene. Sekä sovelluskehittäjät, että loppukäyttäjät saavat etua tekniikan tarjoamasta mahdollisuudesta tarkastella sovelluksen koko tilahistoriaa.

Event Sourcing -tekniikkaa hyödynnetään yleensä ympäristöissä, joissa on vakaat verkkoyhteydet, sekä keskitetty tietokantaratkaisu. Tämä diplomityö pyrki tuottamaan tarpeelliset keinot, jotta Event Sourcing -tekniikan käyttö olisi mahdollista myös ajoittaisen yhteyden järjestelmissä.

Diplomityössä tutkimus aloitettiin olemattamalla Event Sourcing-tekniikkaa hyödyntävä ajoittaisen yhteyden järjestelmä, joka käyttää hajautettua, usean johtajan tietokantaa. Ensiksi tästä lähestymistavasta esiinnousevat ongelmat tunnistettiin, jonka jälkeen ne ratkaistiin konseptuaalisella tasolla olemassaolevasta kirjallisuudesta ja tutkimuksesta löydetyin metodein. Lopuksi suoritettiin fiktiivinen tapaustutkimus, jonka lopputuloksena saatiin järjestelmä, joka osoitti ratkaisujen olevan sovellettavissa käytäntöön.

Yhteensä kolme pääasiallista ongelmaa tunnistettiin. Lisäämällä Event Sourcing -tekniikalla toteutettuun tietomalliin toinen aika-akseli, retroaktiivinen tapahtumien jakaminen mahdollistui rikkomatta Event Sourcing -tekniikan tapahtumalokin muuttamattomuusperiaatetta. Yhden johtajan tietokantajärjestelmästä usean johtajan tietokantajärjestelmään siirtyminen nosti esiin ongelmia ristiriitaisten ja rinnakkaisten päivitysten tunnistamisen ja käsittelyn suhteen. Nämä ongelmat paljastuivat olevan entuudestaan tunnistettuja ongelmia hajautettujen järjestelmien tutkimuksen alle kuuluvan datan replikoimisen parissa. Viimeinen ongelma liittyi siihen, miten järjestelmä kykenee antamaan takuut siitä, ettei ulkopuolisille järjestelmille toimitettuun dataan tule enää päivityksiä. Tämä osoittautui ratkaistavaksi soveltamalla hajautettujen järjestelmien stabiliteetti ominaisuuksia Event Sourcing -tekniikalla toteutettuun tietomalliin, mikä mahdollisti tapahtumalokin jakamisen vakaaseen ja epävakaaseen osioon. Yhdessä nämä tulokset muodostavat perustan Event Sourcing -tekniikkaa hyödyntävien, ajoittaisten yhteyksien järjestelmien rakentamiselle.

Avainsanat: Event Sourcing, hajautetut järjestelmät, CRDT, bi-temporal, ajoittaisen yhteyden järjestelmät

CONTENTS

LYHENTEET JA MERKINNÄT

CmCRDT	Commutative Replicated Data Type, Operation-Based CRDT
CQRS	Command Query Responsibility Seggregation
CQS	Command Query Separation
CRDT	Conflict-free Replicated Data Type
CRUD	Create Read Update Delete
CvCRDT	Convergent Replicated Data Type, State-Based CRDT
DDD	Domain-Driven Design
JSON	JavaScript Object Notation
LPC	Local Procedure Call
OCS	Occasionally Connected System
SEC	Strong Eventual Consistency
TAU	Tampere University
TUNI	Tampere Universities
URL	Uniform Resource Locator

1. INTRODUCTION

Traditionally object-oriented software systems are developed around datastore models that try to accurately model only the most recent state of the world. When data is updated, it is updated in place, causing loss of the data the update replaces. Also often enough, without an explicit audit log, user intention behind the update is lost. Data was updated, but data itself does not describe why. When data is deleted, data is physically lost, the result is as the data never existed, and again without an explicit audit log nobody can tell the reason why data was removed.

In some cases, this kind of data loss is unacceptable, one such example is accounting. Accountants have been using append-only ledgers thorough the known history, where transactions once written into a ledger are never removed nor they are ever updated. If somehow an incorrect transaction is recorded into the ledger, another transaction is written to compensate the incorrect one, letting the ledger to show that a mistake was made, and it was corrected.

Significance and value of information for businesses has been rising for decades and continues to rise. Accurate data is no longer providing only a competitive advantage but is vital for the success in the markets. Simultaneously cost of storage space is dramatically lowering, allowing more and more data to be collected and stored. As a result, data storage techniques that prevents described kind of data loss at the cost of storage space have been adopted.

One of such techniques is called event sourcing. It is an idea that instead of storing a single state that is being updated arbitrarily in place, software should store changes to the state represented as a sequence of semantically meaningful, self-describing events that carries the business meaning with them.

At the same time environment for software systems have been changing dramatically. Computers are everywhere, and with the computers, software. Consequently, software systems are getting more and more distributed. Unfortunately, while networks are also getting better and their coverage is increasing, fundamental problems behind networking remains. Distributed software systems must be built to tolerate these fundamentals. There is no guarantee that devices like mobile phones, vehicles or laptops would have constant network connection available. In despite of that users of these devices more and more often expect the ability to work while being offline. Software must be designed around the fact that connection to network is available only occasionally.

This master's thesis is a study on applying event sourcing, an information storing architecture, to occasionally connected information systems.

1.1 Motivation

Event sourcing provides many businesses as well technical benefits to its adopters. Strong auditability is not only a luxury, but a must have property in heavily regulated sectors such as finance and health care. Being able to derive new projections from the event streams makes software extremely adaptable to future's business requirements. Ability to reach every state of an application by replaying changes against system provides developers a very powerful debugging tool. [9] [16, pp. 457-458] [18] [21, pp. 595-597]

Motivation for this thesis is that event sourced systems are often developed to run in an environment with a consistent and stable network, with a centralized database. The architecture for such systems is illustrated in the figure 1.1.

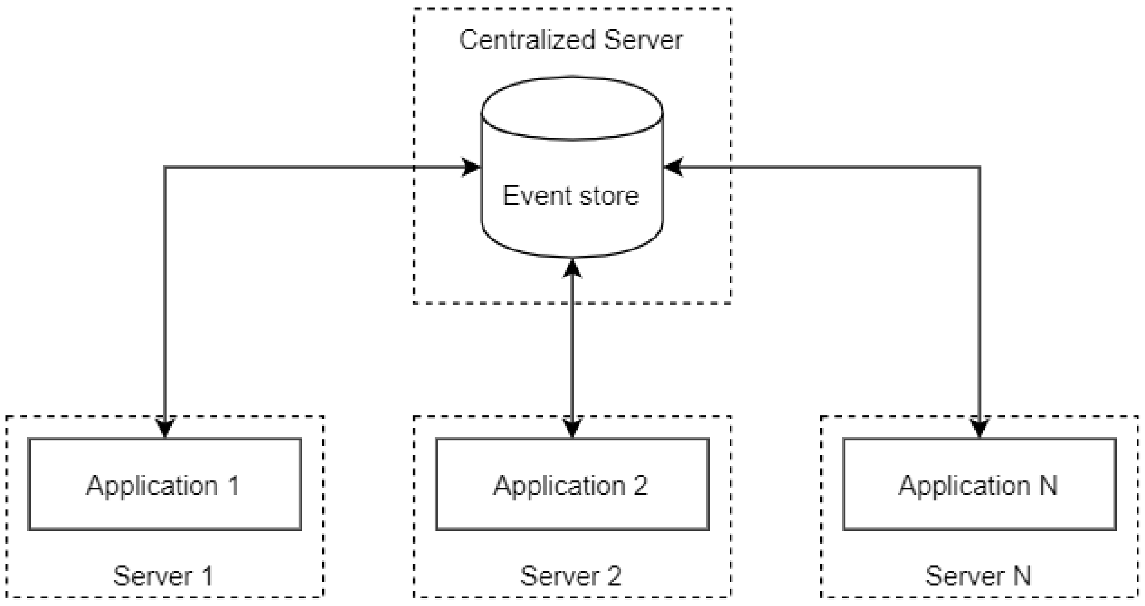

Figure 1.1. *A system with a centralized event store. Every application requires a constant connection to function properly.*

However, many software systems working in distributed occasionally connected environments where autonomy for decision making is needed and centralization of information is not an option could also benefit from event sourcing. The architecture for such systems is illustrated in the figure 1.2.

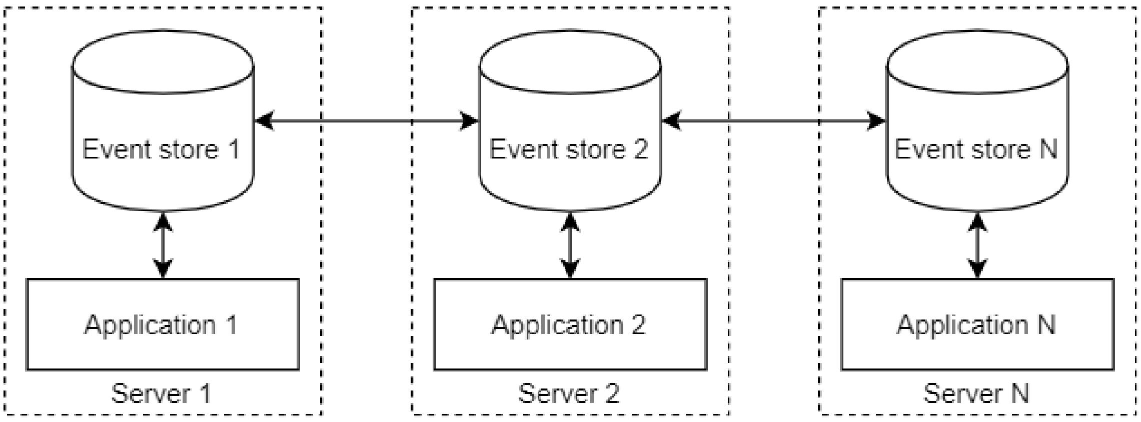

Figure 1.2. *A system with distributed event stores. Every application can work independently regardless of the connection.*

While migrating from a system with a single centralized event store to a distributed system of multiple event stores relaxes requirements for network conditions, distributing the system also brings up many challenges, such as keeping the data consistent and conflict-free. This thesis targets to identify practices and rules which would allow systems running in above-mentioned environments to gain benefits of event sourcing.

1.2 Goals and Non-goals

Research done in this thesis is applied research. First goal is to identify problems and limitations emerging from distributed event log and decision making, and to provide solutions or workarounds for them if such solutions exist. Expectation is that most of the problems are already well-known in research. This part of the thesis will be performed by use of literature and research around related topics. Second goal is to build a simple proof-of-concept, which works as a simple reference application that represented solutions can be implemented in practice.

Thus, research question is: How and with what limitations and alterations can event sourcing be used as a replicated persistence mechanism in a distributed, occasionally connected system?

This thesis does not consider aspects of solutions and problems regarding to computational complexity, networking, or performance. Problems related to development of the software, such as versioning and deprecation strategies are also disregarded. Proof-of-concept system does not aim to provide production ready solutions.

2. BACKGROUND AND DEFINITIONS

This section provides theoretical background and definitions of terms relevant to this thesis.

2.1 Occasionally Connected System

Occasionally connected system (OCS) is a term used to describe a distributed system where there is an occasional connection between the nodes, and nodes can continue working offline with local data. A node within OCS can disconnect from the network and continue working offline and is able to synchronize with the rest of the system upon connection. [18] [4]

2.2 Properties of Operations

To reason about event sourcing and related techniques, some mathematical properties of operations must be defined.

Commutativity A binary operation $*$ is deemed as commutative, if $a * b = b * a$ holds. [28]

Associativity A binary operation $*$ is deemed as associative, if $(a * b) * c = a * (b * c)$ holds. [27]

Idempotence A binary operation $*$ is deemed as idempotent, if $a * a = a$ holds. [29]

2.3 Persistence

In context of this thesis, persistence of an event means the act of appending that event to the event log in a durable way. After event is persisted into an event log, it is part of the application state. During persistence of an event the event store may add additional metadata to the persisted event, such as timestamps.

2.4 Active Record and CRUD

Active record is a design pattern, where objects represented in the software code contains both logic for acting on the data and loading and saving the data from and to the database. In addition to the pattern itself, the objects following this design are also called active records. Using active record pattern classes are modeled around the database structure, and each field of the class should be represented as a column in an SQL-database. [13, p. 160]

Models such as active record are often called CRUD-based models, where the CRUD is an acronym from Create, Read, Update and Delete -operations used to model interactions with the data.

2.5 Replication and Consistency

In distributed systems consistency refers to the problem keeping replicas of the same data consistent between different nodes of the system. Because replicas are copies of each other, updates to one replica should be reflected to other replicas as well. [30, p. 357]

Need for replication may raise from many different reasons. First primary reason is that replicating data can be used to improve system reliability. Second primary reason is performance, as data access may be a performance bottleneck for large systems. [30, pp. 357-359]

The case of occasionally connected systems falls into the category of second reason. Within occasionally connected systems there just isn't a centralized location where data could be stored so it could be accessed by every node within a reasonable time.

There are many ways to keep replicas consistent called consistency models, and each model have different limitations they impose to the system. Consistency models relevant to this research are introduced here:

Tight consistency or synchronous replication is a model, where every replicate is updated in a single, atomic operation. This is often not a feasible model in systems distributed over large networks, where quick response times are expected. [30, pp. 359-360]

Causal consistency states that potentially causally related writes must be seen in the same order by all processes. Write B is said to be potentially causally dependent on write A, if write B may have had an effect to outcome of write A. Events are said to be potentially causally related if one is potentially causally dependent on the other. Events that are not potentially causally related are said to be concurrent. [16, pp. 186-187] [30, pp. 370-372]

Eventual consistency is a weak-consistency model, with a guarantee that in absence of conflicting writes replicas will converge towards an identical state, as long as every update is propagated to all replicas. It is often seen used within large-scale distributed systems, as replication delay does not affect write operations. [30, pp. 374-376]

Strong eventual consistency (SEC) is a special case of eventual consistency. SEC guarantees that replicates will converge towards an identical state without any conflicts. However, SEC requires that underlying datatype of the replica satisfies set of conditions. Datatype that satisfies these conditions is called Conflict-free Replicated Data Type (see 2.6). [26]

2.6 Conflict-free Replicated Data Types

Different applications require different kind of consistency levels. Standard strong consistency in centralized applications may be sacrificed for better performance and scalability provided by eventual consistency in distributed applications. By accepting lower consistency, risk for conflicts rise, and with them the need for conflict resolution mechanisms such as rollbacks. [26]

Conflict-free replicated data types (CRDT) are data types, that provides strong eventual consistency (see 2.5) when they satisfy set of conditions, meaning that convergence of replicas of an CRDT is guaranteed. There are two basic types of CRDTs [26]:

1. State-based CRDTs.
2. Operation-based CRDTs.

Above-mentioned can and has been used to implement non-trivial types of CRDTs such as JSON CRDTs. [17]

2.7 Immutability

Immutability is a principle that states after some piece of data is written, it is never changed anymore. Immutable object is an object, which state does not change after it has been initialized. [24, pp. 233-235]

There exists many levels and kinds of immutability. On level of a programming language, immutability may be shallow and deep, and abstract or concrete. In deep immutability every object an immutable object references must be itself deeply immutable, whereas shallow immutability forbids reassignments only to fields of the object itself but does not prevent editing states of referenced objects. Concrete immutability mandates that an objects in-memory representation must stay the same, whereas abstract immutability only forbids changes that would cause the abstraction to change. [24, pp. 233-235]

In context of databases immutability generally implies to existence of an append-only log. Write-operations do not update objects (such as rows in SQL-database) in place, but instead write-operations are appended into a sequential log, and current state of the object is then derived from the log. Write-operations and history of the log are immutable. [16, pp. 70-71, 458–460]

Immutability provides many benefits from software engineering standpoint. As the immutability of an object guarantees that object does not change, by extension any replicas of said object are guaranteed not to change as well, and after creation of the replica no change detection is needed, making replication of immutable data trivial. Also, any deterministically derived information from immutable data will be immutable as well. [3, p. 766] [24, pp. 233-235]

As in the heart of event sourcing is an immutable, append-only log, discussion about its benefits is left for the section 2.8.

2.8 Event Sourcing

Event sourcing is a technique for storing application state as a sequence of domain events. Events are defined within the application, and views providing information about application state are derived from the events themselves. [16, pp. 457-461] [21, pp. 595-597] [9]

Current system state ($state(now)$) in event sourcing can be defined in a pseudo mathematical format as an integral over a stream of events over time ($stream(t)$) as seen in the formula 2.1. [16, p. 460]

$$state(now) = \int_{t=0}^{now} stream(t)\, dt \tag{2.1}$$

Events are considered to be immutable, and they are stored into a database in an append-only manner [16, p. 461]. By default, events in event sourcing are non-commutative and non-idempotent [22, p. 40].

There are many concepts that are central for the event sourcing. Below are definitions used in this thesis.

Event An immutable fact about something that has happened in the past. Events can be categorized to domain events, integration events and external events. None of these are exclusive to event sourcing, but domain events are in the center of it and as a result in context of event sourcing domain events are usually referenced just as events. Details related to different types of events are described in 2.9 and 2.10.

Event log An ordered sequency of domain events describing history of changes within the application state.

Event store A database storing the event log. [1]

Event stream Logical stream of events. A common solution is to have a single event stream per aggregate instance.

Actor A term originating from actor model, which is a mathematical model of concurrent computation. Actor is a computational primitive, that may have a state, and interacts with other actors using only messages. [16, pp. 138-139]

Aggregate Term originating from domain driven design. Write model of an event sourced application. Contains domain logic to be executed based on the input and generates events as output. More detailed description given in 2.10. Aggregate can be seen as a specific type of an actor.

Projection Read-only model generated from a stream of events. Queries are executed against projections. [21, pp. 599-600] Projection can be seen as a specific type of an actor.

Event processor A stateless object, that transforms incoming external events into internal representation. Event processor can be seen as a specific type of an actor.

Rehydration Process of applying historical events to an aggregate so it can achieve desired state. [21, pp. 605-608]

By designing events semantically meaningful, and by treating events and event log as immutable many benefits are gained over CRUD-based models such as active record described in 2.4. Event log not only tells what the current state is, but also how the current state was achieved. Every historical state becomes reproduceable.

From business perspective this can be useful, as new insights can be derived from the data retroactively even if the need was not identified during development. Another business benefit is software's ability to answer to temporal queries. [21, pp. 595-597].

Audit log As an event sourced system derives system state from a sequence of immutable events, the event log itself works as a strong audit log. [16, p. 461] [18]

Time travel Application state can be inspected in any point of time. From technical perspective this can be used as a powerful debugging tool, and from business perspective it may provide retrospective information about historical decision making as it enables temporal queries. [16, p. 461] [18]

Replay and reshape Existing events can be used to derive new insights from the data. As the read model is just a view or a projection for the application state,

[1] Not to be confused with Eventstore, which is a commercial implementation of an event store. [18]

and all changes to the state are recorded into events, new views can be generated retroactively. [16, pp. 461-462] [18]

Alternative realities Event sourced data can be used in "what if"-analysis for simulating outcomes of future scenarios. [16, pp. 461-462] [18]

2.9 Command Query Responsibility Segregation

Command query responsibility segregation (CQRS) is an extension of command query separation (CQS) principle. CQS states that there are two types of functions, queries, and commands. Queries are used to get information about the object and should be free of any side-effects, whereas commands are used to modify object's state. [20, pp. 748-749]

CQRS extends CQS by stating that read and write operations should be handled by different models. Reasoning behind this is that a single model would usually be a compromise between different and asymmetrical needs for write and read operations. By separating models with different set of responsibilities, both can be optimized more precisely without need for compromise. [21, pp. 669-670] [16, p. 462]

2.10 Domain-Driven Design

The name for domain-driven design (DDD) comes from a book called Domain-Driven Design: Tackling Complexity in the Heart of Software written by Eric Evans in 2003 [7]. It is an approach that emphasizes building a rich domain model with a deep understanding of business rules and processes of a complex domain. DDD guides its practitioners to build ubiquitous language which can be used to describe things in the model, but also to discuss with domain experts. It also describes many patterns for developers to take advantage of, classifies different kind of objects to categories, and offers tools how to split large domains into more manageable pieces called bounded contexts. [8] [21, pp. 3-4]

In DDD aggregate is a term used to describe a group of related stateful domain entities, that can be treated as a single unit. Generally, there is a single domain object defined within an aggregate which is called an aggregate root and performs as a single point of access to the aggregate. [8]

An aggregate's internal state is always guaranteed to be consistent, and every operation executed against an aggregate transactionally either succeeds completely or no changes are persisted. As aggregates are also responsible for protecting their internal invariants and business rules, aggregates provide a consistency boundary. Inside this boundary changes are made using strong consistency, but between aggregates changes are usually done with eventual consistency (see 2.5). [21, pp. 434-442]

Aggregates are often a central piece in event sourced applications. In event sourcing aggregate can be seen as a function that takes in event history and a command as inputs and produces new events. [21, pp. 600-602]

$$aggregate([e_0, e_1, ..., e_{t-1}], command) => [e_t, ..., e_{t+n}] \tag{2.2}$$

From event history aggregate can rehydrate its internal state and projections it needs to react to given command. [21, pp. 600-602]

DDD definition for the policy is similar to the strategy-pattern, which enables selection of an implementation of an algorithm during runtime. Policy is a pattern that is used to model the process of making a domain decision automatically when something has happened that fulfills predefined conditions. In other words, a policy listens for events, and emits commands as result. [7]

2.11 Modeling Time

In distributed system three types of clocks exists:

Physical clock or a wall clock is a clock representing time of the day.

Logical clock is a mechanism for capturing causalities between events in distributed systems.

Hybrid clock is a hybrid of the two above, used in special kind of distributed systems.

Problem with physical clocks is that they are not precise. Even if different physical clocks synchronized themselves to the same time, as soon as synchronization is finished, clocks will start to drift. If order of events generated by different nodes within a distributed system is defined by the physical timestamp, difference between clocks of nodes is included into event timestamps, and this may break causal ordering. To ensure causal ordering is preserved within timestamps, logical clocks such as vector clock were invented. [16, pp. 291-293]

Vector clock is a logical clock, where each node has its own named counted within a vector. Rules for vector clock are [3, pp. 608-609]:

1. Each counter starts from zero.
2. Whenever internal event happens, process increases it's counter by one.
3. Whenever message is sent by process, process first increases it's counter by one and then version clock is attached to the message.
4. Whenever message is received by process, it increases it's counter by one, executes cellwise max-operation between it's local version clock and message's ver-

sion clock, and saves this result as new local version clock value.

Rules for comparing vector clocks are [3, pp. 609-610] [16, p. 191] [12]:

A is equal with B if both have same value for each of the counters.

A is greater than B if at least one of A's counters has greater and every counter has equal or greater value.

A is concurrent with B if both vectors have some value greater than in the other.

Matrix clocks are a generalization of the notion for vector clocks allowing local node to know what other nodes in the system knows. [6] [3, p. 610] Matrix clock can be represented as a matrix where each column is a version clock timestamp of a single node.

2.12 Versioning

Purpose of versioning is to track changes made to different replicas of the same data. With a proper versioning mechanism in addition of detecting which of the replicas is ahead and which is behind, one can detect concurrent updates.

Version vectors and version matrix are very similar to vector clocks and matrix clocks (see 2.11). Difference comes from the different purpose of use, and slightly simpler rules of update. Whereas vector clocks are a mechanism for providing a partial order for events within a distributed system, version vectors and matrices are used to track updates on replicated data. Rules for comparison are same as in vector clocks, but rules for updates differs a bit [12]:

1. Each replica has its own counter within the vector, each counter starts from zero.
2. When an update happens, replica increases it's counter by one.
3. When two replicas are synchronized, resulting version vector is a cellwise maximum of the two synchronized vectors. In case version vectors are concurrent, conflict resolution may be needed (see 4.3).

Version matrix is similarly a generalization of version vectors, where as matrix clock is generalization of vector clock.

2.13 Stability in Distributed Systems

A function which maps the set of global state of nodes to a Boolean value is called a global state predicate. A property of the system is defined to be stable, if once the system enters to a state where predicate associated with property returns true, it will return true in all future states. [3, pp. 614-615]

3. METHODOLOGIES

Research problem in this thesis is both a computer science problem and a software engineering problem, and thus approach used in this thesis is both theoretical and practical. System under research shares characteristics across many known types of system, and as such shares many known properties and problems of these types of systems.

First goal was to first identify these problems by their properties and then to review what literature exists around them. Second goal was to find solutions for these problems from the literature. At last final goal was to build a proof-of-concept system to showcase that these concepts can be implemented.

3.1 Assumptions and Restrictions

To define goals, solution space must be constrained first. There are following assumptions made about the system under research:

1. System consists of nodes that requires autonomy in decision making and can operate offline.
2. Nodes are distributed and databases within nodes can be seen as a distributed database with a multi-leader replication.
3. Connection between nodes is occasional.
4. System nodes are running the same application code.
5. Information is learned in real-time, but nodes may exchange it retroactively.
6. Amount and identities of the system nodes is known and fixed.

Proposed solutions have to adhere following restrictions:

1. System nodes must use event sourcing or it's variant to store their state.
2. System nodes must be able to share information between each other.
3. System nodes must be able to build a common view of the world's state.
4. System nodes must not lose information when sharing information between nodes. This includes system's ability to tell what was known when each decision was made.

3.2 The Conceptual Model

In the chapter 4 problems were analyzed, and solutions were provided on a conceptual level. Literature was used as a main source for information. However, this was not a pure literature review, and reflection is present in the chapter. This was a necessary approach as the subject is one that presents combination of multiple problems known under distributed system research, and literature alone couldn't provide ready made answers.

3.3 Proof of Concept

During the research a system was built for testing and proving the solutions. Resulted system was divided into two parts. First part was a framework that provides necessary tools to work with the data in a generic manner. Second part was an application using said library to implement functionalities for an imaginary domain. Both the framework and the application were implemented using C#-programming language and Microsoft Visual Studio.

4. THE CONCEPTUAL MODEL

There is no silver bullet solving all the problems within distributed systems, and occasionally connected systems as a subset of distributed system are not an exception. During this research no generic solutions that fits all systems were found, but many problems identified in this thesis proved to be solvable.

This chapter goes through problems and possible solutions for them in a conceptual level. It forms a theoretical toolbox for developers working with systems similar to one described. These solutions are then implemented in practice in the chapter 5.

4.1 Retroactive Updates

First problem to be solved was to give nodes ability to receive new data retroactively, without compromising auditability which is provided by an append-only log. Different nodes need to be able to share their understanding of the history between each other. Information that is old for one node may be new for another, and that must be accounted in the modeling.

Simple way to join two event logs of different nodes together would be doing a sort-merge join between them, especially as both event logs are already sorted by definition. [19] [9] There are two reasons why this wouldn't work as one could hope:

1. Many event sourcing benefits are gained from the assumption that event log is append-only, and thus editing event logs is considered to be a bad practice and even forbidden. [5] [16, p. 457]
2. Knowledge is encoded into the events, but also into the ordering of the sequence which the events were applied, which is information that doing a simple sort-merge join would lose. [9]

In order to retain information stored into the ordering of the events and thus full auditability, a different solution is needed.

4.1.1 Bi-temporal Event Sourcing

As nodes learn retroactively about the past events from each other, normal uni-temporal event sourcing is inadequate to retain all necessary information such as causalities. Solution to make editing history possible with append-only event log is making the event log bi-temporal.

Making event log bi-temporal means that each event has two timestamps instead of one:

1. Actual time, valid time or "occurred at".
2. Record time, transaction time or "recorded at".

Valid time of an event tells when the event occurred whereas record time tells when the event was recorded, or in other words when did a node learn about each event. [11] [31] As there are multiple nodes each recording its knowledge about history, each will have a unique record time for each of the events. With this solution auditability of the event log and context of decisions made by each node are not lost, as each node is able to know what they knew at the moment of each decision, even when the history is supplemented with new data retroactively.

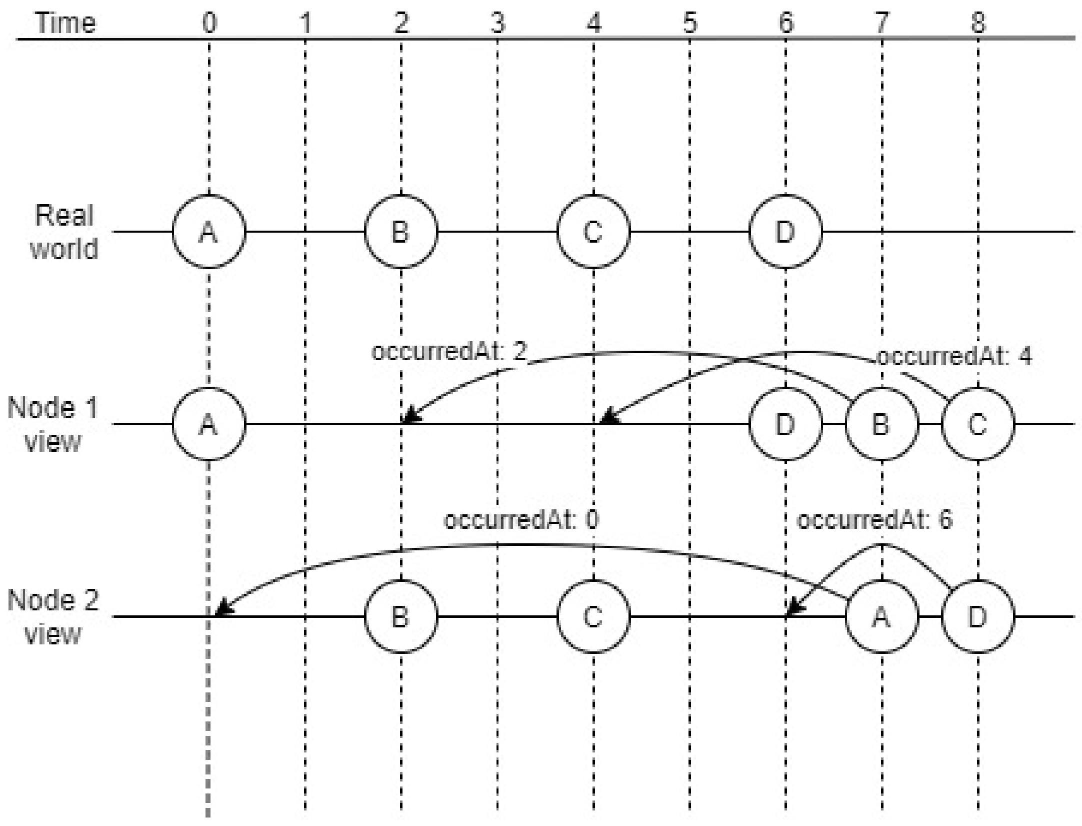

Figure 4.1. *Bi-temporal event logs of two nodes. Location of events shows record time, and arrows point to actual time.*

Visualization of this can be seen in the figure 4.1. At point of time $t = 8$ both nodes 1 and 2 has the full knowledge of the history. However, if either one of them made a decision at $t = 6$, the decision would be made based only on partial knowledge. With a uni-temporal model the context of decision would be lost during synchronization at $t = 8$, but bi-temporal model is able to preserve this information.

Reading Bi-temporal Event Log

Uni-temporal event log can be queried by simply by reading and applying events sorted by occuration time $t_{occurred}$ starting from the oldest. In order to query what the state was at specific time system simply needs to stop reading events when $t_{occurred} > t_{projection}$ becomes true. [21, pp. 595-597]

In bi-temporal model there is another aspect to consider. Instead of only asking what system state was at specific projection time $t_{projection}$, query must also provide the time for the viewpoint $t_{viewpoint}$. This means that when in a uni-temporal system query can be described as following:

What does system think state was at $t_{projection}$?

In a bi-temporal system query also includes the viewpoint time and can be described as following:

At $t_{viewpoint}$ what did system think state was at $t_{projection}$?

This type of bi-temporal query allows three different projections. The first two can be thought as special cases of the third one.

1. "As at": System state at given time as seen at given time ($t_{projection} = t_{viewpoint}$).
2. "As of": System state at given time as seen at current time ($t_{viewpoint} = t_{now}$).
3. "As of until": System state at given time as seen at other specific time. [31] [23]

To make such queries algorithm for reading uni-temporal event log must be edited only slightly. By adding condition to skip events for which is true $t_{recorded} > t_{viewpoint}$ to the algorithm.

Retroactive Events

Software rarely can count on that there wouldn't be any erroneous inputs. Problem is that inputs with errors usually means that there will be errors in the outputs too. When there are problems in the output, tools to fix those errors retroactively are needed. For event sourcing that tool is retroactive events. There are three types of errors occurring in events:

1. Out-of-order event that was processed late. When bi-temporal model is used, this means an event with incorrect $t_{occurred}$.
2. Rejected event that never should have been processed.
3. Incorrect event that contained incorrect data. [10]

As editing event log is discouraged, to correct these events system must support appending retroactive events. For uni-temporal system this would violate append-only mechanic, but bi-temporal system can handle these without violations, as history can be edited with appended events. As events are processed in order defined by $t_{occurred}$, incorrect events can be corrected by using retroactive by injecting suitable events:

1. Rejected event: Inject an event that makes the system ignore the rejected event at process time.
2. Incorrect event: Inject a correction event with corrected information and reject the incorrect event.
3. Out-of-order event: Inject an event that makes the system process event at given time and ignore the original process time. [10]

Use of retroactive events needs careful consideration in regard to causalities. If incorrect events already have affected to more recent events making them causally dependent, editing incorrect ones retroactively could cause problems that needs to be attended.

4.2 Detecting Concurrent Updates

If aggregates can be modified by multiple nodes or users, concurrent modifications will appear. To detect concurrent modifications, and possible causal dependencies between events, a strategy is needed.

4.2.1 Concurrency Detection Within a Single Event Store System

Within a system with a single event store possible causality tracking and concurrency detection with optimistic concurrency is quite trivial. Because there is a centralized entity, concurrency can be detected during operation time. This can be implemented in a following way:

1. Each aggregate should have a version number.
2. When aggregate generates an event, it increments version number by one, and this new expected version is included to the event.
3. When event is sent to event store for persisting, event store compares event's version number to the latest stored one. If latest version is equal or greater than the

expected version in the event being persisted, event store knows that events were concurrent and can return an error response to the client.

4. Client/aggregate can then handle the conflict as it sees fit. It may reload aggregate from the memory and reapply the command, or it may return an error to the user.

As concurrent events are not accepted by event store, event streams are totally ordered and free of conflicts. [21, pp. 618-620]

4.2.2 Version Vectors for Detecting Concurrent Modifications

Within an OCS concurrent updates cannot be detected during persisting. Instead, all updates must be persisted into event log, and event log should record necessary metadata which can be then used for concurrency detection during synchronization. To detect concurrent updates between replicas of an aggregate within an OCS, version vectors can be used.

Version vector is an object containing a named counter for each node that can update the replica of the aggregate. Every time a node updates it's local replica of an aggregate, it increments its counter inside the version vector. [16, p. 191] [12]

Comparison between version vectors is done by comparing counters within the vectors. Absence of a counter can be thought to be equal to zero. Rules of comparison and updating are described in 2.11 and 2.12.

In an event sourced system every update applied to an aggregate generates events, so version vectors can be included to events themselves. During a synchronization of two streams between two nodes, following actions should be taken:

1. If both streams have equal versions, no synchronization is needed.
2. If one stream has greater version than other, changes should be replicated to the one with smaller version.
3. If streams were updated concurrently, conflict resolution may be needed. See 4.3 for details.

After synchronization current version is the one of the latest updates, or in case of concurrent updates cell-wise max-operation between latest concurrent vectors. With this approach consumer of an event stream can detect whether the events happened concurrently or not. [16, p. 191] [12]

4.3 Resolving and Avoiding Conflicts from Concurrent Updates

Previous chapter represented solutions for conflict detection. However, system's ability to detect conflicts is not enough, and application must be able to avoid or handle them as well. This chapter introduces a few ways to avoid or to resolve conflicts.

4.3.1 Dynamic Ownership

One way to avoid conflicting updates between aggregates is to ensure that updates are never applied concurrently. If system can ensure that all updates to a single aggregate are generated by the same node, this is achieved. [16, p. 172]

This can be done using a mechanism called dynamic ownership or mutual exclusion token, which is a mutual exclusion mechanism implemented by using token passing. [3, pp. 634-635] [2]

Applying these principles to system under research, implementing dynamic ownership should comply with following rules:

1. Ownership initially belongs to the node that created the aggregate.
2. Ownership belongs only to a single node at time.
3. Ownership can be transferred to another node only during synchronization. This effectively happens if transferring of ownership is modeled as an event.
4. Only node with ownership is allowed to modify aggregate's state.
5. Nodes without ownership are allowed only to read aggregate's state.

With these rules conflicting updates are avoided.

4.3.2 Conflict-free Replicated Data Types

By designing aggregates as CRDTs, states of replicas will always converge without conflicts. [26]

Aggregates designed using CQRS and event sourcing area already good candidates for commutative replicated data types (CmRDTs) / operation-based CRDTs. This is because two-phase operation of CmCRDTs matches to command-event -separation, where command does not mutate the state but prepares an event based on the invocation arguments (command) and current state of the aggregate. [25]

4.3.3 Priority Groups

Priority group is a method for automatic conflict resolution which can be used when amount possible values are limited, and each can be given a unique priority. It can be useful when target property models a workflow. [1]

4.3.4 Human Assisted Resolution

Sometimes conflict avoidance or automatic conflict resolution is just not possible, and human interaction is required. To ensure that conflicts are handled deterministically, only one of the nodes can generate the resolution event. To enforce this there must be rules which node and when conflicts can be resolved. Some possibilities for this are:

1. Conflicts are always resolved during synchronization, or synchronization is rejected.
2. Conflict resolver is decided during synchronization. Appending to stream is forbidden until conflict is resolved. Node without resolver status will have to synchronize with resolver node again after conflict is resolved.
3. Conflict resolver is decided based on static rule. Otherwise, rules are the same as above.

4.4 Information Stability for External Systems

Second problem was that not all systems can deal with volatile information. For example, software systems may be expected to be able to generate weekly/monthly/annual reports providing a snapshot about the current state. If the state is volatile, data derived from it will be volatile too. For a system to be able to answer queries with stable responses, the system must be able to identify when its state is stable.

4.4.1 Stable Timestamp

Timestamp is defined to be stable if every node has seen updates predating said timestamp. With vector clocks described in 2.11 stable timestamp can be defined as a cellwise minimum over current timestamp of each node. With matrix clocks described in 2.11 stable timestamp from perspective of a single node can be defined as a row wise minimum over current timestamp. [3]

In this thesis no vector clocks or matrix clocks are used. However same rules apply to version vectors and version matrices used here.

4.4.2 Stable History

History (event log) can be divided into stable and unstable parts. Stable part is the part preceding stable timestamp (including event with stable timestamp).

Temporal queries can take advantage of this property. As long as query targets to only the stable part of the history, result is guaranteed to be stable as well.

Table 4.1. *Illustration of a stable timestamp. Synchronization is defined as an atomic operation*

T	Event	A	B	Stable
0	-	[0, 0]	[0, 0]	[0, 0]
1	E1 at A	[1, 0]	[0, 0]	[0, 0]
2	E2 at B	[1, 0]	[0, 1]	[0, 0]
3	E3 at B	[1, 0]	[0, 2]	[0, 0]
4	Sync A <-> B	[1, 2]	[1, 2]	[1, 2]

In table 4.1 there would be two stable timestamps: $[0, 0]$ until sync, and $[1, 2]$ after.

4.4.3 Stable History and Retroactive Updates

For history to gain stability, retroactive updates (see 4.1.1) cannot be allowed to be appended further than the unsynchronized part of the event log. It is also good to note that unsynchronized part is not the same as unstable part mentioned in the 4.4.2. If nodes were allowed to add any new information that predates any synchronization operation, stable timestamp could not anymore be calculated.

4.5 Event Processors: Events as Inputs

As aggregates are constructs that take in commands to update state while protecting invariants (see 2.10), events as inputs must be treated differently. Source of external events may be a physical entity the system is monitoring, or it may be another software application. In both cases from system's perspective external event is just a notification received from outside about something that has happened.

Even with assumption that every external event will eventually be received, important thing to consider is that there may not be guarantee that external events are delivered in causal order. Thus, there are times when the system is operating with partial information only, and the system can not guarantee that invariants are valid. Problem is same as the one described in 4.4, with the exception that with external events requirements for stability comes from outside.

In search of solutions a few assumptions were made:

1. All external events are eventually received.
2. External events can be treated as unique. In case of an external source, the source should provide a unique identifier for each event. In case of a physical entity

being monitored, duplicate observations made simultaneously by different nodes should produce (nearly) identical events which can be differentiated by tuple of $(timestamp, payload)$, where $payload$ includes necessary information to identify the physical entity.

3. External events are delivered in causal order to the system, even if events are received by different node.
4. External events contain necessary information for determining causal order, or it can be determined during receiving.

4.5.1 Event Processors: Rules for Persistence

To maintain full world view, every incoming event must be persisted (see 2.3). System most likely needs to add data to incoming events before persisting them, but there are some limitations what kind of data can be included. Because causal delivery order is not guaranteed, until event history achieves stability, any computed state can't be guaranteed to be causally consistent. This limits use of events, as users will have to deal with potential changes until stability is reached. This also means that transformation of incoming events to persisted events cannot rely on a computed, potentially unstable state, before stable timestamp within the system reaches the event, as persisted events should be treated immutable.

4.5.2 Event Processors: Ownership of Event Source

In the special case when system can guarantee that only one specific node is the listener for a stream of events, node has guarantees about ordering of events and limitations from previous section do not apply. This is because in this case events are always observed in a causal order.

4.6 Summary

Results found from the literature show that most of the problems identified in this thesis do have known solutions.

First problem identified here was maintaining auditability provided by event sourcing when receiving data retroactively from other nodes. Solution for this problem turned out to be making event log bi-temporal.

Second problem was related to detecting and resolving concurrent and conflicting updates. This problem could be generalized as multileader replication, which is a well-known research problem within context of distributed systems and databases, and thus there were plenty of literature around the topic. Version vectors described in the literature provided a good solution for detecting concurrency and thus possible conflicts. For automatic conflict resolution no universal solution was found, but JSON-CRDTs were the closest one.

Third problem emerged from solving the first problem with bi-temporality: if data can change retroactively, how can system provide robust information for external systems to consume? Answer was provided from distributed system research, where matrix clocks are used to keep track of other nodes vector clocks. Same technique could be used with version vectors, providing nodes an ability to know what other nodes at least know.

Fourth part was focused on events and event flow. They were identified to be a special case, but after all identified problems related to them could be solved with solutions above.

5. PROOF OF CONCEPT

Proof of concept product created as a part of this thesis contains three main parts. First part is an implementation for an event store. To keep things simple, event store uses only in-memory persistence, and communication happens in-process using local procedure calls (LPC). However, abstraction of the event store client could also be used also with remote protocols. Second part is a framework with some generic components that provides application developers necessary building blocks. As a third part an application using said framework and event store was implemented. Source code for the application is provided in a Github-repository [14].

5.1 Goals

As the topic of research in this thesis was quite wide and goal was to collect known methods from literature, not every solution described in theoretical level in this thesis were included in the proof-of-concept work. Goals for the implementation were following:

1. System should retain auditability and not lose information during synchronization.
2. System should allow temporal queries.
3. System should be able to detect conflicts on stream level.
4. System should be able to detect when and what information is stable.

Library provides an implementation for dynamic ownership for conflict avoidance, but generic CRDT-implementation (such as Automerge[1]) or interactive conflict resolution was not provided. CRDTs are already formally proven to avoid conflicts when correctly implemented and re-implementing well-known CRDTs here does not increase research value of this thesis.

5.2 Overview

Overview of system architecture of a system with three nodes is presented in figure 5.1. As seen from the picture, each node is running its own instances of an application and an event store.

[1]Martin Kleppman's library implementation for generic JSON-CRDTs. [15]

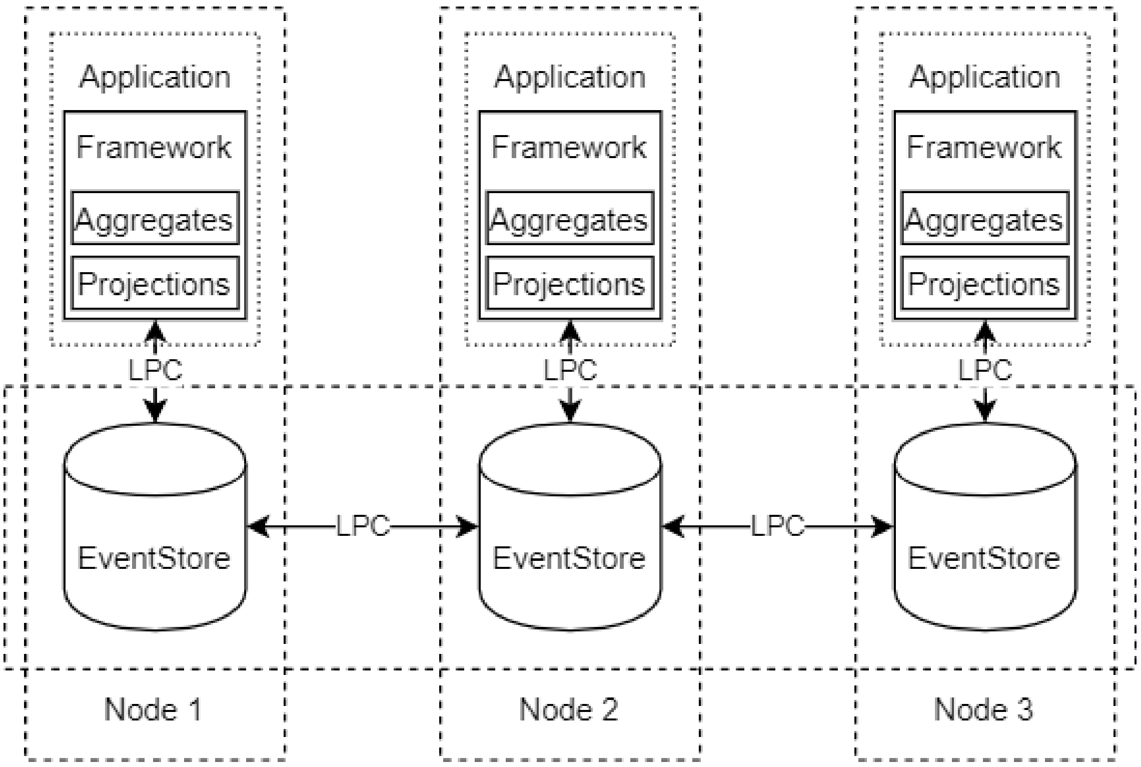

Figure 5.1. *Overview of system architecture with three nodes. Communication channels between nodes 1 and 3 are not drawn to keep the picture clear.*

Application defines projections and aggregates (as they are domain specific constructs), but framework provides frame for them. Application itself can interact with aggregates and projections only through framework, which handles rehydration and other generic issues (see 5.4).

5.3 Event Store

In addition to storing the events, event store implementation is responsible for tracking versions of nodes and streams. As discussed in 4.4 and 4.3, these features are needed to detect conflicts and stable state of the system.

Event store organizes events into streams, and every event appended to the event store needs to specify to which stream (other than main stream) it belongs to. Stream structure is shown in the figure 5.2. Each stream indexes events in the order they are recorded by the event store. Main stream indexes all the events, and other streams only the events appended to that particular stream.

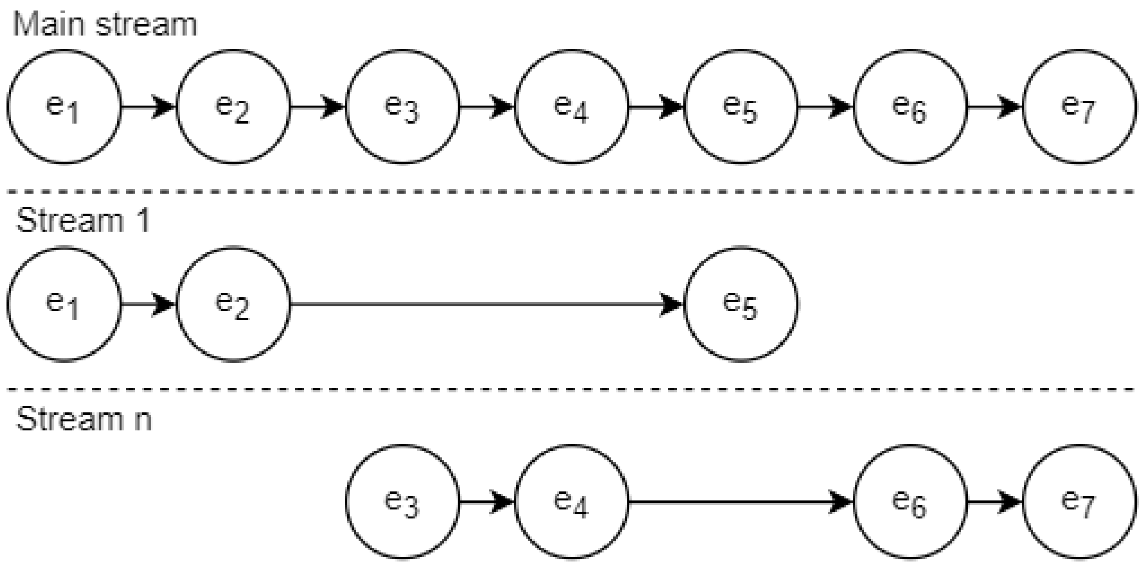

Figure 5.2. *Overview of logical streams in event store. Streams are identified by `StreamId`.*

Primary reason behind dividing events into streams is to isolate events with potential causal dependencies from each other. Each stream has its own version vector, which is updated every time event is appended into the stream. Having this version vector allows event store to detect differences and concurrent writes between replicated streams during synchronization. Secondary reason for separation is to allow performant rehydration for stateful entities such as aggregates. If there were only a single stream (main stream), every rehydration would require reading all of the events in the event store, but with stream per aggregate approach this can be avoided.

Because the main stream is also a stream, it also has a version vector. This version vector is updated every time the event store instance receives an event and is used for synchronization and stability.

These versions are also stored within the events themselves. There are total of three different version vectors and two physical clock timestamps that the event store stores within events. While physical timestamps are not necessary from system perspective, they are much more convenient to use from business perspective that version vectors.

`OccurredAtNodeVersion` which is node version which existed when the event was raised. This is used for stability and for concurrency detection between nodes.

`RecordedAtNodeVersion` which is node version which existed when the event was recorded by a particular node. This is used for providing information about in what order did node learn about events. `RecordedAtNodeVersion` is unique for each node.

`StreamVersion` which is stream version of the stream when the said event was raised. This is used for concurrency detection. This is stream level equivalent for `OccurredAtNodeVersion`. Because `RecordedAtNodeVersion` already provides total order on node level, it also provides total order on stream level, and no stream level equivalent is needed.

`OccurredAt` is physical time when event originally occurred. This is used to allow temporal queries against projections.

`RecordedAt` is physical time when event was recorded by a particular node. This is used together with `OccurredAt` to allow bi-temporal queries against projections. `RecordedAt` is unique for each node.

5.3.1 Synchronization and Stability

Synchronization is initiated by nodes, but the protocol itself is included within the event store. In the implementation node sends a synchronization command to its own event store (referenced as $local$ now on) and includes URL for the target event store (referenced as $remote$ now on). $Local$ event store then handles the protocol, which consists of following steps:

1. Get current node versions (main stream versions) of both $local$ and $remote$.
2. Read all events from $local$ that have `OccurredAtNodeVersion` that is newer or concurrent to node version of $remote$.
3. Read all events from $remote$ that have `OccurredAtNodeVersion` that is newer or concurrent to node version of $local$.
4. Store all events read from $remote$ to $local$.
5. Send all events read from $local$ to $remote$.
6. Read version matrices from both $local$ and $remote$.
7. Send version matrix read from $local$ to $remote$, and version matrix read from $remote$ to $local$.

After synchronization both event stores have same set of events, and identical system version matrices. As stable timestamp is derived from the system version matrix, stable timestamp is also updated. This mechanism guarantees that all events that are older than current stable timestamp are known by every node (see 4.4.2).

5.3.2 Concurrency Detection

Event store allows detection of concurrent updates within a stream by using version vectors (`StreamVersion`). Figure 5.3 shows how concurrency is detected within a single stream with two replicas being updated concurrently. Arrows in the figure represents potential causality where potential cause points to an effect.

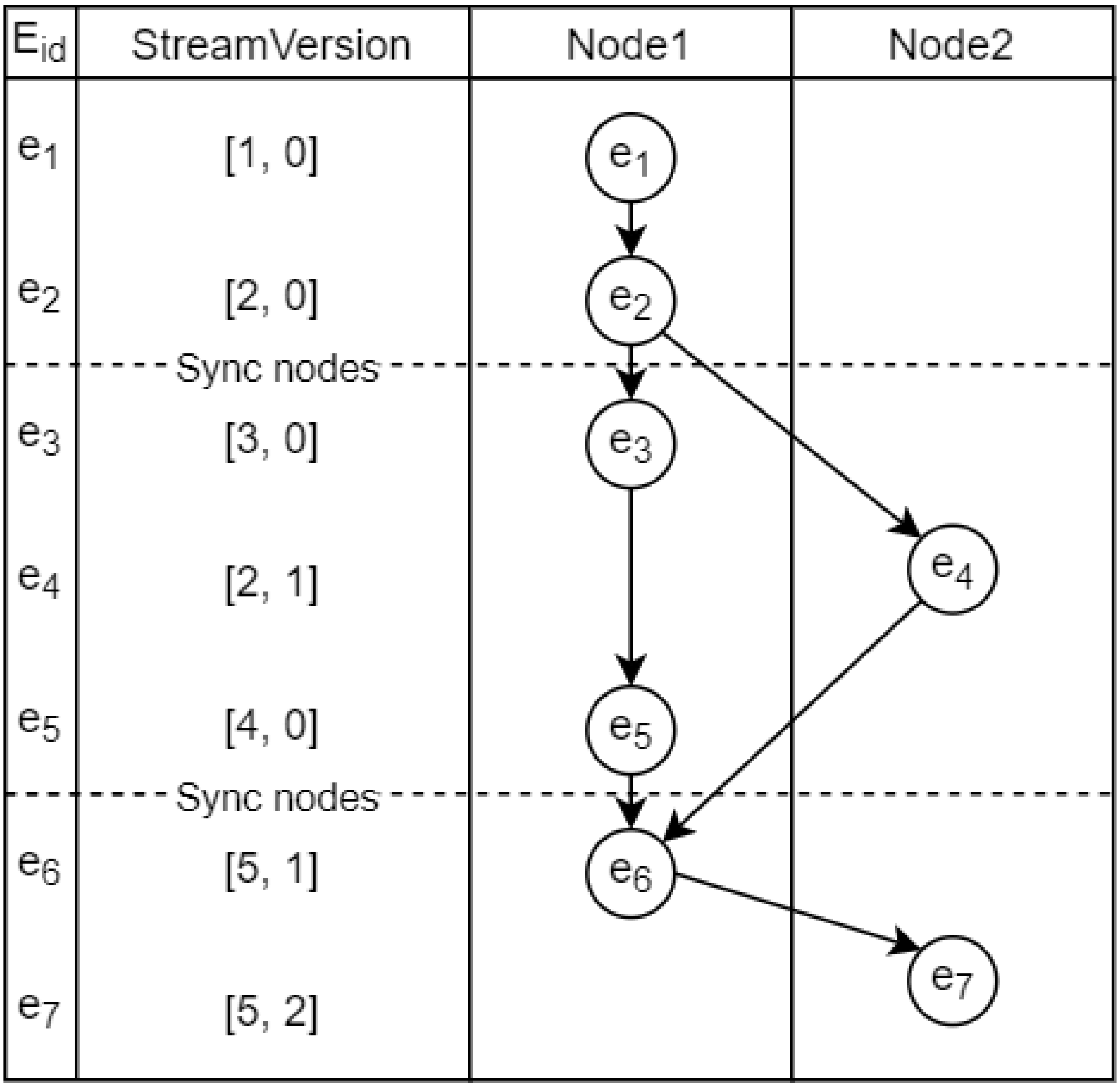

Figure 5.3. *Stream versioning. Picture shows a single stream that is replicated to two nodes. `StreamVersion` can be used to detect concurrent updates and potential causality between events. Events e_3 and e_5 raised by node 1 are concurrent with event e_4 raised by node 2.*

`StreamVersion` saved to each event is provided to the framework and application when rehydrating, allowing conflict handling mechanisms to utilize this information.

5.4 Framework

Framework defines a collection of generic components and a frame, that allows building occasionally connected systems using event sourcing without mixing the details of topics discussed in this thesis into the business logic.

Communication within the framework is built around different types of messages: commands, events, and queries. Information is stored only as events (except for version matrix within the event store), and information from events is derived using aggregates and projections. For event store framework defines an interface and provides an in-memory implementation by default.

5.4.1 Node

Entry point for the framework is called `Node`. It was designed to work as a simple facade, that applications built around the framework can easily take use of. `Node` exposes three methods:

- `Handle` which is a method for executing commands.
- `Query` which is a method for executing queries.
- `Synchronize` which is a method for initiating a synchronization.

Because business logic is defined by the application, and `Node` needs to have access there, some integration is required. Frameworks solution to define integration is to have a `NodeBuilder` which provides a convenient and guided way to do the wiring. The `NodeBuilder` exposes following methods:

- `AddThisNode` which registers identity of this current node and a URL for it's event store.
- `AddKnownNode` which registers another node in the system and a URL for that nodes event store.
- `RegisterSerializer` which registers a serializer that handles serialization for a specific event.
- `UseEventstore` which registers client used for connecting to the event store.
- `RegisterSimpleAggregate` which registers an aggregate that doesn't implement kind of conflict handling mechanism.
- `RegisterDynamicallyOwnedAggregate` which registers an aggregate with dynamic ownership conflict avoidance model.
- `RegisterCommand` which maps a command to an aggregate.
- `RegisterProjectionQueryHandler` which registers a query and a projection that handles the said query.
- `Build` which returns an instance of the node with registered capabilities.

Builder-pattern was chosen as it makes building process of complex objects a more straight forward and clean from the application perspective. Details are described in later sections.

5.4.2 Aggregates

Aggregates represents individual, independent pieces of state that can be changed atomically. Framework defines an interface that aggregates must implement to integrate with the framework. This interface consists of four methods/properties:

`StreamId` is the ID that identifies the aggregate. This is needed for rehydration and event persistance logic, so existing events can be read from, and any new events can be appended to the correct stream.

`UncommittedEvents` is a list containing all the events a command execution created before they are stored to the event store.

`Commit` is a method that commits the uncommitted events. If aggregates are cached in-memory and not re-instantiated for every command, this needs to at least clear the uncommitted event list. Framework will call this after successfully saving events to the event store.

`Rehydrate` is called before executing commands. Rehydrate method should playback given events so aggregate reaches the state that the command should be executed against, which is usually the current state.

In addition to the bare interface, framework offers a few base classes that has some features implemented.

`AggregateBase` which requires inheritors to implement only `Rehydrate`-method.

`DynamicallyOwnedAggregate` which implements dynamic ownership model.

`OperationBasedCRDTAggregate` which works as an operation-based CRDT as long as user defined commands and events satisfies requirements of operation-based CRDTs (see 2.6).

Workflow with the aggregates is simple. User sends a command via `INode`, this is the only way how user can interact with aggregates. Command contains a `StreamId` that identifies the aggregate and matches it to a stream. Framework queries the event store for events from that stream and rehydrates the aggregate state to its current state. Then command is executed against the current state, raising new events executing the state changes. Each command is an atomic operation, meaning that all the events raised by a single command will be accepted and stored, or none will. After command execution is finished, framework commits new events to the events store.

5.4.3 Projections

For implementing projections to serve users and external systems in the application side library defines an interface `IProjectionQueryHandler<TQuery, TResult>`. Projections in this library supports two query modes: Latest and stable. Each projection can support only one of these. Based on this mode projection is rehydrated with either latest available data, or data that is guaranteed to be stable system wide. Design choice was made to restrict projections to these two modes, as this frees queries from carrying version vectors making implementation of queries and projections simpler. Considered factor in favor of restriction was also that it is probable that those business cases that would benefit from using both kind of data from the same projection are rare.

Workflow for projections is similar to aggregates, with some differences. Basic flow is quite similar: User sends a query via `node`, application rehydrates projection state, projection handles the query and returns query result to the user.

First difference to aggregates is that projections support temporal properties in query. This means that queries contain `asAt` and `asOf` properties that are typed as `DateTime`. By using these properties applications are given ability to perform temporal queries. These are forwarded to the event store, which handles them as described in 4.1.1 during rehydration process.

Second difference is that projections filter events by type instead of `StreamId`. Each projection describes which types of events it accepts to, and only these are read during rehydration. It is good to note that rehydration delivers events in order of recordation, and causal delivery order is guaranteed only within a single stream.

5.4.4 Policies

Policies (see 2.10) were implemented in a very similar way as projections. Policies declares which events they are interested in, and those events are then provided for them for rehydration of state. Using this state policies can then make decisions which commands, if any, should be dispatched. Framework calls the `Trigger`-method for each policy, and as a response each policy should return a list of commands to be executed.

5.5 Case Study

Scenario considered in this case study as an example here is a fictional case set into the forestry industry. Forestry was chosen as an example industry because forestry sites are often in rural environments with poor network conditions, meaning they are prime example for occasionally connected systems.

Goal in this scenario is to enable accurate tracking of wood from forests to warehouses. This data could be used for example to track down from which site and even from which tree in the site did wood-eating bugs come to the wood warehouse. Let's start by defining some terms.

Site is an artificially divided geographical area that is a forest or a piece of forest which is required to be cut down.

Log is a piece of a full tree log that was cut to length.

Bunch is a pile cut-to-length logs.

Harvester is a machine with a human operator that cut downs trees into bunches.

Harvester manager is a harvester on the site, who is responsible for reporting completion of the harvest in the site.

Forwarder is a machine that transports trees out of the forest and loads them onto trucks for transportation.

Log truck is a truck that transports logs and bunches on roads.

Warehouse is a physical location, where wood is transported before shipping to factories and other buyers.

Order is an order to cut down a site. In system context it is issued by a salesperson.

In this scenario workflow starts by customer ordering a harvest to customer's site, and finishes when site is reported to be completed, logs gathered from the site are transported to the warehouse, and the salesperson has generated a completion report.

At first order is inserted into the system by a salesperson. After insertion of the order, order is forwarded to a harvester, appointing said harvester to act as a harvester manager for the target site. There can be one or more harvesters on a site, but only the harvester manager will be able to report when the order is completed.

After order is issued, harvesters on the site cuts down trees to make logs and drops logs onto the ground forming bunches. Forwarders collects these bunches, transports bunches next to the road for the trucks to pick up, which then transports logs to warehouse. After whole site is harvested, harvester manager will report the site as harvested. Then salesperson can generate accurate reports about the harvest and mark the order as completed.

5.5.1 System model

Most of the complexities and logic are handled by the framework described in the chapter 5.4. Because of framework doing majority of the work there are only two things to implement, which are the interface to interact with the system, and the domain model.

As this implementation is based on a fictional environment, there are no external systems feeding inputs to the system. For this reason, only interface that was built into system was a simple command line interface. This interface can used by giving inputs by hand, or it can be given a special command that reads commands from a text file.

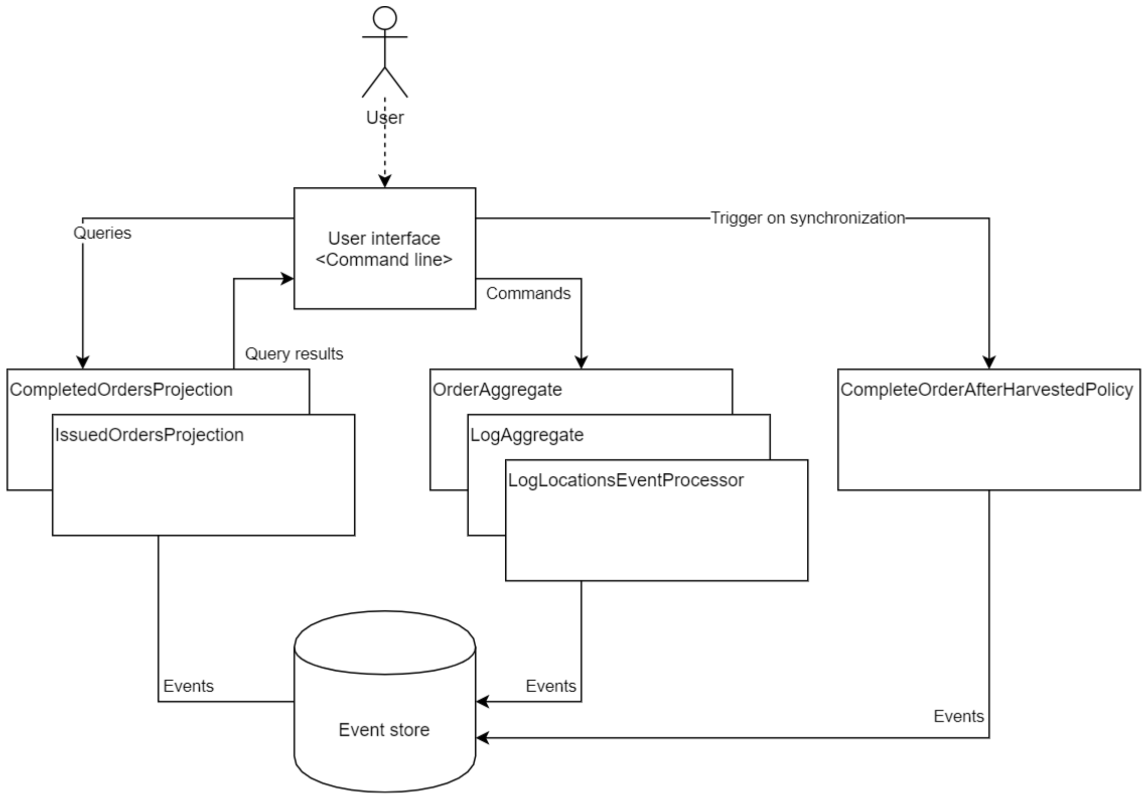
User
User interface <Command line>
Queries
Query results
Commands
Trigger on synchronization
CompletedOrdersProjection
IssuedOrdersProjection
OrderAggregate
LogAggregate
LogLocationsEventProcessor
CompleteOrderAfterHarvestedPolicy
Events
Events
Events
Event store

Figure 5.4. *Message flow in the proof-of-concept-system. User interacts with the system using a command line interface.*

Figure 5.4 shows an overview how user interaction with the system works. User is able to send commands and queries by using CLI (command line interface) to be processed by the aggregates, the event processors and the projections within the system. Results for queries are printed on the screen.

OrderAggregate

Order represents a workflow involving multiple actors, where each state transition is done by a different actor. This makes order a prime example to be modeled as a dynamically owned aggregate. By applying the dynamic ownership to orders, system can enforce that each state transition of the order is made only once, and by the correct actor. Workflow for the order is following:

1. Salesperson issues an order.
2. Salesperson assigns the order to a harvester, transferring ownership of the order to the harvester.
3. After site is harvested, assigned harvester sets order into a harvested state, and transfers ownership back to the salesperson.

4. After system state is stabilized, harvesting report is generated and attached to the order, and order state is set to completed.

Model and message flow for order created on workflow above contains following messages:

`IssueOrderCommand` which creates an order and assigns the ownership to the initial salesperson. Emits an `OrderIssuedEvent`.

`AssignToHarvesterCommand` which sets order into `InHarvest`-state and then transfers ownership to assigned harvester. Emits an `AssignedToHarvesterEvent` and an `OwnershipTransferredEvent`.

`ReportOrderSiteHarvestedCommand` which sets order into `Harvested`-state and then transfers ownership to the salesperson. Emits a `SiteHarvestedEvent` and an `OwnershipTransferredEvent`.

`CompleteOrderCommand` which assigns a report to the order and sets order into `Completed`-state. Emits an `OrderCompletedEvent`.

LogAggregate

`LogAggregate` is an aggregate that transforms harvester's actions of cutting down a tree and into logs to events.

Harvester workflow to cut down a tree into logs starts by first identifying a tree. In the identification process harvester saves the location and the species of the tree. After identification it cuts the tree down, and then cuts it to logs of specific length.

It could also be modeled as an event processor because it is only a monitoring system monitoring the actions taken by the operator of the harvester. However, it was modeled as an aggregate, because assumption was made that the same harvester will finish the workflow from identifying the tree to cutting the log, so statefulness can be allowed. `LogAggregate` deals with following messages:

`IdentifyTreeCommand` emits a `TreeIdentifiedEvent`.

`CutDownTreeCommand` emits a `TreeCutDownEvent`.

`CutLogToLengthCommand` emits a `LogCutToLengthEvent`.

LogLocationsEventProcessor

In regard of tracking log movements and to track which log ended up where, first assumption made was that a forwarder operator or a forwarder does not possess an ability to identify the logs on the ground. Another assumption was that the forwarder may move the logs without the system having information about them, as the operator can see the logs physically on the ground without help of the monitoring system.

Keeping these assumptions in mind, only way to identify log at its current location at the site is to have a record of complete chain of movements starting from the location of the cut down. `LogLocationsEventProcessor` was modeled as an event processor because it must be stateless, as it cannot rely on having received information about movements executed by other forwarders. `LogLocationsEventProcessor` list of messages is:

`PickBunchCommand` emits a `BunchPickedByForwarderEvent`

`GroundBunchCommand` emits a `BunchGroundedByForwarderEvent`

`LoadBunchToTruckCommand` emits a `BunchLoadedToTruckEvent`

`StoreToWarehouseCommand` emits a `StoredToWarehouseEvent`

As `LogLocationsEventProcessor` cannot differentiate between the logs with the same coordinates, and an assumption is made that in the case where the logs are indistinguishable by location, assumption is made that forwarder/truck will pick them all as a single bunch. Without this assumption system would not be able to accurately track the logs, unless forwarder were able to identify logs it separated from the bunch.

CompleteOrderAfterHarvestedPolicy

To automate the process of completing orders after all the required information is gathered, instead of requiring the salesperson to manually generate and request order completion, a policy (see 2.10) can be used. In that case policy's state would be rehydrated just like projection's state, but instead of replying to queries it causes a dispatch for a `CompleteOrderCommand` directly.

Implementation of `CompleteOrderAfterHarvestedPolicy` is quite simple. As order completion requires a stable state, the policy subscribes only for stable events. It subscribes for events of type `SiteHarvestedEvent` and `OrderCompletedEvent`, and all the movement events generated by `LogLocationsEventProcessor`. Each order that is harvested, but not yet completed, should be generated a order completion report and given a `CompleteOrderCommand` with the generated report. This report fulfills the requirement for the case by providing information from which tree were each of the logs was cut down, and what was the route each log did take to get from the tree to the warehouse.

Report generation ignores default event ordering defined by `NodeVersion`, and instead uses physical timestamp `OccurredAt` to order them. It can do this, because movement events from `LogLocationsEventProcessor` are generated by a physical source, meaning that even though events may happen concurrently from the system's viewpoint, in the reality they are causally dependent, and because `SiteHarvestedEvent` is always generated only after `OrderIssuedEvent` has already occurred. Thus, as long as physical timestamps are correct, no causality violations will happen.

A minor issue that this approach will cause is commands to be generated on multiple nodes. However, because targeted `OrderAggregate` is a dynamically owned aggregate, commands executed on nodes without the ownership will be rejected, and possibility for duplicated actions to be performed is avoided.

For simplicity, in the system policies are triggered on both nodes after every synchronization by the CLI. This is an acceptable simplification, as the only policy in the system is `CompleteOrderAfterHarvestedPolicy`, which subscribes to a stable state (hence triggers only on synchronization) and is known not to cause any kind of chain reaction (e.g., a policy sends commands, which cause new events, which cause another policy to send commands). In an event store implementation targeting to an actual production system, some kind of event streaming solution would probably be required.

CompletedOrdersProjection

`CompletedOrdersProjection`'s responsibility is to allow user to query completed orders and see reports attached to them. As all the information it needs from each order can be found from `OrderCompletedEvents` alone, and thus there are no causality related concerns to keep in mind, `OrderReportProjection` does not need to use stable projection mode. Reports are generated using stable projection mode, so they can be shared externally.

IssuedOrdersProjection

`IssuedOrdersProjection`'s responsibility is to allow user to query issued orders and see who issued them, and when. This projection listens only for `OrderIssuedEvents`, which contains all the information it needs. Report is generated using latest projection mode.

5.5.2 System Evaluation

Evaluation of how system met criterias set for it was conducted using a suite of automated tests. These test scenarios were built in order to produce situations, in which problems demonstrated in this thesis would occur. In these scenarios nodes simulate the work, share information between each other and then resulting state is queried and query results validated. Testing was focused to validate three things:

1. Bi-temporal data should work as expected. This allows system to retain auditability, prevent information loss on synchronization and makes temporal queries possible.

2. Stable and unstable parts of the event log should identified correctly.
3. Concurrent updates to streams should be detected.
4. Concurrent updates to streams with dynamic ownership should be prevented.

The axes of bi-temporal data in the system were `OccurredAt` and `RecordedAt`, or in other words when did an event occur and when did the node learn about it. Idea behind validating bi-temporality is exemplified by following:

1. Let's assume a system of two nodes, A and B.
2. At t_0 do work e_1 on a node A.
3. At t_2 synchronize nodes A and B.
4. Validate that when queried from A, work e_1 occurred at t_0 and was recorded at t_0.
5. Validate that when queried from B, work e_1 occurred at t_0 and was recorded at t_2.
6. Validate that when queried from A, using t_1 as a viewpoint (`RecordedAt <=` t_1), work e_1 occurred at t_0 and was recorded at t_0.
7. Validate that when queried from B, using t_1 as a viewpoint (`RecordedAt <=` t_1), no work has occurred.

A node considers a timestamp to be a stable timestamp, when the node knows for sure that every node in the system has seen all the events prior to the stable timestamp. To validate information stability properly, an example of a system of three nodes can be studied.

1. Let's assume a system of three nodes, A, B and C.
2. At t_0 do work e_1 on a node A.
3. At t_1 synchronize nodes A and B.
4. At t_2 synchronize nodes A and C.
5. Validate that when queried from A or C, e_1 is considered stable. A and C knows that both of them and B knows about e_1.
6. Validate that when queried from B, e_1 isn't considered stable. B doesn't know if C does know about e_1.
7. At t_3 synchronize nodes A and B.
8. Validate that when queried from A, B, or C, e_1 is considered stable, as B now knows that C knows about e_1.

Synchronization protocol detects concurrent updates by checking, if both nodes involved in synchronization contains uncommited events for the same stream. This can be detected during synchronization.

1. Let's assume a system of two nodes, A and B.

2. At t_0 do work e_1 on a node A, where e_1 targets stream s.
3. At t_1 do work e_2 on a node B, where e_1 targets stream s.
4. At t_2 synchronize nodes A and B. Validate that synchronization protocol reported that there were concurrent updates to stream s.

These testing concepts were put into practice by using matching automated test scenarios which are described in appendices D, E, and F.

5.6 Summary

Proof of concept demonstrates how the concepts introduced works together. Framework and concepts presented in this thesis were used successfully to build a system providing required capabilities:

1. System nodes are capable of independent decision making.
2. System is capable of sharing the information between nodes, while detecting concurrent updates and avoiding data conflicts.
3. System uses bi-temporal event sourcing preserving context for each decision made.
4. System is able to provide stable information for external systems to consume.

As such this proof-of-concept demonstrates that solutions introduced in chapter 4 can be applied to a real system on a logical level. Whether limitations regarding performance or other non-functional characteristics makes demonstrated solutions unfeasible for production systems is left unknown, as that question was outside of the scope of this thesis.

6. CONCLUSIONS

This chapter gives an overview of the results and provides some considerations about limitations of the proposed solutions. After summarising results, possible future research ideas are discussed.

6.1 Summary of Results

There was a total of four fundamental problems to be solved, when moving from centralized to distributed event log. Bi-temporal event sourcing described in section 4.1 provided a way to share events between nodes without compromising the append-only nature of the event log. Bi-temporal data has been around for a long time, but bi-temporal event sourcing seemed to be quite rare occurrence. Section 4.2 presented version vectors as a solution for the concurrency detection. Conflict handling in event sourcing is an application-level issue, and section 4.3 provided a few different ways to deal with them. Both concurrency detection and conflict handling are well-known problems, and a lot of research and literature can be found around the topics. Final conceptual problem solved was related to the information stability in the section 4.4. Without being able to tell when the information is subject to change, external software systems would need to take this into account as well, rendering use of event sourcing in occasionally connected systems infeasible.

With the tools above, a simple proof-of-concept was built. It consisted of an event store with support for bi-temporal events, an application framework providing conflict handling mechanisms and a demo application showcasing that all these parts do work together. While there remains lot of other things to consider when building systems in the real world, results did show that event sourcing can be applied successfully to occasionally connected systems.

6.2 Known Problems and Future Research

While solutions for the identified problems were found, some limitations exist. Some of these limitations are well known and ways over these can be found from the literature, but others may require more research.

6.2.1 Limitations of Proposed Solutions

Because stable timestamp is calculated by default using all the nodes from the system, even a single node being offline for a long time can prevent the system as a whole reaching a stable state for long periods of time. To allow large systems to reach stability, large systems should be divided into smaller sets of nodes in order to allow relevant sub-states to stabilize faster. Mechanisms for this are not in the scope of this thesis but should be considered if results are applied into practice.

A well-known problem in the distributed system research related to version vectors is state explosion. In large systems where may be over hundreds or thousands of nodes, version vectors become quite large, consuming a lot of space. There are mechanisms such as dotted version vectors and interval tree clocks that solves some practical issues related to version vectors.

This thesis considered mainly systems with fixed number of known members. In the actual production environment, there often is a need to allow nodes to join or leave the system, which would require consideration from stability and version vector perspective.

6.2.2 Known Issues in Event Sourced Systems

Practical drawback for event sourced systems is that they require more resources from the infrastructure, than equivalent non-event sourced system. Rehydration of the state is quite expensive operation, and events requires a lot of disk space. Storing the rehydrated state in a form of snapshots periodically can lower the cost for rehydration, as not all events in the stream will have to be reapplied from the start of time. If old events are not an area of interest, combined with snapshotting old event logs can be pruned, trading some benefits gained from event sourcing to reduced disk usage.

Software systems are rarely ready when they are first released. In event sourced systems known issue is related to event versioning. To have future systems to work with legacy events, and sometimes other way around, careful planning is required.

6.3 Conclusions

Event sourcing is a powerful technique, that provides many capabilities absent in traditional systems. It is traditionally applied in domains requiring strong audit capabilities, operating in environment with constant network. To give application developers ability to use event sourcing in new domains, this thesis provides a way to apply event sourcing to occasionally connected systems. It collects a set of methods from the field of distributed systems research, provides known limitations of these methods, and describes how to apply those to allow use of event sourcing in occasionally connected systems.

REFERENCES

[1] Oracle Corporation. *Conflict Resolution Concepts and Architecture*. Accessed 13.02.2022. URL: https://docs.oracle.com/cd/B12037_01/server.101/b10732/repconfl.htm#23858.

[2] Oracle Corporation. *Using Dynamic Ownership Conflict Avoidance*. Accessed 06.02.2022. URL: https://docs.oracle.com/cd/A81042_01/DOC/server.816/a76959/repadv.htm#1767.

[3] George Coulouris et al. *Distributed Systems: Concepts and Design, Fifth edition*. Addison-Wesley, 2011. ISBN: 978-0-13-214301-1.

[4] Udi Dahan. *Occasionally Connected Systems Architecture*. Accessed 22.01.2022. Apr. 2007. URL: https://udidahan.com/2007/04/04/occasionally-connected-systems-architecture/.

[5] Microsoft documentation. *Event Sourcing pattern*. Accessed 12.12.2021. URL: https://docs.microsoft.com/en-us/azure/architecture/patterns/event-sourcing.

[6] Lúcia M.A. Drummond and Valmir C. Barbosa. “On reducing the complexity of matrix clocks”. In: *Parallel Computing* 29.7 (2003), pp. 895–905. ISSN: 0167-8191. DOI: https://doi.org/10.1016/S0167-8191(03)00066-8. URL: https://www.sciencedirect.com/science/article/pii/S0167819103000668.

[7] Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley, 2004. ISBN: 978-0321125217.

[8] Martin Fowler. *DomainDrivenDesign*. Accessed 01.02.2022. Apr. 2020. URL: https://martinfowler.com/bliki/DomainDrivenDesign.html.

[9] Martin Fowler. *Event Sourcing*. Accessed 12.12.2021. Dec. 2005. URL: https://martinfowler.com/eaaDev/EventSourcing.html.

[10] Martin Fowler. *Retroactive Event*. Accessed 19.12.2021. Feb. 2005. URL: https://martinfowler.com/eaaDev/RetroactiveEvent.html.

[11] Martin Fowler. *Temporal Patterns*. Accessed 13.12.2021. Feb. 2005. URL: https://martinfowler.com/eaaDev/timeNarrative.html.

[12] Martin Fowler. *Version Vector*. Accessed 06.02.2022. June 2021. URL: https://martinfowler.com/articles/patterns-of-distributed-systems/version-vector.html.

[13] Martin Fowler et al. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, 2002. ISBN: 978-0-3211-2742-6. URL: https://www.martinfowler.com/books/eaa.html.

[14] Jami-Petteri Kimpimäki. *Forester Application*. Source code for the proof-of-concept implementation. URL: https://github.com/kimpimaj/forester.

[15] Martin Kleppmann. *Automerge*. Accessed 29.03.2022. URL: https://github.com/automerge/automerge.

[16] Martin Kleppmann. *Designing Data-Intensive Applications*. O'Reilly Media, Inc., 2017. ISBN: 978-1-4493-7332-0. URL: https://www.oreilly.com/library/view/designing-data-intensive-applications/9781491903063/.

[17] Martin Kleppmann and Alastair R. Beresford. "A Conflict-Free Replicated JSON Datatype". In: *CoRR* abs/1608.03960 (2016). URL: http://arxiv.org/abs/1608.03960.

[18] Event Store Ltd. *Event Store*. Accessed 22.01.2021. 2022. URL: https://www.eventstore.com/.

[19] *Merge Join Algorithm*. Accessed 12.12.2021. URL: https://www.javatpoint.com/merge-join-algorithm.

[20] Bertrand Meyer. *Object-Oriented Software Construction*. 2nd ed. CQS. 1997, pp. 748–751.

[21] Scott Millett and Nick Tune. *Patterns, Principles, and Practices of Domain-Driven Design*. Wrox, 2015. ISBN: 978-1-1187-1470-6. URL: https://www.wiley.com/en-ie/Patterns%2C+Principles%2C+and+Practices+of+Domain+Driven+Design-p-9781118714690.

[22] Michael L. Perry. *The Art of Immutable Architecture: Theory and Practice of Data Management in Distributed Systems*. Apress, 2020. ISBN: 978-1-4842-5954-2. URL: https://link.springer.com/book/10.1007/978-1-4842-5955-9.

[23] Thomas Pierrain. "As Time Goes By...(a Bi-temporal Event Sourcing story)". Domain-Driven Design Europe 2018. Sept. 2018. URL: https://2018.dddeurope.com/speakers/thomas-pierrain/#talk2.

[24] Alex Potanin et al. "Immutability". In: *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*. Ed. by Dave Clarke, James Noble, and Tobias Wrigstad. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 233–269. ISBN: 978-3-642-36946-9. DOI: 10.1007/978-3-642-36946-9_9. URL: https://doi.org/10.1007/978-3-642-36946-9_9.

[25] Marc Shapiro et al. *A comprehensive study of Convergent and Commutative Replicated Data Types*. 2011. URL: https://hal.inria.fr/file/index/docid/555588/filename/techreport.pdf.

[26] Marc Shapiro et al. "Conflict-Free Replicated Data Types". In: *Stabilization, Safety, and Security of Distributed Systems*. Ed. by Xavierand Petit Dfago and Vincent Franckand Villain. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 386–400. ISBN: 978-3-642-24550-3.

[27] European Mathematical Society Springer Verlag GmbH. *Associviaty*. Accessed 30.01.2022. 2016. URL: http://encyclopediaofmath.org/index.php?title=Associativity&oldid=37385.

[28] European Mathematical Society Springer Verlag GmbH. *Commutativity*. Accessed 30.01.2022. 2014. URL: http://encyclopediaofmath.org/index.php?title=Commutativity&oldid=34081.

[29] European Mathematical Society Springer Verlag GmbH. *Idempotence*. Accessed 30.01.2022. 2016. URL: http://encyclopediaofmath.org/index.php?title=Idempotence&oldid=39755.

[30] Maarten van Steen and Andrew S. Tanenbaum. *Distributed Systems*. Third edition. Maarten van Steen, previously Pearson Education, Inc., 2020. ISBN: 978-90-815406-2-9.

[31] Jan Stenberg. *Retroactive and Future Events in an Event Sourced System*. Accessed 13.12.2021. Feb. 2018. URL: https://www.infoq.com/news/2018/02/retroactive-future-event-sourced/.

APPENDIX A: IMPLEMENTATION OF VERSION VECTOR

Listing A.1. *Implementation for version vector.*

```
public class VersionVector
{
  public enum Comparison
  {
    AreSame,
    IsOlder,
    IsNewer,
    AreConcurrent
  }

  private readonly Dictionary<string, int> _value;

  public VersionVector()
  {
    _value = new Dictionary<string, int>();
  }

  public VersionVector(Dictionary<string, int> value)
  {
    _value = Copy(value);
  }

  /// <summary>
  /// Updates value for a node
  /// </summary>
  /// <param name="node">Node to be updated</param>
  /// <returns>VersionVector with updated value</returns>
  public VersionVector Next(string node)
  {
    var copy = Copy(_value);

    if (!copy.ContainsKey(node))
    {
      copy[node] = 0;
    }

    copy[node]++;

    return new VersionVector(copy);
  }

  /// <summary>
  /// Compares two version vectors.
  /// </summary>
  /// <param name="another">Comparison target</param>
  /// <returns>Comparison result</returns>
  public Comparison ComparedTo(VersionVector another)
  {
    var a = Copy(_value);
    var b = Copy(another._value);

    var notInB = a.Where(kvp => !b.ContainsKey(kvp.Key)).ToList();
    var notInA = b.Where(kvp => !a.ContainsKey(kvp.Key)).ToList();

    // Empty value equals zero value
    notInA.ForEach(kvp => a.Add(kvp.Key, 0));
    notInB.ForEach(kvp => b.Add(kvp.Key, 0));

    var areSame = a.All(kvp => b[kvp.Key] == kvp.Value);
    if (areSame)
    {
      // All values were same.
      return Comparison.AreSame;
    }

    var aIsAfterB = a.All(kvp => kvp.Value >= b[kvp.Key]);
    if (aIsAfterB)
    {
      // Every value in A was greater
      // than or equal to values in B.
      return Comparison.IsNewer;
    }

    var bIsAfterA = b.All(kvp => kvp.Value >= a[kvp.Key]);
    if (bIsAfterA)
    {
      // Every value in B was greater
      // than or equal to values in A.
      return Comparison.IsOlder;
    }

    // Both contained values greater
    // than in other.
    return Comparison.AreConcurrent;
  }

  public static bool operator >(VersionVector a, VersionVector b)
  {
    return a.ComparedTo(b) == Comparison.IsNewer;
  }

  public static bool operator <(VersionVector a, VersionVector b)
  {
    return a.ComparedTo(b) == Comparison.IsOlder;
  }

  public static bool operator ==(VersionVector a, VersionVector b)
  {
    return a.ComparedTo(b) == Comparison.AreSame;
  }

  public static bool operator !=(VersionVector a, VersionVector b)
  {
    return a.ComparedTo(b) != Comparison.AreSame;
  }

  public static bool operator <=(VersionVector a, VersionVector b)
  {
    var comparison = a.ComparedTo(b);
    return comparison == Comparison.AreSame || comparison == Comparison.Is
```

```
}

public static bool operator >=(VersionVector a, VersionVector b)
{
  var comparison = a.ComparedTo(b);
  return comparison == Comparison.AreSame || comparison == Comparison.Is
}

/// <summary>
/// Does cell wise maximum of two version vectors.
/// Used for synchronizing versions.
/// </summary>
/// <param name="another"></param>
/// <returns>Synchronized version</returns>
public VersionVector Ceil(VersionVector another)
{
  Dictionary<string, int> a = Copy(_value);
  Dictionary<string, int> b = Copy(another._value);

  // Transforming into hashset removes duplicates.
  var keys = a.Keys.Union(b.Keys).ToHashSet();

  var result = new Dictionary<string, int>();

  foreach (string key in keys)
  {
    int valueA = a.ContainsKey(key) ? a[key] : 0;
    int valueB = b.ContainsKey(key) ? b[key] : 0;
    result[key] = Math.Max(valueB, valueA);
  }

  return new VersionVector(result);
}

/// <summary>
/// Does cell wise minimum of two version vectors.
/// Used for calculating stable version.
/// </summary>
/// <param name="another"></param>
/// <returns>Stable version</returns>
```

```
  public VersionVector Floor(VersionVector another)
  {
    Dictionary<string, int> a = Copy(_value);
    Dictionary<string, int> b = Copy(another._value);

    // Transforming into hashset removes duplicates.
    var keys = a.Keys.Union(b.Keys).ToHashSet();

    var result = new Dictionary<string, int>();

    foreach (string key in keys)
    {
      int valueA = a.ContainsKey(key) ? a[key] : 0;
      int valueB = b.ContainsKey(key) ? b[key] : 0;
      result[key] = Math.Min(valueB, valueA);
    }

    return new VersionVector(result);
  }

  /// <summary>
  /// Returns a copy of the version vector
  /// </summary>
  /// <returns></returns>
  public VersionVector Copy()
  {
    return new VersionVector(_value);
  }

  private Dictionary<string, int> Copy(Dictionary<string, int> original)
  {
    return original.ToDictionary(k => k.Key, k => k.Value);
  }

  public override string ToString()
  {
    return $"[{String.Join(", ",␣_value.OrderBy(v␣=>␣v.Key).Select(kvp␣=>␣
  }
}
```

APPENDIX B: IMPLEMENTATION OF VERSION MATRIX

Listing B.1. *Implementation for version matrix.*

```
public class VersionMatrix
{
  private Dictionary<string, VersionVector> _versions;

  public VersionMatrix()
  {
    _versions = new Dictionary<string, VersionVector>();
  }

  public VersionMatrix(Dictionary<string, VersionVector> versions)
  {
    _versions = versions;
  }

  /// <summary>
  /// Returns a version vector that describes observed
  /// state for a node.
  /// </summary>
  /// <param name="node"></param>
  /// <returns></returns>
  public VersionVector this[string node]
  {
    get => _versions.ContainsKey(node)
      ? _versions[node]
      : new VersionVector();
  }

  /// <summary>
  /// Returns a version that describes state of
  /// a target node observed by an observing node
  /// </summary>
  /// <param name="key"></param>
  /// <returns></returns>
  public int this[string observingNode, string targetNode]
  {
    get
    {
      if (!_versions.ContainsKey(observingNode))
      {
        return 0;
      }

      return _versions[observingNode][targetNode];
    }
  }

  /// <summary>
  /// Creates a deep copy of the matrix.
  /// </summary>
  /// <returns>
  /// Deep copy of the version matrix
  /// </returns>
  public VersionMatrix Copy()
  {
    return new VersionMatrix(_versions.ToDictionary(
        x => x.Key,
        x => x.Value.Copy()));
  }

  /// <summary>
  /// Returns version vectors as dictionary.
  /// </summary>
  /// <returns>
  /// Mappings of node <-> version vector
  /// </returns>
  public Dictionary<string, VersionVector> GetVectors()
  {
    return _versions.ToDictionary(
        x => x.Key,
        x => x.Value.Copy());
```

```
}

/// <summary>
/// Replaces version vector for given node.
/// </summary>
/// <param name="node">Node to be updated</param>
/// <param name="version">Updated version vector</param>
/// <returns>Updated version matrix</returns>
public VersionMatrix Update(string node, VersionVector version)
{
  var copy = Copy();

  copy._versions[node] = version;

  return copy;
}

/// <summary>
/// Cell wise maximum between two version matrices,
/// and takes cell-wise maximum between versions of
/// the two given nodes putting them into same state.
/// </summary>
/// <param name="other">
/// Version matrix to be compared.
/// </param>
/// <returns>
/// Synchronized version matrix.
/// </returns>
public VersionMatrix Sync(VersionMatrix other,
  string node1,
  string node2)
{
  var ceiled = Ceil(other);

  var first = ceiled.GetVectors()[node1];
  var second = ceiled.GetVectors()[node2];

  var synced = first.Ceil(second);

  ceiled = ceiled.Update(node1, synced);
```

```
    ceiled = ceiled.Update(node2, synced);

    return ceiled;
  }

  /// <summary>
  /// Cell wise maximum between two version matrices.
  /// </summary>
  /// <param name="other">
  /// Version matrix to be compared.
  /// </param>
  /// <returns>
  /// Version matrix with maximum cell values.
  /// </returns>
  public VersionMatrix Ceil(VersionMatrix other)
  {
    var results = new Dictionary<string, VersionVector>();

    var first = Copy()._versions;
    var second = other.Copy()._versions;

    var nodes = new HashSet<string>(
      first.Keys.Union(second.Keys)
    );

    foreach (var node in nodes)
    {
      var firstVersion = first.ContainsKey(node)
        ? first[node]
        : new VersionVector(new Dictionary<string, int> {
            { node, 0 }
          });
      var secondVersion = second.ContainsKey(node)
        ? second[node]
        : new VersionVector(new Dictionary<string, int> {
            { node, 0 }
          });

      var ceilVersion = firstVersion.Ceil(secondVersion);
      results.Add(node, ceilVersion);
    }

    return new VersionMatrix(results);
  }

  /// <summary>
  /// Cell wise minimum between two version matrices.
  /// </summary>
  /// <param name="other">
  /// Version matrix to be compared.
  /// </param>
  /// <returns>
  /// Version matrix with minimum cell values.
  /// </returns>
  public VersionMatrix Floor(VersionMatrix other)
  {
    var results = new Dictionary<string, VersionVector>();

    var first = _versions;
    var second = other._versions;

    var nodes = new HashSet<string>(
      first.Keys.Union(second.Keys)
    );

    foreach (var node in nodes)
    {
      var firstVersion = first.ContainsKey(node)
        ? first[node]
        : new VersionVector(new Dictionary<string, int> {
            { node, 0 }
          });

      var secondVersion = second.ContainsKey(node)
        ? second[node]
        : new VersionVector(new Dictionary<string, int> {
            { node, 0 }
          });

      var ceilVersion = firstVersion.Floor(secondVersion);
      results.Add(node, ceilVersion);
    }

    return new VersionMatrix(results);
  }

  /// <summary>
  /// Returns stable timestamp among given nodes.
  /// If any of the nodes is not known, an empty
  /// version vector with all zero values will be
  /// returned.
  /// </summary>
  /// <param name="nodes">
  /// Nodes the stability is checked among.
  /// </param>
  /// <returns>Stable timestamp</returns>
  public VersionVector Stable(List<string> nodes)
  {
    VersionVector? stableStamp = null;

    foreach (var node in nodes)
    {
      var version = _versions.ContainsKey(node) ?
        _versions[node]
        : new VersionVector(new Dictionary<string, int> {
           { node, 0 }
          });

      if (ReferenceEquals(stableStamp, null))
      {
        stableStamp = version;
      }
      else
      {
        stableStamp = version.Floor(stableStamp);
      }
    }

    return stableStamp ?? new VersionVector();
  }
```

```
}
```

APPENDIX C: IMPLEMENTATION OF EVENT

Listing C.1. *Definition for a uncommitted event.*

```
public class UncommittedEvent
{
  /// <summary>
  /// Type of the event.
  /// </summary>
  public string Type { get; }

  /// <summary>
  /// Application data related to the event.
  /// </summary>
  public byte[] Payload { get; }

  public UncommittedEvent(string type, byte[] payload)
  {
    this.Type = type;
    this.Payload = payload;
  }
}
```

Listing C.2. *Definition for a committed event.*

```
  public class CommittedEvent
  {
    /// <summary>
    /// Unique identifier for the event.
    /// Identifier is generated when event is created
    /// for the first time, and does not change during
    /// replication.
    /// </summary>
    public Guid EventId { get; }

    /// <summary>
    /// The stream event was generated from.
    /// </summary>
    public string StreamId { get; }

    /// <summary>
    /// Node that generated the event.
    /// </summary>
    public string Issuer { get; }

    /// <summary>
    /// Type of the event.
    /// </summary>
    public string Type { get; }

    /// <summary>
    /// Application data related to the event.
    /// </summary>
    public byte[] Payload { get; }

    /// <summary>
    /// Logical version of stream when event was generated.
    /// Is used for ordering events within the stream, and
    /// for concurrency/conflict detection.
    /// </summary>
    public VersionVector StreamVersion { get; }

    /// <summary>
    /// Logical version of node when event was generated.
    /// Is used mainly for replication.
    /// </summary>
    public VersionVector OccurredAtNodeVersion { get; }

    /// <summary>
    /// Logical version of node when event was recorded.
    /// Is used to provide total order within node.
    /// Is unique for each node, and set when event is
    /// received during synchronization.
    /// Could be replaced with Version Matrix for completion.
    /// </summary>
    public VersionVector RecordedAtNodeVersion { get; }

    /// <summary>
    /// Physical time when event was generated.
    /// Used for bi-temporal queries.
    /// </summary>
    public DateTime OccurredAt { get; }

    /// <summary>
    /// Physical time when event was recorded.
    /// Used for bi-temporal queries.
    /// Replaced during replication.
    /// </summary>
    public DateTime RecordedAt { get; }

    public CommittedEvent(
      Guid eventId,
      string type,
      byte[] payload,
      string streamId,
      string issuer,
      VersionVector streamVersion,
      VersionVector occurredAtNodeVersion,
      DateTime occurredAt)
    {
      Payload = payload;
      EventId = eventId;
      Type = type;
      StreamId = streamId;
      Issuer = issuer;
      StreamVersion = streamVersion;
      OccurredAtNodeVersion = occurredAtNodeVersion;
      RecordedAtNodeVersion = occurredAtNodeVersion;
      OccurredAt = occurredAt;
      RecordedAt = occurredAt;
    }

    public CommittedEvent(
      Guid eventId,
      string type,
      byte[] payload,
      string streamId,
      string issuer,
      VersionVector streamVersion,
      VersionVector occurredAtNodeVersion,
      VersionVector recordedAtNodeVersion,
      DateTime occurredAt,
      DateTime recordedAt)
    {
      Payload = payload;
      EventId = eventId;
      Type = type;
      StreamId = streamId;
      Issuer = issuer;
      StreamVersion = streamVersion;
      OccurredAtNodeVersion = occurredAtNodeVersion;
      RecordedAtNodeVersion = recordedAtNodeVersion;
      OccurredAt = occurredAt;
      RecordedAt = recordedAt;
    }

    /// <summary>
    /// Copies the event for replication.
    /// Replaces recordedAt stamps.
    /// </summary>
    /// <param name="recordedAtNodeVersion"></param>
    /// <param name="recordedAt"></param>
    /// <returns>Replicated event</returns>
    public CommittedEvent AsRecordedAt(
      VersionVector recordedAtNodeVersion,
      DateTime recordedAt)
    {
      return new CommittedEvent(
        EventId,
        Type,
        Payload,
        StreamId,
        Issuer,
        StreamVersion,
        OccurredAtNodeVersion,
        recordedAtNodeVersion,
                OccurredAt ,
                recordedAt ) ;
        }
    }
```

APPENDIX D: SCENARIO TEST FOR BI-TEMPORALITY

Listing D.1. *Scenario test for bi-temporality.*

```
1  ########################################
2  # This scenario tests bi-temporality.
3  ########################################
4
5  # Setup phase
6  init-add-node salespersonA
7  init-add-node harvesterB
8
9  # Use controlled clock to manipulate time during scenario
10 init --use-controlled-clock
11
12 # Issue order at t0
13 time-set 2022-01-01|13:00:00
14 cmd-order-issue salespersonA order1 site1 salespersonA
15
16 # Sync at t2
17 time-set 2022-01-01|14:00:00
18 cmd-sync salespersonA harvesterB
19
20 validate-output Scenarios/Bi-Temporality/output-1.txt query-issued-orders
21 validate-output Scenarios/Bi-Temporality/output-2.txt query-issued-orders
22
23 # Do validations using t1 as viewpoint.
24 validate-output Scenarios/Bi-Temporality/output-3.txt query-issued-orders
25 validate-output Scenarios/Bi-Temporality/output-4.txt query-issued-orders
```

Listing D.2. *Expected output (line 20).*

```
GetCompletedOrdersQueryResult {
IssuedOrder { OrderId = order1, Issuer = salespersonA, occurredAt = 01/01/
}
```

Listing D.3. *Expected output (line 21).*

```
GetCompletedOrdersQueryResult {
IssuedOrder { OrderId = order1 , Issuer = salespersonA , occurredAt = 01/01/
}
```

Listing D.4. *Expected output (line 24).*

```
GetCompletedOrdersQueryResult {
IssuedOrder { OrderId = order1 , Issuer = salespersonA , occurredAt = 01/01/
}
```

Listing D.5. *Expected output (line 25).*

```
GetCompletedOrdersQueryResult {
}
```

APPENDIX E: SCENARIO TEST FOR STABILITY

Listing E.1. *Scenario test for stability.*

```
########################################
# This scenario tests information stability.
##########################################

# Setup phase
init-add-node salespersonA
init-add-node harvesterB
init-add-node harvesterC

# Use controlled clock to manipulate time during scenario
init --use-controlled-clock

# Order 'order1' issued by 'salespersonA' in
# node 'salespersonA' to harvest 'site1'.
time-set 2022-01-01|13:00:00
cmd-order-issue salespersonA order1 site1 salespersonA

# Assing (transfer ownership) or 'order' at
# node 'salespersonA' to node 'harvesterB'
time-set 2022-01-01|13:05:00
cmd-order-assign salespersonA order1 harvesterB

# Do sync to get the order placed to harvester
time-set 2022-01-01|13:10:00
cmd-sync salespersonA harvesterB

cmd-order-harvested harvesterB order1 salespersonA

# Sync nodes harvesterB and salespersonA
cmd-sync salespersonA harvesterB
```

```
31 cmd-sync harvesterC salespersonA
32
33 # Check completed orders (uses stable-mode), there
34 # should not be any, as A thinks that B does not
35 # yet know that C knows about completion of order.
36 # Note, \ is not part of the syntax, just a line break.
37 validate-output Scenarios/Stability/output-1.txt \
38 query-completed-orders salespersonA
39
40 # Synchronize so Completion of order1 is stabilized,
41 # as B gets information about what C knows, and can
42 # guarantee that every node knows about the completion.
43 cmd-sync salespersonA harvesterB
44
45 # A now knows that B knows about that C knows order1
46 # is completed.
47 # Note, \ is not part of the syntax, just a line break.
48 validate-output Scenarios/Stability/output-2.txt \
49 query-completed-orders salespersonA
```

Listing E.2. *Expected output (line 37) before completion of order is stabilized.*

```
CompletedOrder { OrderId = order1, Report = Total of 0 logs:
 }
}
```

Listing E.3. *Expected output (line 48) after completion of order is stabilized.*

```
GetCompletedOrdersQueryResult {
}
```

APPENDIX F: SCENARIO TEST FOR CONFLICT DETECTION

Listing F.1. *Scenario test for conflict-detection.*

```
#########################################
# This scenario tests for conflict detection.
#########################################

# Setup phase
init-add-node salespersonA
init-add-node harvesterB
init-add-node harvesterC

# Use controlled clock to manipulate time during scenario
init --use-controlled-clock

# Order 'order1' issued by 'salespersonA' in node 'salespersonA' to harves
time-set 2022-01-01|13:00:00
cmd-order-issue salespersonA order1 site1 salespersonA

# Assing (transfer ownership) or 'order' at node 'salespersonA' to node 'h
time-set 2022-01-01|13:05:00
cmd-order-assign salespersonA order1 harvesterB

cmd-identify-tree harvesterB site tree0 1000 1000 spruce
cmd-cut-down harvesterB tree0
cmd-cut-to-length harvesterB tree0 tree0-log1 1000 1100 6 2
cmd-cut-to-length harvesterB tree0 tree0-log2 1000 1100 6 2

cmd-sync salespersonA harvesterB
cmd-sync harvesterB harvesterC

# Both harvesters identifies tree1. This causes concurrent update to the s
```

```
30 cmd-identify-tree harvesterC site tree1 1000 1000 spruce
31 cmd-identify-tree harvesterB site tree1 1000 1000 spruce
32 cmd-identify-tree harvesterC site tree2 2000 2000 spruce
33
34 # Validate that conflict is detected.
35 validate-output Scenarios/Conflict-Detection/output.txt cmd-sync harvester
```

Listing F.2. *Expected output (line 35).*

```
1 Conflict at tree1.
```