TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

Consistency and Robustness in an Event-sourced System

Bachelor Thesis

von

Daniel Balke

March 31, 2022

Technische Universität Kaiserslautern,
Department of Computer Science,
67663 Kaiserslautern,
Germany

Examiner: Prof. Dr. Klaus Schneider
M. Sc. Martin Köhler

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die von mir vorgelegte Arbeit mit dem Thema "Consistency and Robustness in an Event-sourced System" selbstständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Kaiserslautern, den 31.3.2022

Daniel Balke

Daniel Balke

Abstract

Many systems accumulate data over time and need to persist that data to non-volatile storage to prevent data loss in case of system crashes, but also to have a stable domain model which can be used for analysis and troubleshooting. In many applications this data is managed by a relational database. The application gives up ownership of its data to the database in exchange for a unified data model and concurrent access. However, this severely limits the developer to the data modeling and querying capabilites of the specific database and makes using data structures and algorithms from the application's native programming language really difficult. The solution to this problem is to bring the domain logic and the data model closer together. However, by embedding the data model directly into the application, it becomes difficult to track state changes which need to be stored to non-volatile memory or replicated across machines for durability. We remedy this by using a technique called event sourcing, where a transaction is not simply a set of state changes to a data structure, but a logical sequence of events with each event representing an atomic change in the domain of the data model. This reduces the problem of tracking changes to appending to an append-only log. As a case study, we look at the backend of the mobile game Game of TUK whose application state is being structured into multiple independent modules. We describe techniques for validating invariants and handling failures in the context of event sourcing.

Zusammenfassung

Viele Systeme sammeln im Laufe der Zeit Daten an und müssen diese Daten in einem nichtflüchtigen Speicher aufbewahren, um Datenverluste bei Systemabstürzen zu vermeiden, aber auch, um ein stabiles Domänenmodell zu haben, das für Analysen und die Fehlersuche verwendet werden kann. In vielen Anwendungen werden diese Daten von einer relationalen Datenbank verwaltet. Die Anwendung gibt die Verwaltung ihrer Daten an die Datenbank ab und erhält dafür ein einheitliches Datenmodell und nebenläufigen Zugriff. Dies schränkt den Entwickler jedoch stark auf die Datenmodellierungs- und Abfragekapazitäten der spezifischen Datenbank ein und erschwert die Verwendung von Datenstrukturen und Algorithmen der Programmiersprache der Anwendung erheblich. Die Lösung für dieses Problem besteht darin, die Domänenlogik und das Datenmodell näher zusammenzubringen. Durch die direkte Einbettung des Datenmodells in die Anwendung wird es jedoch schwierig, Zustandsänderungen nachzuverfolgen, die aus Gründen der Langlebigkeit in einem nichtflüchtigen Speicher abgelegt oder über mehrere Rechner hinweg repliziert werden müssen. Wir schaffen hier Abhilfe, indem wir eine Technik namens Event Sourcing verwenden, bei der eine Transaktion nicht einfach eine Reihe von Zustandsänderungen einer Datenstruktur ist, sondern eine logische Folge von Ereignissen, wobei jedes Ereignis eine atomare Änderung in der Domäne des Datenmodells darstellt. Dadurch wird das Problem der Nachverfolgung von Änderungen auf das Anhängen an ein Protokoll reduziert, das nur Anhängeoperationen unterstützt. Als Fallstudie betrachten wir das Backend des mobilen Spiels Game of TUK, dessen Anwendungszustand in mehrere unabhängige Module strukturiert wird. Wir beschreiben Techniken zur Validierung von Invarianten und zur Behandlung von Fehlern im Kontext von Event Sourcing.

Contents

List of Figures vii

1 Introduction 1
1.1 Related Work . . . 1
1.2 Outline . . . 2
1.3 Contribution . . . 2

2 Background: Replicated Stateful Reactive Systems 3
2.1 Replication . . . 4
2.2 Consistency . . . 5
2.3 Concurrency . . . 6
2.3.1 Pipelined Architecture . . . 7
2.4 Classes of state . . . 7
2.5 Write efficiency . . . 9
2.6 Storage formats . . . 10
2.6.1 State-based . . . 10
2.6.2 Difference-based . . . 10
2.7 Event sourcing . . . 11
2.7.1 Projections . . . 11
2.7.2 Command Query Responsibility Separation . . . 11
2.7.3 Snapshotting . . . 12
2.7.4 Reproducibility over time . . . 12

3 Background: Game of TUK 13
3.1 History . . . 13
3.2 Domain concepts . . . 13
3.2.1 Quiz . . . 14
3.2.2 Paper Chase . . . 14
3.2.3 Fitness Courses . . . 14
3.3 Current architecture . . . 14
3.3.1 Protocol . . . 14
3.3.2 Frontend . . . 14
3.3.3 Backend . . . 15

4 Reproducibility, Robustness and Observability in Event Sourcing 17
4.1 Partitioning the domain into domain modules . . . 17
4.2 Separating input validation and state mutation . . . 17
4.3 Tracking generated events in a transaction . . . 22
4.3.1 Direct mutation . . . 22
4.3.2 Delayed mutation . . . 22

4.3.3 Iterative delayed mutation 22
4.3.4 Encapsulated mutation 23
4.4 Responsibility of a module 24
4.4.1 Managing entities and relationships 25
4.5 Dealing with invalid events 26
4.5.1 Observing warnings 29

5 Conclusion **31**
5.1 Future Work 31

Bibliography **33**

List of Figures

2.1 The transaction pipeline. 8

1 Introduction

Modern systems increasingly face issues of reproducibility and debuggability. These problems lead to further symptoms like slow development and hard-to-find inconsistencies in data, which can result in untraceable data loss or corruption. These robustness issues are often caused by CRUD (create/read/update/delete) based data storage which pushes developers into making systems whose state changes cannot be easily validated once they occurred. To cope with these issues, a technique called *event sourcing* [Fow05] can be used where conceptually only a sequence of state changes called *events* are held in persistent storage and derived state can be computed deterministically from these events only. This idea is not new, however. The simplest example is an accounting book where the individual transactions are the events and the current balance is the derived state.

When we reimplemented *Game of TUK* in 2020 (see section 3.1), we decided to use event sourcing as the primary storage for both server and client. With it came many benefits, including the ability to compute previously unanticipated state for existing data, like player velocities between recorded locations for statistics and progress data for achievements. It also helped with faster iteration on these features, as only the server has to be restarted to recompute derived state in memory. However, because this system allowed us to reproduce and fix bugs indefinitely, we were not pressured to validate invariants for state changes as soon as possible. This decreased overall confidence in the robustness of our logic, allows clients to send forged events with otherwise impossible data, and hides possible bugs in event generating code.

To fix these issues, we first need to understand the principles underlying event sourcing. Based on these principles, we identify the problems in Game of TUK's Rust backend and mitigate them by developing specific patterns for the Rust programming language, some of which may be easily translated into other languages. We illustrate these patterns using selected examples, but we refactored the rest of the game logic using the same style as well.

1.1 Related Work

Many of the covered concepts have been researched for decades, including replication and concurrency. Replicated stateful reactive systems are described by Schneider [Sch90] as replicated state machines with two different failure models, but we do not cover Byzantine failures. The ACID properties were first formulated by Haerder et. al. [HR83]. Difference-based or log-based storage formats are used by distributed consensus algorithms like Raft designed by Ongaro et. al. [OO14] and by The LMAX Architecture [Fow11b] designed by

the team at the LMAX financial exchange in London. Fowler introduced event sourcing in 2005 [Fow05]. Domain Driven Design by Evans [EE04] describes general domain modeling concepts that can be useful when using event sourcing.

1.2 Outline

In chapter 2 we explain the rationale behind event sourcing in a logical progression. We start with explaining the principles of replication and why it is necessary. We then transition to consistency models and the ACID properties of database systems. We then give an overview of different state representations, which leads directly into the key idea of event sourcing.

In chapter 3 we give a quick overview of our domain, the mobile game *Game of TUK*.

In chapter 4 we discuss how to partition the domain of Game of TUK into multiple domain modules to achieve encapsulation and separation of concerns. We go over specific problems when event sourcing is used as a replication mechanism, and patterns which can be used to overcome these limitations. We describe a mechanism for observing the validity of those events.

1.3 Contribution

We hope that the logical progression in chapter 2 helps readers to understand the principles behind database systems better and why they can often be replaced by a system with simpler semantics and more natural domain modeling. We do not claim in any way to have pioneered or invented these concepts.

We have not seen the implementation patterns discussed in chapter 4 applied to event sourcing before. The patterns may be helpful for writing similar systems to Game of TUK, especially using the Rust programming language.

2 Background: Replicated Stateful Reactive Systems

The primary purpose of computer systems is to transform information through computation and exchange it with the outside world. A computation does not just happen spontaneously, it needs to be initiated by a stimulus, which we call *external input*. When initiated, a computation either runs forever or terminates in a finite amount of time and produces a result. An example of a system with a single computation is a compiler: It receives source code as input and terminates with compiled code. This thesis is about *reactive* systems. These systems do not terminate on their own and are ready to receive external inputs at any time. Each external input initiates a computation which can produce multiple outputs during its execution. The system cannot do anything with these outputs except send them to a system, possibly itself, as an external input, therefore they are called *external outputs*. A reactive system that manages its own state over time is called *stateful* and is essentially a state machine. Each initiated computation represents a state transition which is also called a *transaction*. A computer system does not exist in a vacuum, but consists of concepts that have a meaning to the people using and developing it. Those concepts are called its *domain* and the structure of its state is called its *domain model*. A transaction is executed by a *process*. The operations in a transaction are executed *sequentially*, that is, one after another. The progress of a process is called its *execution context*. State can be distributed over multiple storage locations. If the same state is held in multiple different storage locations it is called *replicated*. The process by which state between those locations is synchronized is known as *replication*. If state is split between different storage locations it is called *sharded*.

Find below an example stateful reactive system: The state is a simple counter value that is stored in volatile main memory. The system can be instructed to increment or read the counter by sending external input, for example through an HTTP API. This is very simple to implement in a general-purpose programming language: We store the counter value in a variable that is accessed by a single thread of execution to execute transactions. For reference, an implementation of the domain in the Rust programming language might look like this:

```
1 struct State {
2     counter_value: u64,
3 }
4
5 fn initial_state() -> State {
6     State { counter_value: 0 }
```

```
7  }
8
9  fn increment(state: &mut State) {
10     state.counter_value += 1;
11 }
12
13 fn read(state: &State) -> u64 {
14     state.counter_value
15 }
```

2.1 Replication

Every storage medium has a non-zero risk of unrecoverable data loss. To decrease the risk of data loss, state has to be stored in multiple or less risky storage locations. In the above example the following scenarios all lead to the loss of state:

- The operating system or the application shuts down gracefully.
- The hardware experiences power loss.
- The system or application crashes due to a hardware or software bug/issue.

Losing state eventually is very likely if we only store it in main memory. Depending on the amount of redundancy and fail-safety required, state can be replicated to many different storage locations, for example to a hard drive in the same computer or to a data center located on the other side of the planet. State can be replicated either proactively or reactively. For example, game servers often save their game state both periodically (proactive) and when instructed to shut down (reactive). The domain needs to inform the decision when replication is performed and how much risk is acceptable. In many domains data loss is considered unacceptable, so reactive replication should not be used in these cases. For completeness, we discuss a few reactive solutions in the next paragraph.

A naive approach for handling power loss proactively would be using non-volatile memory modules as main memory. Aside from the fact that there are very few non-volatile memory modules available as a replacement for DRAM (for example Intel Optane), this does not solve the problem. There is additional internal state that is not stored in main memory, for example in CPU registers and caches, for which there is no guarantee that it is ever written back to main memory. The hardware would need to react to a power loss and save its internal state to main memory. As far as we know, there is not any production-level hardware that can resume execution after a power loss, but it is theoretically possible. In case of a planned shutdown, modern operating systems can suspend their state to disk (hibernation). A reactive solution for handling power loss is to use an uninterruptible power supply (UPS) to notify the operating system

or the application of a power loss and give it enough time to save its state to non-volatile memory.

To handle unpredictable failures like crashes or hardware defects we must turn to proactive replication. Applications need to be restarted from time to time, for example to perform updates. For this, we need a stable state representation that is not dependent on the internal representation in compiled code. We cannot rely on the operating system, therefore we need to perform replication in the application layer. Because we have to assume that the system can crash anytime without notice, naively we would have to replicate the domain state and execution context after every state change. To reference a famous quote: If a state change occurs, but no one observes it, did it actually happen (yet)? State can only be observed externally through external outputs. Because of this it is sufficient to ensure that state changes that were read or are being acknowledged by an external output are replicated before that output is relayed. These are the concepts of atomicity and durability as discussed in the next section. If we execute transactions sequentially, we can simply replicate the domain state after each transaction or batch multiple transactions together into one replication unit to amortize, for example, slow disk access. Because domain state can only be accessed inside a transaction, there does not exist any execution context outside a transaction that would need to be replicated alongside the domain state.

2.2 Consistency

Consistency can have two different meanings depending on the context:

Self-consistency or Data integrity State is self-consistent if its internal references are valid and its invariants and integrity constraints are satisfied.

Operational consistency This defines which values a read operation can produce, specifically under which circumstances the value of a write operation may be produced by a read operation. Alternatively, it can be seen as defining the ordering of concurrent operations.

From this point, we use the term consistency to refer to operational consistency. There are many different consistency models available to place constraints on the visibility of write operations to read operations.

A common quality metric for correctness and robustness of stateful reactive systems is *ACID* [HR83] (Atomicity, Consistency, Isolation, Durability).

- Atomicity: Guarantees atomic consistency (linearizability) of transactions to external observers. If a transaction is not atomic, its effects may be partially visible and observers may see self-inconsistent state.

- Consistency: Guarantees that if the state before the transaction is self-consistent, the state after the transaction is also self-consistent.

- Isolation: Guarantees sequential consistency (serializability) of transactions to other transactions. Prevents anomalies in case transactions are executed concurrently (see section 2.3).
- Durability: Guarantees monotonic read consistency of state changes to external observers.

An interesting fact is that these properties all describe operational consistency models except consistency which describes self-consistency. Atomicity makes transactions instantaneous to external observers, so that at every point they either see the effects of the transaction completely or not at all. Durability makes transactions monotonic to external observers, so that once they have seen the effects of the transaction, they will see them until an unrecoverable data loss (or forever). Logically, durability is a subset of atomicity, but the two are often separated to emphasize that observable state is replicated to at least one non-volatile storage location. Isolation guarantees that the effects of a transaction are visible to other transactions as if they were executed sequentially in some arbitrary order. If transactions are executed sequentially, isolation is trivially satisfied. [Bai14a]

2.3 Concurrency

It is notoriously difficult to reason about the correctness of concurrent operations in simple data structures, but is even harder if applied to complex business data with many relationships and complex invariants. Nevertheless, many database systems try to optimize throughput and latency by executing transactions concurrently and in parallel. This stems from the desire to provide a general-purpose data store that is independent of the domain model. This decision can pay off for long-running read-only transactions, for example backups, that can execute in parallel with write transactions using multi-version concurrency control (MVCC). Concurrent write transactions that access different state can also benefit. However, as soon as concurrently running transactions that write and read the same state, for example incrementing a counter, conflicts can occur as a result. The database can try to avoid conflicts using pessimistic synchronization primitives like locks or it can optimistically execute a transaction and when conflicts are detected abort its execution and rollback its changes. The operational consistency guaranteed for accesses is determined by the database isolation level. The default isolation level of PostgreSQL and MySQL, some of the most popular databases, is *read committed*, which allows transactions to commit with conflicts, giving rise to anomalies which can violate data integrity. When using the *serializable* isolation level, anomalies are not possible, but the application must be able to retry aborted transactions after a serialization failure. [Bai14b]

We think that in most systems with a complex domain, atomicity and isolation are desirable to be able to trust the correctness of its operations to ensure self-consistent state, which makes these systems easier for developers and domain experts to understand and evolve. The simplest way to achieve

that is to execute at least write transactions sequentially, which is called the *Single Writer Principle* [Tho11]. However, this limits concurrency and therefore parallelism, which reduces the performance of operations on unrelated data. As a solution, developers and domain experts can often partition the domain into multiple separate domains that communicate with each other through message passing (often referred to as *microservices*) and execute their own transactions sequentially. This exploits the natural concurrency that exists between different domains and can enable higher performance by eliminating the overhead of the database system by managing state in main memory.

For transactions that require atomicity with automatic failover to another machine, distributed consensus algorithms like Raft [OO14] can be used. They work by coordinating between different machines to agree on a single ordering of external inputs.

For transactions that only require eventual consistency, conflict-free replicated data types [Sha+11] (CRDTs) can be used instead. They are suited well to systems in which multiple concurrent writers are natural, for example collaborative editing. CRDTs guarantee that all writers eventually converge on the same state regardless of the execution order. [Sha+11]

2.3.1 Pipelined Architecture

This architecture can be implemented using pipelining, which is visualized in figure 2.1. The LMAX Architecture [Fow11b] uses this pattern to achieve high throughput and low latency by executing each stage of the pipeline in parallel using multiple threads.

Each external input is processed in a pipeline of sequential processes. When an external input is received, it is then processed by mutating state stored in main memory. Those writes are recorded, along with any outputs the system wants to pass to external systems as a response. The writes are then replicated to storage atomically. Only after all writes from that transaction are replicated, the external outputs are relayed to their destination. Because transactions are executed sequentially, rollback is never needed, so destructive operations inside a transaction can be safely allowed. If a transaction fails because of programmer error, the program crashes and the replicated state is unchanged.

2.4 Classes of state

We classify state as follows:

- stored or non-stored (also replicated or non-replicated)
- reproducible or non-reproducible
- derived or non-derived
- online and/or offline (offline is equivalent to stored)

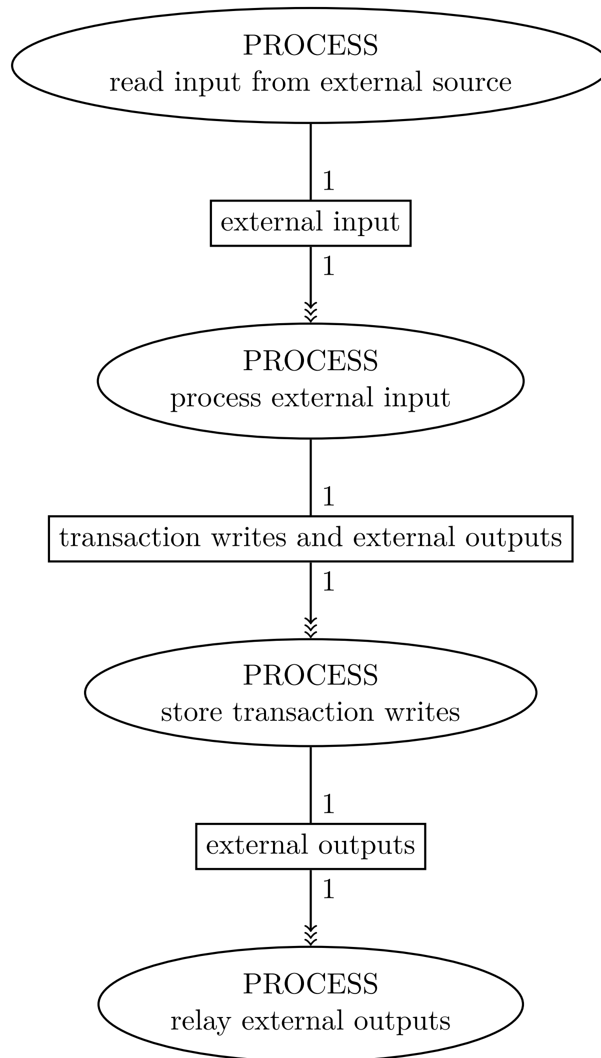

Figure 2.1: *The transaction pipeline.*

Stored state is all state that is replicated to disk or other non-volatile storage and is preserved across restarts of the application. Reproducible state is computed from stored state by a function referred to as the *derive* function. Therefore all stored state is also reproducible state, by means of the identity function. State computed this way which does not contain stored state is called derived state. Non-stored state is therefore composed of derived state and non-reproducible state. All non-reproducible state is contained in the state of the running system (called online state) and cannot be reconstructed once the system is shut down. Another way to phrase it is that online state is allowed to contain reproducible state, but offline state may not contain non-reproducible state.

Here are the above properties in more formal terms:

$$\begin{aligned} \text{reproducible} &= \text{derive(stored)} \\ \text{stored} &\subseteq \text{reproducible} \\ \text{derived} &= \text{reproducible} \setminus \text{stored} \\ \text{non-stored} &= \text{derived} \uplus \text{non-reproducible} \\ \text{non-reproducible} &\subseteq \text{online} \\ \emptyset &= \text{offline} \cap \text{non-reproducible} \end{aligned}$$

2.5 Write efficiency

The importance of write efficiency depends on how much state is stored and how frequently stored state needs to be updated. One of the simplest storage algorithms which guarantees atomicity and durability is *snapshot copy-on-write*. For every transaction snapshot to be stored, perform the following steps:

1. Serialize a snapshot of the state as a sequence of bytes.
2. Write that byte sequence into a temporary file.
3. Instruct and wait for the operating system to move the temporary file into the snapshot file atomically.
4. Only now make the state changes from the snapshot observable to external systems.

The algorithm guarantees:

- Atomicity because step 3 ensures that the snapshot file contains either the previous or the current snapshot.
- Durability because step 4 ensures that only stored state can be observed externally.

However, this method is only maximally write-efficient if two subsequent snapshots have almost no state in common. In practice, this almost never happens for larger state as it tends to accumulate over time, so a better approach is to store only the data that changed between snapshots. Expressed formally,

if m is the number of changed bytes between snapshot 1 and snapshot 2, and snapshot 1 is already stored, storing snapshot 2 must run in $O(m)$ time and must take $O(m)$ additional storage space, both on average to allow amortization. This means we cannot compare the full serializations of subsequent snapshots to compute the differences (which is basically how compression and deduplication work), as this does not satisfy the runtime requirement. Instead changes need to be tracked as they are applied, which is a reason why databases are widely used today.

2.6 Storage formats

There are two conceptual storage/replication formats for domain state, state-based and difference-based. In general, stored state cannot be overwritten in-place atomically, as the atomic operations provided by almost all underlying platforms operate on a fixed amount of data. This implies that updates need to be stored to an unused memory location before potentially overwriting previous state.

2.6.1 State-based

Snapshot copy-on-write uses a state-based storage format. The snapshot state is stored directly, but differences between snapshots must be computed if needed. It is primarily optimized for reading, but to fulfill the time and space requirement for writing as described above, we can use persistent data structures (sometimes called purely functional data structures). Persistent data structures enable efficient access and modification of multiple versions of state with a common history, and do so by structurally sharing unchanged data between snapshots. This approach is used by database systems like Datomic, copy-on-write filesystems, backup software and even Git. Many persistent data structures use a method called path copying, where a snapshot is represented by a tree, so that when values (possibly subtrees) in the tree are modified, the nodes on the paths from these values to the root are instead copied with the modified values and the new tree shares unchanged nodes with the previous tree.

2.6.2 Difference-based

In a difference-based storage format the differences between snapshots are stored directly in an append-only log, but the snapshot state must be computed by applying the differences. It is optimized for writing. Technically a log is a persistent data structure as well. This format is used for Git patches, in database systems, and especially in *event sourcing*.

2.7 Event sourcing

The core idea of event sourcing is to use different internal state representations for writing and reading domain state. We write domain events to an append-only log (difference-based) stored in non-volatile memory and read from the derived domain state (state-based) stored in volatile main memory.

The primary advantage of storing the domain state in main memory is that we can access it using a general-purpose programming language. We can use the data structures, algorithms and modeling capabilities of the programming language of our choice and do not need to worry about minimizing the network roundtrips to an external database like *Redis* or about using the asynchronous interface of integrated databases like *SQLite*. [Tho20]

When modeling the domain, it is important to think about which information to store inside an event and which information can be (deterministically) derived from it. As a general rule, minimize the amount of redundant information in an event and store only information in it that is necessary to represent the meaning of the event in the domain faithfully, but try to keep future modifications to the domain model in mind and rely only on stable determinism defined in section 2.7.4. The last point is a key difference between event sourcing and other difference-based storage formats.

2.7.1 Projections

The state-based volatile state is called projection, for which we define two operations:

- `init` returns the initial projection
- `project` applies an event to the projection

Both functions should be deterministic and free from side-effects. The stored state is a list of events. We then define the `derive` function referenced in section 2.4 of type `fn(Vec<Event>) -> State` in terms of `project` as `fn(events) { events.fold(project, init()) }`. The advantage over the direct definition is that when appending an event to the log, we do not need to start the computation from the initial state again because events are applied one at a time, which reduces runtime from quadratic to linear time.

2.7.2 Command Query Responsibility Separation

It is possible to have multiple projections which derive different state. A projection can be a write model or a read model, depending on the use of its state. A write model holds state needed to process transactions and validate inputs. A read model only holds state for answering queries, but not for processing transactions. The separation of state into write and write models is known as *CQRS* (Command Query Responsibility Separation). Using CQRS allows executing read-only queries in parallel with write transactions for horizontal scaling, similar to MVCC. It can be beneficial to compute read

models with eventual consistency to ensure the system is always available to answer queries. [Fow11a]

2.7.3 Snapshotting

By using an event log for replication, we get good write efficiency, but replaying the events on startup can take a long time. We can balance write and read efficiency by periodically storing a snapshot of the domain state to non-volatile memory using snapshot copy-on-write defined in section 2.5, for example once a day, and get fast replay on startup while amortize the snapshotting cost over many transactions. Database systems often take a similar approach by storing transactions in a write-ahead log, but then applying them non-atomically in-place to stored state directly.

2.7.4 Reproducibility over time

When an event is generated, it adheres to a schema given implicitly by the validation and generation code. However, if the `project` function is changed so that it is no longer equivalent to the previous definition, the event might become invalid under the new definition. Invalid events can also be generated if there is a bug in the transaction processing code or events were inserted or deleted manually. For robustness and increased confidence in the system, it is important to check invariants both when an event is applied to a projection and when processing a transaction. However, loosening the invariants in the generating code may be desirable to allow the system behavior to adapt without needing to introduce a new version of a previous event or migrate events altogether. Therefore a fine balance needs to be found between enforcing too many/strict and too few/loose invariants. This decision can only be made with adequate domain knowledge, as it is highly dependent on the changes to the domain model that the domain expert anticipates for the future.

We differentiate between stable and unstable determinism. The stability of a system is how predictable its behavior changes when changing its inputs, state, implementation or configuration. Examples of stable determinism are integer addition, (stable) sorting algorithms and linear functions. Examples of unstable determinism are hash functions, floating-point operations, pseudorandom number generators. The `project` function should exhibit stable determinism to ensure reproducibility from one execution to the next and from one program version to the next.

3 Background: Game of TUK

The theoretical framework in this thesis should be useful in general to developers of reactive applications with complex domain logic. However, the next chapter focuses on specific examples from the *Game of TUK* Rust backend, therefore this chapter provides a short summary of Game of TUK.

3.1 History

Game of TUK at its core is a gamified mobile app to increase physical activity among students at TU Kaiserslautern [Mül+20]. It provides several game modes to students and employees, some of which are location-based, and is supposed to be played over a course of roughly four weeks once a year. Since its inception is has seen several changes, both technical and non-technical. Started as a research project in 2018, its popularity has allowed it to remain in active development until today. In the first season in 2018, players discovered various security vulnerabilities and bugs which enabled cheating on a wide scale. This resulted in the rewrite of the backend using the Python framework Django in 2019, however the original codebases for the Android and iOS apps were simply adjusted to talk to the new backend [The19]. After the second season in 2019, it was decided that the apps needed to be rewritten as well, as bugs and separate codebases made it difficult to develop and extend further. The cross-platform framework *Flutter* from Google was chosen to develop the new frontend. New features and game modes were devised which needed big changes in the backend as well. After careful consideration we decided to rewrite the game logic code using event sourcing with in-memory state and the database as the replication storage, as Django's process-per-request model did not allow sharing state between requests without exchanging it through the database. New features like the leaderboard required computing a fair amount of derived state, which we neither wanted to compute for every request nor store in the database. The authentication code was not rewritten, as it is not as complex and the database still proved adequate to store shared state directly.

3.2 Domain concepts

Game of TUK consists of six independent game modes. *Quiz* and *Donkey Race* are round-based and have limits for how many rounds can be played each day, *Coin Collector* can vary its content each day, *Mr. Z* and *Paper Chase* can be completed anytime in the season and *Fitness Courses* can have a combination

of daily and weekly limits. *Day*s are non-overlapping dense time intervals which define when the season begins and when it ends.

3.2.1 Quiz

Every day each player is assigned a configurable number of random questions. A question cannot be assigned to the same player more than once. The backend randomly generates questions for a given player if that player does not already have assigned questions for the current day. The assigned questions are then stored in an event on the backend.

3.2.2 Paper Chase

This game mode consists of a series of riddles that players can solve to gain points. Those riddles are partitioned into a series of stages and players gain additional points for the completion of each stage. Players can play the game offline and each time they reach the location of the riddle, an event is generated and uploaded to the backend.

3.2.3 Fitness Courses

This game mode consists of a set of codes with an associated number of points that are awarded to the first player who scans the code. This action is called *redeeming* a code. Each code can only be redeemed once and a single event is stored for every successfully redeemed code.

3.3 Current architecture

3.3.1 Protocol

The frontend communicates with the backend via a request-response protocol. Ignoring authentication and only considering game logic, the client sends its event log and the server responds with the newest game state. The client polls the server periodically or when it appends a new event to its event log. A request-response protocol is sufficient for infrequently changing data, but we already have the real-time game mode *Mr. Z* for which an asynchronous, bidirectional protocol would be beneficial to reduce latency. We do not cover the development of such a protocol, however it is important to minimize coupling between the game logic and network protocol to make such a switch more seamless in the future.

3.3.2 Frontend

When the app is running, it has much non-reproducible state as it is not necessary to keep a record of all user interactions, like the user closing a dialog box. Only the interactions that are relevant to game progress are saved, like answering a question or collecting a coin. Stored state also contains authentication data, the last state the server sent and its own event log. State

is stored to disk using the inefficient snapshot copy-on-write algorithm defined in section 2.5. The topics in this thesis may be helpful for switching to a more efficient storage algorithm in the future.

3.3.3 Backend

The backend currently consists of two parts: Authentication and database schema management in the Python framework Django and game logic in Rust. Communication with the frontend happens with the Python backend which forwards authenticated requests to the Rust backend. From this point, *backend* refers only to the Rust backend, unless specified otherwise.

The backend is in essence a stateful reactive system using the LMAX Architecture and event sourcing. On startup, it connects to the database and loads all stored events into main memory, then applies them to the projection which holds all derived state. At this point, it is able to respond to external inputs, for example requests from the Python backend or elapsed time. As almost every request needs to change domain state and the read volume is low enough, we decided not to employ CQRS and instead have all reads execute sequentially with writes. We generally do not directly record external inputs as events, but try to transform them into natural domain events. An exception are events that the frontend generates while being offline as these already are domain events. We do not employ snapshotting because the lifetime of our domain state is limited to an interval of four weeks in which a season takes place and the number of generated events can usually be measured in single-digit millions, which is generally fast enough to replay in tens of seconds.

The backend has very little non-reproducible state because it was a requirement that server restarts or failovers are handled transparently to clients. Non-reproducible state is restricted to the storage and network layer, the domain logic exposes only derived state to the outside world. From the definition it follows that for a state change to become externally observable, stored state must be altered first.

The backend's domain logic is currently defined within a single file and is unorganized and ad-hoc. The domain state is defined using a projection that contains all relevant state for executing transactions and answering queries. The functionality of the game modes is scattered across the single file. Their state definitions are not complete and their events are only sparely validated. Their behavior is often defined in the `project` function, in some cases it is split into a Rust module. Their state is not encapsulated, so it is accessed from different unrelated functions directly. This leads to high coupling which makes the logic hard to understand and evolve in isolation. It is difficult to keep the event generation code deterministic because transactions access the projection through a mutable reference.

4 Reproducibility, Robustness and Observability in Event Sourcing

In this chapter, we describe issues with the current module structure and overall architecture of the domain logic in the Rust backend using three Game of TUK game modes (Quiz, Fitness Courses, Paper Chase) as examples and we intend to provide generally useful patterns and lines of thinking for tackling these problems. Code examples have been simplified to focus on the relevant aspects.

4.1 Partitioning the domain into domain modules

Game of TUK is characterized by diverse types of state and behavior with functionality that spans across different game modes, for example awarding points to players. The challenge is to extract each game mode into a separate unit to be able to understand and test it independently, while retaining the flexibility to handle cross-cutting concerns without resorting to hacks. The basic idea is to split the projection into multiple smaller projections. The projections are then organized in a shallow hierarchy of modules (for example Rust modules in separate files) in which parent modules interact with their child modules in an encapsulated way: A parent module's state contains the state of its child modules, and it calls the functions defined within its child modules to implement its own behavior. Direct access to a child module's state is restricted to the functions defined within it, and its parent module can only access its state indirectly by calling those functions. This is done to prevent coupling of a parent module to its child module's internal state representation.

4.2 Separating input validation and state mutation

As an example, we look at the module of the game mode *Fitness Courses*. This is the state definition we settled on:

```
1 struct State {
2     codes: HashMap<String, CodeState>,
3 }
4
5 enum CodeState {
6     Unused { points: u32 },
7     Used(PlayerId),
8 }
```

It contains the set of valid codes that can be either used or unused. Associated with each player is the list of points that were awarded to them by redeeming codes. The naive first attempt at implementing the module's functionality resulted in the `try_redeem` function, which is defined like this:

```
10 enum RedeemResult {
11     Success { points: u32 },
12     NotFound,
13     AlreadyUsedBy(PlayerId),
14 }
15
16 fn try_redeem(
17     state: &mut State, code: &str, player: PlayerId,
18 ) -> RedeemResult {
19     match state.codes.get_mut(code) {
20         None => RedeemResult::NotFound,
21         Some(&mut CodeState::Used(prev_player)) => {
22             RedeemResult::AlreadyUsedBy(prev_player)
23         }
24         Some(code_state @ &mut CodeState::Unused { points }) => {
25             *code_state = CodeState::Used(player);
26             RedeemResult::Success { points }
27         }
28     }
29 }
```

It receives a player and code as input and returns the result of the operation. There are three possible cases: Either the scanned code is invalid, meaning it does not exist, or the same or another player already redeemed the code, or the redemption was successful. The thing to note here is that the function combines input validation with subsequent state mutation. Because both take place in a single code path, lookups only need to be performed once, which is good for performance, and the Rust type system can be used for ensuring that all cases are covered, which is known as exhaustiveness checking.

The parent module's state is defined like this:

```
1 struct State {
2     scanner: scanner::State,
3     points_history: HashMap<PlayerId, Vec<(Timestamp, u32)>>,
4     // ...
5 }
```

For brevity, the *Fitness Courses* module is called `scanner`. The parent module's state contains the state of the game mode's module. For illustration of the problem, we also store the points history of each player in the attribute `points_history`.

We then define the parent module's `project` function:

```
7 enum Event {
8     CodeRedeemed {
9         time: Timestamp, player: PlayerId, code: String,
10     },
11     // ...
```

```
12 }
13
14 fn project(state: &mut State, event: &Event) {
15     match *event {
16         Event::CodeRedeemed { time, player, ref code } => {
17             match scanner::try_redeem(
18                 &mut state.scanner, code, player,
19             ) {
20                 scanner::RedeemResult::Success { points } => {
21                     state.points_history
22                         .entry(player).or_insert(Vec::new())
23                         .push((time, points));
24                 }
25                 _ => unreachable!(),
26             }
27         }
28         // ...
29     }
30 }
```

First, we define the domain events of Game of TUK. The `CodeRedeemed` event tracks successful code redemptions. The `project` function takes a mutable reference to the state and an event as input, and applies the event to the state by mutating the state. If `try_redeem` indicates success, the awarded points are added to the player's points history. If the redemption was not successful, we crash the program using `unreachable!()`, as such an event should have never been generated. Alternatives to this strategy are discussed in section 4.5.

When a player scans a code, the client sends a request to the server and expects to receive the result of the transaction as a response. The resulting transaction is defined by the function `player_scanned_code` in the parent module:

```
32 fn player_scanned_code(
33     state: &mut State,
34     player: PlayerId,
35     code: String,
36 ) -> (String, Vec<Event>) {
37     let time = time_now();
38     match scanner::try_redeem(
39         &mut state.scanner, &code, player,
40     ) {
41         scanner::RedeemResult::Success { points } => {
42             state.points_history
43                 .entry(player).or_insert(Vec::new())
44                 .push((time, points)); // duplicated
45             (
46                 format!("success:{}", points),
47                 vec![Event::CodeRedeemed { time, player, code }],
48             )
49         }
50         _ => (format!("failure"), vec![]),
51     }
```

```
52 }
```

It takes a mutable reference to the state, a valid player, and a code as input. It returns the response for the client and any generated events. Those events are then appended to the event log. To adhere to the idea of event sourcing, we would strongly prefer for every mutation to go through `project` to ensure that the state is reproducible deterministically, however that requires that an event exists first. Because the validation and mutation logic is intertwined, `player_scanned_code` is forced to call `try_redeem` directly, which is now called twice on *different* code paths, when processing the external input and when replaying the events on startup. This requires logic to be duplicated, in this case the accumulation of all player points. This makes it really easy to introduce non-reproducible state and subtle nondeterminism into the system.

We fixed these issues by splitting `try_redeem` into two separate functions:

```
16 fn can_redeem(
17     state: &State,
18     code: &str,
19     player: PlayerId,
20 ) -> RedeemResult {
21     match state.codes.get(code) {
22         None => RedeemResult::NotFound,
23         Some(&CodeState::Used(prev_player)) => {
24             RedeemResult::AlreadyUsedBy(prev_player)
25         }
26         Some(&CodeState::Unused { points }) => {
27             RedeemResult::Success { points }
28         }
29     }
30 }
31
32 fn on_code_redeemed(
33     state: &mut State,
34     code: &str,
35     player: PlayerId,
36 ) -> u32 {
37     match state.codes.get_mut(code) {
38         None => unreachable!(),
39         Some(&mut CodeState::Used(_)) => unreachable!(),
40         Some(code_state @ &mut CodeState::Unused { points }) => {
41             *code_state = CodeState::Used(player);
42             points
43         }
44     }
45 }
```

`can_redeem` only checks that the input is valid, in this case that the scanned code exists and was not redeemed yet. `on_code_redeemed` is responsible for actually performing the mutation, and implicitly assumes that the event is valid. If the event is invalid, the program still crashes, but this time `unreachable!()` is used in the child module instead of the parent module. This means that the child module is now responsible for deciding what to do when encountering

an invalid event, which is relevant for the alternative strategies described in section 4.5. Also, `on_code_redeemed` returns the awarded points, which are used by the adjusted `project` function in the parent module:

```
14 fn project(state: &mut State, event: &Event) {
15     match *event {
16         Event::CodeRedeemed { time, player, ref code } => {
17             let points = scanner::on_code_redeemed(
18                 &mut state.scanner, code, player, time,
19             );
20             state.points_history
21                 .entry(player).or_insert(Vec::new())
22                 .push((time, points));
23         }
24         // ...
25     }
26 }
```

The `project` function is much simpler now because handling the event does not require conditional branches anymore. The `player_scanned_code` function in the parent module is adjusted as follows:

```
32 fn player_scanned_code(
33     state: &mut State,
34     player: PlayerId,
35     code: String,
36 ) -> (String, Vec<Event>) {
37     let time = time_now();
38     match scanner::can_redeem(state, &code, player) {
39         scanner::RedeemResult::Success { points } => {
40             let event = Event::CodeRedeemed { time, player, code };
41             project(state, &event);
42             (format!("success:{}", points), vec![event])
43         }
44         _ => (format!("failure"), vec![]),
45     }
46 }
```

There is now only one code path for mutation: through the `project` function. However, `player_scanned_code` still takes a mutable reference to the state, and it has to make sure that generated events are correctly applied using `project` and also returned. A better solution using encapsulated mutation is presented in section 4.3.4.

The disadvantage is that we now have to duplicate code between `can_redeem` and `on_code_redeemed` that performs lookups and validation, but that logic can be extracted into helper functions if necessary. In some cases it can be useful to have the mutation function perform less checks than the validation function. However, if they diverge we need to be careful to ensure that the checks in the mutation function are a subset of the checks in the validation function to prevent invalid events from being generated, which may result in a crash or a mismatch of external output and domain state.

Ultimately, we think that the benefits of this refactoring outweigh the

drawbacks. It is generally better to have a deterministic code path than to prevent (small) duplicate work inside a transaction.

4.3 Tracking generated events in a transaction

Because the derived state is modeled with Rust data types and data structures, it must be held fully in main memory and therefore needs to be kept in sync with the database. Because the backend is assumed to have exclusive write access to the data it writes [Tho11], it is sufficient to only replicate changes from main memory to the database, not vice versa. Currently, the stored state consists only of events, however it is planned to mix state-based and difference-based storage in the future to integrate the functionality of the Python backend into the Rust backend and allow state to be overwritten, for example if data needs to be deleted for data protection reasons. There are multiple patterns for keeping track of state changes represented as events, characterized by their consistency (if queries can see recent state changes), robustness and ease of use. The patterns are presented in logical progression.

4.3.1 Direct mutation

Mutation of the projection cannot be delayed indefinitely and must be performed somewhere in the transaction. The last `player_scanned_code` function from the previous section is an example of this. However, it should generally be performed at the outermost layer to isolate possible desynchronization and nondeterminism issues to one place. We want to make sure that all generated events are applied to the projection and also appended to the event log later. Direct mutation is the base operation that the following patterns build on. Code that generates events is defined in a function (the callee) which is called by another function (the caller).

4.3.2 Delayed mutation

The callee cannot mutate the projection, but can read it and has to communicate a description of the state changes it wants to perform to the caller. The caller has the choice whether actually to perform the changes or not. The callee should perform a pure computation without any side-effects or non-local mutation. The advantage is that the callee cannot accidentally perform state changes and forget to record them because it does not directly perform them. The disadvantage is that it cannot observe the state modified by a generated event. However, this is not a problem for many use cases where state changes are not dependent on previous changes or only a few state changes are made.

An example of this is the `can_redeem` function from the previous section.

4.3.3 Iterative delayed mutation

This pattern can be used to perform one or more state changes where computation of a state change or the final value needs to read the previous state. The

caller calls the callee in a loop. Each iteration, the callee can either terminate the loop with a value or it can request the caller to perform a state change.

As an example we look at the *Quiz* game mode of Game of TUK. The event that describes the randomly generated questions for a given player is defined as a simple list for clarity:

```
type QuestionsAssignedEvent = Vec<QuestionId>;
```

We use this to retrieve and possibly assign questions in one function:

```
enum QueryResult<Change, Value> {
    Yield(Change),
    Finished(Value),
}
fn retrieve_or_assign_questions(state: &State, day_id: DayId)
    -> QueryResult<QuestionsAssignedEvent, Vec<QuestionId>>;
```

Because questions are generated only once for each day, we can also use non-iterative delayed mutation by splitting into two functions:

```
fn assign_questions(state: &State, day_id: DayId)
    -> Option<QuestionsAssignedEvent>;
fn retrieve_questions(state: &State, day_id: DayId)
    -> Vec<QuestionId>;
```

There is a tradeoff here: The iterative solution can accidentally lead to an infinite loop. The non-iterative solution avoids this by having `assign_questions` return a more specific type. However, `retrieve_questions` has to assume that the questions have already been generated, so it crashes on lookup when they have not. In this case crashing is better than having an infinite loop, as it is easier to diagnose.

If additional state needs to be passed between iterations, these patterns should not be used, because the execution context is thrown away after each iteration. The additional state may then need to be recomputed each iteration, which performs unnecessary work and reduces performance.

4.3.4 Encapsulated mutation

This pattern provides the flexibility of iterative delayed mutation with the ability to define the number of ”iterations” at compile-time and execute different logic each ”iteration”. Additional state can be passed between ”iterations” using normal variables. In essence, the callee does not return the intended changes, but passes them to the caller through a callback while keeping its execution context. When using delayed mutation, additional state needs to be passed in and out of the callee or integrated into the callee’s state, which breaks separation of concerns and makes the code harder to understand and maintain.

The caller provides a data structure with the following interface to the callee:

```
trait EventBuffer {
    type State;
    type Event;
    fn state(&self) -> &Self::State;
    fn apply(&mut self, event: Self::Event);
```

```
6 }
```

It is initialized at the outermost layer and passed to domain logic code which applies changes using `apply`. However consider the following sequence of actions:

1. The projection is read using `event_buffer.state()` and stored in a local variable
2. An event is generated and applied using `event_buffer.apply(event)`
3. External output is computed, but using the projection stored in the local variable instead of calling `event_buffer.state()` again

There is a problem here: The previous value may be invalid, especially if properties of the projection are further extracted into local variables. If the value is garbage-collected or reference-counted, this usually results in reading stale data, which may compromise the consistency of the system as a whole. If the value contains raw pointers, reading those will result in undefined behaviour if they have been invalidated by `event_buffer.apply(event)`, which may result in a crash at best or data corruption at worst. In contrast, Rust's borrowing system prevents this bug by invalidating all references to the state obtained by `event_buffer.state()` and disallows using them at compile-time after `event_buffer.apply(event)` is called.

We can use the *Fitness Courses* game mode from the end of section 4.2 as an example:

```
32 fn player_scanned_code(
33     event_buffer: &mut impl EventBuffer<State=State, Event=Event>,
34     player: PlayerId,
35     code: String,
36 ) -> String {
37     let time = time_now();
38     let state = event_buffer.state();
39     match scanner::can_redeem(state, &code, player) {
40         scanner::RedeemResult::Success { points } => {
41             let event = Event::CodeRedeemed { time, player, code };
42             event_buffer.apply(event);
43             format!("success:{}", points)
44         }
45         _ => format!("failure"),
46     }
47 }
```

The state is now supplied as a read-only reference, preventing direct mutation at compile-time. The return type is simpler, as generated events are now tracked automatically by the event buffer. The `on_code_redeemed` function is transparently called by the event buffer through the `project` function.

4.4 Responsibility of a module

We define the state of the *Paper Chase* module as follows:

```
struct State {
    stages: Vec<Stage>,
    players: HashMap<PlayerId, PlayerState>,
}

struct Stage {
    riddles: usize,
}

struct PlayerState {
    riddles_solved: usize,
}
```

It seems that there is much state missing here. There is no text or image describing the location of the riddles. There also are no riddle coordinates which are needed to check if a player reached a riddle location. If all that information is missing, where is it defined instead and what is the exact purpose of a module?

In our case, a module is not supposed to implement all functionality of the game mode, for example networking and configuration management. Its only responsibility is to track state for validation and communicate significant information back to its parent module. By separating the domain logic from other system functionality, we are able to tailor the module interface specifically to the game mode, resulting in better understandability and clearer communication of intent to other developers and increased testability with unit tests, due to its more narrow scope. The technical term for this narrow and specific scope is *cohesion*, the complement of *coupling*. This comes with a downside: Depending on the degree of specificity of the module interface, there may be a lot of glue code needed to integrate it into the full system. However, this is acceptable because that code is needed anyway and the only question is where it should be located. The purpose of a parent module includes connecting its subsystems and possibly interacting with external systems. All side-effects should be performed near the top of the hierarchy to retain the mentioned benefits.

The module still needs to provide operations for the parent to use, which we cover in section 4.5.

4.4.1 Managing entities and relationships

Relational databases represent state using a set of tables where each table contains a set or list of tuples. Each tuple can have a unique identifier, often a serial number, which is used to attach identity to it. Identity means that after the tuple's state is changed, it can still be related back to the previous version using its identifier. A tuple with identity is an *entity*. An entity can be referenced by other tuples to represent relationships between data. For example, a Game of TUK *player* is an entity.

Instead of automatically managing entities, we need to assign unique identifiers to entities explicity. When an entity is created, the identifier must be

contained in or derivable from the event that describes the entity's creation. The identifier can also be referenced in subsequent events. Each entity has an owner module that is responsible for validating it. In this case, players are managed by the parent module and child modules like *Paper Chase* can assume that entities received from the parent module are valid. A player is also guaranteed never to become invalid, so *Paper Chase* can safely reference them within its state using their identifier.

4.5 Dealing with invalid events

What if for some reason an event is invalid or contains implausible information? In section 4.2 we assumed that all `CodeRedeemed` events have been correctly generated after external inputs were validated, or else it is sufficient to assume the situation is unrecoverable and crash. However, what if we do not want to check an event for plausibility before it is appended to the event log, but instead treat it similar to an external input? For example, we use this for synchronizing events from the client to the server that the client generated while possibly being offline. The idea is that the event already happened on the client and therefore needs to be simply stored on the server. Because clients can produce invalid or even malicious events, the server must not trust them when interpreting their information. Additionally, although crashing is the easiest option if events are known to be plausible, it is not robust to the other factors described in section 2.7.4. To discuss, we look at the still incomplete *Paper Chase* module:

```
14 fn on_riddle_solved(
15     state: &mut State,
16     player: PlayerId,
17     expected_last_riddle_in_stage: bool,
18 ) {
19     let player_state = &mut state.players[player];
20     let stages_solved =
21       calculate_stages_solved(&state.stages, player_state);
22     if stages_solved >= state.stages.len() {
23         // All riddles have been solved already
24         return;
25     }
26     player_state.riddles_solved += 1;
27     // A riddle was solved
28     let last_riddle_in_stage =
29       stages_solved !=
30         calculate_stages_solved(&state.stages, player_state);
31     if last_riddle_in_stage {
32         // A stage was completed
33     }
34     if last_riddle_in_stage != expected_last_riddle_in_stage {
35         // The client computed different progress
36     }
37 }
```

The red comment lines indicate that something is missing. We discuss this in the following paragraphs.

The function describes what happens to the state when a riddle is solved by a player. It is called for each `RiddleSolved` event the server received from the client. It takes as input a mutable reference to the current internal state and information associated with the event. The information does not need to be in the same form as the event itself, as the parent module can add additional validation or transformation. The `on_riddle_solved` function only provides a contract that the parent module needs to comply with. This is discussed in more detail in section 4.4.1.

First the state of the current player is looked up and it is verified that the event could have happened under the preconditions of the domain model, in this case whether another riddle exists that could have been solved. If this is not the case, we have multiple options for how to deal with that violation:

Crash We can abort the execution by killing the operating system process and freeing all its resources. This prevents invalid operations from being performed, but makes the system unavailable even if it may be possible to (partially) recover from the violation. In this case, we cannot use this method anyway, because the event is generated externally and we must not trust external input, as mentioned above.

Ignore We can just pretend the violation did not happen. This ensures that the system stays available. However, as violations cannot be monitored directly by administrators or the system itself, diagnosis is more difficult and faults may go unnoticed.

Escalate The situation that a riddle was supposedly solved although there are none left to solve is not fully recoverable and can be caused by a bug in the client in which an invalid event was generated, a bug in the server in which the situation is wrongly diagnosed, an attempt at cheating by a player, or a synchronization/concurrency issue in which the client generated a genuine event using outdated information from the server with a larger amount of riddles. When we do not know how to handle a situation, we can choose not to decide right now and delay the decision. In this context this means escalating to a higher level, as the decision depends on multiple factors that are out of scope for this module. As this situation cannot be resolved automatically, we may want to escalate all the way to a human observer. The easiest form is printing text to the console and in some cases may be sufficient. However, we would like to have a more user-friendly monitoring tool for observing the system's status. Because this situation is not critical to the system's operation, the system can keep running.

To escalate instead of using the ignore strategy, we define a new data type `Out` that defines the notifications that are relevant to the parent module:

```
1 enum Out {
2     NoNextRiddle,
```

```
3     RiddleSolved,
4     StageCompleted,
5     UnexpectedLastRiddleInStage,
6 }
```

The cases `NoNextRiddle` and `UnexpectedLastRiddleInStage` represent a notification to the parent module that the event is invalid. The notification `UnexpectedLastRiddleInStage` is communicated to the parent when client and server disagree on the fact that the current riddle is the last one in its stage. This is important because they both compute the number of points for the player. The client does this to display the points to the player even if they are offline. There currently is no mechanism in place to re-synchronize these if they get out of sync, so manual action needs to be taken. On the other hand, `RiddleSolved` and `StageCompleted` are different because they do not signal an unusual condition. Rather, the information is important to the parent module to award the correct points to the player and correct their rank in the leaderboard. Before the refactoring the parent module just counted the number of `RiddleSolved` events, but this had the problem that the maximum number of riddles was not enforced.

Now take a look at the modified `on_riddle_solved` function:

```
14 fn on_riddle_solved(
15     state: &mut State,
16     player: PlayerId,
17     expected_last_riddle_in_stage: bool,
18     mut emit: impl FnOnce(Out),
19 ) {
20     let player_state = &mut state.players[player];
21     let stages_solved =
22       calculate_stages_solved(&state.stages, player_state);
23     if stages_solved >= state.stages.len() {
24         emit(Out::NoNextRiddle);
25         return;
26     }
27     player_state.riddles_solved += 1;
28     emit(Out::RiddleSolved);
29     let last_riddle_in_stage =
30       stages_solved !=
31         calculate_stages_solved(&state.stages, player_state);
32     if last_riddle_in_stage {
33         emit(Out::StageCompleted);
34     }
35     if last_riddle_in_stage != expected_last_riddle_in_stage {
36         emit(Out::UnexpectedLastRiddleInStage);
37     }
38 }
```

The function takes an additional callback function as a parameter, which is called to notify the parent module. The problem with this is that if the parent module wants to pattern match on the notification, it must handle all cases, which is probably the wrong thing to do for warnings. The solution is to separate the warnings from informational notifications. There are multiple

ways to implement this: The parent module could provide multiple callbacks, however those are limited. Because of Rust's borrowing system, they cannot access the same mutable reference. Another solution would be to annotate the different cases so that the parent module knows which one to treat as useful information. However, we decided to split `Out` into two separate types and communicate warnings as a return value. This communicates useful intent to the parent module.

```
1 enum Warning {
2     NoNextRiddle,
3     UnexpectedLastRiddleInStage,
4 }
5
6 enum Info {
7     RiddleSolved,
8     StageCompleted,
9 }
```

We change the parameter type of the `emit` callback function from `Out` to `Info`. The return type is also set to the `Result` type, so that we must communicate any warning by returning from the function:

```
18     mut emit: impl FnOnce(Info),
19 ) -> Result<(), Warning> {
```

If we want to emit a warning, we are forced to return it from the function using the `Err` variant of the `Result` type:

```
23     if stages_solved >= state.stages.len() {
24         return Err(Warning::NoNextRiddle);
25     }
```

We then change the type from `Out` to `Info`:

```
28     emit(Info::RiddleSolved);
```

```
33     if last_riddle_in_stage {
34         emit(Info::StageCompleted);
35     }
```

And at the end, we return the warning using the `Err` variant of the `Result` type:

```
36     if last_riddle_in_stage != expected_last_riddle_in_stage {
37         return Err(Warning::UnexpectedLastRiddleInStage);
38     }
```

4.5.1 Observing warnings

When the parent module receives a warning from its child module, it needs to forward it to an administrator who can diagnose the problem. We simply convert the warning to a string using the `Debug` trait, which can be automatically derived by the Rust compiler, and then print it to the console, along with the current event.

By observing the printed warnings we noticed a concurrency bug in the client where an event was generated twice. We could have observed the relevant events by directly looking at the event log, but the warnings made it much easier to filter out unnecessary information and provided a useful hint for diagnosing the problem at the same time.

5 Conclusion

We looked at replication in detail, learning that it appears in a variety of different systems, and that primarily consistency and concurrent writes define its characteristics, whereas there is no fundamental difference between different storage locations. We discussed the ACID properties and evaluated concurrency in transactional database systems and have seen an alternative way to design replicated stateful reactive systems known as event sourcing. This technique conceptually represents state changes as a series of events and differentiates between read-only state stored in volatile memory, called the projection, and write-only state, the replicated event log. When the system is restarted, the event log is replayed to recompute the projection. This process should exhibit stable determinism to ensure the system behaves predictably.

Game of TUK's backend uses event sourcing and implements its domain model and domain logic using the Rust programming language. The implementation had a number of problems, from occasionally crashing, to not detecting bugs in the event generation code of the client, because events were not validated by the server. We fixed these issues by modeling the domain properly and separating different concerns into different modules. We described why transactions should be defined separately from the projection and devised a technique called encapsulated mutation to ensure this at compile-time. To verify that the system behaves correctly, we looked at the warnings produced by invalid events. Overall, the domain logic is now more robust because its code is more understandable and focused, and because it is observable in case things do go wrong.

5.1 Future Work

In section 2.3, we touched on distributed systems with more complex communication and consistency requirements, for example distributed consensus or eventual consistency. Finding patterns for reasoning about such behavior in a general-purpose programming language may be an interesting research topic.

In section 4.3.4 we only provided a way to track generated events. If we allow arbitrary state changes to occur, the problem becomes a lot more complex, as there is no simple way to ensure deterministic execution. There a domain-specific language (DSL) may be useful for tracking state changes more transparently, similar to SQL for relational database systems.

For Game of TUK, it would be beneficial to show the warnings produced by invalid events in its webinterface and be able to observe the domain state at previous events. This makes observability more prevalent and allows for interesting monitoring functionality.

Bibliography

[Bai14a] Peter Bailis. *Linearizability versus Serializability*. 2014. URL: http://www.bailis.org/blog/linearizability-versus-serializability/. (last accessed: 28.03.2022).

[Bai14b] Peter Bailis. *Understanding Weak Isolation Is a Serious Problem*. 2014. URL: http://www.bailis.org/blog/understanding-weak-isolation-is-a-serious-problem/. (last accessed: 28.03.2022).

[EE04] Eric Evans and Eric J Evans. *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional, 2004.

[Fow05] Martin Fowler. *Event Sourcing*. 2005. URL: https://martinfowler.com/eaaDev/EventSourcing.html. (last accessed: 28.03.2022).

[Fow11a] Martin Fowler. *CQRS*. 2011. URL: https://www.martinfowler.com/bliki/CQRS.html. (last accessed: 28.03.2022).

[Fow11b] Martin Fowler. *The LMAX Architecture*. 2011. URL: https://martinfowler.com/articles/lmax.html. (last accessed: 28.03.2022).

[HR83] Theo Haerder and Andreas Reuter. “Principles of Transaction-Oriented Database Recovery”. In: *ACM Comput. Surv.* 15.4 (Dec. 1983), pp. 287–317. ISSN: 0360-0300. DOI: 10.1145/289.291. URL: https://doi.org/10.1145/289.291.

[Mül+20] Julia Müller, Max Sprenger, Tobias Franke, Paul Lukowicz, Claudia Reidick, and Marc Herrlich. “Game of TUK: Deploying a Large-Scale Activity-Boosting Gamification Project in a University Context”. In: *Proceedings of the Conference on Mensch Und Computer*. MuC ’20. Magdeburg, Germany: Association for Computing Machinery, 2020, pp. 169–172. ISBN: 9781450375405. DOI: 10.1145/3404983.3410008. URL: https://doi.org/10.1145/3404983.3410008.

[OO14] Diego Ongaro and John Ousterhout. “In Search of an Understandable Consensus Algorithm”. In: *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. Philadelphia, PA: USENIX Association, June 2014, pp. 305–319. ISBN: 978-1-931971-10-2. URL: https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro.

[Sch90] Fred B. Schneider. “Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial”. In: *ACM Comput. Surv.* 22.4 (Dec. 1990), pp. 299–319. ISSN: 0360-0300. DOI: 10.1145/98163.98167. URL: https://doi.org/10.1145/98163.98167.

[Sha+11] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. "Conflict-free Replicated Data Types". In: *13th International Conference on Stabilization, Safety, and Security of Distributed Systems*. SSS 2011. Springer LNCS volume 6976, Oct. 2011, pp. 386–400. DOI: 10.1007/978-3-642-24550-3_29.

[The19] Daniel Theis. "Restrukturierung der API und Serverkomponente eines mobilen Spiels zur Optimierung der Sicherheit". Bachelor. MA thesis. Department of Computer Science, University of Kaiserslautern, Germany, Oct. 2019.

[Tho11] Martin Thompson. *Single Writer Principle*. 2011. URL: https://mechanical-sympathy.blogspot.com/2011/09/single-writer-principle.html. (last accessed: 28.03.2022).

[Tho20] Martin Thompson. *Event Log Architectures: when quality matters*. 2020. URL: https://www.youtube.com/watch?v=RlwO6CJbJjQ. (last accessed: 28.03.2022).