FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Replication in Loosely Coupled Systems

Simulation and Evaluation of Replication Strategies for their use in Loosely Coupled IT Systems

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Business Informatics

eingereicht von

Alexander Schenk, BSc
Matrikelnummer 0825353

an der Fakultät für Informatik
der Technischen Universität Wien
Betreuung: Ao.Univ.Prof. Mag. Dr. Horst Eidenberger

Wien, 8. Mai 2018

Alexander Schenk	Horst Eidenberger

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Replication in Loosely Coupled Systems

Simulation and Evaluation of Replication Strategies for their use in Loosely Coupled IT Systems

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Business Informatics

by

Alexander Schenk, BSc
Registration Number 0825353

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof. Mag. Dr. Horst Eidenberger

Vienna, 8th May, 2018

Alexander Schenk | Horst Eidenberger

Erklärung zur Verfassung der Arbeit

Alexander Schenk, BSc
Reindorfgasse 35, 1150 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 8. Mai 2018

Alexander Schenk

Danksagung

Ihr Text hier.

Kurzfassung

Moderne IT-Systeme stellen oft lose gekoppelte verteilte Systeme dar, in denen einzelne Applikationen unabhängig voneinander erstellt, weiterentwickelt und ausgeliefert werden können. Nachteilig an dieser Architektur ist der hohe Kommunikationsaufwand zwischen den Applikationen. Dieser kann durch den Einsatz von Replikation verringert werden. Der aktuelle Stand der Technik in diesem Bereich konzentriert sich auf die Daten-Ebene, bei der enge Kopplung der Applikationen entsteht und der daher nicht für lose gekoppelte Systeme angewandt werden kann.

In dieser Arbeit wurden Replikationsstrategien für lose gekoppelte IT-Systeme identifiziert und auf Basis von qualitativen und quantitativen Kennzahlen bewertet. Die Auswahl wurde mit einer bottom-up Ansatz durchgeführt, in der die allgemeinen Eigenschaften von Replikationsmechanismen als Basis dienten. Die konkreten Implementierungen der Replikationsstrategien bauten primär auf Event-Driven Architectures mit Messaging und Atom-basierte Web Feeds auf.

Basis der Bewertung bildete eine analytische Betrachtung der Replikationsstrategien und die Ergebnisse einer Simulation. Die qualitativen Kennzahlen wurden rein analytisch auf Basis der ISO 25010 Software-Qualitätseigenschaften ermittelt. Als quantitative Kennzahlen wurden die client-seitige Latenz von Leseoperationen, die server-seitige Latenz von Schreiboperationen, die Zeit bis alle Replikate konsistent sind und der Bandbreitenverbrauch verwendet. Diese wurden sowohl analytisch ermittelt, als auch im Zuge der Simulation in einer eigens dafür erstellten Testumgebung gemessen. In der Testumgebung wurden konkrete Implementierungen der Replikationsstrategien in Java EE und Ruby on Rails mit mehreren synthetisch erstellten Workloads getestet.

Im Zuge der Arbeit wurde festgestellt, dass keine einzelne Replikationsstrategie optimal für alle Anwendungsfälle geeignet ist. Replikationsstrategien die Atom-basierte Web Feeds verwenden werden grundsätzlich nicht empfohlen, da die verfügbaren Bibliotheken in diesem Bereich nicht ausgereift sind. Replikation über push-basierte Benachrichtigungen des gesamten Zustands eines Datensatzes, über eine publish-subscribe message queue, stellt sich als die beste Lösung für die meisten Anwendungsfälle dar.

Abstract

Modern IT systems are often implemented as loosely coupled systems, consisting of independently developed and deployed components. These systems require a high degree of communication within and can be optimized by the replication of data. Actual research on replication mainly focuses on the data layer, which leads to tight coupling and is therefore not suitable for loosely coupled systems.

This thesis identified replication strategies which may be used for loosely coupled systems by a bottom-up approach (based on general characteristics of replication mechanisms) and evaluates their use in respect to qualitative and quantitative indicators. The considered replication strategies mainly build on Event-Driven Architectures with Messaging and Atom-based Web Feeds.

Data was gathered by an analytical evaluation and a simulation. Qualitative indicators were solely examined analytically, in accordance with the quality indicators of ISO 25010. Quantitative indicators are the latency of a client-side read operation, the latency of a server-side update operation, the time until all replicas are in a consistent state and the consumed bandwidth. These indicators were examined analytically and measured during a simulation in a custom-built simulation environment with implementation of the replication strategies in Java EE and Ruby on Rails. To emulate the behavior of a real IT system, the simulation uses several synthetic workloads which are based on real-world observations and assumptions made in related research.

The evaluation concludes that there is no single best replication strategy for all cases of applications. Replication strategies which use Atom-based Web Feeds are generally not recommended because of the insufficient availability of tools in this area. The evaluation indicates that replication by push-based notifications of the full state of a data item with a publish-subscribe message queue can be used for most cases of applications.

Contents

CHAPTER 1

Introduction

1.1 Problem Statement

Modern IT systems are mostly implemented as distributed systems of isolated applications that transfer data and operations over defined interfaces like RESTful webservices. The distribution of applications also leads to a distribution of data and the challenge to exchange it between the *server*, who has the data, and *clients*, who need the data for operations. The transmission of data for each operation leads to bad runtime performance and does not scale when the system evolves. A solution for this problem is replication (mainly in form of *caches*), where data is stored for read-only access on the client-side and transmissions are minimized.

A challenge of replication is to keep replicates 100% up-to-date at all times. This constraint is often limited with *optimistic replication* or *lazy replication* that guarantees only *eventual consistency* [Tan07] for the benefit of better runtime performance. To ensure business gets actual data to perform their work and generate value, it is essential to actualize or invalidate replicas in short time and ensure a high consistency of the overall system.

Replica actualization and invalidation in eventually consistent systems increases the complexity of introducing caching in IT systems. Companies planning to use replication therefore need well-founded decision criteria to find an appropriate solution for the respective requirements of their IT system.

1.2 Motivation

This work addresses business IT systems which are operating inside companies and organizations to handle their business cases and operations. Business IT systems consist

of one or more individual applications that are typically implemented with standard technologies like *HTML*, *RESTful webservices* and *Messaging*.

Over the last decades, there is a trend from *monolithic* architectures [Ric17] to smaller applications that form a distributed system. Modern business IT systems follow this trend and are therefore often implemented as distributed systems. These systems allow individual applications to be developed and deployed independently, expecting to suite business requirements more quickly, effectively and efficiently [New15] [Wol15]. Most architectural trends in the last years like *Service-Oriented Architecture* (*SOA*) [SOA], *Event-Driven Architecture* [EDA], *Microservices* [Fow14] and *Self Contained Systems* (*SCS*) [SCS] share these characteristics.

These architectural styles propagate *loose coupling* [Hoh04] [Val09] to achieve the independence of individual applications in the system. Replication also enhances loose coupling: applications can operate without the need of communication and even if parts of the systems are not available. Even though loose coupling has been an architectural and design principle in monolithic architectures, there is greater need for it in a distributed system. In a distributed system, tight coupling leads not only to problems in operations, but also during development and deployment [CDA] [Wol15].

Current integration and replication techniques built primarily for monolithic architectures, does not focus on loose coupling. Most of them are located on a data level, where data-schemes are shared between individual applications. This leads to tight coupling: a change in the data-scheme of a single application requires changes in other applications too [Wol15].

There is little reliable research available which focuses on replication, replica actualization and invalidation for loosely coupled business IT systems. Most information is published in internet blogs, magazines and developer conference talks and is not founded on scientific research methods. Moreover, these publications only deal with certain concepts or technologies and do not cover critical comparison and evaluation. The absence of reliable decision criteria increases the risk of applying an inappropriate solution which may cause excessive costs or a situation where caching is simply not used.

Eventually, TU Wien is currently evaluating replication for the purpose of introducing the concept into its IT systems, which provides the practical application background for this thesis.

1.3 Aim of the Work

The aim of this work is to evaluate concepts and technologies for replication for their use in loosely coupled business IT systems, based on quantitative and qualitative criteria. Under the assumption that there is no single best solution for the requirements of all IT systems, concepts and technologies are further evaluated for their use in different cases of application. These evaluations should provide decision makers with the information needed to implement replication in their IT systems.

The focus of this work is the use of replication to decrease latency for read operations on the client. Other aspects like consistency of the client replicas, the size of transmitted data and quality parameters (e.g. reliability and maintainability) are also considered.

1.3.1 Research Questions

The research questions shall address and cover the before mentioned topics and are therefore formulated as follows:

1. *Which concepts and technologies can be used for replication in modern business IT systems?*
2. *How do concepts and technologies for replication perform in typical scenarios of business IT systems?*
3. *What are the characteristics, advantages and disadvantages of concepts and technologies for replication and how do they relate to different cases of applications?*

To answer *question 1*, replication is discussed from a theoretical perspective by its characteristics and its use in modern loosely coupled business IT systems. Based on this discussion, concrete replication strategies are presented and implementations based on modern technologies are composed.

To answer *question 2*, concepts and technologies are implemented and used in a simulation where client-side requests and server-side updates, that are replicated from the server to a client-side replica, are performed. Concepts are implemented for *Java EE* (representing enterprise platforms) and *Ruby on Rails* (representing more light-weight rapid-prototype platforms) to take the heterogeneity of modern IT systems into account.

To answer *question 3*, an evaluation of qualitative and quantitative indicators is performed. The qualitative indicators are evaluated based on an analysis of the ISO 25010 quality characteristics. The quantitative analysis is based on the results of the simulation and on a theoretical discussion of the characteristics and processes of the individual replication strategies (based on the recommendation of [Jai90] to validate results of a simulation by an analytical approach).

1.3.2 Delimination

In order to set the focus of this thesis on loosely coupled IT systems and limit the scope, the following constraints were defined:

- **Evaluation of technologies**: The aim of this work is to evaluate individual concepts for replication. The thesis does not cover evaluations and benchmarks of technologies and tools (e.g. Java EE in comparison to Ruby on Rails or a comparison of different message brokers).

- **Writeable replicas**: Write operations are only performed on the server. It is assumed, that the clients send write requests to the server (e.g. over webservices) for data updates. Replicas are read-only.
- **Strict consistency**: The discussed replication strategies partially build on eventual consistency and therefore inconsistencies in the replicas may occur. It is only guaranteed that replicas are consistent at some point of time in the future. Since write operations are only performed by the server and thus the server has a consistent state, this does not lead to a fundamental problem. Areas of application where strict consistency is crucial (like banking systems) are explicitly not covered by this thesis.
- **Autonomous invalidation**: Data items in replicas are invalidated by the server only. Other invalidation techniques (e.g. invalidation by time-to-live) are not examined but considered in course of the analysis of qualitative and quantitative indicators.
- **Persistence technologies**: The thesis focuses on replication. Technologies to implement replicas (like relational databases) are not examined.
- **Initial data import**: Some replication strategies need an initial data import to fill the replica. This operation can be performed in the data-layer more efficiently than on the application layer (e.g. with a direct import into the database). As persistence technologies are not dealt with, this subject is not covered either.
- **Horizontal scaling**: It is assumed, that every component exists only once. The investigated replication strategies can also be used when components are running redundantly multiple times. Specific requirements emerging through horizontal scaling are not in the scope of this work.

1.4 Methodological Approach

The methodological approach consists of the following steps:

1. **Analysis**: In the first step, concepts and technologies are selected and parameters to measure their performance are specified.
   a) **Selection of replication strategies**: Concepts and technologies that are used for caching and replication are examined. Starting point is a theoretical discussion of replication and characteristics of replication methods. Based on this discussion, concrete replication strategies for loosely coupled IT systems are presented and implementations with modern technologies are composed.
   b) **Specification of quantitative indicators and metrics**: Quantitative indicators and metrics to measure them are determined. These indicators

are meant to examine characteristics of replication strategies like runtime performance and consistency.

2. **Implementation**: In the second step, a simulation environment to measure the performance of replication strategies is implemented.
   a) **Design and implementation of a simulation environment**: Design of a simulation environment where replication with the replication strategies are simulated and quantitative indicators are measured. The simulation environment must be capable to run different test objects, execute different scenarios and measure the specified metrics. Further, it must be capable to run test objects in Java EE and Ruby on Rails. Therefore, parts of the simulation environment will be implemented two times, once for each platform.
   b) **Design and implementation of replication strategies**: Replication strategies are implemented two times, once in Java EE and once in Ruby on Rails. Replication strategies are designed and implemented, including the selection of suitable libraries and tools for the platform.
3. **Simulation**: In the third step, scenarios for the simulations are specified and the simulation is executed.
   a) **Specification of scenarios**: Scenarios for the simulation are specified. Scenarios differ in the number and characteristics of read and write operations. It is important to find realistic scenarios that represent operations inside real business IT systems. To achieve this, the specification is done in cooperation with the representatives of the information system of TU Wien and based on typical use cases of this system.
   b) **Execution**: Specified scenarios are simulated for each replication strategy while specified metrics are measured. The simulation must run in an isolated environment where no other processes or network traffic influence the results.
4. **Evaluation**: In the last step, replication strategies are evaluated.
   a) **Analysis of qualitative indicators**: Replication strategies are analyzed based on the quality model of ISO 25010 considering findings in the course of the implementation.
   b) **Evaluation of replication strategies**: Replication strategies are evaluated based on their performance in the simulation related to the specified use cases. Results are further analyzed on their overall performance, including qualitative indicators.

1.5 Structure of the Work

The following two chapters form the theoretical background of the work: *Chapter 2* summarises current research in the area of replication including characteristic, consistency,

levels of application, related evaluations and related benchmarks. *Chapter 3* describes communication technologies and paradigms used in this work.

Based on the fundamental research, *Chapter 4* discusses replication characteristics in the area of loosely coupled business IT systems and selects replication strategies for this domain. Concrete implementations are composed based on current technologies and discussed in detail.

Replication strategies and implementations are further analyzed in *Chapter 5*. It identifies and discusses quantitative indicators, metrics to measure them and analyses how replication strategies meet these indicators. Furthermore, qualitative characteristics of the replication strategies are addressed based on the quality model of ISO 25010.

After the theoretical analysis, *Chapter 6* presents the simulation. This includes the experimental setup, consisting of the selection of the used data and inspected use cases, the description of the implemented simulation environment, the implementation of the replication strategies in Java EE and Ruby on Rails and the simulation itself. The chapter concludes with the presentation of the simulation results.

The final evaluation of the qualitative and quantitative analysis and the simulation is done in *Chapter 7*. *Chapter 8* summarises the results of the work and discusses possible future work in this area.

CHAPTER 2

Related Work

Replication is a well-researched topic in computer science. This chapter summarises fundamental knowledge of the subject and the current state of the art research. In order to do so, *Section 2.1* gives a brief overview over replication. *Section 2.2* deals with consistency in systems that implement replication, followed by a discussion of how to implement replication on different layers of an application in *Section 2.3*. The chapter ends with a survey on related evaluation methods and benchmarks in *Section 2.4*.

2.1 Replication

Replication is the *"maintenance of copies of data at multiple computers"* [Cou05]. It is used in distributed system to distribute data on the **components** of which the system is composed of. Data is stored in a **data store** that consists of individual **data items**. A data item is characterized by individual attributes. Data items are usually clustered by attributes to **entity types**. In a database system, an entity type represents a database table and data items represent rows inside the table. Replication can be implemented on the level of entity types. In this case, a component on which data is replicated holds a replica where it stores copies of the data items in the data store.

The replication of data over multiple components is generally used to enhance performance, improve availability and improve fault-tolerance [Cou05] [Tan07]. The resulting distribution of data leads to the problem to keep replicas consistent [Tan07]. To ensure consistency, updates have to be propagated to all replicas to ensure that data items are identical in the entire system [Tan07].

The following subsections describe replication in detail. *Subsection 2.1.1* gives a general overview over types of replication based on content replication and placement. *Subsection 2.1.2* discusses caches and differentiates them as a special form of replicas. The last three subsections contain characteristics of methods to ensure consistency, based on [Tan07].

2.1.1 Content Replication and Placement

[Tan07] and [Pie02] give a general overview over types of replication by distinguishing replicas based on where they are placed at and by whom they are initiated. There are three different types, logically organized as shown in Figure 2.1:

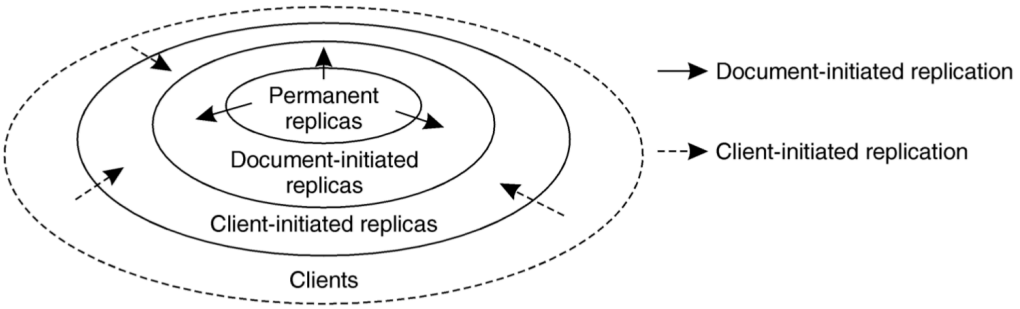


Figure 2.1: The logical organization of different kinds of copies of a data store [Pie02]

- **Permanent Replicas**: These replicas form an initial set of replicas which constitute a distributed data store. An example of permanent replicas is a distributed database, where a database is distributed over a number of servers that form a cluster. For website replication, it may be further distinguished based on the geographical distribution of content: content can be placed on a limited number of servers on a single location or can be spread geographically around the internet (mirroring).
- **Document-Initiated Replicas**: (also known as *Server-Initiated Replicas*) Are copies of a data store which are created on the initiative of the (owner of the) data store to enhance performance. These can be spread geographically and may be only temporarily available in order to handle peak load over a limited period of time (e.g. *Content Delivery Networks*).
- **Client-Initiated Replicas**[1]: Are local storage facilities which are used by a client to store a copy of data temporarily. These replicas are primarily used to improve data access times.

Permanent replicas form a distributed, joint data store. Write operations can be performed on each replica and are replicated to other replicas in the background. To do so, it can be distinguished between*primary-based* protocols where one replica is responsible for the state of a data item and changes are forwarded to this node (in the background – this communication is non-transparent to clients for which it looks like a local operation) and *replicated-write* protocols where replicas coordinate the state of the overall system among themselves (e.g. by a quorum based approach). Figure 2.2 illustrates permanent replicas and their interactions. [Tan07]

[1] [Tan07] also used the term "cache" for client-imitated replicas. In the terminology of this thesis, the term "cache" is used in a narrower context (see *Subsection 2.1.2*).

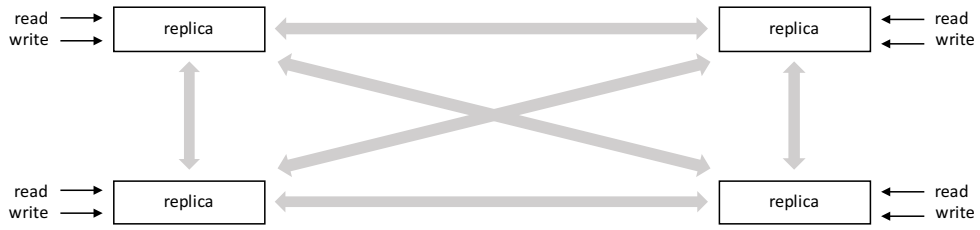


Figure 2.2: Permanent replicas and their interactions

Initiated replicas are typically read-only. Write operations are performed on a permanent replica. In the context of this thesis, the permanent replica where write operations take place is named **server**. Changes are distributed to the replicas (see *Subsection 2.1.4* how this distribution can be implemented). Figure 2.3 illustrates such a system. [Tan07]

Figure 2.3: Read-only initiated replicas and their interactions

This thesis only considers read-only replicas where update operations are only performed on the server (see *Subsection 1.3.2*). Such systems can be implemented with initiated replicas. The further discussion will therefore focus on initiated replicas.

2.1.2 Caching and Replication

In general, a replica is a copy of a data set on another machine to improve performance, availability and fault tolerance [Bae97] [Lou02] [Tan07]. A **Cache** is a special form of a replica. In the words of [RFC-7234], *"a cache stores cacheable responses in order to reduce the response time and network bandwidth consumption on future, equivalent requests."* A cache is a client-initiated replica that contains only data that was requested before. It can be held locally (e.g. in a web browser) or on another component of the system (e.g. a proxy server) [Bae97] [Tan07].

In this thesis, **Completeness** of a replica is defined as the amount of the server-side data that is contained in the client-side replica. From the point of view of completeness, a replica always contains a data item X, a cache may contain it or not [Pie01]. Caching therefore is a special case of replication, where a replica only holds parts of the data [Lou02].

In the context of this thesis, a **Full Replica** is the representation of all server-side data and therefore all data items and a **Cache** stores only requested data items so that there is no need to request them again.

Because caches hold only parts of the data, they cannot be used to improve availability and fault tolerance. As [RFC-7234] outlines, they are primarily used to improve performance.

2.1.3 Change Distribution

[Pie02] defines change distribution as *"how changes between replicas are distributed."* For initiated replicas, it characterizes which information is distributed between the server and the replicas in order to achieve consistency. [Pie02] and [Tan07] distinguish four change distribution types:

- **Notification** (also known as *invalidation protocols*): Replicas are notified which data item has changed and invalidate this data item locally. The new version has to be loaded from the server separately. The main advantage of invalidation protocols is that they use little network bandwidth. They are very efficient when the read-to-write ratio is relatively low and so the probability that every update is read from the replica (and has to be loaded from the server) is low as well [Tan07].
- **Full state**; The new version of the data item is propagated to the replicas. This type is useful when the read-to-write ratio is relatively high [Tan07].
- **State difference** (only in [Pie02]): The delta between the old version and the new version is propagated to the replicas. If changes are very small compared to the whole data item, this approach leads to lower bandwidth consumption than the propagation of the whole data item [Ban97].
- **Operations**: The operation that leads to the change in data is propagated to the replicas. Replicas must be capable to perform the operations on their data autonomously. The main benefit of this approach is that operations require very little network bandwidth compared to the propagation of the whole data item [Tan07].

2.1.4 Replica Reaction

[Pie02] defines replica reaction as *"how an update is propagated to replicas."* For initiated replicas, it characterizes how changes on the server-side data store are propagated to client-side replicas. [Pie02], [Tan07] and [Yin99] distinguish *server-based / push-based* protocols, where updates are actively pushed to the replicas and *client-based / pull-based* protocols, where a replica pulls the updates from the server. [Yin99] further distinguishes client-based protocols, leading to the following three types:

- **Server-driven** (PUSH): Active notification of replicas when data changes. These protocols are used when replicas need a high degree of consistency [Tan07].

- **Polling on each read** (PULL): Replicas are actively requesting the current state of a data item when it is requested. These protocols supply strict consistency but increase the load on the server and the latency of each read [Yin99]. They are efficient when the read-to-write ratio is relatively low [Tan07].
- **Periodic polling** (PULL): Replicas are periodically requesting the server state. Replicas are in an inconsistent state between the update on the server-side data store and the next poll of the server state (leading to eventual consistency, see *Subsection 2.2.1*) [Yin99].

Table 2.1 describes the types of replica reaction by distinguishing pull and push-based approaches and the event that leads to the change:

Replica Reaction	**PULL / PUSH**	**Event**
server-driven	PUSH	change of data
polling on each read	PULL	request of data
periodic polling	PULL	interval

Table 2.1: Types of Replica Reaction

2.1.5 Addressing Method

The addressing method determines how messages are sent to multiple receivers. For initiated replicas, it characterizes how the server can propagate updates to multiple replicas. [Tan07] distinguishes two addressing methods:

- **Unicast** If an update must be propagated to n replicas, n messages are sent.
- **Multicast** If an update must be propagated to n replicas, only one message is sent. The system takes care of sending messages efficiently to multiple receivers (e.g. with *IP Multicast*).

Recommendations for the addressing method used can be made based on the replica reaction (*Subsection 2.1.4*). With a pull-based approach, a replica is directly communicating with the initial data set. Benefits of multicast cannot be used here because of the one-to-one communication. [Tan07] notes that unicast *„may be the most efficient solution.“* With a pull-based approach, every client has to be notified. Multicast can lead to a more efficient solution, where the server and the network can be relieved of pressure because of the optimization of the used multicast implementation [Tan07] [Yu99]. Furthermore, multicast can enable loose coupling when clients can independently subscribe to multicast groups (depending on the multicast implementation).

As a consequence of these recommendations, **push-based replication is implemented by unicast** and **pull-based replication by multicast** in the course of this work.

2.2 Consistency

One of the major problems with replication is to maintain consistency [Tan07]. Data are stored in many places and updates must be propagated to all places viz. to all replicas. This section discusses consistency in replicated, distributed systems. *Subsection 2.2.1* discusses consistency models for read-only replicas and *Subsection 2.2.2* summarises current research on trade-offs between consistency and other properties.

2.2.1 Consistency Models

Most consistency models in literature assume that data is written directly in the replicas and thus are very complex (see [Tan07]). This thesis only considers read-only replicas where update operations are only performed on the server (see *Subsection 1.3.2*). Consistency models for writeable replicas are not considered in this section.

Since write operations are only performed by the server, the server always has a consistent state. Only the client is affected by inconsistencies. From the client-side point of view, there are three types of consistency [Vog09]:

- **Strong consistency**: After an update, all subsequent access to data will return the updated value. Strong consistency can be implemented in a distributed system by *Two Phase Commits* [Ami02].
- **Eventual consistency**: Guarantees that after an update, when no new updates take place, access to data will eventually return the updated value. After a long time without updates, all replicas will gradually become consistent [Tan07].
- **Weak consistency**: Does not guarantee that after an update a subsequent access to data will return the updated value.

Strong consistency is widely used because it is implemented in modern database management systems by the *ACID*[2] properties [Bre12]. [Flo09] argues, that *Service Oriented Architectures* (*SOA*) and ACID constraints do not mix well, because ACID requires full control of all data management activities (via transactions), whereas SOA implies autonomy of services and their data. Techniques like Two Phase Commits help to support ACID-style transactions in SOA, but are *"painful"* to use [Flo09]. The problems with strong consistency arise from the distribution of data and the autonomy of components. Therefore, this discussion is not solely valid for SOA but may be applied to distributed and loosely coupled IT systems in general.

[Hel07] and [Vog07] argue that strong consistency and ACID transactions are rarely needed. [New15] and [Wol15] are also discussing eventual consistency and transaction and draw the conclusion that the use of eventual consistency depends on the use case. It is assumed, that many use cases can be redesigned to fit to an eventual consistent model

[2] *ACID = atomicity (A), consistency (C), isolation (I), durability (D)*

[Wol15]. For the course of the work, it is assumed that eventual consistent models meet the needs of loosely distributed IT systems.

Eventual consistency can be further classified into four models[3] [Vog09]:

- **Causal consistency**: If a process is notified about a new version of a data item, any subsequent access to the data item will return the new version.
- **Read-your-writes consistency**: If a process updates a data item to a new version, any subsequent access to the data item will never return an older version.
- **Monotonic read consistency**: If a process reads a version of a data item, any subsequent access to the data item will never return an older version.
- **Monotonic write consistency**: *"The systems guarantees to serialize writes by the same process."* [Vog09]

2.2.2 Trade-Offs

When designing distributed systems, there are three properties that are commonly desired [Ada12] [Bre00] [Gil02]:

- **Consistency** (C): The system provides strong consistency.
- **Availability** (A): Data is highly available.
- **Partition-Tolerance** (P): When the system is distributed to several network partitions, it can handle the failure of partitions and messages lost between individual partitions.

In 2000, *Brewer* [Bre00] conjectured that it is impossible to achieve all three properties. This conjecture was formally proved by *Gilbert* and *Lynch* [Gil02] in 2002 and is known as the *CAP theorem*. Based on the CAP theorem, there are three possible system types:

- **CA systems**: Consistent and highly available, but not partition-tolerant. Applications are e.g. relational database systems with ACID constraints.
- **CP systems**: Consistent and partition-tolerant, but not highly available. Applications are e.g. banking applications.
- **AP systems**: Highly available and partition-tolerant, but not consistent. Applications are e.g. the domain name system (DNS) and cloud computing.

[3] [Vog09] also describes *session consistency* as a fifth variation, that is not considered in this work.

Later discussions on the CAP theorem clarify that this decision is not binary [Bre12]. There is no exclusive choice for two of the three properties and no need to sacrifice the third, but it is also possible to weaken one or two of the properties. Even though the described systems are oversimplified, they help to understand the trade-offs.

Adabi [Ada12] defined the *PACELC*[4] theorem (not proofen) which extends CAP and also considers replication. PACELC distinguishes between systems where network partitions exist and systems with no partitions. In the case of network partitions, the conditions apply according to the CAP theorem (choice between *consistency* and *availability*). When partitions need not to be considered, *Adabi* argues that high availability can only be provided by replication. *Adabi* described several cases and concludes that replication has a trade-off between *consistency* and *latency*. Strong consistency needs communication between the components, leading to higher latency. Lower consistency models need less communication and therefore have lower latency.

2.3 Implementation of Replication on different Layers

Communication between client and server is necessary to ensure consistency in the replicas. This section discusses different layers of communication and the situation where replication can be implemented. The discussion is based on the well-known three-layer model for enterprise applications [Bor05] [Fow02]:

- **Presentation**: Integration with the end-user (e.g. *HTML* web interface)
- **Application** (or *Domain*): Business logic (e.g. *Enterprise Java Beans*[5])
- **Data**: Persistence and access to data (e.g. database)

The individual layers and replication on these layers are discussed in the next subsections in detail. It is concluded that replication in loosely coupled IT system must be implemented on an application layer.

The discussion in this section partially bases on works in the area of *Microservices*. A Microservice architecture is a distributed system formed of many small and loosely coupled applications [Fow14]. Because applications in a Microservice architecture are very small, these systems need more individual applications to implement the same number of use cases than a traditional system. More individual applications also lead to more communication between applications and therefore research on Microservice architectures focuses on the communication and integration of applications. Even though not every distributed IT system follows a Microservice architecture, research done on communication and integration can be used for loosely coupled IT system in general.

[4] PACELEC can be read as: "*if there is a partition (P), how does the system trade off between availability and consistency (A and C); else (E), when the system is running normally in the absence of partitions, how does the system trade off latency (L) and consistency (C)*" [Ada12].

[5] `https://jcp.org/en/jsr/detail?id=345` (last checked 26.03.2018)

2.3.1 Replication on the Data Layer

The easiest way to integrate applications on the data layer is by a *shared database*, where all applications access the same database and database tables [Hoh04]. Replication can be implemented by standard functionality like *distributed databases*, where replication is realized by the *database management system* [Wol15].

Replication on the data layer leads to a shared data schema [Hoh04]. Even though these techniques are often used in practice, they violate data encapsulation and lead to tight coupling [Ric15]. Tight coupling results because a change in the data schema of a single application requires changes in other applications too [Wol15]. Other problems with shared databases are the difficulty to create a shared data schema that satisfies the requirements of all applications and the risk of turning the database into a performance bottleneck [Hoh04]. Integration on this layer is not recommended [Hoh04] [Ric17b] [Wol15] and will not be done in this thesis.

2.3.2 Replication on the Application Layer

Replication on the application layer can be accomplished *synchronously* (e.g. with *RESTful* webservices) or *asynchronously* (e.g. push-based with *Messaging* [Wol15] or pull-based with *Web Feeds* [Kar07] [Web10]).

Research on the use of replication in modern IT systems showed that replication is mostly implemented asynchronously on an application layer with Messaging or Web Feeds. Examples are *Apache Airavata* [Dha17] (Messaging), the *ORACLE ATG Web Commerce Platform* [ATG] (Messaging) and the e-commerce system of the German online retailer *OTTO* [Ste15] (Web Feeds).

Asynchronous integration of applications is highly related to *Event Driven Architectures*, where interaction between systems is based on events. Events are used for changes in data or state and transmitted to subscribed components by event notifications [EDA]. Event Driven Architectures can be implemented by a pull or push-based approach [San15] [Gie15]. [Fow17] distinguishes individual types of Event Driven Architectures and defines a type specifically for replication: *Event-Carried State Transfer*. The aim of Event-Carried State Transfer is to decouple components in a distributed system. A client holds a replica of the server-side data store and needs no server communication to read data from the data store. Modification of data items in the server-side data store lead to events that are communicated to the client to update the client-side replica.

The application layer is mostly implemented individually and not by standard software. Individual implementations are necessary because the application layer technically implements the business logic that constitutes the unique selling proposition and generates the actual product value. Therefore, it is assumed that replication (or parts of it) has to be implemented individually in order to integrate to the individual parts of the application layer. This assumption corresponds to the finding that there are no standard libraries available that completely implement replication at the application layer.

2.3.3 Replication on the Presentation Layer

It is not possible to implement replication on the presentation layer because data cannot be replicated into the layers below (application and data layer).

2.4 Evaluations and Benchmarks

There is currently no comparable comprehensive scientific survey or evaluation for replication specifically for loosely coupled systems. There are already surveys and evaluations that cover parts of the topic and tools for benchmarking that can be used to measure replications effects. This section gives an overview over these works.

2.4.1 Evaluations

Analytical surveys for replication in loosely coupled IT systems can mostly be found in unstructured, non-academic form (e.g. blog posts). These will not be listed here. However, there is standard literature in the area of *Microservices* and *Self Contained Systems* (a special type of Microservices) where replication is covered [New15] [SCS] [Wol15]. *Event Driven Architecture*, *Event-Carried State Transfer* and the comparison of pull and push-based concepts are discussed in the developer conference talks of [San15] and [Gie15]. These sources provide the basis for the requirements and as an indication for the selection of the replication strategies and their implementation.

There are currently no simulations of replication on the application layer. A related area of research, even if the used technologies differ, is replication of documents over several webservers. The requirements are similar, there is also no need for strong consistency and replicas are read-only. [Bae97], [Cao98], [Gwe96], [Pie01], [Siv07], [Yin99] and [Yu99] discuss related replication strategies like the ones considered in this thesis and observed their behavior using simulations. The results provided by these evaluations build the basis of this thesis.

Simulations to measure consistency metrics in eventual consistent systems were especially performed for *NoSQL* and *cloud databases* in [Ber13], [Ber14] and [Wad11]. Findings of these simulations were specific for NoSQL and cloud databases and therefore results could not be used in the context of this thesis. The setup of the simulations environment provided an indicator for the simulation performed in this work.

Furthermore, there are many evaluations and simulations for replication on the data-layer (e.g. *distributed file systems* and *distributed databases*). These cover another layer of communication (see *Subsection 2.3.1*) and build on writeable replicas (see *Subsection 1.3.2*), which are excluded from the scope of this thesis. Therefore, they were not considered.

2.4.2 Benchmarks

There is a wide range of benchmarks and tools available to measure web application performance. Most insights into system and implementation behavior can be obtained from benchmark applications that offer a server side web application and generate workloads by emulating user sessions per HTTP requests. They measure metrics from a client and a server perspective. The most popular benchmark applications are:

- **TPC-W**[6] [Gar03] [Smi00]: Is an industry-standard e-commerce benchmark by the *Transaction Processing Performance Council* that simulates an online store where users browse and buy products from a website. TPC-W bases on a complex infrastructure with many servers performing different functions (e.g. Web Servers, Web Caches, Image Servers or a Database Server). The main metric is Web Interactions Per Seconds (WIPS) that can be sustained by the system or individual servers. Workloads are generated by independent shopping sessions following a model-based sequence of interactions (like search, browse, add to shopping cart and purchase).
- **RUBBoS**[7] [Amz02]: Is a bulletin board benchmark by the *OW2 Consortium*, modeled after *slashdot.org*. Implementations are available for PHP and Java Servlets. Users can browse stories, submit stories, comment stories, review stories and rate comments. The workload exhibits a high locality to represent the properties of a bulletin board where users are usually interested in the latest news [Siv07].
- **RUBiS**[8] [Amz02]: Is an auction site benchmark by the *OW2 Consortium*, modeled after *ebay.com*. Implementations are available for PHP, Java Servlets and Enterprise Java Beans (EJB). The auction site implements core functionality (selling, browsing and bidding) by 26 interactions.

Another web benchmark is *SPECweb99*[9] [Nah02] by the *Standard Performance Evaluation Corporation*. It includes no benchmark application but provides a standard workload that can be executed on existing websites and applications. The primary metric used by SPECWeb99 is the number of simultaneous conforming connections (unlike other benchmarks measuring server throughput or response time).

There are also *Load Testing Tools* for web applications that simulate a large number of clients by producing HTTP request on an application or website. They can be scripted to customize the workload to the individual needs. Because these tools are not integrated in the application but act purely as HTTP clients, they only provide metrics from the client perspective. Popular examples are *gatling*[10], *Apache Bench*[11] and *Apache JMeter*[12].

[6] `http://www.tpc.org/tpcw/` (last checked 26.03.2018)
[7] `http://jmob.ow2.org/rubbos.html` (last checked 26.03.2018)
[8] `http://rubis.ow2.org/` (last checked 26.03.2018)
[9] `https://www.spec.org/web99/` (last checked 26.03.2018)
[10] `https://gatling.io/` (last checked 26.03.2018)
[11] `https://httpd.apache.org/docs/2.4/programs/ab.html` (last checked 26.03.2018)
[12] `https://jmeter.apache.org/` (last checked 26.03.2018)

All these benchmarks and tools implement a *closed system model* where new requests are triggered by the completion of a previous request [Sch06]. A load testing tool that uses an *open system model*, where new requests are independent of former requests, is *httperf* [13].

There are also benchmarks to measure the performance of eventually consistent systems. The *Yahoo! Cloud Serving Benchmark* (*YCSB*)[14] [Coo10] was created to analyze cloud based databases (that provide online write/read access). Based on its open architecture it can be extended to handle other databases or application as well. YCSB initially measures only two tiers: Performance and Scaling. *YCSB++*, a set of extensions to YCSB, includes eventual consistency measurement [Pat11].

The survey in this chapter concludes that replication in loosely coupled IT systems is preferable implemented on the application layer. Technologies to do so were mentioned throughout the chapter. The next chapter describes these technologies and their features needed to implement replication in detail.

[13] `https://github.com/httperf/httperf` (last checked 26.03.2018)
[14] `https://github.com/brianfrankcooper/YCSB` (last checked 26.03.2018)

CHAPTER 3

Communication Paradigms

This thesis uses existing technologies and communication paradigms to implement replication strategies. This chapter describes these technologies and paradigms. The description includes a survey on libraries and tools to use the technologies. The focus of this survey is on Java EE and Ruby on Rails because these platforms form the basis for the practical part of this thesis.

3.1 REST - Representative State Transfer

Representative State Transfer (*REST*) was introduced by *Roy Fielding* in his dissertation [Fie00] in 2000 to condense the foundation of the world wide web in a general architectural style. In practice, REST is implemented with HTTP. Other implementations are possible as well, but "*attempts to do so can be an expensive proposition*" [All10]. REST has been established as the standard for modern web *Application Programming Interfaces* (*APIs*).

Key elements are *resources*. A resource is any information that can *named* [Fie00] and *referenced* as a thing itself [Ric07]. Resources can be anything from a text file, media file (e.g. an image), a row in a database or a business process (e.g. a procedure) [Dai12].

HTTP and RESTful webservices are standard functionalities of modern web-based application platforms. As an example, Java EE provides the *Java API for RESTful Webservices* (*JAX-RS*)[1] standard to create RESTful servers and clients. The Ruby on Rails framework entirely builds on the concept of RESTful webservices to provide web pages and web services[2].

The subsequent subsections describe architectural constraints of RESTful architecture, provide examples of RESTful resources and explain how caching can be implemented using HTTP.

[1] `https://jcp.org/en/jsr/detail?id=339` (last checked 26.03.2018)

[2] `http://guides.rubyonrails.org/getting_started.html` (last checked 26.03.2018)

3.1.1 Architectural Constraints

[Fie00] defined a set of constraints a system must meet to be RESTful. The following section describes the constraints including considerations on their use in HTTP:

- **Client-Server**: REST implements a client-server architecture.
- **Stateless**: Each request from the client to the server must contain all information necessary to understand the request on the server-side. Session state is stored entirely on the client and the server cannot take advantage of any stored context. In the case of HTTP based web application, server-side sessions are not allowed.
- **Cache**: Data within a request or response can explicitly be labeled as cacheable or non-cacheable. The client can read cacheable responses from a cache and reuse it for later, equivalent requests. HTTP provides standard mechanisms which are described in *Subsection 3.1.3*.
- **Uniform Interface**: A uniform interface is used between all components. This constraint is further subdivided into four interface constraints:
  - **Identification of Resources**: Resources are identified by a unique *uniform resource identifier* (*URI*). In the case of HTTP, this is the HTTP URI [Dai12].
  - **Manipulation of Resources Through Representations**: Resources are represented in different formats to encapsulate the resource and its information (e.g. encoded in a markup language such as XML or JSON). Manipulation of resources is performed with representations of the resource. In HTTP, the representation format can be defined via the `Content-Type` and `Accept` headers of HTTP requests and responses [All10].
  - **Self-Descriptive Messages**: Messages must be self-descriptive. The uniform interface provides all information needed to perform actions on resources and no other, proprietary information is necessary. This constraint is implemented in HTTP with the HTTP methods (e.g. GET, POST and DELETE) [Ric13].
  - **Hypermedia as the Engine of Application State** (*HATEOAS*): Hyperlinks are used in messages to guide clients through the application's state [Dai12]. This is similar to the world wide web, where users start at an entry point (e.g. a search engine) and navigate the web through hyperlinks. HATEOAS allows clients and servers to evolve independently and enables loose coupling.

    Though HATEOAS is recommended in nearly every publication about RESTful webservices (e.g. [All10], [Ric07], [Ric13] and [Web10]), it is hardly adopted in public web APIs[3] [Bul14] [Dai12] [Ren12]. HATEOAS has no specific advantage for the examined case of replication and is therefore not further considered in this work.

[3] To the author's knowledge, there are no surveys of the use of HATEOAS in private web APIs available.

- **Layered System**: [Fie00] describes a layered system as "*an architecture to be composed of hierarchical layers by constraining component behavior such that each component cannot "see" beyond the immediate layer with which they are interacting.*" In the case of HTTP, it allows proxy servers and gateways to act intermediates in the communication [Ric13]. Adding this components into the system is nearly transparent to the clients [Ric13].

- **Code on Demand** (optional): Code on demand allows "*client functionality to be extended by downloading and executing code in the form of applets or scripts*" [Fie00]. It is not recommended to use this functionality as a general solution [Ric13] and is only considered as an optional constraint in [Fie00]. Therefore, it is not considered in this work.

3.1.2 The use of HTTP Methods in REST

The practical use of RESTful webservices is illustrated by a web shop that manages orders. A typical resource in such a system is an order. It can be represented by a unique URI such as `/orders/123`. To access the order and perform actions on it, it can be accessed using HTTP methods. Table 3.1 gives an overview of HTTP methods and their effect on the order resource (the table shows only a subset of possible methods).

HTTP method	**Action**
GET	returns the order
POST	not supported in this context
PUT	replaces the order with the representation of a single order in the request body
DELETE	deletes the order

Table 3.1: HTTP methods and their effect on the order resource

A resource can also represent a collection of resources. The URI `/orders` would represent all orders in a web shop. Table 3.2 gives an overview of HTTP methods and their effect on a collection of orders (the table shows only a subset of possible methods).

HTTP method	**Action**
GET	returns all orders
POST	adds a new order
PUT	replaces all orders with the representations of a collection of orders in the request body
DELETE	deletes all orders

Table 3.2: HTTP methods and their effect on a collection of orders

It is not required to provide all methods for any resource. The deletion of all orders in a web shop is an unlikely use case of application and would presumably not be implemented. Resources can also be organized hierarchically. All items in an order could be represented by the URI `/orders/123/items` and accessed and manipulated via HTTP methods as described before.

3.1.3 Caching

Caching is a primary constraint in RESTful architectures. HTTP supports caching on multiple levels of the layered system (e.g. for client-side caches or for proxy servers that serve multiple machines). This survey focuses on HTTP message headers for client caches that can be applied for replication. For a detailed overview over caching mechanisms see [Web10]. HTTP supports client caching by the following three headers [RFC-7234]:

- **Expires Header**: A response can be equipped with a timestamp that defines its validity (*time to live*) in the `Expires` header of the server response. Clients have to invalidate data from their cache independently after the time to live has expired. In practice, it is hard to assign a appropriate time to live for a data item [Cao98]. Too small values lead to lots of requests to the server and minimize the advantage using a cache, too high values lead to high inconsistencies in the data [Cao98].

- **Last Modified Header**: A response can be equipped with the timestamp of the last modification in the `Last-Modified` header of the server response. Clients can store this information and perform a *conditional GET* to the server including a `If-Modified-Since` header, telling the server only to send data if it has changed in the meantime. If data has not changed and the modification date is the same, the data in the cache is still valid and can be used. Therefore, no data has to be transmitted. In this case, the server returns an empty request body with the HTTP status code `309 Not Modified` to the client. Otherwise the server returns the data and the new modification date. Figure 3.1 illustrates a scenario where data is initially loaded (1) and requested two times: the first time without modifications (2) and the second time after a modification on the server side (3).

- **ETag Header**: A response can be quipped with an unique identification key (e.g. a hash) of the data in the `ETag` header of the server response. Clients can store this information and perform a *conditional GET* to the server including a `If-None-Match` header, telling the server only to send data if it differs from the version identified by the identification key. If data has not changed and the ETag is the same, the data in the cache is still valid and can be used. Therefore, no data has to be transmitted.. In this case, the server returns a empty request body with the HTTP status code `309 Not Modified` to the client. Otherwise the server returns the data and the new ETag. Figure 3.2 illustrates a scenario where data is initially loaded (1) and requested two times: the first time without modifications (2) and the second time after a modification on the server side (3).

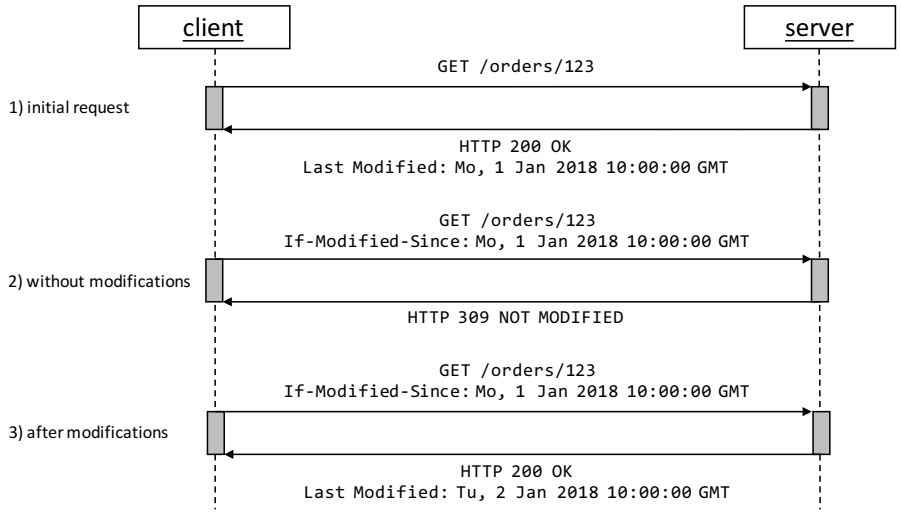


Figure 3.1: Example of the use of the Last-Modified HTTP header

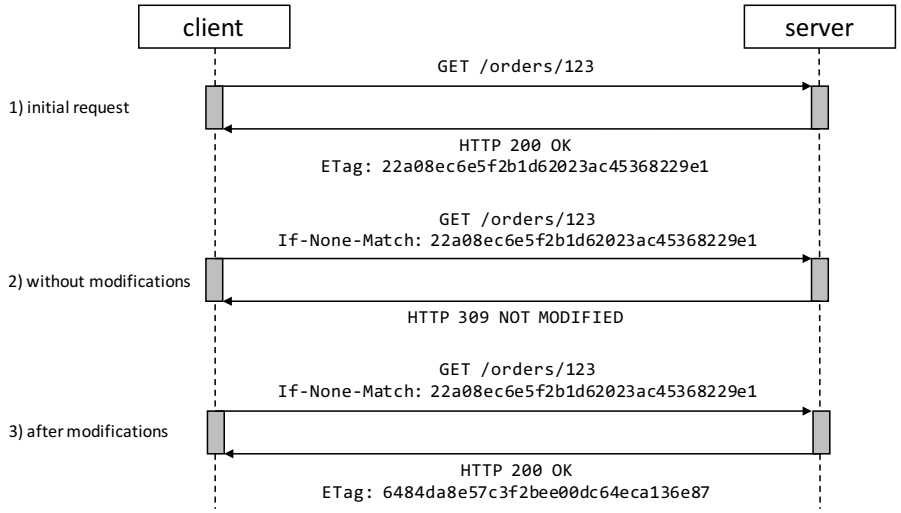


Figure 3.2: Example of the use of the ETag HTTP header

3.2 Web Feeds

Web feeds are traditionally used for publication of news and changes on websites such as news-sites, blogs and wikis. Interested readers (e.g. human users) can regularly poll the feed and are notified when additional or updated content is available. Web feeds can also be used for publishing changes for machine-to-machine integration inside IT systems, containing changes in REST resources or database tables [Gie15] [Kar07] [Web10]. In this case, readers are not human users but other components and applications in the IT system.

Technically, a web feed represents a structured list of items [Web10]. Items in a web feed are called *entries* and associated document metadata with web content [Web10]. The web feed is provided by a server via HTTP. Clients regularly read (poll) the feed in order to be notified when changes happen. Figure 3.3 shows the use of web feeds.

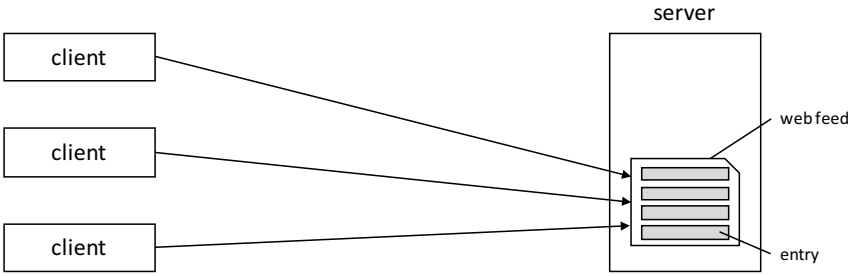


Figure 3.3: Example of the use of web feeds

The subsequent subsections describe web feed standards and one of the standards, the *Atom Syndication Protocol*, in detail.

3.2.1 Standards

There are several standards for web feeds available. The most popular standards are:

- **Really Simple Syndication**, formerly *Rich Site Summary* (*RSS*)[4]: Is a XML-based web content syndication format. RSS was developed at Netscape and got popularity with the rise of web blogs.

- **Atom Syndication Protocol** (*Atom*): Is a XML-based web content syndication format, defined in [RFC-4287]. It was introduced as the successor of RSS.

- **Atom Publishing Protocol** (*AtomPub*): Is an extension of Atom for publishing timestamped lists of web content, defined in [RFC-5023]. The RFC describes AtomPub as "*an application-level protocol for publishing and editing Web resources. The protocol is based on HTTP transfer of Atom-formatted representations.*" AtomPub defines rules for a client and a server to create and edit web resources by using HTTP to implement a domain-specific application protocol [Web10]. It can be used for manipulating the contents of Atom feeds in a standardized way [Web10]. A AtomPub server (e.g. *AtomHopper*[5]) offers this functionality.

- **JSON Feed**[6]: Is a JSON-based web content syndication format. JSON has established itself as the de facto standard exchange format for web services [Ric13].

[4] `http://www.rssboard.org/rss-specification` (last checked 26.03.2018)
[5] `http://atomhopper.org/` (last checked 26.03.2018)
[6] `https://jsonfeed.org/` (last checked 26.03.2018)

A problem with the presented XML-based standards is that they do not fit into the existing JSON-based environment and are therefore hardly used [Ric13]. The motivation of JSON Feed was to supply a format similar to RSS and Atom in JSON. JSON Feed was introduced in 2017. Due to the recent introduction, implementations for supplying and reading feeds are not fully developed yet and not available for all platforms (e.g. there is no implementation for Java)[7].

The Atom Syndication Protocol is currently the most mature standard for providing web feeds and is therefore used in this thesis.

3.2.2 Atom Syndication Protocol

The Atom Syndication Protocol provides a format for web feeds. The structure of the web feed is defined by the Atom specification (RFC 4287). The content of the web feed is customizable and depends on the domains requirements [Web10]. The following example shows a simplified web feed that provides orders in a web shop.

```
<?xml version="1.0" encoding="UTF-8"?>
<feed xmlns="http://www.w3.org/2005/Atom">
  <id>http://www.webshop.com/orders</id>
  <title>orders</title>
  <updated>2018-01-01T10:00:00Z</updated>
  <link rel="self" href="http://wwww.webshop.com/orders" />
  <entry>
    <id>orders/2</id>
    <title>order #2</title>
    <updated>2018-01-01T10:00:00Z</updated>
    <author>
      <name>user 1256</name>
    </author>
    <link rel="self" href="http://www.webshop.com/orders/2" />
    <category term="data" />
    <content type="application/xml">
      <order xmlns="http://schemas.webshop.com/orders">
        <customer>
          <user id="1256" />
        </customer>
        <article id="1001" />
        <article id="2002" />
      </order>
    </content>
  </entry>
</feed>
```

[7] https://jsonfeed.org/code (last checked 26.03.2018)

Atom feeds can be used for machine-to-machine communication. The presented web feed is an example for such an use case. The meta-data in the Atom feed sets the context for orders and includes the following elements (based on the description in [Web10]):

- `<atom:id>`: Is a permanent and unique identifier for the feed.
- `<atom:title>`: Is the human-readable name of the feed.
- `<atom:updated>`: Is the last date the feed changed.
- `<atom:link>`: Is a reference to related content. The `self` link contains "*the canonical URI for retrieving the feed*" [Web10]. Other link types include `previous` and `next` (to navigate through linked feeds) [Web10]. [Web10] referenced *IANA*s link registry[8] for recognized links that are recommended to use.

The actual information is represented by the `<atom:entry>` elements. In this example, there is just one order which is represented by a single entry. Each entry consists of metadata and the actual content. The metadata includes the following elements (based on the description in [Web10]):

- `<atom:id>`: Is a unique identifier for the entry.
- `<atom:title>`: Is the human-readable name of the entry.
- `<atom:updated>`: Is the last date the entry has changed. In this example, it is the time the order was made.
- `<atom:author>`: Provides information about the author that created the entry. In this example, it is the user who made the order.
- `<atom:link>`: Is a reference to related content. The `self` link contains "*the URI for addressing this entry as a standalone document*" [Web10]. Other link types include `related` (for a related document) and `alternate` (for an alternative representation) [Web10].
- `<atom:category>`: Is an specification to organize entries. The specification explicitly assigns no meaning to the content of this element. Categories therefore can be used for categorizing in the context of the particular use case [Web10].

The actual content is located in the `<atom:content>` element. In this example, the order is mapped as XML in an arbitrary, foreign namespace. There are no restrictions on the content and structure of the content. Content can be of any content type (e.g. XML, JSON or images). This allows Atom feeds to be customized for a particular use case.

There exist several libraries to implement Atom feeds. Ruby supports the RSS library in its core packages[9], that supports reading and writing of RSS and Atom Feeds. Even if

[8] `https://www.iana.org/assignments/link-relations/link-relations.xhtml` (last checked 26.03.2018)

[9] `https://ruby-doc.org/stdlib-2.1.3/libdoc/rss/rdoc/RSS/Atom/Feed.html` (last checked 26.03.2018)

the library is mature, the documentation is insufficient and incomplete, making it difficult to use. There are other libraries for reading web feeds[10] or writing them[11] available for Ruby as well. These are mostly private projects and often not maintained. Therefore, it cannot be recommended to use these libraries in business IT-Systems. For Java, there are two popular libraries to read and write Atom feeds: *ROME*[12], an independent library which is used by the spring framework and others and *Apache Abdera*[13], which was widely used but retired in 2017. Because Atom is based on XML, it is also possible to parse it with a standard XML parser.

3.3 Messaging

[Hoh04] describes messaging as „*a technology that enables high-speed, asynchronous, program-to-program communication with reliable delivery.*“

With Messaging, programs communicate by sending *messages*. These messages are send through *channels* (also known as *queues*) that are the logical pathways to connect programs among each other. A message is sent by a *producer* or *sender* by writing the messaging into the channel. A *consumer* or *reader* receives the message by reading (and deleting) it from the channel. Communication is asynchronous by design: producers send messages through channel and proceed. They do not wait until the message is processed by the consumers. Figure 3.4 illustrates communication via messaging. [Hoh04]

Figure 3.4: General structure of communication via messaging

Messages can be any sort of data structure, like a string, a byte array or an object. Complex data types can be serialized (directly to byte code or in a markup language such as XML or JSON) and transmitted to the consumers. A message usually consists of two parts: a *header*, which contains meta-information and a *body*, which contains the real payload. [Hoh04]

Messaging capabilities are typically implemented by a *message broker* (also *Messaging System* or *Message Oriented Middleware*). It provides channels and moves the messages from the consumer to the producer in a reliable fashion. Administrators manage channels and thereby define paths of communication, as database administrators provide persistence layers by populating a database schema. [Hoh04]

[10] `https://github.com/aasmith/feed-normalizer` (last checked 26.03.2018),
`https://github.com/feedjira/feedjira` (last checked 26.03.2018),
`https://github.com/feedparser/feedparser` (last checked 26.03.2018),
`https://github.com/cardmagic/simple-rss` (last checked 26.03.2018),
`https://github.com/swanson/stringer` (last checked 26.03.2018)

[11] `https://github.com/seangeo/ratom` (last checked 26.03.2018)

[12] `https://rometools.github.io/rome/` (last checked 26.03.2018)

[13] `https://abdera.apache.org/` (last checked 26.03.2018)

As mentioned before, messaging enables asynchronous communication. There is no need for the producer and the consumer to work at the same time. When the consumer is not available, messages are stored and subsequently delivered when the client is available again. The message broker stores the messages and repeatedly tries to send the message to the client. To ensure the delivery of a message, clients send notifications when they have received the message. Messaging can also ensure *guaranteed delivery* by persisting messages on the message broker and ensure delivery even if the message broker restarts or fails. [Hoh04]

3.3.1 Message Channel Types

There are basically two approaches to send messages over a channel [Hoh04]:

- **Point-to-Point Channel**: Ensures that each message is sent to a single consumer.
- **Publish-Subscribe Channel**: Broadcasts a message to multiple "*interested*" consumers.

Point-to-point channels implement a *Remote Procedure Call* (*RPC*) [Tan07] like type of communication. Messages sent through channels are consumed by only one consumer. If the channel only has one consumer, the channel ensures that the message is delivered to this consumer. A channel can also have multiple consumers (e.g. for better scaling). In this case, the message broker ensures that the message is delivered only to one of the consumers and consumers do not have to manage coordination themselves. Figure 3.5 illustrates communication with point-to-point channels and multiple consumers. [Hoh04]

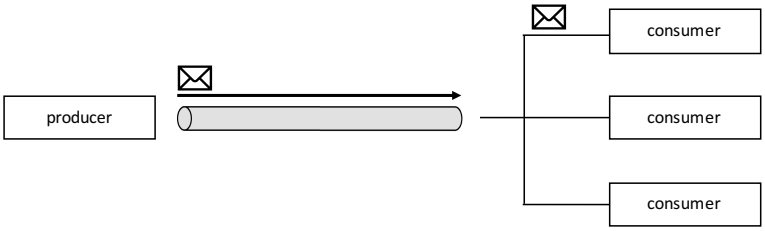


Figure 3.5: General structure of communication via a point-to-point channel

Publish-subscribe channels implement the *Publisher-Subscriber Pattern* [Bus96], a well-established pattern for implementing broadcast. This pattern allows subjects (that is, the source of an event) to decouple from its observers (who are interested in a subject). Interested observers can independently subscribe to a subject and are notified on events. With messaging, events can be packaged as messages and the message broker ensures delivery to the interested subscribers. To do so, the publish-subscribe channel has one input channel that is split into multiple output channels. When a message is sent through channel, the publish-subscribe channel delivers copies of the message to all output channels. Each output channel has a single subscriber that gets the message once. As only interested consumers are addressed, publish-subscribe channels are an application

level[14] implementation of multicast [Eug03]. Figure 3.6 illustrates communication with publish-subscribe channels. [Hoh04]

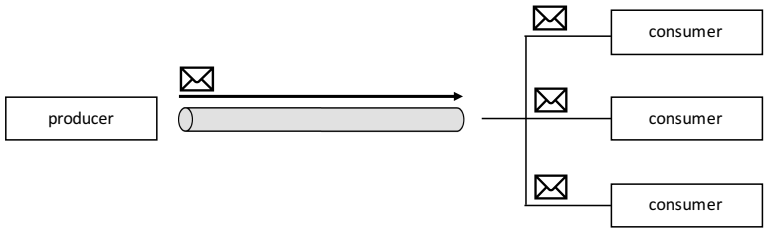


Figure 3.6: General structure of communication via a publish-subscribe channel

In modern message brokers, messages can be annotated by *topics*[15]. A topic is a categorization of a message. Consumers subscribe not only to a channel but also to an individual topic. The producer assigns each message with a topic and messages are delivered to consumers who subscribed to the channel and the respective topic. Topic subscriptions can also implement *wildcard* characters. Topics and wildcards are described based on RabbitMQ in more detail in *Subsection 3.3.3*. [Hoh04]

The next subsections describe messaging standards and the AMQP, respectively RabbitMQ, a message broker that uses this standard, in detail.

3.3.2 Standards

There are several standards for message brokers and the exchange of messages available. The most popular standards are:

- **Advanced Message Queuing Protocol** (*AMQP*)[16]: AMQP is an open networking standard. It is used by many message brokers (e.g. RabbitMQ, ActiveMQ and the Microsoft Azure Service Bus) and in many critical systems in Telecommunications, Defense, Manufacturing, Internet and Cloud Computing (e.g. Deutsche Börse and NASA)[17].

- **Message Queuing Telemetry Protocol** (*MQTT*)[18]: MQTT is a lightweight and open networking standard which is used by many message brokers (e.g. RabbitMQ and ActiveMQ)[19]. It was designed for constrained devices and low-bandwidth,

[14] Publish-subscribe is not necessarily implemented with network multicasting (like *IP multicast*). Depending on the implementation, messages may be sent to each client independently. Therefore, it implements multicast only on the application layer.

[15] `https://www.rabbitmq.com/tutorials/tutorial-five-java.html` (last checked 26.03.2018)

[16] `https://www.amqp.org/` (last checked 26.03.2018)

[17] `https://www.amqp.org/about/examples` (last checked 26.03.2018)

[18] `https://mqtt.org/` (last checked 26.03.2018)

[19] `https://github.com/mqtt/mqtt.github.io/wiki/servers` (last checked 26.03.2018)

high-latency or unreliable networks and often used for machine-to-machine communication[20]. Based on the resources published by MQTT itself [21], it is mostly used in the *Internet of Things* (*IoT*) and for integration of low level devices.

- **Simple Text Oriented Messaging Protocol** (*STOMP*)[22]: STOMP is a simple protocol, similar to HTTP. Because of its simplicity, clients can also be implemented with light-weight tools like *telnet* and *web-sockets*. It is applied by many message brokers (e.g. RabbitMQ, ActiveMQ and HornetMQ)[23]. There is no information on projects that use STOMP published by the project itself.

There are also other messaging-like technologies available, which support parts of the features of messaging. An example is *Redis Pub/Sub*[24], which also implements the Publisher-Subscriber Pattern but does not guarantee delivery.

For Java EE exists the *Java Message Service* (*JMS*) standard[25]. This standard generalizes the access from a Java application to a message broker. Where standards like AMQP and MQTT allows different message brokers to interact, JMS only provides a general Java API. The main benefit of a general API is the opportunity to change the message broker behind the API without changing the code.

Table 3.3 gives and overview over the standards and their support of the previously described features of messaging:

	AMQP	**MQTT**	**STOMP**	**Redis Pub/Sub**
Point-to-Point Channels	X		X	
Publish-Subscribe Channels	X	X	X	X
Topics	X	X	X	X
Wildcards	X	X	X	X
Guranteed Delivery	X	X	X	

Table 3.3: Supported features of messaging standards

Based on the available information on areas of application and projects respectively organizations that use individual standards, it is assumed that AMQP suits best for the use in business IT systems. Therefore, it is used in the thesis for implementing replication strategies with web feeds

[20] `https://mqtt.org/faq` (last checked 26.03.2018)
[21] `https://mqtt.org/projects` (last checked 26.03.2018),
`https://github.com/mqtt/mqtt.github.io/wiki/things` (last checked 26.03.2018)
[22] `https://stomp.github.io/` (last checked 26.03.2018)
[23] `https://stomp.github.io/implementations.html` (last checked 26.03.2018)
[24] `https://redis.io/topics/pubsub` (last checked 26.03.2018)
[25] `https://www.jcp.org/en/jsr/detail?id=914` (last checked 26.03.2018)

3.3.3 AMQP and RabbitMQ

The use of AMQP will be described in detail based on RabbitMQ, a popular message broker that used AMQP as the underlying network protocol for message delivery.

RabbitMQ uses *exchanges*, *bindings* and *queues* to implement channels. An *exchange* represents the input of a channel, where messages are sent to. A *queue* represents the output of a channel. Queues can also be *persistent* to provide guaranteed delivery. The integration of exchanges and queues is implemented by a *binding*.

Messages in RabbitMQ are equipped with a *routing key* - representing the topic (sent in the header of the message). A binding can be assigned with a binding key. Messages equipped with a particular *binding key* are delivery to queues that are bonded with the matching binding key. Binding keys can also be implemented as wildcards. Figure 3.7 illustrates this structure.

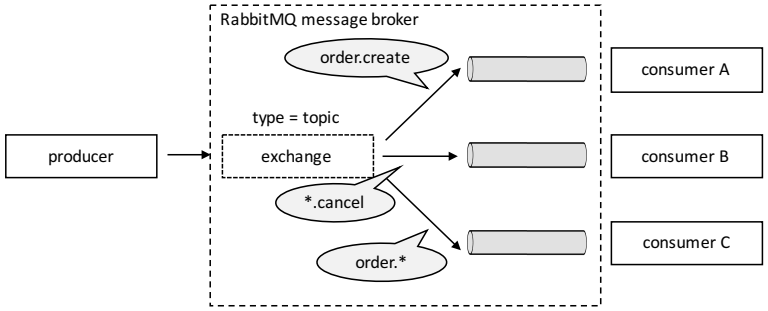


Figure 3.7: Example of messaging with RabbitMQ

A message is sent directly from the producer to an exchange. The exchange must be assigned with the type `topic` to provide support for topic-based delivery (simple publish-subscribe and one-to-one channels are supported as well). Consumers are bound to a queue. Bindings (illustrated as arrows) connect queues and exchanges. Binding keys (illustrated as bubbles) manage the topic subscription for an individual binding.

To further explain the procedure, imagine the following scenarios based on Figure 3.7:

- When a message with the routing key `order.create` is sent from the producer, it is delivered to consumer A (who subscribed exactly this topic) and consumer B (who subscribed all topics regarding orders).
- When a message with the routing key `order.cancel` is send from the producer, it is delivered to consumer B (who subscribed all topics regarding cancellations) and consumer C (who subscribed all topics regarding orders).
- Messages with any other cancellation would only be send to consumer C (because he is the only one who is not solely subscribing topics regarding orders).

Advanced topics like error-handling and monitoring are not covered in this survey. A detailed overview over system designs in these areas can be found in [Hoh04].

Several libraries exist for communication with a RabbitMQ message broker and use messaging in an application. An official Java client is available at the RabbitMQ homepage[26]. This client can be used for AMQP based messaging in general. AMQP can be used in Ruby with *bunny*[27], a library that is recommended in the official RabbitMQ homepage. Libraries for other platforms are either available direct from RabbitMQ (e.g. for *Erlang* and *.Net*) or recommended there (e.g. for *Python*, *PHP*, *JavaScript* and *Go*).

In principle, these libraries can be used for consuming messages as well. An examination of exiting libraries showed that they block the current process while listening. Therefore, the process must be implemented independently from other processes, especially the user interaction. For example, a Java based implementation could start a separate thread that listens to the queue. Messaging in Java EE and JMS outsource the communication to the message broker to the Java EE container that takes care of the listening threads that are blocked. In Ruby on Rails it is necessary to start an independent job. A widely-used library to consume message queues in Ruby is *Sneakers*[28], which implements the process based on the standardized *Active Job* interface.

The last two chapters formed the theoretical foundation of this thesis. Based on these surveys, the next chapter composes concrete replication strategies to implement replication in an IT system.

[26] `https://www.rabbitmq.com/java-client.html` (last checked 26.03.2018)
[27] `https://github.com/ruby-amqp/bunny` (last checked 26.03.2018)
[28] `https://github.com/jondot/sneakers` (last checked 26.03.2018)

CHAPTER 4

Replication Strategies

This chapter provides a definition of concrete replication strategies for the use in loosely coupled IT systems. Based on the definition of the aim of this work in *Section 1.3* and the findings in *Chapter 2*, requirements for the use of replication strategies in this context can be summarized as followed:

1. The main objective is to improve performance for client-side read operations.
2. Components are loosely coupled.
3. Client-side replicas are read-only. Write operations are performed only in the server-side data store. Therefore, consistency has to be ensured for client-side replicas only.
4. Strict consistency is not crucial. It is sufficient to provide eventual consistency.
5. To achieve loose coupling between components, replication is performed on the application level.

Even though implementations which fulfill these requirements exist and are already used in existing IT systems, there is no general catalogue available. Implementations are often used as parts of a more general architectural style (e.g. Event-Driven Architecture or Microservices) or specialized local optimizations of individual procedures (e.g. caching of HTTP requests). It is assumed that previous research focused particularly on the individual characteristics of technologies used and not on the aspects of replication in general. Furthermore, replication on the application layer is an applied topic, specific for a particular use case. As a result, a catalogue of replication strategies has to be composed in the course of this thesis.

The composition of replication strategies is described in *Subsection 4.1.* The following sections describe replication strategies and their concrete implementations in detail.

Replication strategies are examined individually for client-side access and server-side updates of data items. The examination bases on the following simplified use case with one server and one client (see also Figure 4.1):

- The server provides data items in a local data store. The data store is accessible via a REST interface. External components update the server-side data store on the server-side.
- External components request data items in the server-side data store through the client. The client provides an interface to do so. Based on the replication strategy, data is read from the server, a cache or a replica.
- Data items consist of two attributes: `data0` and `data1`. Updates only modify the attribute `data0`. Each data item is identified by a unique ID.

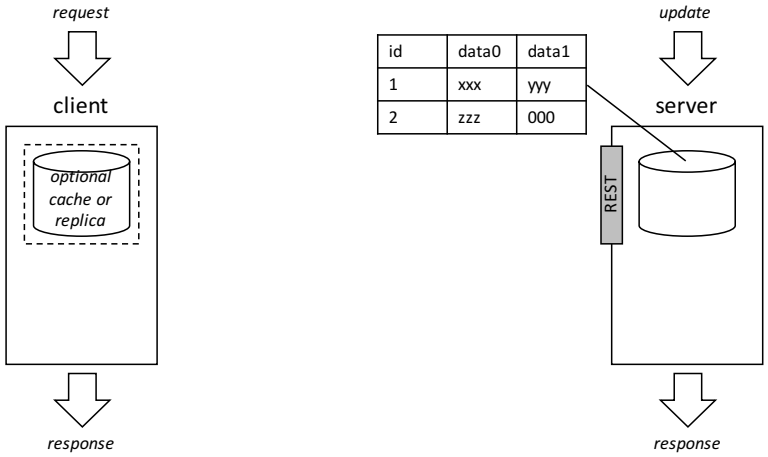


Figure 4.1: Structure of the use case

4.1 Selection of Replication Strategies

This section provides a selection of replication strategies with a bottom-up approach, based on the characteristics described in *Section 2.1*. The use of a top-down approach based on existing implementations was rejected. Documentation of existing IT systems hardly focuses on replication and does not cover the topic specifically. Therefore, it is hard to extract relevant information that can be used as a basis for an analysis. Furthermore, only few organizations publish insights to their IT systems, making it even harder to gather a representative number of high-quality sources.

The following subchapters are based on the individual characteristics discussed in *Section 2.1* and examine which options can be used for the requirements described below. Each chapter builds on the results of the previous chapter. *Subsection 4.1.4* presents concrete replication strategies as composed combinations of fundmental options.

4.1.1 Change Distribution

Change distribution defines which information is distributed between the server and the client (see *Subsection 2.1.3*). To meet the previously defined requirements, changes can be distributed by *invalidation* notifications, *full state* and *state difference*.

Distribution of changes by sending *operations* leads to tight coupling and therefore does not meet the requirements. Clients must be capable of performing operations on their data autonomously. Therefore, operations must be implemented on the server-side and on the client-side. Changes of operations must be coordinated between server and client which leads to tight coupling in development and especially deployment.

4.1.2 Replica Reaction

Replica reaction defines how server-side changes are propagated to the replicas (see *Subsection 2.1.4*). Basically, all methods are appropriate but not all combinations. Table 4.1 shows appropriate combinations of replica reaction methods and change distribution methods determined below. Marked combinations are considered below.

	Invalidation	**Full State**	**State Difference**
server-driven	X	X	X
polling on each request	X		
periodic polling	X	X	X

Table 4.1: Appropriate combinations of change distribution and replica reaction

The other combinations are not appropriate because:

- The combination of polling on each request and full state leads to the transmission of an entire data item on each request. This is equivalent to no replication.
- Polling the state difference on each request is as an optimization of the proposed implementation for Cache Validation. It is discussion at the end of *Subsection 4.3.1*.

4.1.3 Replica and Cache

It can be distinguished between full replicas that hold all server-side data and caches that hold only parts of the data (see *Subsection 2.1.2*). An examination of all combinations showed full replicas are appropriate for propagation with full state and state differences, caches are appropriate for invalidation notifications. Other combinations are not considered because:

- Invalidation of caches with notifications leads to a state where the full replica does not contain all data items. This contradicts the definition in *Subsection 2.1.2*.
- Propagation of full state or state differences can be used with caching, but is not efficient. Caches only hold parts of the data. Nevertheless, all updates are sent over the network. This leads to potentially unnecessary network traffic.

Tables 4.2 and 4.3 show which combinations determined in *Subsection 4.1.2* are suitable for caches and full replicas.

	Invalidation	**Full State**	**State Difference**
server-driven	X		
polling on each request	X		
periodic polling	X		

Table 4.2: Caches: appropriate combinations of change distribution and replica reaction

	Invalidation	**Full State**	**State Difference**
server-driven		X	X
polling on each request			
periodic polling		X	X

Table 4.3: Full replica: appropriate combinations of change distribution and replica reaction

4.1.4 Results

Based on the analysis in the last chapters, seven replication strategies are chosen for the evaluation. Table 4.4 shows these replication strategies and their characteristics.

	Name	**Change Distribution**	**Replica Reaction**	**PULL / PUSH**	**Cache / Full Replica**
RS1	*Cache Validation*	invalidation	polling on each request	PULL	cache
RS2	*Event-Driven Cache Invalidation*	invalidation	server-driven	PUSH	cache
RS3	*Event-Driven Replication*	full state	server-driven	PUSH	full replica
RS4	*Event-Driven Delta Replication*	state difference	server-driven	PUSH	full replica
RS5	*Poll-Based Cache Invalidation*	invalidation	periodic polling	PULL	cache
RS6	*Poll-Based Replication*	full state	periodic polling	PULL	full replica
RS7	*Poll-Based Delta Replication*	state difference	periodic polling	PULL	full replica

Table 4.4: Replication strategies and their characteristics

In addition, synchronous communication without replication will also be examined (RS0). This replication strategy is described in the next chapter.

4.2 Synchronous Requests

In a system without caching and replication, the client has to request data from the server whenever it is needed. Each access on a data item requires a synchronous request to the server respectively its data store. This approach is often used in IT systems because it offers integration of a client and a server without any optimization like caches and replication. To enable this type of integration, the server has to provide an interface to read data from its data store. Usually the server will also provide interfaces to modify data for typical CRUD (*Create*, *Read*, *Update*, *Delete*) operations.

4.2.1 Solution Design

Synchronous communication in modern distributed IT systems is mostly implemented with RESTful webservices. For the described scenario, a resource `/data` is used. The CRUD functionality can be implemented as described in Section 3.1. The resource `/data` has to provide at least the methods described in Table 4.5.

URI	**HTTP method**	**Description**
`/data/{id}`	GET	Returns the data item with the id `{id}`
`/data/{id}`	PUT	Replaces the data item with the id `{id}` with the representation in the request body
`/data/{id}`	DELETE	Deletes the data item with the id `{id}`
`/data`	POST	Inserts a new data item in the data store

Table 4.5: HTTP methods needed for synchronous requests

Communication between the client and the server uses the GET method. Other methods are used to enable modifications of data only.

4.2.2 RS0 – No Replication

RS0 - No Replication implements synchronous requests with RESTful webservices, without a cache or replica. It is used as the baseline for further observations. Figure 4.2 illustrates the structure of the implementation.

Figure 4.2: Structure of RS0

The implementation consists of the following components:

- A server-side REST interface to access data items. For communication with the client, only read access to the resource (with HTTP GET) must be provided.
- A client-side HTTP client library to access the REST resource.

Request

When a data item is requested on the client-side, the client passes the request to the server. The server reads the data item from its local data store and returns it to the client. Figure 4.3 illustrates the procedure.

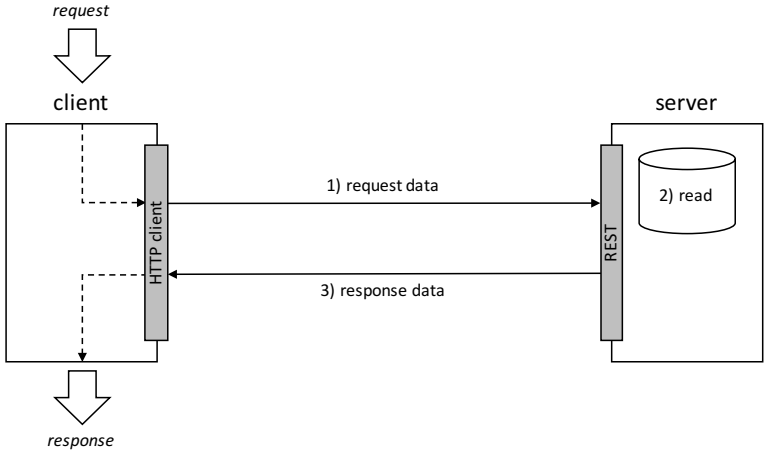


Figure 4.3: Procedure of a request with RS0

Requesting a data item on the client-side leads to the following operations:

1. The client receives a request and fetches the requested data item from the server. A HTTP GET on the resource `/data/{id}` is performed.
2. The server reads the data item from its local data store.
3. The server returns the data item to the client.

There are no additional operations necessary on the client-side.

When the server-side data store does not contain the requested data item, the server responds with an empty message body and the HTTP status code `404 NOT FOUND`. The client has to respond to his request correspondingly.

Update

Data items are updated in the server-side data store. No additional operations are necessary. Figure 4.4 illustrates the procedure.

Figure 4.4: Procedure of an update with RS0

4.3 Cache Validation

Cache Validation introduces a client-side cache which contains data items which were requested before. Data items are validated each time they are read. The validation is performed by the comparison of the data item (respectively its version) in the client-side cache and in the server-side data store. Cache Validation is widely used in the internet because it decreases the required bandwidth in comparison to the use of no replication and no caching.

4.3.1 Solution Design

RESTful webservices, respectively HTTP provides functionality to implement Cache Validation with *conditional GET* and the HTTP `Modified` or HTTP `ETag` header. Existing RESTful webservices can easily be extended to support these technologies. Implementation on the client-side is more expensive because the client needs an additional cache to store data items.

Conditional GET is an extension of the HTTP GET method on the `/data` resource described in *Subsection 4.2.1*. Other methods of the `/data` resource (PUT, DELETE, POST) need no modification to implement this replication strategy.

In practice, the HTTP `ETag` header is preferred over the HTTP `Modified` header, because it allows arbitrary validation strings where the HTTP `Modified` header only allows the date of the last modification [Web10]. It is even possible to use the date of the last modification with the HTTP `ETag` header, making it a more general solution of the HTTP `Modified` header.

The cache can be implemented by any persistence technology. Specialized implementations offer an advantage when concurrent requests on the same data item occur: a general solution would validate the data on each request, leading to multiple similar requests to

the server. A specialized implementation can condense the requests, send only a single request to the server and stress the network respectively the server less. Technologies to implement caches are not covered by this thesis (see delimitations in *Subsection 1.3.2*).

Based on the discussion in *Section 4.1*, Cache Validation represents invalidation of data on each request. Methods like conditional GET and ETags optimize the process so that there is no need for a separate request that fetches the new version of a data item. The process could be optimized further by sending only the state difference of the current client-side version and the version on the server-side. This optimization is not used in practice respectively not implemented in the widely used HTTP standard and therefore not considered in this thesis.

4.3.2 RS1 – Cache Validation

RS1 - Cache Validation implements synchronous requests with validation by ETags and a client-side cache. It is an extension of RS0. Figure 4.5 illustrates the structure of the implementation.

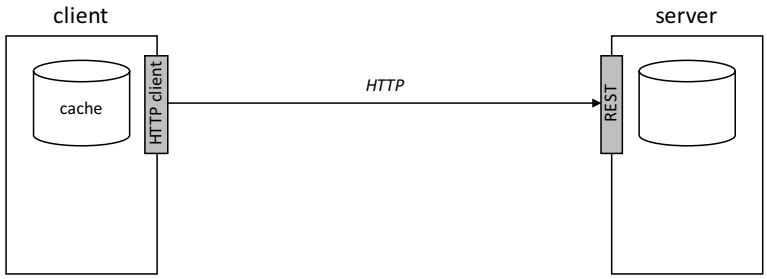


Figure 4.5: Structure of RS1

The implementation consists of the following components:

- A client-side cache
- A server-side REST interface to access data items. For the purpose of replication, only read access to the resource (with HTTP GET) must be provided.
- A client-side HTTP client library to access the REST resource.

Request

The processing of client-side requests depends on 1) the existence of the data item in the cache and when the data item exists in the cache 2) the validity of the data item in the cache and 3) the existence of the data item in the server-side data store. This leads to five possible scenarios:

1. The cache does not contain the data item.
2. The cache contains the data item but versions of the data item in the cache and in the server-side data store differ.

3. The cache contains the data item and it has the same version as in the server-side data store.
4. The cache contains the data item but the server does not. This practically means that it was deleted from the server-side data store.
5. Neither the cache nor the-server-side data store contain the data item.

These scenarios will be discussed in more detail in the next paragraphs.

Scenario 1: When the cache does not contain a data item, it has to be loaded from the server. No additional headers must be sent. Figure 4.6 illustrates the procedure.

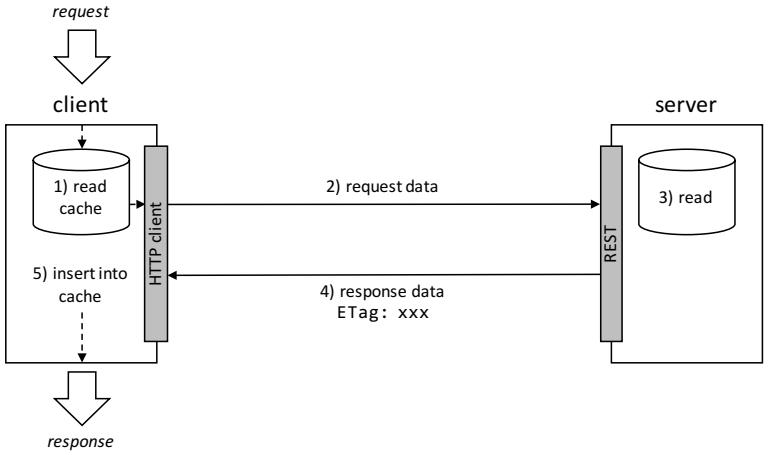


Figure 4.6: Procedure of a request with RS1 in Scenario 1

In this scenario, requesting a data item on the client-side leads to the following operations:

1. The client searches the cache for the data item. The cache does not contain the data item.
2. The client requests the data item from the server. A HTTP GET on the resource `/data/{id}` is performed. No additional HTTP headers are added.
3. The server reads the data item from its local data store.
4. The server returns the data item to the client. A `ETag` header with the ETag of the data item is added.
5. The client saves the data item **and the ETag** in its cache.

Scenario 2: When a requested data item exists in the client-side cache, it is validated on the server-side. In this scenario, it is assumed that the data item changed since it was read for the first time and therefore the data item differs in the client-side cache and the server-side data store. Therefore, the server returns the new version of the data item. Figure 4.7 illustrates the procedure.

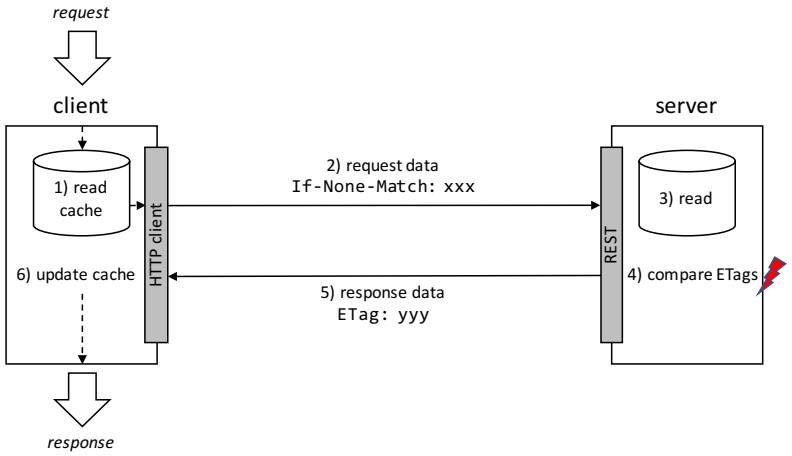


Figure 4.7: Procedure of a request with RS1 in Scenario 2

In this scenario, requesting a data item on the client-side leads to the following operations:

1. The client searches the cache for the data item. The cache contains the data item.
2. The client validates the data item at the server. A HTTP GET on the resource `/data/{id}` is performed, including the `If-None-Match` header with the ETag of the cached data item.
3. The server reads the data item from its local data store.
4. The server compares the ETag in the request with the ETag of his data item. This may require the calculation of the ETag based on the contents of the data item in the server-side data store.
5. Because ETags differ, the server returns the new version of the data item to the client including an `ETag` header with the ETag of the new version.
6. The client updates the data item **and the ETag** in its cache.

Scenario 3: When a requested data item exists in the client-side cache, it is validated on the server-side. In this scenario, it is assumed that the data item in the server-side data store is equal to the data item in the cache. Therefore, there is no need to transmit the data item from the server to the client. Figure 4.8 illustrates the procedure.

In this scenario, requesting a data item on the client-side leads to the following operations:

1. The client searches the cache for the data item. The cache contains the data item.
2. The client validates the data item at the server. A HTTP GET on the resource `/data/{id}` is performed, including the `If-None-Match` header with the ETag of the cached data item.
3. The server reads the data item from its local data store.

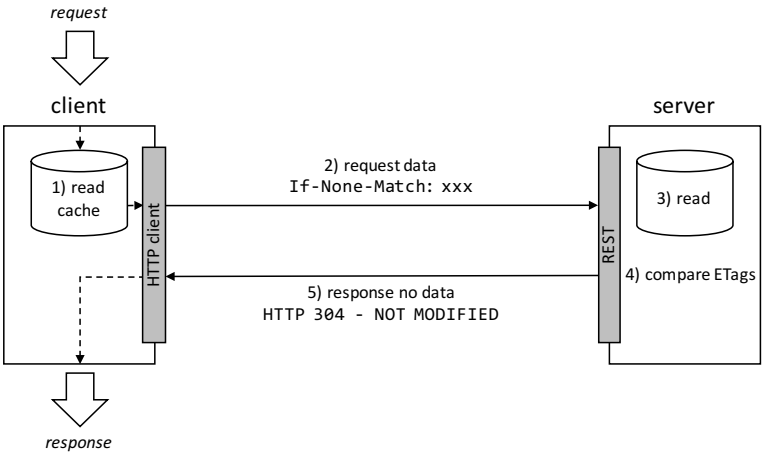


Figure 4.8: Procedure of a request with RS1 in Scenario 3

4. The server compares the ETag in the request with the ETag of his data item. This may require the calculation of the ETag based on the contents of the data item in the server-side data store.
5. Because the ETags are the same, the server returns an empty message body and sets the HTTP status Code to `304 NOT MODIFIED`.

No updates in the cache are necessary.

Scenario 4: When a requested data item exists in the client-side cache, it is validated on the server-side. In this scenario, it is assumed that the data item was deleted in the server-side data store. Figure 4.9 illustrates the procedure.

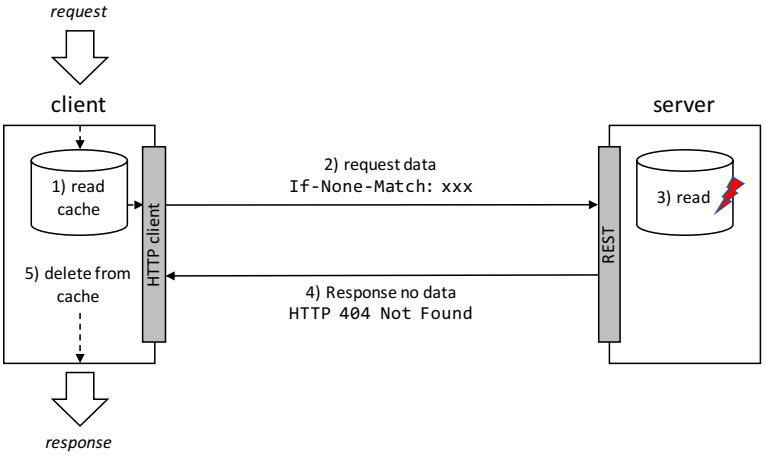


Figure 4.9: Procedure of a request with RS1 in Scenario 4

In this scenario, requesting a data item on the client-side leads to the following operations:

1. The client searches the cache for the data item. The cache contains the data item.
2. The client validates the data item at the server. A HTTP GET on the resource `/data/{id}` is performed, including the `If-None-Match` header with the ETag of the cached data item.
3. The server searches its local data store for the data item. The data store does not contain the data item.
4. The server returns an empty message body and sets the HTTP status code of the response to `404 NOT FOUND`.
5. The client deletes the data item from its cache and returns a corresponding response.

Scenario 5: In the case that the client requests a data item which does neither exist in the client-side cache nor in the server-side data store, the process is similar to Scenario 4. The client requests the data item from the server, gets a response with `404 NOT FOUND` and returns a corresponding response. The client accesses the cache once but no other access of the cache is performed.

Update

The process on updates is the same as in Subsection 4.2.2.

4.4 Event-Driven Replication Strategies

Event-Driven replication strategies comprise replication strategies where the server actively notifies the client (by push) when changes in the server-side data store take place. Based on the type of change distribution (see *Subsection 4.1.4*), these replication strategies can be distinguished into:

- **Event-Driven Cache Invalidation** (*RS2*): The server sends notifications to the client to **invalidate** a client-side cache.
- **Event-Driven Replication** (*RS3*): The server sends the **full state** of a new data item version to the client. The client updates its full replica to the new version independently.
- **Event-Driven Delta Replication** (*RS4*): The server sends the **state difference** between the old and the new version of a data item to the client. The client updates its full replica to the new version independently.

These replication strategies can be implemented by an event-based approach, leading to a system similar to Event-Driven Architectures (see *Subsection 2.3.2*). In an Event-Driven Architecture, replication strategies RS3 and RS4 would implement *Event-Carried State Transfer* (see *Subsection 2.3.2*).

4.4.1 Solution Design

The Event-Driven replication strategies presented in this work are implemented with publish-subscribe based messaging. Publish-subscribe based messaging was chosen over other technologies because it is mainly used for push-based multicast communication in the highly related area of Event-Driven Systems. Messaging standards and message brokers have to meet the following requirements to implement these replication strategies:

- *Publish-Subscribe*: The publish-subscribe pattern implements multicast for messaging (see *Subsection 3.3.1*).
- *Guaranteed delivery*: Consistency of a cache or replica can only be achieved when every change in the server-side data store is propagated to the clients. Guaranteed delivery ensures that every message is delivered to the clients (see *Subsection 3.3*).

This thesis proposes RabbitMQ and AMQP because of their wide and proven usage (see *Subsection 3.3.2*). The presented implementation uses topic-based publish-subscribe channels (see *Subsection 3.3.1*). Figure 4.10 illustrates the general structure.

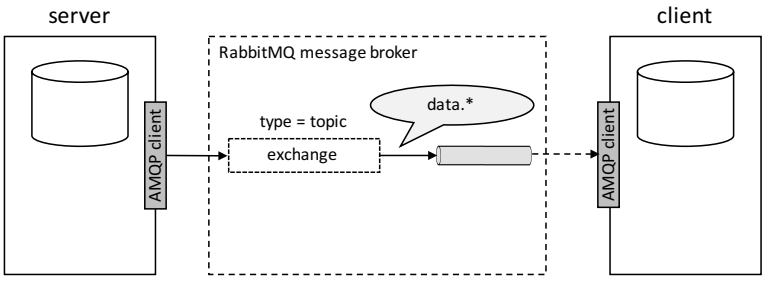


Figure 4.10: Event-Driven replication with RabbitMQ

RabbitMQ provides a topic-based publish-subscribe exchange. The server sends messages with an AMQP client to the RabbitMQ message broker whenever data items in the data store are changed. A client can subscribe to the queue and get messages and notifications on changes in the server-side data store. The queue is implemented as a persistent queue to guarantee message delivery.

Routing keys are used to provide support for different replication events over a single queue. Table 4.6 gives describes routing keys that can be used for replication.

routing key	**event**
`data.insert`	data item was created
`data.modification`	data item was modified
`data.deletion`	data item was deleted

Table 4.6: Routing keys for Event-Driven replication

The client in Figure 4.10 subscribed all events for changes of the data entity type. For the use case of replication, three routing keys and the binding key `data.*` are sufficient.

Routing keys and binding keys can be extended to implement complex Event-Driven Systems. Because the focus of this work is on replication, these options are not considered.

4.4.2 RS2 – Event-Driven Cache Invalidation

RS2 - Event-Driven Cache Invalidation uses a client-side cache that holds data items that were requested before. Data items in the cache are invalidated (and thus deleted from the cache) when data items on the server-side data store are updated or deleted. Newly inserted data items are loaded on the first access. Cache Invalidation is implemented asynchronously by push notifications over a publish-subscribe channel as described in Subsection 4.4.1. Figure 4.11 illustrates the structure of the implementation.

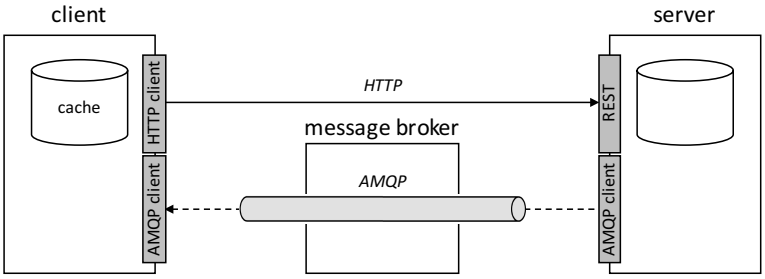


Figure 4.11: Structure of RS2

The implementation consists of the following components:

- A client-side cache
- A server-side REST interface to access data items. For the purpose of replication, only read access to the resource (with HTTP GET) must be provided.
- A client-side HTTP client library to access the REST resource.
- An AMQP message broker which provides a persisted publish-subscribe channel.
- Client Libraries for AMQP to access the message broker on client and server.

Request

The process of client-side requests depends on the existence of the data item in the cache. Therefore, there are two possible scenarios:

1. The cache does not contain the data item and has to request it from the server.
2. The cache contains the data item and it can be loaded directly from the cache.

These scenarios will be discussed in more detail in the next paragraphs. Scenarios where the server-side-data store does not contain the data item follow the same process as *Scenario 5* in *Subsection 4.3.2*.

Scenario 1: When the data item does not exist in the cache, it has to be loaded from the server via the REST interface. Figure 4.12 illustrates the procedure.

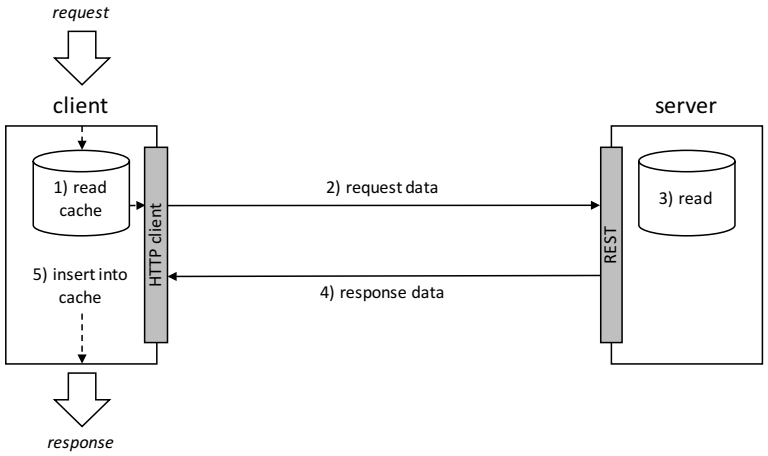


Figure 4.12: Procedure of a request with RS2 in Scenario 1

In this scenario, requesting a data item on the client-side leads to the following operations:

1. The client searches the cache for the data item. The cache does not contain the data item.
2. The client requests the data item from the server. A HTTP GET on the resource `/data/{id}` is performed. No additional HTTP headers are added.
3. The server reads the data item from its local data store.
4. The server returns the data item to the client.
5. The client saves the data item in its cache.

Scenario 2: When the data item exists in the cache it is directly loaded. There are no additional operations necessary. Figure 4.13 illustrates the procedure.

Figure 4.13: Procedure of a request with RS2 in Scenario 2

Update

After an update is executed in the server-side data store, the server sends a notification to the clients to invalidate the data item in their caches. Notifications are sent on update and delete operations. There is no need to notify a client on an insert, because the client-side cache does not hold the data item and therefore nothing has to be invalidated. Insert operations are only performed in the server-side data store without any other operations needed. Routing keys can be used but are not crucial: data items are removed from the cache on updates as they are on deletions. Figure 4.14 illustrates the procedure of an update of a data item.

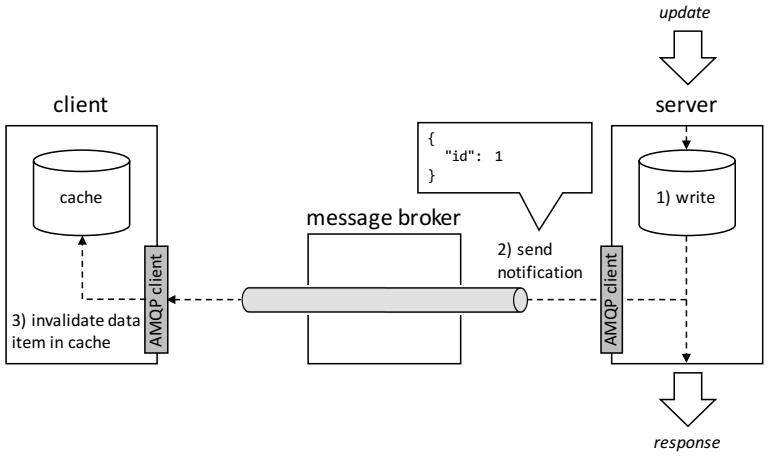


Figure 4.14: Procedure of an update with RS2

Updating a data item on the server-side leads to the following operations:

1. The server updates the data item in its local data store.
2. The server sends a notification to the publish-subscribe channel on the message broker with the topic `data.modification` and the ID of the modified data item. The message broker takes care of the delivery and the server finishes the response.
3. The client asynchronously receives the messages and invalidates the data item in its cache. To do so, the client removes the data item from the cache. When the cache does not contain the data item, no operations are performed.

4.4.3 RS3 – Event-Driven Replication

RS3 - Event-Driven Replication uses a client-side replica that holds all data items of the server-side data store. Changes on the server-side are propagated asynchronously by push notifications over a topic-based publish-subscribe channel. Notifications include the whole content of the data item. The client-side replica is initially loaded by a manual data import. Figure 4.15 illustrates the structure of the implementation.

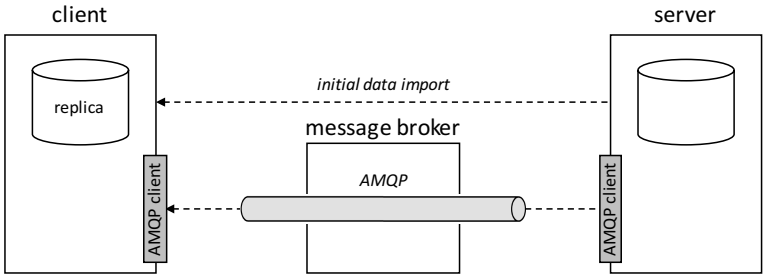


Figure 4.15: Structure of RS3

The implementation consists of the following components:

- A client-side replica
- An AMQP message broker which provides a persistent publish-subscribe channel.
- Client Libraries for AMQP to access the message broker on client and server.

RS3 requires an initial, one-time data import. Replication in RS3 only notifies clients about changes. Notification does not start until a client is put into operation and therefore changes before this point are not visible to the client. To establish a full replica of the server-side data store, the client-side replica needs an initial set-up. It is assumed that the set-up is done as a manual data import with methods on the persistence layer (like export / import of a database dump). Methods to perform the data import are not part of this thesis (see *Delimitations* in *Subsection 1.3.2*).

Request

The client-side full replica contains all data items. When a data item is requested on the client-side, it is directly read from the replica. Figure 4.16 illustrates the procedure.

Figure 4.16: Procedure of a request with RS3

Update

After an update was executed in the server-side data store, the server notifies clients by sending a message to the publish-subscribe channel. The client independently updates its full replica, based on the message. The routing key of the message provides the information if a data item has to be inserted, updated or deleted. All messages include the ID of the data item. Insert and update operations also include the full state of the data item (in the assumed scenario `data0` and `data1`). Delete operations do not contain any other information than the ID. Figure 4.17 illustrates the procedure of the update of a data item.

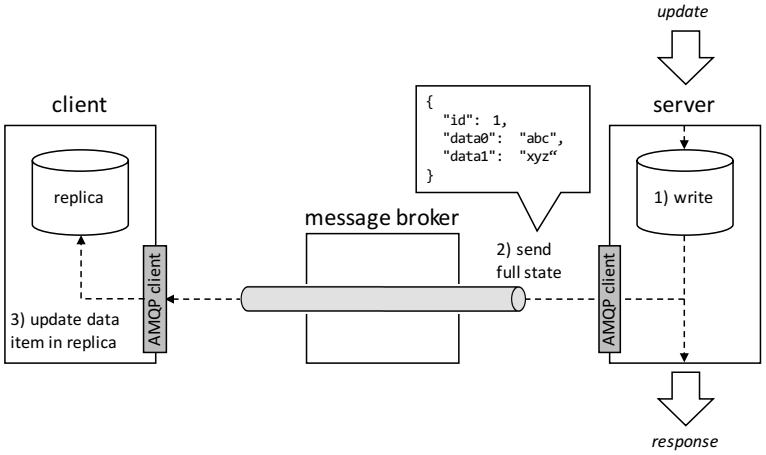


Figure 4.17: Procedure of an update with RS3

Updating a data item on the server-side leads to the following operations:

1. The server updates the data item in its local data store.
2. The server sends a notification to the publish-subscribe channel on the message broker with the topic `data.modification`. The message contains the ID and the full state of the new version of the data item, encoded in JSON. The message broker takes care of the delivery and the server finishes the response.
3. The client asynchronously receives the message and updates the data item in the replica with the contents of the message.

4.4.4 RS4 – Event-Driven Delta Replication

RS4 - Event-Driven Delta Replication uses a client-side replica that holds all data items of the server-side data store. Changes on the server-side are propagated asynchronously by push notifications over a topic-based publish-subscribe channel. Notifications include only the state difference (e.g. delta of the old version to the new version). The client-side replica is initially loaded by a manual data import operation (see *Subsection 4.4.3*).

The structure is the same as for RS3, described in *Subsection 4.4.3*. RS4 is an optimization of RS3 to reduce bandwidth consumption. Because only state differences and not the

whole state of a data item have to be transmitted, message size decreases and so does bandwidth consumption.

Request

The client-side full replica contains all data items. When a data item is requested on the client-side, it is read from the replica. The process is the same as in *Subsection 4.4.3*.

Update

After an update was executed in the server-side data store, the server notifies the clients by sending a message to the publish-subscribe channel. The client independently updates its full replica, based on the message. The routing key of the message provides the information if a data item has to be inserted, updated or deleted. All messages include the ID of the data item. Insert operations also include the full state of the data item (as in RS3). Update operations only include the changed data (in the assumed scenario `data0`). Delete operations do not contain any other information than the ID. Figure 4.18 illustrates the procedure of the update of a data item.

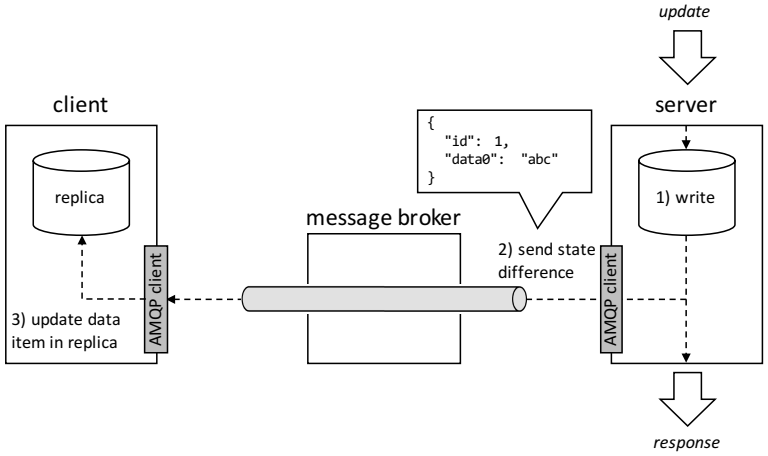


Figure 4.18: Procedure of an update with RS4

Updating a data item on the server-side leads to the following operations:

1. The server updates the data item in its local data store.
2. The server sends a notification to the publish-subscribe channel on the message broker with the topic `data.modification`. The message contains the ID and the delta of the old version to the new version of the data item (in the assumed scenario `data0`), encoded in JSON. The message broker takes care of the delivery and the server finishes the response.
3. The client asynchronously receives the messages and updates the data item in the replica according to the delta in the message.

4.5 Poll-Based Replication Stratgies

Poll-Based replication strategies comprise replication strategies where the client actively requests the server to be notified about changes in the data store (by pull). Responsibility for consistency of the cache is therefore solely on the client-side.

A simple implementation of polling is the extraction of the whole server-side data store and the import of all data on the client-side. An example of this approach is the *File Transfer* integration pattern [Hoh04], in which the server provides dumps of its data store on the file systems. Clients read the dump and import it into their own data store. This approach can also be implemented with direct communication (like RESTful webservices), in which the client would request all data directly from the server. [Hoh04] stated that it requires a lot of effort to export and import data. Therefore, these tasks are usually performed infrequently (e.g. during the night or on a weekly basis) which leads to long times of inconsistency. Better consistency can be achieved with an Event-Driven approach.

An Event-Driven Architecture can also be used for Poll-Based replication. To do so, the server files each change as an event in a repository (e.g. as a file on a file system, a row in a database or as an entry in a web feed). This repository must be accessible to clients. Clients regularly read the repository for new events and process them by actualizing their replicas or caches. Replication therefore needs less effort: The server has to file a single event for each change and the client has to import only modifications and not all data items. As a result, replication can be performed more often which leads to better consistency.

Based on the type of change distribution (see *Subsection 4.1.4*), Poll-Based replication strategies can be distinguished into:

1. **Poll-Based Cache Invalidation** (*RS5*): The server files notifications on every change in an accessible repository. The client regularly reads the repository and processes events to **invalidate** its cache.
2. **Poll-Based Replication** (*RS6*): The server files the **full state** of the new version of the data item in an accessible repository. The client regularly reads the repository and processes events to update its replica.
3. **Poll-Based Delta Replication** (*RS7*): The server files **state differences** between the old and the new version of a data item in an accessible repository. The client regularly reads the repository and processes events to update its replica.

With the Event-Driven approach as described above, replication strategies RS6 and RS7 implement Event-Carried State Transfer (see *Subsection 2.3.2*).

4.5.1 Solution Design

The implementation used in this thesis bases on the Event-Driven replication approach described in [Web10]. This approach uses a linked list of Atom web feeds.

The basic element is a *working feed* that includes all events between the present moment and a cutoff point in the past. *Archive feeds* contain all events that happened before the cutoff point. Every event is written to the working feed. The working feed is regularly archived and a new working feed is created. The archived working feed gets immutable at that point. In addition to these feeds, there is the "*feed of recent events*". It contains the same events as the working feed and is always up-to-date. When a working feed is archived, the recent feed shows the contents of the new working feed.

The solution in [Web10] groups feeds and events by their date of occurrence. The working feed contains today's events, the archive feeds contain events of individual days in the past. For the implementation in this thesis, a more general approach was chosen which is independent of the date of occurrence: Every feed has a defined length of n entries. When the working feed has reached the maximum size, it is archived.

Feeds are linked using *hypermedia* controls, concretely with the Atom `<link>` element. Its `rel` attribute defines the link relation and semantic context for the link. The relations `prev-archive` and `next-archive` are used to implement linking between web feeds.

Figure 4.19 shows the structure and linking of feeds. It includes only linking information and no other content or meta-information.

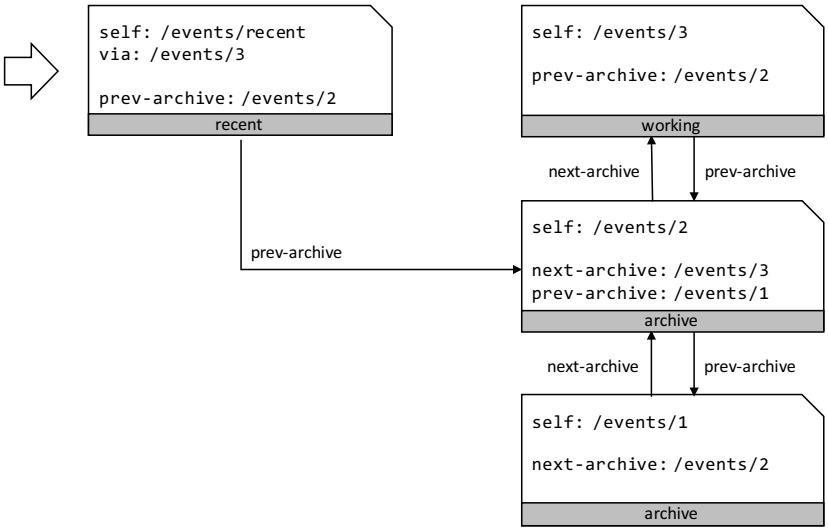


Figure 4.19: Structure of linked web feeds

Clients always access the recent feed. Feeds are generally ordered descending by the time an event occurred. The client reads the contents of the feed until it reaches an event that has already been processed before. Because every entry in a feed has a unique ID, comparison of events is easy to establish. When the client finds an event it has processed before, it stops reading and processes all events in a reverse order (the oldest first). When the client processed none of the events in a feed before, it has to read the previous archive feed and searches it for unprocessed events as well. The same procedure

applies to the archive feeds: if the client processed none of the events before, it has to read the previous archive feed. Events are processed at the very end. The oldest event of all feeds (including the recent feed and all read archive feeds) is processed first.

4.5.2 RS5 – Poll-Based Cache Invalidation

RS5 - Poll-Based Cache Invalidation uses a client-side cache to hold data items that were requested before. The server files an event in the web feed for each change of a data item. Clients read the web feed in intervals and invalidate data items based on the events in the web feed. The client-side cache is therefore updated asynchronously. Newly inserted data items are loaded on the first access on the client-side. Modified data items respectively their IDs are written to a web feed that is polled by clients. Figure 4.20 illustrates the structure of the implementation.

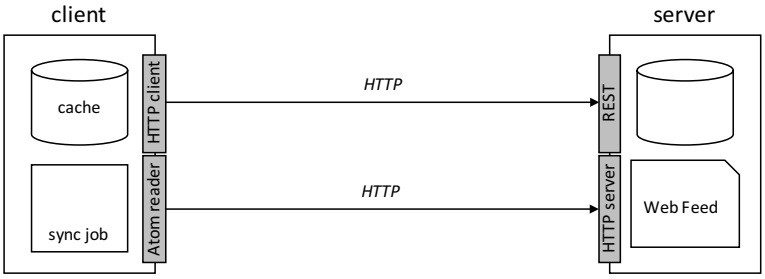


Figure 4.20: Structure of RS5

The implementation consists of the following components:

- A client-side cache
- A server-side REST interface to access data items. For the purpose of replication, only read access to the resource (with HTTP GET) must be provided.
- A client-side HTTP client library to access the REST resource.
- A server-side web feed, provided by a HTTP server.
- A client-side Atom reader to request the Atom feed over HTTP and parse it.
- A client-side synchronization job to update the cache. The job runs the synchronization process in intervals.

Request

The process is the same as in *Subsection 4.4.2*.

Update

After an update was executed in the server-side data store, the server adds a new entry to the web feed. Figure 4.21 illustrates the procedure of an update of a data item.

Figure 4.21: Procedure of an update with RS5

The web feed entry contains the ID of the data item that has been updated or deleted. The following listing shows a web feed containing a single event of an update operation in the server-side data store.

```
<?xml version="1.0" encoding="UTF-8"?>
<feed xmlns="http://www.w3.org/2005/Atom">
  <id>http://www.application.com/events</id>
  <title>events</title>
  <updated>2018-01-01T10:00:00Z</updated>
  <link rel="self" href="/events" />
  <entry>
    <id>events/1</id>
    <updated>2018-01-01T10:00:00Z</updated>
    <author>
      <name>user</name>
    </author>
    <link rel="self" href="/events/1" />
    <link rel="related" href="/data/10" />
    <category term="data" />
    <category term="update" />
    <content type="application/xml">
      <invalidation>
        <id>10</id>
      </invalidation>
    </content>
  </entry>
</feed>
```

4.5.3 RS6 – Poll-Based Replication

RS6 - Poll-Based Replication uses a client-side replica that holds all data items of the server-side data store. The server files the new state of data items (after the update) as events to the web feed. Clients read the web feed in intervals and update their replicas based on the events in the web feed. The client-side replica is therefore updated asynchronously. Figure 4.22 illustrates the structure of the implementation.

Figure 4.22: Structure of RS6

The implementation consists of the following components:

- A client-side replica
- A server-side web feed, provided by a HTTP server.
- A client-side Atom reader to request the Atom feed over HTTP and parse it.
- A client-side synchronization job to update the full replica. The job runs the synchronization process in intervals.

Provided that events are not deleted after time, Poll-Based Replication can be used without a manual initial data import. Each operation in the data store is filed as an event. The current state of the data store can therefore be merged as the results of all events. To establish the current state of the data store, the client needs to process all events from the start on.

Request

The client-side full replica contains all data items. When a data item is requested on the client-side, it is read from the replica. The process is the same as in *Subsection 4.4.3*.

Update

The process of an update in the server-side data store is illustrated in *Subsection 4.5.2*.

Each web feed entry includes the ID of the data item. Entries of insert and update operations also include the full state of the data item (in the assumed scenario `data0` and `data1`). Delete operations do not contain any other information than the ID. The following listing shows a web feed containing a single event of an update operation in the server-side data store.

```
<?xml version="1.0" encoding="UTF-8"?>
<feed xmlns="http://www.w3.org/2005/Atom">
  <id>http://www.application.com/events</id>
  <title>events</title>
  <updated>2018-01-01T10:00:00Z</updated>
  <link rel="self" href="/events" />
  <entry>
    <id>events/2</id>
    <updated>2018-01-01T10:00:00Z</updated>
    <author>
      <name>user</name>
    </author>
    <link rel="self" href="/events/2" />
    <link rel="related" href="/data/10" />
    <category term="data" />
    <category term="update" />
    <content type="application/xml">
      <data>
        <id>10</id>
        <data0>abc</data0>
        <data1>xyz</data1>
      </invalidation>
    </data>
  </entry>
</feed>
```

4.5.4 RS7 – Poll-Based Delta Replication

RS7 - Poll-Based Delta Replication uses a client-side replica that holds all data items of the server-side data store. The server files state differences (e.g. delta of the old version to the new version) as events to the web feed. Clients read the web feed in intervals and update their replicas based on the events in the web feed. Client-side replica are therefore updated asynchronously. The structure is the same as for RS6, described in *Subsection 4.5.3*.

RS7 is an optimization of RS6 to reduce bandwidth consumption. Because only state differences and not the whole state of a data item have to be transmitted, message size decreases and so does bandwidth consumption.

Request

The client-side full replica contains all data items. When a data item is requested on the client-side, it is read from the replica. The process is the same as in *Subsection 4.4.3*.

Update

The process of an update in the server-side data store is illustrated in *Subsection 4.5.2*.

Each web feed entry includes the ID of the data item. Insert operations also include the full state of the data item (as in RS6). Update operations only include the changed data (in the assumed scenario `data0`). Delete operations do not contain any other information than the ID. The following listing shows a web feed containing a single event of an update operation in the server-side data store.

```
<?xml version="1.0" encoding="UTF-8"?>
<feed xmlns="http://www.w3.org/2005/Atom">
  <id>http://www.application.com/events</id>
  <title>events</title>
  <updated>2018-01-01T10:00:00Z</updated>
  <link rel="self" href="/events" />
  <entry>
    <id>events/2</id>
    <updated>2018-01-01T10:00:00Z</updated>
    <author>
      <name>user</name>
    </author>
    <link rel="self" href="/events/2" />
    <link rel="related" href="/data/10" />
    <category term="data" />
    <category term="update" />
    <content type="application/xml">
      <delta>
        <id>10</id>
        <data0>abc</data0>
      </delta>
    </content>
  </entry>
</feed>
```

4.6 Summary

Table 4.7 summarizes the described replication strategies.

The next chapter investigates the presented replication strategies and their use in IT systems in more detail by examining qualitative and quantitative indicators of individual replication strategies.

	Name	Description
RS0	*No Replication*	Clients directly and synchronously request data items from the server. No replicas or caches are used.
RS1	*Cache Validation*	Clients directly and synchronously request data items from the server. Requested data items are stored in a client-side cache which is validated with conditional get and HTTP ETags.
RS2	*Event-Driven Cache Invalidation*	Clients directly and synchronously request data items from the server. Requested data items are stored in a client-side cache which is invalidated asynchronously by the server by sending push notifications over a publish-subscribe messaging channel.
RS3	*Event-Driven Replication*	Clients use a full replica of the server-side data store. Changes on the server-side are propagated asynchronously by push notifications over a publish-subscribe messaging channel. Notifications include the full state of the new version of a data item.
RS4	*Event-Driven Delta Replication*	Clients use a full replica of the server-side data store. Changes on the server-side are propagated asynchronously by push notifications over a publish-subscribe messaging channel. Notifications include state differences (e.g. delta of the old version to the new version).
RS5	*Poll-Based Cache Invalidation*	Clients directly and synchronously request data items from the server. Requested data items are stored in a client-side cache. The server files data items which have changed as events to a web feed. Clients read the web feed in intervals and invalidate data items in their caches based on the events in the web feed.
RS6	*Poll-Based Replication*	Clients use a full replica of the server-side data store. The server files the full states of the new versions of data items as an events to a web feed. Clients read the web feed in intervals and update their full replicas based on the events in the web feed.
RS7	*Poll-Based Delta Replication*	Clients use a full replica of the server-side data store. The server files state differences (e.g. delta of the old version to the new version) as events to a web feed. Clients read the web feed in intervals and update their full replicas based on the events in the web feed.

Table 4.7: Concrete implementations of the replication strategies

CHAPTER 5

Analytical Evaluation

This chapter provides an analytical evaluation of the replication strategies presented in *Chapter 4*. The evaluation bases on qualitative and quantitative indicators. It uses a simple scale to grade replication strategies related to the indicators. The scale uses the grades ++, +, -, -- where ++ represents the best grading possible while -- represents the worst. Empty grading represents a neutral grade between + and -.

5.1 Qualitative Indicators

Qualitative indicators are chosen based on the characteristics of the product quality[1] model in [ISO-25010] *"Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models."* These characteristics "*relate to static properties of software and dynamic properties of the computer system.*" [ISO-25010] distinguishes the following eight characteristics:

1. Functional suitability
2. Performance efficiency
3. Compatibility
4. Usability
5. Reliability
6. Security
7. Maintainability
8. Portability

Functional suitability und *usability* are not considered in the analysis of qualitative indicators, because this thesis provides a general examination, independent of a particular

[1] The norm is chosen because it describes important properties of software systems. The occurrence of "quality" in its name is a coincidence and does not relate to qualitative indicators.

use case and its functionality (which excludes *functional suitability*) and replication can be used even in systems without a user interface (which excludes *usability*). The following sections examine how replication strategies perform related to the six remaining characteristics of [ISO-25010].

5.1.1 Performance efficiency

[ISO-25010] defines Performance Efficiency as the *"performance relative to the amount of resources used under stated conditions"*. It further defines three sub-characteristics:

- **Time Behavior**: The efficiency related towards response and processing time.
- **Resource Utilization**: The efficiency towards the amount of resource usage.
- **Capacity**: The maximum limits of the system.

The next paragraphs examine how replication strategies perform related to the sub-characteristics.

Resource Initialization

The examination on resource utilization is based on the following questions[2]:

- How are client-side read operations (requests) processed?
- How are server-side write operations (updates) processed?
- On which places is data stored and how much data is stored?

Requests require operations on the client-side and in some replication strategies on the server-side as well. Client-side processes can be distinguished between simple redirections to the server (*Redirect - R*) and client-side read operations in the cache or full replica (*Client Read - CR*). With some replication strategies, the client has to request data from the server. It can be distinguished between cases in which the server receives no request, cases in which the server receives a partial request when a data item was not found in the client-side cache (*Partially Server Read - PSR*) and cases in which the server receives a request on every client request (*Server Read - SR*). Based on the description in *Chapter 4*, Table 5.1 shows which operations are performed on client and server.

	RS0	**RS1**	**RS2**	**RS3**	**RS4**	**RS5**	**RS6**	**RS7**
client	R	CR	CR	CR	CR	CR	CR	CR
server	SR	SR	PSR			PSR		

Table 5.1: Description of operations performed on client-side requests on client and server

The utilization on client and server was estimated under the assumption that $R < CR$ and $PSR < SR$. The results are presented in Table 5.2

[2] Additionally, the network utilization must be considered. To do so, bandwidth consumption is examined during the analysis of quantitative indicators in *Section 5.2*.

	RS0	**RS1**	**RS2**	**RS3**	**RS4**	**RS5**	**RS6**	**RS7**
client	-	--	--	--	--	--	--	--
server	--	--	-			-		

Table 5.2: Grading of *Resource Utilization* of client-side requests

Table 5.2 shows that with the use of full replicas (RS3, RS4, RS6, RS7) only the client is utilized. In environments where many clients use the same data, replication with full replicas therefore may decrease load on the server and distributes it over the clients. Cache Validation (RS2, RS4) has the same benefit as long as data items exist in the cache. RS0 and RS1 utilize the server on every request and therefore the server may become a bottleneck when data is requested very frequently.

Updates are performed on the server. Some replication strategies perform additional operations to notify clients about changes. Operations for notification are performed on the server-side, the message broker and the client. The server updates its local data store (*Update - U*) and optionally writes data into its local web feed (*Web feed - W*) or sends a message to the message broker (*Message - M*). When sending a message, the message broker has to process and deliver the message. In those cases, in which the client is notified, it has to process the received notification. Based on the replication strategy, the client has to invalidate a data item in the cache (*Invalidation - I*) or to update the full replica (*Update - U*). Based on the description in *Chapter 4*, Table 5.3 shows which operations are performed on the client, server and message broker.

	RS0	**RS1**	**RS2**	**RS3**	**RS4**	**RS5**	**RS6**	**RS7**
client			I	U	U	I	U	U
server	U	U	U+M	U+M	U+M	U+W	U+W	U+W
message broker			M	M	M			

Table 5.3: Description of operations performed on server-side updates on client and server

Under the assumption that $I < U$ and $M < W$ the following estimation of utilization of the client, server and message broker was made.

	RS0	**RS1**	**RS2**	**RS3**	**RS4**	**RS5**	**RS6**	**RS7**
client			-	--	--	-	--	--
server			-	-	-	--	--	--
message broker			-	-	-			

Table 5.4: Grading of *Resource Utilization* of server-side updates

Table 5.4 shows that RS0 and RS1 do not include any operations on the client and therefore do not utilize the client. Notifications lead to additional operations and therefore increase load on the client and server.

For the calculation of **storage** usage, the following variables are used:

- n as the number of data items
- m as the number of changes
- s as the mean size of a data item
- t as the mean size of the delta of a change
- i as the mean size of the ID of a data item

Table 5.5 shows the storage usage of individual replication strategies. The client can store no data items (RS0), parts of the data items in a non-complete cache (RS1, RS2, RS5) or all data items in a full replica (RS3, RS4, RS6, RS7). The server saves all data items in its data store and saves events, containing the full state (RS6), state differences (RS7) or the ID (RS5) in a web feed.

	RS0	**RS1**	**RS2**	**RS3**	**RS4**	**RS5**	**RS6**	**RS7**
client	0	<n*s	<n*s	n*s	n*s	<n*s	n*s	n*s
server								
- for data items	n*s	n*s	n*s	n*s	n*s	n*s	n*s	n*s
- for changes						m*i	m*s	m*t

Table 5.5: Calculation of storage utilized on client and server

Under the assumption that $n < m$ and $i < t < s$ the following estimation of storage utilization on client and server was made.

	RS0	**RS1**	**RS2**	**RS3**	**RS4**	**RS5**	**RS6**	**RS7**
client		-	-	--	--	-	--	--
server						-	--	-

Table 5.6: Grading of *Resource Utilization* of storage on client and server

Capacity

The limits of the system where not examined. It is assumed that business logic and processes on the presentation layer mainly utilize a system. Therefore, capacity planning mainly focuses on these areas and replication plays only a subordinate role in this context.

5.1.2 Compatibility

[ISO-25010] defines Compatibility as the degree to which a system can share an environment with other systems and exchange information with them. It further defines two sub-characteristics:

- **Co-Existence**: Indicates if a system affects other systems when they share the same environment.
- **Interoperability**: Indicates the degree to which a system can exchange information with other systems.

Co-Existence

Replication strategies described in this thesis can be implemented with standard software and libraries. *Chapter 4* described the technologies used. It is assumed that these software and libraries provide a high degree of co-existence. Replication only impacts the utilization of shared resources which was described in the course of the examination of *Resource Utilization* in *Subsection 5.1.1*.

Interoperability

It is assumed that interoperability can be ensured by the use of public standards. Replication strategies described in this thesis can be implemented based on such standards. Direct communication can be fully implemented by the public and widely used HTTP Standard, based on the de-facto standard of RESTful webservices. Standards for messaging and web feeds are described in *Chapter 3*. The concrete implementations presented in *Chapter 4* build on public standards. Interoperability should therefore be provided.

5.1.3 Reliability

[ISO-25010] defines Reliability as the ability of a system to perform its functionality *"under specified conditions for a specified period of time"*. It further defines four subcharacteristics:

- **Maturity**: Indicates how a system *"meets needs for reliability under normal operations"*.
- **Availability**: Indicates to which degree a system is available.
- **Fault Tolerance**: Indicates how a system behaves in the case of hardware or software faults.
- **Recoverability**: Indicates if a system recovers to a valid state after a fault is fixed.

The next paragraphs examine how replication strategies perform related to the subcharacteristics.

Maturity

It is assumed that maturity of a system can be determined by the complexity of the implementation of the system and the maturity of technologies and libraries used. Based on the description of replication strategies in *Chapter 4* the implementation of replication processes may consist of a client-side storage (cache or full replica), libraries to establish different types of communication (with RESTful webservices, messaging or web feeds) and the replication process itself.

Replicas can be implemented with standard persistence technologies such as relational databases. There exist widely used and mature technologies in this area.

Caches can be implemented with any persistence technology including relational databases or in-memory storage systems (e.g. *HashMaps* in Java). Furthermore, there are specialized cache solutions available that offer additional optimizations (e.g. to minimize requests to the server when a single data item is requested concurrently or to remove unused data items independently). For both cases, there exist widely used and mature technologies.

Communication between client and server with HTTP and **REST** can be established with standard functionalities of modern platforms, including advanced caching methods like ETags (see *Section 3.1*).

Software in the area of **messaging**, can be distinguished in message brokers and libraries which enable the use of messaging inside an application. There exist mature message brokers that are widely used. Popular examples are RabbitMQ and ActiveMQ. These message brokers support different messaging protocols to communicate with other message brokers and clients. In the course of this thesis, AMQP was used a messaging protocol. *Subsection 3.3.2* gives an overview of libraries that can be used for AMQP based messaging inside an application. These libraries are not part of the standard functionalities of modern platforms but are mostly provided by the vendors of the message brokers. Because of their origin and their widespread use, these libraries may also be considered mature.

This thesis focuses on Atom-based **web feeds** and therefore this examination also focuses on this standard. *Subsection 3.2.2* gives an overview of libraries that can be used to create and read Atom feeds in Java and Ruby. The survey concludes that there is just a single mature and maintained library available for Java. Ruby comes with an insufficiently documented RSS library in the standard extent. Other libraries are either retired or just private projects, not suitable for business projects. It is assumed that there are hardly any mature libraries available for other platforms as well.

The libraries presented in *Subsection 3.2.2* provide basic functionality to create and read web feeds but do not implement functionality to read new events so that they can be processed like the Event-Driven replication process described in [Web10]. A library that implements this process is available for Java[3]. This library is just a private project and was hardly maintained over the last years. To the knowledge of the author, other libraries are not available. The process has therefore to be implemented independently when RS5, RS6 or RS7 are used in an IT system. The process was implemented for Java and Ruby for the use in the simulation environment. It showed that an independent implementation is complex and error-prone.

There are only few mature libraries available for writing and reading of Atom feeds and no mature and maintained libraries that implement an actualization process like the process described by [Web10]. It is assumed that companies which use web feed inside their systems build and maintain proprietary solutions that are kept inside the company. An online survey based on publicly available information gave no further results. As

[3] `https://github.com/ICT4H/atomfeed` (last checked 26.03.2018)

an example, the German online retailer *OTTO* uses a AtomPub-based Event-Driven Architecture [Ste15]. OTTO provides several public repositories[4] and lots of information on their architecture and tooling, but no information on their usage of Atom or AtomPub was found. It can be concluded that libraries for Atom and Atom-based replication cannot be considered mature.

Replication is implemented by server-side processes for notification (filing changes into a web feed or sending messages) and client-side processes to actualize the cache or replica.

Server-side processes have to be embedded into the existing application layer and cannot be completely provided by standard functionality. The libraries described below offer technical functionality to support these processes. From the view of complexity, there are hardly any differences in the implementations of individual replication strategies.

Technical functionality of **client-side processes** is also covered for the most part by the libraries described below. Replication logic itself mainly consists of simple insert, update and delete operations. Replication strategies which distribute only state differences (RS4, RS7) need additional but simple programming code to update the state of an object. Therefore, the use of these replication strategies is more complex.

The examination results in the following grading:

	RS0	**RS1**	**RS2**	**RS3**	**RS4**	**RS5**	**RS6**	**RS7**
Maturity	+	+	+	+	-	-	-	--

Table 5.7: Grading of *Maturity* of the replication strategies

Availability

Replication improves availability because data items can be read locally. Otherwise, communication over the network is required. Because it must be assumed that the network is not reliable and communication over the network may fail even in normal operation mode (e.g. packages are lost), communication over the network reduces availability [Tan07]. Effects when an entire communication channel fails are considered during the examination of *Fault Tolerance*.

With replication, data items are stored locally and therefore network communication is not required. Replication strategies that use a client-side full replica (RS3, RS4, RS6, RS7) provide the highest level of availability because data items can be read without any communication to the server. Replication strategies which use a client-side cache (RS2, RS5) provide a lower level of availability. Server communication is required every time a data item is requested that is not contained in the cache. From the perspective of availability, they are advantageous over replication strategies that need server communication on each access on a data item (RS0, RS1).

[4] `https://github.com/otto-de` (last checked 26.03.2018)

The examination results in the following grading:

	RS0	**RS1**	**RS2**	**RS3**	**RS4**	**RS5**	**RS6**	**RS7**
Availability	-	-		+	+		+	+

Table 5.8: Grading of *Availability* of the replication strategies

Fault Tolerance

The examination on fault tolerance bases on an analysis of the consequences of failure of individual communication channels. Figure 5.1 shows communication channels between the components based on the description in *Chapter 4*[5].

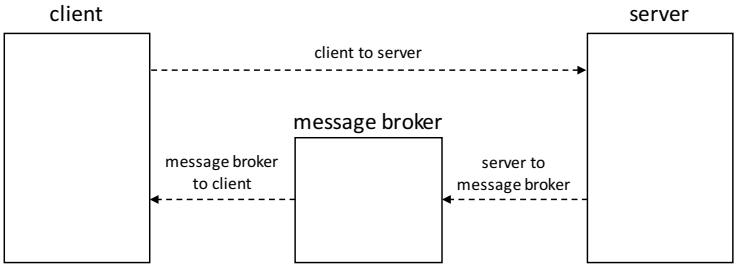


Figure 5.1: Communication channels between the components

Communication channels from and to the message broker are only used in Event-Driven replication strategies (RS2, RS3, RS4). Therefore, the examination of communication channels concerning the message broker only considers these replication strategies. The following paragraphs examine the effects of failure of individual communication channels in detail:

- **Client to server**: If a client cannot access the server (e.g. if the server is down), the client is limited. When data items are solely requested from the server without any form of replicas (RS0), fault tolerance is not given. These systems implement a CP system according to the CAP theorem. When the server is not available, the client is not available as well. Replication improves fault tolerance because data items or at least parts of them are stored on the clients. For further examination of fault tolerance in systems using replication, it is distinguished between effects when data items are accessed on the client-side and effects on replication processes.

  Access on data items on the client-side is only affected when caches are used (RS2, RS5). Data items which are not contained in the cache have to be loaded from the server. When the server is not available, it is not possible to load them. Fault tolerance is therefore given only partially (for the extent in which data items

[5] There is no direct communication from the server to the client and therefore this communication channel is not considered.

are available in the cache). Full Replicas (RS3, RS4, RS6, RS7) contain all data items and need no server communication. They are therefore also not affected when communication with the server fails.

Replication processes are only affected when Poll-Based replication strategies (RS5, RS6, RS7) are used. These replication strategies require access to the server-side web feed to get notified about changes. When the server is not available, changes are not delivered to the client. Event-Driven replication strategies (RS2, RS3, RS4) need no direct server communication and are therefore not affected.

RS1 was not considered so far. It uses a cache but validates data items on each access. When the client cannot access the server, the client can choose one of two options based on its business requirements. The client can prohibit access to all data items. This behavior is an implementation of a CP system according to the CAP theorem. The other option is to read data items solely from the cache without any validation. The state of the client-side cache may differ from the server-side state and this therefore weakens the strict consistency constraint in favor of availability. Therefore, this behavior is an implementation of an AP system according to the CAP theorem.

- **Message broker to client**: If the message broker cannot access a client (e.g. if the client is down), the particular client is not notified about changes. The server is still able to process updates in its data store and still sends messages intended for this client to the message broker. The message broker cannot deliver messages to the unreachable client but persists them inside the message queue (see *Subsection 4.4.1*). These messages are delivered when the client is reachable again. The client therefore still works in an eventually consistent mode.

- **Server to message broker**: If the server cannot access the message broker (e.g. when the message broker is down), replication is not possible. Clients are still able to read data items from their cache or full replica and to request data from the server. To ensure eventual consistency even if the message broker is not available, the server has two options: It can prohibit updates or it has to store all changes locally and sends them to the message broker when it is available again (e.g. with a local persistent queue similar to the queue on the message broker).

The examination results in the following grading:

	RS0	**RS1**	**RS2**	**RS3**	**RS4**	**RS5**	**RS6**	**RS7**
Fault Tolerance	--	-	+	++	++	+	++	++

Table 5.9: Grading of *Fault Tolerance* of the replication strategies

Recoverability

In the case of replication, it is assumed that recoverability is the recovery of a consistent system state after the failure of one of the three communication channels described below. It is guaranteed by all replication strategies. The following examination of individual replication strategies builds on the discussion on fault tolerance above:

- **Synchronous Requests (RS0)** RS0 cannot lead to an inconsistent system state, because data items are stored on the server only.
- **Cache Validation (RS1)**: Inconsistencies can only occur when the client weakens the strict consistency constraint. In this mode, data items can be validated as soon as the server is accessible again. At this juncture, strict consistency is recovered
- **Event-Driven replication strategies (RS2, RS3, RS4)**: Notifications are persistent on the server or message broker when communication channels from or to the message broker failed. As soon as the communication channels are working again, notifications are delivered and the system recovers to a consistent state. If the server decides to prohibit updates no inconsistencies can occur and there is no need for recovery.
- **Poll-Based replication strategies (RS5, RS6, RS7)**: Updates are always filed as events in the web feed, independently of the availability of the communication channels. Clients read the web feed when the channel is available again and a consistent system state is recovered.

The examination results in the following grading:

	RS0	**RS1**	**RS2**	**RS3**	**RS4**	**RS5**	**RS6**	**RS7**
Recoverability	++	++	++	++	++	++	++	++

Table 5.10: Grading of *Authenticity* of the replication strategies

5.1.4 Security

[ISO-25010] defines Security as the degree to which a system protects information and data. It further defines five sub-characteristics:

- **Confidentiality**: Indicates if data is only accessible to the ones who are authorized.
- **Integrity**: Indicates that unauthorized access and modifications are prevented.
- **Non-repudiation**: Indicates that events can be proven to have taken place.
- **Accountability**: Indicates that actions can be uniquely traced to the one who performed the action.
- **Authenticity**: Indicates that a subject or resources can uniquely be identified as the one it claims to be.

The next paragraphs examine how replication strategies perform related to the sub-characteristics. The characteristics *Confidentiality* and *Integrity*, *Non-repudiation* and *Accountability* share many attributes in their definition and will be examined together as in [Xu13].

Confidentiality and Integrity

[Xu13] distinguishes confidentiality and integrity into five system properties: *secure data transport*, *secure data storage*, *authorized data access*, *secure authorization* and *input and output verification*.

Secure Data Transport can be established with *Transport Layer Security* (*TLS*) [RFC-5246]. Secure HTTP based communication (used for RESTful webservices and web feeds) can be ensured via HTTPS which uses TLS [RFC-2818]. Messaging protocols also support mechanisms to secure data transfer. As an example, AMQP also supports TLS as a security layer[6].

HTTP and AMQP are network protocols and do not provide mechanisms for **Authorized Data Access** out of the box. Since HTTP is widely used, many solutions exist for authorization. It can be implemented directly on a webserver or web application server with technologies like the *Java Authentication and Authorization Service* (*JAAS*) [Ora] in Java EE or provided by a distributed solution like *OAuth* [7]. For messaging, authorization has to be implemented on the message broker. As an example, RabbitMQ supports authorization on the level of a virtual host (a logical group of entities a set of resources belongs to), resources (e.g. exchanges and queues) and topics[8]. The described methods are widely used in industry, several of which are industry standards. **Secure Authorization** can therefore be assumed as well.

Secure Data Storage and **Input and Output Verification** are important security concerns, but not special for the case of replication and the presented replication strategies. These system properties are therefore not further examined.

The examination results in the following grading:

	RS0	**RS1**	**RS2**	**RS3**	**RS4**	**RS5**	**RS6**	**RS7**
Confidentiality and Integrity	++	++	++	++	++	++	++	++

Table 5.11: Grading of *Confidentiality and Integrity* of the replication strategies

[6] `https://docs.oasis-open.org/amqp/core/v1.0/amqp-core-security-v1.0.html` (last checked 26.03.2018)

[7] `https://oauth.net/2/` (last checked 26.03.2018)

[8] `https://www.rabbitmq.com/access-control.html` (last checked 26.03.2018)

Non-Repudiation and Accountability

Most concerns in this area are independent of replication: The server has to trace read and update operations in his data store as the client for read operations. Specific concerns in the area of replication are tracing of data items and their versions that are read by the clients and the proof that events are successfully delivered.

Clients read data items from the server via RESTful webservices (for RS0, RS1, RS2, RS5). Communication can only be retraced on an HTTP level (e.g. with HTTP access logs). The content of HTTP requests and therefore the version of a data item is not visible. It can only be reproduced based on internal timestamps (e.g. the data of the last change of a data item). This information can be corrupted and is therefore not reliable.

Reception of events can be distinguished between messaging (RS2, RS3, RS4) and web feeds (RS5, RS6, RS7). Messaging with AMQP and RabbitMQ provides an acknowledgement mechanism[9]. Clients confirm that they received a message and therefore also confirm that they received an event notification. Historical web feed contents can only be traced based on timestamps (e.g. the create date of a web feed entry) that can be corrupted. Reception of individual events can therefore not be ensured.

The examination of these two characteristics result in the following grading:

	RS0	**RS1**	**RS2**	**RS3**	**RS4**	**RS5**	**RS6**	**RS7**
Non-Repudiation and Accountability	-	-		++	++	--	-	-

Table 5.12: Grading of *Non-Repudiation and Accountability* of the replication strategies

Authenticity

The HTTP standard provides authentication mechanisms out of the box [RFC-7235]. In addition to the standard there are other products available. A widely used alternative is *OpenID*[10], a decentralized authentication protocol. Authentication for messaging depends on the protocol used. AMQP supports the public *Simple Authentication and Security Layer* (*SASL*) [RFC-4422] standard[11]. All these standards are widely used in industry and can be considered secure.

	RS0	**RS1**	**RS2**	**RS3**	**RS4**	**RS5**	**RS6**	**RS7**
Authenticity	++	++	++	++	++	++	++	++

Table 5.13: Grading of *Authenticity* of the replication strategies

[9] `https://www.rabbitmq.com/confirms.html` (last checked 26.03.2018)
[10] `https://openid.net/` (last checked 26.03.2018)
[11] `https://docs.oasis-open.org/amqp/core/v1.0/amqp-core-security-v1.0.html` (last checked 26.03.2018)

5.1.5 Maintainability

[ISO-25010] defines Maintainability as the degree to which a system can be changed and extended. It distinguishes five sub-characteristics:

- **Modularity**: Indicates the degree to which a system is composed of discrete and independent components.
- **Reusability**: Indicates to which degree assets can be used in other systems.
- **Analysability**: Indicates if changes and failures in the system can be localized and identified.
- **Modifiability**: Indicates how difficult it is to modify the system and how likely defects are introduced with modifications.
- **Testability**: Indicates to which degree the system supports testing in a given test context.

The next paragraphs examine how replication strategies perform related to the sub-characteristics.

Modularity

Replication strategies do not require modularity of components or the overall system. From a system perspective, each application can be considered an independent module of the overall system. Replication enables loose coupling and enforces modularity on this level. From the internal perspective of an application, the structure and modularity inside an application is independent of the replication strategy used.

Reusability

Individual replication strategies do not differ in their level of reusability. From a system perspective, all replication strategies are designed to replicate data items to an arbitrary number of clients. The replication service itself is therefore reusable. From the perspective of a single application, the implementation of a replication strategy can build on existing assets. Libraries and technologies that can be used for this purpose are described in the course of the examination of *Maturity* in *Subsection 5.1.3*.

Analyzability

Analyzability decreases with asynchronous and indirect communication. Synchronous processes and direct communication that fails or behaves strangely can easily be monitored. Asynchronous and indirect communication cannot be monitored by a single application but must be monitored from a system perspective. Replication strategies use asynchrony and indirect communication with messaging (RS2, RS3, RS4) and web feeds (RS5, RS6, RS7).

Message-based systems are hard to monitor. Communication is asynchronous by nature and completely handled by the message broker. The flow of messages over the network is hard to trace. There are patterns like *Invalid Message Channel* and *Dead Letter Channel* in this area. [Hoh04] examines this topic and describes further patterns in this area. These solutions are complex and need expert knowledge.

The main problem with replication based on web feeds occurs if the server does not file a change into the web feed. The absence of the change is not visible to the client and the client cannot react and report a problem. Other processes are easy to monitor because they are synchronous: the client synchronously reads the web feed and the server synchronously writes changes to it.

Additionally, most concerns that where examined during the examination of *Non-Repudiation and Accountability* in *Subsection 5.1.4* also apply here. They are not as critical as the concerns described below and not considered in the grading. The examination results in the following grading:

	RS0	**RS1**	**RS2**	**RS3**	**RS4**	**RS5**	**RS6**	**RS7**
Analysability			--	--	--	-	-	-

Table 5.14: Grading of *Analyzability* of the replication strategies

Modifiability

Defects are most likely introduced with breaking changes in the signature of a data item. Processes for replication inside an application are not complex (see the examination of *Maturity in Subsection 5.1.3*). Therefore, modifications made are not likely to introduce defects into the application. Furthermore, modifications inside an application can easily be tested with unit tests. On the system level, replication integrates individual applications and client and server must therefore share a common definition of the structure of data items. Whenever the server changes the signature of data items (like the data type of an attribute), clients may fail when they do not support the new signature.

This problem is not unique to replication but an integration problem in general. Integration on the data layer (e.g. with a shared database) which uses a shared data schema is affected even more. Changes on the data schema need a coordinated deployment of all applications that share the data schema. Integration on the application layer provides several techniques to bypass the problem and enable independent deployments.

A widely used pattern in distributed IT systems is the *Data Transfer Object* (*DTOs*) [Dai12]. A DTO is a representation of internal data for the communication with other applications. DTOs are used to hide the internal data structure which can even be changed without affecting other applications. Webservices may use DTOs in combination with *versioning* [All10] of an API to provide old and new schemas of data items simultaneously. Another widely used pattern is *Tolerant Reader*[Fow11]. It bases on *Postel's Law "be conservative in what you do, be liberal in what you accept from others"* [Fow11]. The

pattern recommends that clients only read the attributes needed and to make minimum assumptions about the schema. This minimizes the chance that the client is affected by a change of the signature of a data item. Beside these patterns, there are testing strategies and tools for avoiding integration problems in general. These are described in this subsection in the course of the examination of *Testing*.

Poll-Based replication strategies with a full replica (RS6, RS7) are more vulnerable to breaking changes than other replication strategies. These replication strategies read the whole web feed including all archive feeds for the initial set-up (see *Subsection 4.5.3*). When the schema changes over time, clients must be aware of all historical schemas to import historical data. The exchange of historical schema information is a complex task. A solution to this problem is not examined in this thesis. To bypass the problem, the initial data import can be done manually (see *Subsection 4.4.3*).

Most problems concerning modifiability can be avoided or bypassed using modern software engineering techniques. The fundamental problem of breaking changes in the common signature of shared data items exists in systems without replication (RS0) and systems with integration on the data layer as well. The only problem found in this examination which is unique to replication is when initializing Poll-Based replication strategies with full replicas. The examination therefore results in the following grading:

	RS0	**RS1**	**RS2**	**RS3**	**RS4**	**RS5**	**RS6**	**RS7**
Modifiability							-	-

Table 5.15: Grading of *Modifiability* of the replication strategies

Testability

Replication integrates individual applications and therefore testing mainly relates to integration testing. Given an appropriate software design, business logic and the invocation of replication processes can be tested within an application with unit tests. Communication from the client to the server and from client and server to the message broker requires integration testing. [Fow18] describes three types of automated integration tests: *broad tests* (including all applications), *narrow tests* (by using *test doubles*) and *contract tests* (where tests are executed against pre-defined contracts). Replication strategies are examined concerning these types:

- **Broad tests**: Broad integration tests require a production-like test environment. All components used in a test case are executed in this environment. In a system that uses replication, this may include a server, a client and a message broker for Event-Driven replication strategies. Tests are run against concrete applications. These tests are very similar to system tests [Fow18]. Based on the *test pyramid* [Fow12] [New15], system tests should only be a small part of all tests, because they are very costly and likely to change. Therefore, the exclusive use of broad tests cannot be recommended.

Broad integration tests can be executed in a test environment without any additional requirements on applications. Therefore, all replication strategies can be tested with these tests.

- **Test doubles**: Test doubles simulate the behavior of an external component [Fow08] like an application [Fow18]. An application can therefore be tested in isolation without the need to run other applications [New15] [Wol15]. These tests would not necessarily require a dedicated test environment and may therefore also be executed on local developer machines. The use of test doubles is favored against broad tests, because individual applications can be tested individually and are not affected by changes in other applications [New15]. This may lead to problems because changes in other applications are not explicitly tested [New15].

  Test doubles can be used for HTTP based replication strategies (RS0, RS1, RS5, RS6, RS7 and parts of RS2). There exist frameworks to create test doubles for HTTP servers (e.g. *MockServer*[12]) that can be used for this purpose. Test doubles for Poll-Based replication strategies need a repository where web feeds are statically filed (which leads to a high maintenance effort when the structure of the web feed changes) or a proprietary test framework that creates corresponding web feeds (which needs a high initial implementation effort and continually maintenance effort). To the author's knowledge, there are no frameworks to create test doubles for messaging and message brokers available. Test doubles for testing Event-Driven replication strategies (RS3, RS4 and parts of RS2) can therefore only be implemented on the level of local unit tests and not at the level of integration tests.

- **Contract tests**: With contract tests, contracts are defined to specify interfaces between client and server. Tests can be run against the contracts respectively test doubles that are created based on the contracts. Contract tests therefore further decouple applications in the test phase. *Consumer-Driven Contracts* (*CDC*) [Dai12] [Rob06] are a popular pattern in this area. With CDC, clients define contracts that represent their requirements on a server-side service. The server can execute contract tests to test its implementation against all client requirements.

  CDC frameworks support contract tests for communication based on HTTP and messaging protocols[13]. Contract tests can therefore be implemented for all replication strategies.

The examination therefore results in the following grading:

	RS0	**RS1**	**RS2**	**RS3**	**RS4**	**RS5**	**RS6**	**RS7**
Testability			--	--	--	-	-	-

Table 5.16: Grading of *Testability* of the replication strategies

[12] `http://www.mock-server.com/` (last checked 26.03.2018)
[13] `https://cloud.spring.io/spring-cloud-contract/` (last checked 26.03.2018)

5.1.6 Portability

[ISO-25010] defines Portability as the degree to which a system can be ported to another environment (e.g. with different hardware or software). It further defines three sub-characteristics:

- **Adaptability**: Indicates if the system can be ported in a different environment and how difficult it is to port it.
- **Installability**: Indicates how difficult it is to install or uninstall the system.
- **Replaceability**: Indicates if the system can be replicated with another system for the same purpose in the same environment and how difficult it is to replace it.

The next paragraphs examine how replication strategies perform related to the sub-characteristics.

Adaptability

Replication strategies described in this thesis can be implemented with standard software and libraries. These software is widely used in IT systems and therefore there are no specific requirements for hardware, operation system or environment.

	RS0	**RS1**	**RS2**	**RS3**	**RS4**	**RS5**	**RS6**	**RS7**
Adaptability	++	++	++	++	++	++	++	++

Table 5.17: Grading of *Adaptability* of the replication strategies

Installability

Replication processes are directly implemented in the application layer of existing applications. Therefore, there are no additional installation tasks necessary - code for replication is deployed with the application that uses it.

Replication strategies using messaging require a **message broker**. The message broker must be provided within the IT system and configured properly. One message broker, respectively one installation can be used for several clients and servers. Configuration (especially the configuration of persistent queues and authorization on queues and topics) must be performed for every server and client separately. RS3 and RS4 require an additional **initial data import** which has to be performed once for every client.

Table 5.18 shows an examination of the installation effort needed for the individual replication strategies. The examination does not consider additional security configuration (e.g. for firewalls, OAuth or OpenID) and installation and configuration of additional components (e.g. a OAuth infrastructure or an AtomPub server).

	RS0	RS1	RS2	RS3	RS4	RS5	RS6	RS7
message broker			-	-	-			
inital data import				-	-			
sum			-	--	--			

Table 5.18: Grading of *Installability* of the replication strategies

Replaceability

It is assumed that replicability can be achieved by the use of standards. When all components support the same public or propriety standards, individual components can be replaced with new components that support the same standards. Public standards were examined during Interoperability in *Subsection 5.1.2*. Propriety standards for replication where defined in the description of replication strategies in *Chapter 4*. The requirement for replicability should therefore be fullfilled.

	RS0	RS1	RS2	RS3	RS4	RS5	RS6	RS7
Replaceability	++	++	++	++	++	++	++	++

Table 5.19: Grading of *Replaceability* of the replication strategies

5.2 Quantitative indicators

This section provides a definition of quantitative indicators used in this thesis and examines them analytically. The selection of quantitative indicators focuses on areas that are not covered by the qualitative analysis in *Section 5.1*. It bases on differences between individual replication strategies and differences to systems that do not use replication at all (like RS0). It is assumed that this approach leads to the most informative results. Based on the description in *Chapter 4*, it can be concluded that request and update operations of individual replication strategies mainly differ in the following areas:

1. Individual replication strategies use different processes to access data on the client-side. Some replication strategies directly read data items locally, others require more time-consuming requests to the server. As a consequence, **Client-Side Data Access Latency** differs.
2. Individual replication strategies use different processes to notify clients about server-side updates. Depending on how and if client notifications are performed and processed, **Server-Side Data Update Latency** differs.
3. Because processes for client-side read operations and server-side notifications differ and use different types of communication, differences in **Bandwidth Consumption** occur.
4. Because the client may read data items from a local replica that is not updated immediately, the time of inconsistency and the implemented **Consistency** models vary between the replication strategies.

The concrete quantitative indicators are composed based on these areas in *Subsection 5.2.1*. The indicators highly depend on request behavior and data characteristics. *Subsection 5.2.2* describes parameters in these areas. The last four subsections examine the indicators for individual replication strategies analytically and provide approximations based on the grading presented in the introduction of this chapter. Specific values are retrieved in the simulation. This approach bases on the recommendation of [Jai90] to validate results of a simulation by an analytical examination.

5.2.1 Definition of Quantitative Indicators

Quantitative indicators are defined for the four areas described above. To find appropriate indicators and metrics, each area is viewed separately. These surveys result in the definition of a main indicator for each area which is a discrete value that can easily be compared. These indicators are also measured in the simulation and described in *Chapter 6*. During the simulation, additional metrics (e.g. distribution functions) are collected as well but are only presented in cases of particular importance.

Client-Side Data Access Latency

[Siv07] defines *end-to-end client latency* as the sum of *network latency* (which is the time needed to send a request over the network and the time to send the corresponding response) and *internal latency* (which is the time to process the request on the client). Internal latency includes possibly necessary communication with the server as well.

Client-side Data Access Latency can be measured from the perspective of the client (which corresponds to the internal latency) or from the perspective of the component which requests a data item from the client (which corresponds to the end-to-end client latency). All evaluations presented in *Subsection 2.4.1* measured the end-to-end client latency. These evaluations tested existing systems as a black box, where it is difficult or impossible to measure internal latency. Even though internal latency is not measured in these surveys, it must be assumed that internal latency is more accurate because it is independent of network behavior (see *Pitfalls* of distributed system design in [Tan07]) and therefore of fluctuations of transfer times. To ensure accurate results, the internal latency is used.

Client-side Data Access Latency is measured in milliseconds for every request. The main indicator is the mean of all measured data points.

Server-Side Data Update Latency

Data Update Latency is the latency of a server-side change of a data item. It can be measured from the perspective of the server (which corresponds to the internal latency) or from the perspective of the component which sends the update to the server (which corresponds to the end-to-end client latency). Internal latency includes processes which

are necessary to notify clients (send a message to the message broker or file a change in the web feed). The same applies as for client-side Data Access Latency: it must be assumed that internal latency is more accurate and therefore internal latency is used.

Server-side Data Update Latency is measured in milliseconds for every update. The main indicator is the mean of all measured data points.

Bandwidth Consumption

The presented replication strategies use the three communication channels which are described during the examination of *Fault Tolerance* in *Subsection 5.1.3*. During the simulation, data transferred over each of these communication channels is measured independently. The measurement includes not only application data viz. data items and replication overhead (e.g. Atom metadata and message headers) but also overhead of the network communication (e.g. overhead of TCP/IP communication).

Bandwidth consumption is measured in Kilo-Bytes transferred over the network. The main indicator is the sum of the transferred data through all three communication channels.

Consistency

There are two popular metrics to measure consistency [Bai12] [Ber13b] [Ber14]:

- *t-visibility*: Defines the time it takes from an update to the point where all replicas are in a consistent state.
- *k-staleness*: Based on individual requests, it measures the number of versions between the version read by a request on a replica and the most recent version.

These metrics are not measured as discrete values but as probabilistic functions. This enables assumptions for the consistency of an individual request [Bai12].

T-visibility is independent of individual requests and therefore provides general validity. It is also used by the YCSB++ benchmark [Pat11]. Because k-staleness is only measured when a request accesses a data item, it highly depends on the access pattern [Ber14b]. Therefore, results have no general validity.

Consistency is measured by the time it takes for a server-side update to be available in the client-side replica for every update in milliseconds. These data also build the basis of the t-visibility metric. K-staleness is not measured because it has no general validity. The main indicator is the mean of the measured data points.

5.2.2 Parameters

Quantitative parameters depend on request behavior and data characteristics. The following areas can be distinguished:

- *Data*: Characteristics of the data items that are replicated from the server-side to the client-side.
- *Requests*: Characteristics of read operations on the client-side.
- *Updates*: Characteristics of write operations on the server-side.

Table 5.20 shows the parameters that are determined in these areas:

Area	**Parameter**	**Description**
data	number	the number of elements of a single entity type that are stored in the server-side data store
	size	the mean size of data items
	size distribution	the statistical distribution of the size over all data items
request	request distribution	the statistical distribution of read operations over all data items
	interval	the number of read operations per time
	time distribution	the statistical distribution of read operations over a time period
update	update distribution	the statistical distribution of write operations over all data items
	interval	the number of write operations per time
	time distribution	the statistical distribution of write operations over a time period
	size of change	the percentage of the content that is changed with a write operation
	size distribution	the statistical distribution of the size of changes over all write operations

Table 5.20: Description of parameters

These parameters are considered in the analytical evaluation and are also used in the simulation which is described in *Chapter 6*.

5.2.3 Client-Side Data Access Latency

Data Access Latency defines the time needed to read a data item on the client-side and depends on whether data items are stored on the client-side (in a cache or a full replica) or have to be requested from the server. Local read operations take less time than communication over the network. Based on the description in *Chapter 4*, the following cases can be distinguished:

- *Server*: Data items are requested from the server only. Communication to the server is therefore mandatory. Data items are read in the server-side data store only. (RS0)
- *Client and server*: Data items are read from the client-side cache and additionally validated at the server-side. Communication to the server is therefore mandatory. Data items are read in the client-side cache and the server-side data store. (RS1)
- *Client and possibly server*: Data items are read from the client-side cache and if the cache does not contain a data item, it is loaded from the server. Communication to the server may be necessary but is not mandatory. Data items are read in the client-side cache and potentially in the server-side data store too. (RS2, RS5)
- *Client*: Data items are only read from the client-side full replica. No communication to the server is necessary. (RS3, RS4, RS6, RS7)

It is assumed that read operations in the server-side data store, the client-side full replica and the client-side cache need the same amount of time, provided that these stores are fully optimized. Data Access Latency therefore depends on the number of places where data items have to be read and on the condition if server communication is necessary. This results in the following grading:

	RS0	**RS1**	**RS2**	**RS3**	**RS4**	**RS5**	**RS6**	**RS7**
Data Access Latency	-	--	+	++	++	+	++	++

Table 5.21: Grading of *Client-Side Data Access Latency*

Data Access Latency depends on the load on server and client (depending where read operations take place). It must be assumed that high load leads to higher latency, because the data storage and the machines where client and server are operated are highly utilized or blocked.

The number of data items and their size can also have a negative effect on latency. Read operations are slower when the size of a data storage increases. Optimizations to counteract these effects exist (e.g. database *indices* in relational databases). Persistence technologies and their optimizations are not covered by this thesis (see delimitations in *Subsection 1.3.2*) and this optimizations are therefore not examined.

5.2.4 Server-Side Data Update Latency

Update latency defines the time needed for a write operation on the server-side, including replication processes. Data items are updated in the server-side data store only. In addition, the server notifies clients about changes. The update in the server-side data store does not differ for individual replication strategies. Therefore, time differences for updates mainly depend on the notification process. It can be distinguished between the following cases:

- *No Notification*: The client requests or invalidates data items on each client-side access. There is no need for notifications. (RS0, RS1)
- *Direct Notification*: The client is notified directly via messaging. To do so, the server sends a message to the message broker. The server is blocked until the message is successfully sent to the message broker. The message broker delivers the message independently to all clients. (RS2, RS3, RS4)
- *Indirect Notification*: The server files changes as events in a web feed. The server is blocked until the entry in the web feed is created. The client reads the web feed independently. (RS5, RS6, RS7)

It is assumed that *No Notification* < *Indirect Notification* < *Direct Notification* because communication over the network is expected to take more time than a local update of the web feed. This results in the following grading:

	RS0	**RS1**	**RS2**	**RS3**	**RS4**	**RS5**	**RS6**	**RS7**
Data Update Latency	+	+	--	--	--	-	-	-

Table 5.22: Grading of *Server-Side Data Update Latency*

Update latency depends on the load on the server-side and the number of data items and their size. A detailed description can be found in *Subsection 5.2.3* below.

5.2.5 Bandwidth Consumption

Bandwidth consumption defines the size of data that must be transmitted through the network. Replication strategies require communication between server and client to exchange data items and notifications. As described in *Chapter 4*, each replication strategy uses different processes for requests and updates and therefore bandwidth consumption differs.

To compare bandwidth consumption, it will be calculated based on different scenarios of client-side read operations and server-side update operations (including bandwidth consumed by notifications) for a single data item. Four scenarios are distinguished: 1) a single server-side update, 2) a single client-side data access, 3) low read to write ratio (one client-side data access and $n > 1$ updates) and 4) high read to write ratio ($n > 1$ client-side data accesses and one update).

The scenarios use the following parameters[14]:

- *d*: size of a single data item
- *c*: size of the change in a data item
- *i*: size of the ID of a data item
- *o*: overhead of communication with empty message body but including headers
- *0*: no communication which means no bandwidth consumption

The following applies: $d > c > i > o > 0$

The calculation uses simplified scenarios for Poll-Based replication strategies (RS5, RS6, RS7). It is preconditioned that the web feed is empty at the beginning and accessed only once. In general, the bandwidth consumption of Poll-Based replication strategies depends on how often the web feed is read, the number of entries in the web feed, the number of feeds that have to be read (recent feed plus zero or more archive feeds) and the size of an entry (depending on the replication strategy this could be *d*, *c* or *i*).

Feeds have to be transmitted completely. The number of feeds that have to be read depends on the number of changes since they were last read (see *Section 4.4*). Other variables are fixed by the server (number of web feed entries) or the client (interval). The number of archive feeds that have to be read can be calculated as follows:

number feeds = ceil (number of changes / web feed length)

Because of the complexity of the calculation and the lack of realistic values, a simplified calculation is used. A general and realistic estimation is performed in the simulation.

	RS0	**RS1**	**RS2**	**RS3**	**RS4**	**RS5**	**RS6**	**RS7**
0 request 1 update	0	0	i	d	c	i	d	c
1 request 0 update	d	d	d	0	0	d	0	0
1 request n updates	d	d	d+n*i	n*d	n*c	d+n*i	n*d	n*c
n requests 1 update	n*d	d+(n-1)*o	d+i	d	c	d+i	d	c

Table 5.23: Calculation of *Bandwidth Consumption*

Table 5.23 illustrates the simplified calculation. It shows that bandwidth consumption mainly depends on the **size of a data** item or the **size of a change**. The benefit of replication strategies where only state differences are transmitted depends on the relation of the size of a change to the size of the data item itself.

Another insight of the calculation is the dependence on the **number and distribution of requests and updates**. Replication strategies that require clients to request the

[14] To simplify the calculation, the overhead of communication for *d*, *c* and *i* is not considered.

server on each client-side request (RS0, RS1) depend on the number of requests, others that use a full replica (RS3, RS4, RS6, RS7) depend on the number of updates.

The distribution of requests and updates affects replication strategies that use a cache (RS1, RS2, RS5), even if it is not visible in the examination. A data item is transmitted, the first time it is accessed after an update. It can be assumed that these replication strategies are most efficient when a part A of the data items is updated, and another part B of the data items is requested and least efficient when a data item is updated each time before it is read.

Table 5.24 summarizes the dependencies between replication strategies and parameters. It shows that bandwidth consumption between requests and updates correlates negatively.

	RS0	**RS1**	**RS2**	**RS3**	**RS4**	**RS5**	**RS6**	**RS7**
request	--	-	+	++	++	+	++	++
update	++	++	+	--	-	+	--	-

Table 5.24: Grading of *Bandwidth Consumption*

5.2.6 Consistency

Consistency in respect of this analysis is defined by the time of inconsistency in the system, based on the consistency model implemented by individual replication strategies. [Ada12] state that consistency in a system is assured by communication between the components and the level of communication depends on the intensity of communication. Based on this statement, consistency will be examined in relation to the types of communication. The following cases can be distinguished:

- *Direct Server Requests (S)*: The client requests or invalidates data on each client-side access. Data is therefore always consistent. These replication strategies implement **strict consistency** (S). (RS0, RS1)
- *Direct Notification (D)*: The client is notified directly via messaging. To do so, the server sends a message to the message broker. The server is blocked until the message is delivered to the message broker. Because new data can be read at the server immediately after the notification is sent but before notifications arrive at the client, data is inconsistent until the notification arrives at the client. These replication strategies implement **eventual consistency** (E). (RS2, RS3, RS4)
- *Indirect Notification (I)*: The server files changes as events in a web feed. The server is blocked until the entry in the web feed is created. The client reads the web feed independently in intervals to update its cache or full replica. Because new data can be read at the server immediatly after the web feed entry is created but before the client reads the web feed, data is inconsistent until the next time the client reads the interval. Therefore, the interval defines duration of inconsistency. These replication strategies implement **eventual consistency** (E). (RS5, RS6, RS7)

It must be assumed that the time needed to deliver a message is shorter than the interval the web feed is read. It follows that the duration of inconsistencies for indirect notifications is higher than for direct notifications.

Replication strategies with direct and indirect notifications guarantee **causal consistency** only. Other eventual consistency models depend on the specific implementation. As an example, *Read-your-writes consistency* can be achieved with RS2 if the client directly invalidates its local cache after it sends an update operation to the server[15].

The following table concludes findings and shows the final grading:

	RS0	**RS1**	**RS2**	**RS3**	**RS4**	**RS5**	**RS6**	**RS7**
Consistency Model	S	S	E	E	E	E	E	E
Duration of Inconsistency	S	S	D	D	D	I	I	I
Grading	++	++	+	+	+	-	-	-

Table 5.25: Grading of *Consistency*

After the analytical examination, replication strategies are further examined in a simulation. The simulation and its results are described in the next chapter.

[15] It is not sufficient for the client to directly update its cache to the new version, because the server could alter parts of the data item, depending on its business logic.

CHAPTER 6

Simulation

The performance of the replication strategies is measured in a simulation environment that emulates an IT system. The simulation environment contains prototypes of a client and a server which implement the replication strategies. Aim of the simulation is to examine characteristics that cannot be covered sufficiently by an exclusively analytical approach. In order to do so, the simulation focuses on the following questions:

1. *How do replication strategies perform under realistic conditions?* The analytical examination of quantitative indicators was simplified: client-side requests and server-side updates are observed in isolation or based on simple access patterns. The simulation uses complex access patterns to emulate realistic behavior.

2. *How does the distribution of requests and updates affect results?* These distributions mainly affect replication strategies which use caches (RS1, RS2, RS5) because their behavior varies depending whether the cache contains a data item or if it has to be requested from the server.

3. *How does the read/write ratio affect results?* The analytical examination indicates that replication strategies are either optimized for client-side read or server-side update operations. Simulations with different read/write ratios provide further insights into this trade-off.

4. *In which order of magnitude are the measured metrics?* The result of the analytical examination was a simple scale which indicates which replication strategies perform better or worse. The simulation provides concrete values that can be compared.

5. *How does the percentage of changed content affect the update results?* The implementation of replication strategies which distribute only state differences (RS4, RS7) are more complex than replication strategies that distribute the full state (RS3, RS6). The effect on the behavior of quantitative indicators is measured to conclude if there is a benefit of introducing this complexity into the IT system.

It is not observed how replication strategies perform related to the number of requests and updates per time. This behavior, especially the behavior of replication strategies under high load, is highly influenced by the configuration of the application server (e.g. when the server reaches its maximum capacity because a *Thread Pool* is fully utilized). It is a complex and application-specific task to find an optimal configuration which is therefore out of the scope of this thesis. It must also be assumed that applications are mainly utilized by other processes than replication and the utilization of the replication strategies is therefore a minor concern.

The simulation implements the use case presented in *Chapter 4*. It includes replication of a single entity type and no other functionality. All requests are handled by the client and all updates are handled by the server. The client itself provides no functionality to change data items and replicas are therefore read-only. Only the size of a data item differs from the description in *Chapter 4*: Where the previously defined use case uses two attributes (`data0` and `data1`), the simulation uses ten attributes (`data0` to `data9`) to enable changes of data items in steps of 10% of their total size.

The simulation performed only update operations. Insert and delete operations were not included to limit the scope and the time needed for the simulation and the evaluation of its results. Procedures for those operations are similar to update operations. The focus on update operations should therefore not influence the significance of the results.

Each replication strategy is used with different, exactly predefined scenarios inside a single simulation run. The scenarios mainly differ in the access patterns of request and update operations. The execution of particular request and update operations was performed with an *open system model* because it guarantees the execution of these operations at an exact point of time, independent of other requests [Sch06]. This guarantees reproducibility and validity of the results as well. The workloads viz. the access patterns are predefined and executed the exactly same way for each replication strategy.

The remainder of this chapter describes the simulation itself. In order to do so, its structure bases on the methodological approach presented in *Section 1.4*. The concrete implementation of the simulation environment and the prototypes of client, server and the replication strategies are described in *Section 6.1*. The workloads which constitute the basis of individual test runs are composed in *Section 6.2*. They base on the parameters defined in *Subsection 5.2.2* and are selected based on a survey on realistic access behavior. *Section 6.3* gives an overview over the execution of the simulation. It describes the concrete procedure that was used to run individual simulation runs, the environment in which the simulation was performed and how the validity was ensured. The chapter ends with the presentation of the measured values and their evaluation in *Section 6.4*.

6.1 Implementation

The simulation used a custom build simulation environment where requests and updates are executed with a custom build workload executer. The use of the existing benchmarking

tools (presented in *Subsection 2.4.2*) was examined but rejected because they do not support the required platforms (Java EE and Ruby on Rails) and are not able to execute a predefined workload in an open system model.

The presented simulation environment represents a specific environment for this thesis, the presented replication strategies and the specific use case. It is no goal to define a standard benchmark for replication or to meet the requirements for such benchmarks[1] defined by [Gra92].

The remainder of the section describes the implementation of the custom simulation environment in detail.

6.1.1 Architecture

The simulation environment consists of several independent components that run inside Docker containers. Docker was used to enable porting of the simulation environment into different platforms and environments. Figure 6.1 illustrates the structure of the simulation environment.

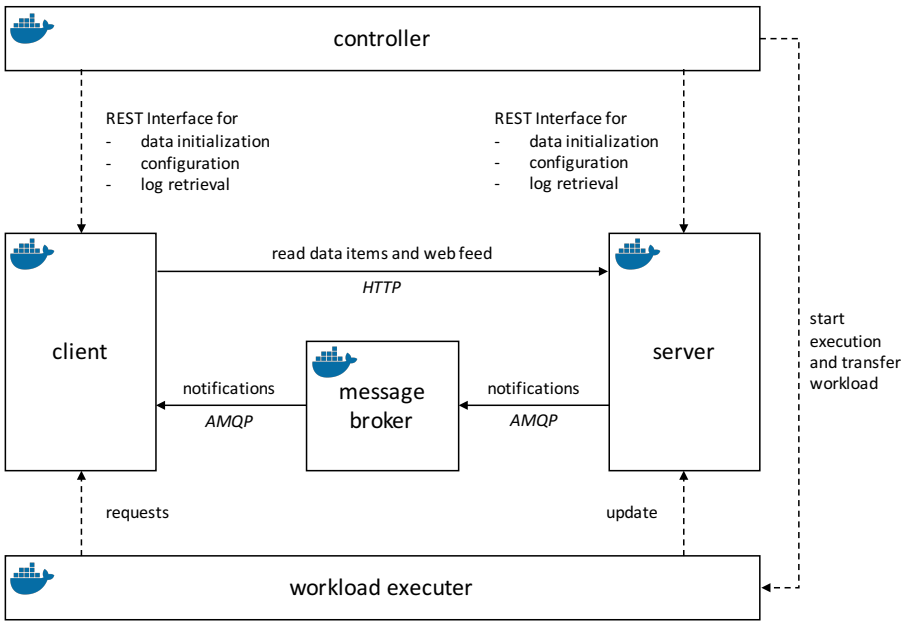


Figure 6.1: Structure of the simulation environment

The process is controlled by the controller component. All controlling communication is implemented by RESTful webservices over HTTP. Controlling communication is represented in the figure with dotted lines. Solid lines represent communication needed for replication which is performed with RESTful webservices and AMQP.

[1] These requirements are *Relevance*, *Portability*, *Scalability* and *Simplicity*.

Client-side requests and server-side updates are performed by the workload executer. The workload is stored at the controller and transferred to the workload executer when the execution is started. The workload executer independently performs all requests and updates. Client and server are implemented for Java EE and Ruby on Rails. Each of these two implementations is provided in a Docker container. A simulation run can use ether the Java implementation of a component or the Ruby implementation.

The next subsection describes the individual components in detail.

6.1.2 Components

The simulation environment consists of several independent components. Most components are presented in Figure 6.1. Additionally, the simulation environment includes a user interface for simple evaluation of the results and an *NTP* server for clock synchronization. The detailed description of the components is provided below:

Controller

The controller is the central component of the simulation environment and implemented as an independent Java EE application. It controls the procedure of a single simulation run (see *Subsection 6.1.3*) and gathers all metrics. The controller provides an interface to start a simulation run with a given configuration by a RESTful webservice. Additionally, it uses a *MariaDB*[2] database to store the definition of the available datasets and the results of the simulation runs. These data are accessible via RESTful webservices.

Workload Executer

The workload executer performs client-side requests and server-side updates based on a pre-defined workload. It provides a RESTful webservice by which the controller transfers a workload, starts the execution and retrieves information about the execution. The workload executer is implemented in *golang*[3] for performance reasons[4]. The implementation of the workload executer bases on a *busy wait loop* [Bli03] that checks for operations every 0.1ms and executes them concurrently[5]. Tests with different implementations showed that this approach ensures the execution of operations most accurately (the deviations are presented in *Subsection 6.3.3*).

[2] `https://mariadb.org/` (last checked 26.03.2018)

[3] `https://golang.org/` (last checked 26.03.2018)

[4] A first implementation in Java was discarded because the planned time of the execution of a request or update highly deviated from the point in time when the execution really took place. After a detailed analysis, the behavior was attributed to garbage collector time-outs of the *Java Virtual Machine*.

[5] In IT systems, busy waiting is bad practice because of its high resource utilization [Bli03]. The workload executer runs only in the simulation environment where it runs on an isolated machine (see *Subsection 6.3.1*) as well. Therefore, utilization is not a concern.

Client

The client handles requests sent from the workload executer. In order to do so, it implements the client-side processes presented in *Chapter 4*. It uses a cache and a full replica to store server-side data locally. The client also implements a synchronization job which reads a server-side web feed asynchronously and is able to subscribe to a queue on the message broker. These two mechanisms can be disabled independently so none of them interferes with other replication strategies. Additionally, the client provides RESTful webservices which the controller uses to control the simulation run (e.g. to set the configuration and reset the client).

The client is implemented in Java EE and Ruby on Rails. *Subsection 6.1.5* presents the technologies and libraries used for the particular implementations.

Server

The server handles updates sent from the workload executer. In order to do so, it implements the server-side processes presented in *Chapter 4*. It stores data in a data store which is accessible for the client via RESTful webservices. The server also provides a web feed via a HTTP interface and is able to send notifications over the message broker. These two mechanisms can be disabled independently so none of them interferes with other replication strategies. Additionally, the client provides RESTful webservices which the controller uses to control the simulation run (e.g. to set the configuration and reset the server).

The server is implemented in Java EE and Ruby on Rails. *Subsection 6.1.5* presents the technologies and libraries used for the particular implementations.

Message Broker

The simulation used RabbitMQ with its default configuration. The configuration was extended to provide a topic-based exchange, a persisted queue, a binding between the exchange and the queue and a user that can access them.

User Interface

The controller can be accessed by a simple *Angular JS*[6] application. This application visualizes the results of a simulation runs in tabular overviews and in diagrams implemented with *plotly.js*[7]. It can also be used to compare several simulations runs.

NTP Server

The simulation environment provides a *Network Time Protocol* (*NTP*) [RFC-5905] server for clock synchronization. The synchronization of the client-side and server-side clock

[6] `https://angularjs.org/` (last checked 26.03.2018)
[7] `https://plot.ly/javascript/` (last checked 26.03.2018)

is required for the calculation of the consistency indicator (see *Subsection 6.1.4*)[8]. It is provided in a independent Docker container and configured as an *Undisciplined Local Clock*[9] to run in an environment isolated from other time servers.

6.1.3 Procedure of a single Simulation Run

Each simulation run is executed in seven steps. The whole process is controlled by the controller component by the execution of RESTful webservices on the client, the server and the workload executer. Table 6.1 presents the individual steps of a simulation run. Operations that are only performed for particular replication strategies are annotated with the particular replication strategies.

1. **Configure**: The controller sets the configuration of the simulation run on the client and the server.
2. **Init**: The controller initializes the server-side data storage, based on the definition of the scenario in its database. For Event-Driven replication strategies with full replicas (RS3, RS4), data items are also created on the client-side to represent the initial data setup (see *Subsection 4.4.3*). For Poll-Based replication strategies, the server files each creation of a data item in a web feed as well.
3. **Start**: Depending on the used replication strategy, the client subscribes to the queue on the message broker or starts the synchronization job to read the server-side web feed. For Poll-Based replication strategies with a full replica, the client reads the events for the creation of data items and sets up its replica.
4. **Execution**: The controller starts the execution of the workload on the workload executer, based on the definition of the workload in its database. The workload executer processes the workload independently. Client and server perform the replication processes described in *Chapter 4*, depending on the used replication strategy.
5. **Stop**: Depending on the used replication strategy, the client unsubscribes from the message queue or stops the synchronization job which reads the web feed.
6. **Log**: The controller collects the results of the simulation run from client and server. *Subsection 6.1.4* describes which data points are gathered on the client-side and server-side. It saves these data in its local database.
7. **Reset**: At the end of the simulation run, client and server are reset. Therefore, all data items and logs are deleted locally.

Whenever an error occurs in one of the steps, client and server are stopped and reset.

[8] It is just required that the clocks are synchronized. The correctness of the time is not necessary.

[9] `https://support.ntp.org/bin/view/Support/UndisciplinedLocalClock` (last checked 26.03.2018)

	Controller	Executer	Client	Server
Configure			set configuration for simulation run	set configuration for simulation run
Init	read data sets		*RS3, RS4: create data items*	create data items *RS5, RS6, RS7: file creation events in web feed*
Start			*RS2, RS3, RS4: subscribe to message queue* *RS5, RS6, RS7: start sync. job*	
Execution	read workload	execute workload	do requests	do updates
Stop			*RS2, RS3, RS4: unsubscribe from message queue* *RS5, RS6, RS7: stop sync. job*	
Log	persist logs		hand out logs	hand out logs
Reset			truncate cache, full replica and logs	truncate data storage and logs

Table 6.1: Procedure of a single simulation run

6.1.4 Measurement and Calculation of Quantitative Indicators

To retrieve the quantitative indicators which are defined in *Subsection 5.2.1*, client and server log the time of particular events. Five timestamps are logged:

(a) **Start of request**: The timestamp of the start of the processing of a client-side request but after the request is interpreted and parsed by the application server.

(b) **End of request**: The timestamp of the end of the processing of a client-side request but before the response is transmitted over the network.

(c) **Start of update**: The timestamp of the start of the processing of a server-side update but after the request is interpreted and parsed by the application server.

(d) **End of update**: The timestamp of the end of the processing of a server-side update but before the response is transmitted over the network.

(e) **Time of notification**: The timestamp when a client is notified about a server-side update.

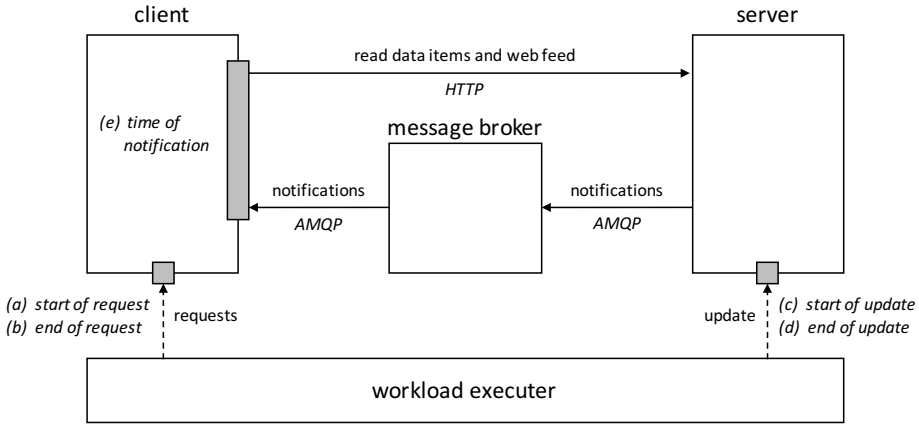


Figure 6.2: Points at which events are logged

Figure 6.2 illustrates where these data are logged. For each of these events, the time of occurrence with an accuracy of milliseconds is filed. The quantitative indicators are calculated based on these data. The calculation bases on the following methodology:

Client-Side Data Access Latency

The Client-Side Data Access Latency is calculated as the difference between the start *(a)* and the end *(b)* of the processing of a client-side request.

Server-Side Data Update Latency

The Server-Side Data Update Latency is calculated as the difference between the start *(c)* and the end *(d)* of the processing of a server-side update.

Bandwidth Consumption

Data transferred over the network is monitored on the client-side and the server-side for particular communication channels (see the examination of Fault Tolerance in *Subsection 5.1.3*) with *tcpdump*[10]. In order to do so, tcpdump is configured to monitor only packets for a particular IP-address and port. The monitored data is interpreted and exported with *capinfos*[11]. Figure 6.3 illustrates where communication data is monitored.

The total bandwidth consumption is calculated as the sum of the data received from the client (at the server-side), the data sent to the message broker (on the server-side) and the data received from the message broker (at the client-side).

Consistency

The time until the replica is in a consistent state is calculate as the time between the start of a server-side update *(c)* and the time the client is notified about a server-side update *(e)*.

[10] `https://www.tcpdump.org/` (last checked 26.03.2018)
[11] `https://www.wireshark.org/docs/man-pages/capinfos.html` (last checked 26.03.2018)

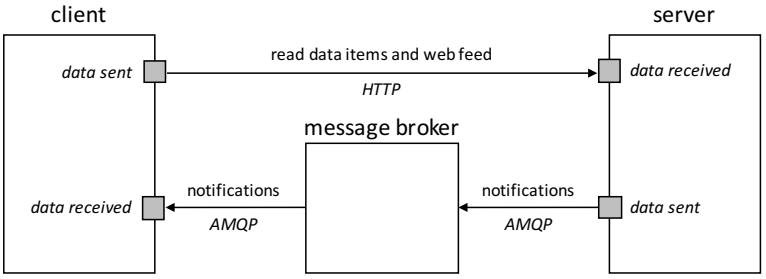


Figure 6.3: Points at which data transfer is monitored

The data items on the client-side are only updated when the client uses a full replica (RS3, RS4, RS6, RS7). For replication strategies that use invalidation (RS2, RS5), the time until the replica is in a consistent state is defined as the time in which a data item is removed from the cache. This indicator is not measured for replication strategies that provide strict consistency (RS0, RS1).

Because the calculation bases on times measured on the client-side and the server-side and these times need to be comparable, clock synchronization with a high level of accuracy is required. The verification of the accuracy of the clocks is described in *Subsection 6.3.3*.

6.1.5 Technologies

Table 6.2 shows which technologies and particular versions were used for the implementation of client and server and Table 6.3 shows these information for the simulation.

	Java	**Ruby**
Platform	OpenJDK 1.8.0	Ruby 2.4.2
Framework	Java Enterprise Edition 7	Rails 5.1.4
Server	Wildfly 10.1.0.Final	Puma 3.10.0
REST		
Server	JAX-RS with Resteasy 3.0.19.Final	-
Client	JAX-RS with Resteasy 3.0.19.Final	rest-client 2.0.2
Messaging		
Server	RabbitMQ AMQP client 4.2.0	bunny 2.7.1
Client	RabbitMQ AMQP client 4.2.0	sneakers 2.6.0
Job Scheduler	*ManagedExecutorService*	ActiveJob 5.1.4
Persistence	*ConcurrentHashMap*	MariaDb 10.2.11

Table 6.2: Technologies used for the implementation of client and server

	Java
Executer	Golang 1.9
Message Broker	rabbitmq-server-3.6.11-1.el7
NTP (client and server)	ntp 4.2.6p5
Operating System	Centos 7.4.1708 Mini
Container	Docker 17.12.0-ce
Network Monitor	tcpdump

Table 6.3: Technologies used for the implementation of the simulation environment

6.1.6 Limitations

There are some known limitations in the implementation of client and server:

- *Cache*: A custom implementation is used for the client-side cache. It provides no optimizations for concurrent access (see *Subsection 4.3.1*). It is assumed that the use of existing cache solutions would improve the behavior of RS1, RS2 and RS5.
- *No usage of JMS*: To ensure that the communication to the message broker is only enabled for particular replication strategies, the Java implementation of client and server use the native AMQP client provided by RabbitMQ. It is assumed that the use of JMS would improve the behavior of Event-Driven replication strategies.
- *Caching of web feeds*: [Web10] proposes the use of caching headers for web feeds. It must be assumed that the use of these headers would not affect the results of the simulation because web feeds change on every access of the client.
- *Use of different storage technologies in Java and Ruby implementations*: The Java implementation stores data items, logged timestamps and the configuration of the simulation run inside the main memory (in a Java `ConcurrentHashMap`) whereas the Ruby implementation uses a relational database. Access of the main memory is faster than access on the database whereby the Java implementation is generally faster. This is evident in the results of the simulation (see *Section 6.4*).

6.2 Scenarios and Workloads

The individual client-side requests and server-side updates constitute the workload. It can be varied to emulate different use cases. Together with the characteristics of the data items, the workload constitutes the variable part of the simulation which can be changed to obtain characteristics of the replication strategies.

Because the simulation implements a custom use case that was defined in the course of this thesis there are no real traces available and a synthetic workload [Fei15] [Jai90] has to be built. There exist two approaches to build a *synthetic workload* [Fei15]: *empirically* (based on real traces) and *analytically* (based on a mathematical model). Even though

empirical workloads are more realistic [Fei15], it is hard to port them into another environment [Fei15] and to change particular parameters [Bah11] [Bus02]. Additionally, these workloads build on traces of specific applications or use cases and are therefore only valid inside this environment [Fei15]. This contradicts the aim of the simulation which is to examine general characteristics. The simulation therefore uses an analytical approach to build synthetic workloads which provides a high flexibility to change parameters in order to constitute workloads with different characteristics [Bah11] [Bus02]. The model to build the workloads bases on the parameters described in *Subsection 5.2.2*.

The analytical creation generally introduces the risk that the workload is not representative [Fei15]. The correctness of the simulation results and their application in real systems therefore depends mainly on the values chosen for the parameters. The simulation follows the approach to find realistic values and maps these values on realistic use cases afterwards. It is assumed that this approach, ensures a realistic representation of real systems on the one hand and provides most insights into the characteristics of particular replication strategies on the other hand. To do so, *Subsection 6.2.1* surveys the parameters and chooses realistic values based on a literature survey and a set of general assumptions. *Subsection 6.2.2* composes realistic use cases from these values.

6.2.1 Values

Concrete values for the parameters presented in *Subsection 5.2.2* were selected based on a literature research and a set of general assumptions. Each parameter was examined independently. Additional to the defined parameters in *Subsection 5.2.2*, specific parameters for Poll-Based replication strategies and the duration of particular simulation runs are examined and selected in this subsection.

Number of data items

Often, IT systems contain millions of data items. A high number of data items therefore leads to more realistic results. Because replication affects only a single data item, observations on a single data item are independent of their total number. On the other hand, fewer data items enable more operations for a single data item in the same amount of time. It can therefore be assumed that a lower number leads to more significant results.

Even though a high number of data items is more realistic, the number of data items mainly affects latency for read operations on the persistence layer (see *Subsection 5.2.2*), which are explicitly not covered by this thesis (see *Subsection 1.3.2*). Varying the total number therefore leads to no insights into the replication strategies themselves. A smaller number of data items was chosen in favor for more significant simulation results. After several manual tests with different quantities, a value of 100 data items was chosen.

Data size distribution

The simulation uses the same, fixed size for all data items. The size of a data item mainly influences the consumed bandwidth. The impact of the size of a single data item on the

overall bandwidth consumption is relative to the overhead of communication itself and the protocol used (e.g. TCP/IP, HTTP, AMQP, Atom metadata). A variation of the data size therefore mainly provides insights into the overhead of the network protocols in general (REST over HTTP versus AMQP versus Atom over HTTP) which constitutes its own field of research. Therefore, a fixed data size was chosen because it enables better comparison of the results.

Size of a data item

The size of a data item depends on the data it represents. An assumption which is valid for all cases of applications is therefore impossible. The mean size was set in consultation with the representatives of the IT system of TU Wien to 1000 alphanumerical ASCII characters. With the use of UTF-8, this corresponds to 1000 Bytes [RFC-3629]. This size is also used by the YCSB benchmark [Coo10].

Distribution of Requests over Data Items

Several surveys showed that the access patterns on websites follow a Zipf or Zipf-like distribution [Ada02] [Bre99] [Cal16] [Lev01] [Men00] [Wil05]. It is therefore mainly used to build synthetic workloads for IT systems (e.g. YCSB [Coo10] and TCP-W [Gar03]).

The Zipf-distribution [Fei15] was originally observed in the distribution of words in natural language [Zip32] [Zip49]. It proposes that the frequency of one word is proportional to the power inverse of its rank. In general, the Zipf-distribution describes a distribution in which high-ranked elements occur proportionally more frequently than low-ranked elements. The highest ranked element will occur twice as often as the second highest ranked element and three times as often as the third highest ranked element. *Figure 6.4* illustrates a example of a Zipf-distribution.

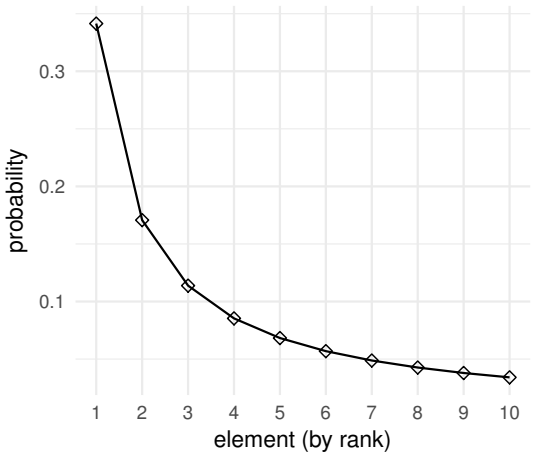


Figure 6.4: Probability density function of a Zipf-distribution with exponent $s = 1$

The default value for the distribution of requests is chosen as a Zipf-distribution with an exponent $s = 1$. Additionally, a uniform distribution is used. This distribution does not represent a widely used access pattern but it is assumed that this simple distribution is easy to evaluate and provides further insights into the behavior of particular replication strategies.

Distribution of Requests over Time

Requests are equally distributed over time. This does not represent realistic behavior because real systems are facing a changing number of requests over time and also face peak loads. Equal distribution of requests simplifies the evaluation of results and was therefore chosen to ensure the correctness of the evaluation. Furthermore, the examination of the behavior of replication strategies referring to the number of requests per time was already rejected in the introduction of this chapter.

Request interval

Based on the discussion above, each simulation run uses a fixed request interval. The concrete number of requests per time differs in real IT systems between individual applications and inside a particular application between individual use cases and entity types. A general assumption is therefore impossible.

Based on the examination in *Subsection 5.2.3*, the request interval mainly effects read operations on the persistence layer. These effects must not be considered because the persistence layer is explicitly not covered by this thesis (see *Subsection 1.3.2*). Therefore, it is assumed that the validity of the simulation results is not affected even if a fixed interval is used for all simulation runs. This is beneficial because it enables a better comparison of the results.

After several manual tests with different request intervals, an interval of 10 requests per second was chosen because it provided a high number of measurement points and can be handled by the standard configuration of the application servers.

Distribution of Updates over Data Items

For the distribution of updates, the Zipf-distribution and the uniform distribution are used. This choice bases on the same assumption as the distribution of requests over data items described above.

Update interval

The simulation uses different update intervals to observe the behavior of replication strategies with respect to the read/write ratio. In order to do so, update intervals are calculated using the fixed read interval (10 requests per second) and different read/write ratios. Ratios are selected based on values used in existing benchmarks (presented in Table 6.4) and data published for existing websites (presented in Table 6.5).

Benchmark	Mode	Read (%)	Write (%)	Ratio
TPC-W [Gar03]	browsing	95.00	5.00	19/1
	shopping	80.00	20.00	4/1
	ordering	50.00	50.00	1/1
YCSB [YCSB]	update heavy	50.00	50.00	1/1
	read heavy	95.00	5.00	19/1
	ready only	100.00	0.00	-
RUBBos [Amz02]		85.00	15.00	5,66/1
RUBis [Amz02]	browsing	100.00	0.00	-
	bidding	85.00	15.00	5,66/1
SPECWeb99 [Nah02]		95.06	4.92	19.32/1

Table 6.4: Read/write ratios of the benchmarks presented in 2.4.2

Website	Year	GET (%)	POST (%)	Ratio
Slashdot [Amz02]	2001	99.50	0.50	199/1
Small Research Institute [Cal10]	2006-2009	90.00	10.00	9/1
Youtube [Gil07]	2007	99.87	0.12	83.191/1
South Korean Blog Hosting Site [Jeo12]	2012	97.35	2.55	3.817/1

Table 6.5: Read/write ratios published for existing websites (used under the assumption that every GET request represents a read operation and every POST requests represents a write operation)

The ratios presented in the tables above show that ratios measured in real applications are more read-intensive than the ratios used in benchmarks. This assumption corresponds to the experience of the representatives of the IT system of TU Wien. The simulation therefore uses more read-intensive ratios in order to correspond to real access behavior. Table 6.6 shows the chosen ratios and the resulting update intervals.

Read (%)	Write (%)	Read/Write Ratio	Update Interval
80.00	20.00	4/1	400ms
95.00	5.00	19/1	1,900ms
99.50	0.50	199/1	19,900ms

Table 6.6: Read/write ratios used in the simulation

The representatives of the IT system of TU Wien observed even more read-intensive ratios. Tests showed that more read-intensive ratios lead to the same results as 1/199 and therefore provide no additional information. As a consequence, the most read-intensive ratio used was 1/199.

Distribution of Updates over Time

Updates are equally distributed over time. This choice bases on the same assumption as the distribution of requests over time described above.

Update size distribution

The size of an update is equal to the percentage of the content of a data item that was changed by an update operation. A data item in the simulation environment consists of 10 attributes, each with the same size. Each update changes the same percentage of the data in a data item. This does not represent realistic behavior but simplifies the evaluation of the results and was therefore chosen to ensure the correctness of the evaluation. To observe the behavior with different change rates, the percentage is changed for particular simulation runs (see below).

Update size of change

No examinations in this area were found in the course of the literature research. The percentage was therefore chosen in consultation with the representatives of the IT system of TU Wien which lead to the following values:

- **10%**: Represents a small change. This percentage is also used in the YCSB benchmark [Coo10]. With the previously chosen data size of 1000 Bytes, a change of 10% = 100 Bytes = 100 Chars may represent the change of the billing address of an order.
- **50%**: Represents a freely chosen percentage between 10% and 100%.
- **100%**: Is a change of the whole data item. This may represent the change of a blog post if the whole text is replaced.

Poll-Based Parameters

Poll-Based replication strategies need two additional configuration parameters (see *Section 4.5*): The client has to define the interval in which it reads the web feed and the server has to define the length of the recent web feed. These parameters highly influence the consistency and the bandwidth consumption.

The **client-side interval** is set to one minute. This ensures that the maximum time of inconsistencies on the client-side is not more than one minute, which is assumed to be a realistic value for most IT systems.

The **server-side web feed length** was optimized for least bandwidth consumption. The optimal value depends on the number of events filed in the web feed between two

client accesses of the web feed. It therefore depends on the client-side interval and the update interval. After a number of tests with different settings, the values presented in Table 6.7 were selected.

Update Interval	Updates per Minute	Web Feed Length
400ms	150.00	180
1,900ms	31.57	40
19,900ms	3.02	5

Table 6.7: Web feed lengths used in the simulation

Duration of a Simulation Run

Another adjustable value is the duration of a simulation run itself. It is a compromise between the number of measured values (and therefore the quality of the results) and the feasibility of a high number of simulation runs (because there is only a limited amount of time to run the simulation). After several manual tests, a duration of 30 minutes was chosen. This time guarantees that all tests can be executed and ensures meaningful results. A simulation run of 30 minutes consists of 18,000 requests and (depending on the read/write ratio) 4,500, 947 or 90 updates.

6.2.2 Definition of Workloads and Use Cases

The survey in the last subsection provides realistic values for the parameters. These values are used to compose concrete workloads in this subsection. The workloads are optimized to emphasize different behaviors of replication strategies under a number of conditions (e.g. with different read/write ratios and different distribution of access patterns). Table 6.8 presents the composed workloads.

	Read/Write Ratio	Change %	Distribution	Used with Replication Strategies
W1	80.0/20.0	100%	zipf	RS0-7
W2	95.0/5.0	100%	zipf	RS0-7
W3	99.5/0.5	100%	zipf	RS0-7
W4	95.0/5.0	100%	uniform	RS0-7
W5	95.0/5.0	10%	zipf	RS3, RS4, RS6, RS7
W6	95.0/5.0	50%	zipf	RS3, RS4, RS6, RS7

Table 6.8: Workloads used in the simulation

Workload W2 was chosen as the baseline for the evaluation. For the other workloads, there was only a single parameter varied to the definition of W2. Because only replication strategies that distribute state differences (RS4, RS7) are affected by the percentage of content changed within an update operation, these are mainly used for workloads where the parameter varies. Replication strategies that distribute the full state (RS3, RS6) are used as a reference.

The workloads can be mapped to real data and use cases in IT systems:

W1 Represents uses case were data is read daily and updated once per week[12], e.g. tours in a logistic company that change in average once per week and are read once every day to calculate the tour and output the tour plans for the drivers.

W2 Represents the personal user information (containing a personal text, interests, etc.) in an online application. These data are read very infrequently and changed even more infrequently.

W3 Represents the contact information (containing a telephone number, e-mail address, etc.) inside a company portal (e.g. TISS for TU Wien). These data are read infrequently and rarely changed.

W4 Represents the personal information of an employee (containing a description of experience, interests, etc.) inside the IT system of a company. These data are used by the human resources department to regularly create promotional material.

W5 Represents the user information of a customer in an e-commerce platform where the ordering address changes for every 19th order.

W6 Represents the personal user information (containing a personal text, interests, etc.) in an online application. These data are read very infrequently and changed even more infrequently. Data is stored in structural form and a change of 50% represents the change of the personal text.

Most real use cases are more read-intensive than the ratio of 99.5/0.5 and are therefore not covered by these workloads (e.g. personal data like the name of a lecturer in the IT system of TU Wien is read very frequently but rarely changed). Because the read/write ratio of 99.5/0.5 only leads to 90 updates per simulation run (while leading to 19.000 requests), a more read-intensive ratio would lead to hardly any write operations (which decreases the significance of the results) or would extend the time for a simulation run enormously. Test showed that the results of simulation runs with a more read-intensive ratio behave similar[13]. The read/write ratio of 99.5/0.5 can therefore be considered as "read/write ratio of 99.5/0.5 or higher".

[12] Data that changes weekly and is read daily would actually having a read/write ratio of 1/5. Considering holidays and other exceptional cases, a read/write ratio of 1/4 is assumed.

[13] The results are more significant but follow the same trend.

6.3 Execution

The final simulation was executed once with 40 independent simulation runs. In consultation with the representatives of the IT system of TU Wien, the Java EE client implementation and the Ruby on Rails server implementation were used because this setting provides most information to them.

This section provide insights in the execution of the simulation runs. *Subsection 6.3.1* describes the environment where the simulation was executed, *Subsection 6.3.2* presents the concrete process how individual simulation runs where executed and *Subsection 6.3.3* describes which verification processes were performed to ensure the correctness of the results.

6.3.1 Environment

The simulation was performed in an isolated environment with no access to the internet or other WANs. The setup consisted of four *HP DL380* servers of the sixth generation with two XEON CPUs and 72 GByte of RAM each and a single *MacBook Pro* (Retina, 13-inch, Early 2015) which are connected over Gigabit Ethernet. Figure 6.5 illustrates how the components viz. the Docker containers were distributed over the machines.

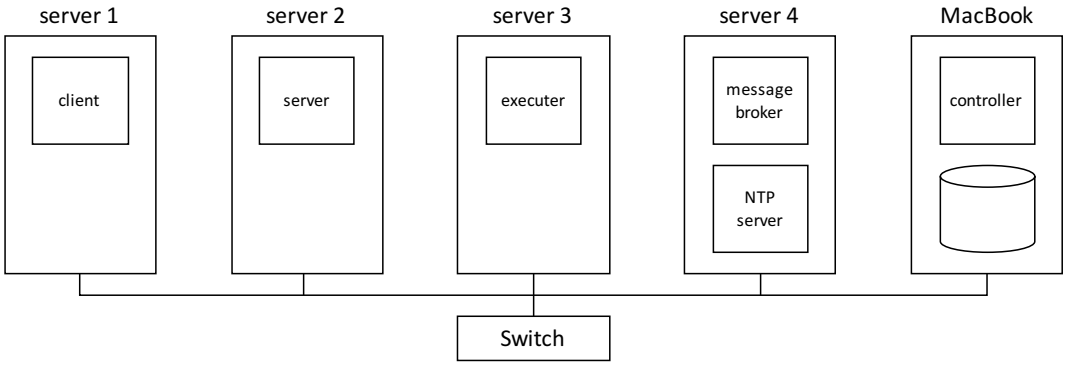


Figure 6.5: Structure of the environment where the simulation was performed

Client and server represent the *system under test* and therefore run on independent machines. To ensure accuracy of the execution of client-side requests and server-side updates, the executer also runs on an independent machine. Because the NTP server utilizes only few resources it was executed on the same machine as the message broker. The controller manages the process but implements no time-critical operations. Therefore, it was executed on the MacBook which has the lowest performance of all machines. The execution of the controller on a desktop machine also enabled better monitoring of the simulation progress. The MacBook also stored the database which persists the results of the simulation runs.

6.3.2 Procedure of the Simulation

To ensure the isolation and reproducibility of particular simulation runs, the whole simulation environment was reset after each simulation run. In order to do so, each simulation run consisted of the following four steps:

1. **Restart**: Restart of all components by restarting the Docker containers (including a wait phase until all components are started). This includes the client, server and message broker.
2. **Warm-up for push-based replication**: Execution of a test set with 250 reads and 50 writes with RS3.
3. **Warm-up for pull-based replication**: Execution of a test set with 250 reads and 50 writes with RS6.
4. **Execution**: Execution of the real workload with a particular replication strategy by the procedure described in *Subsection 6.1.3*.

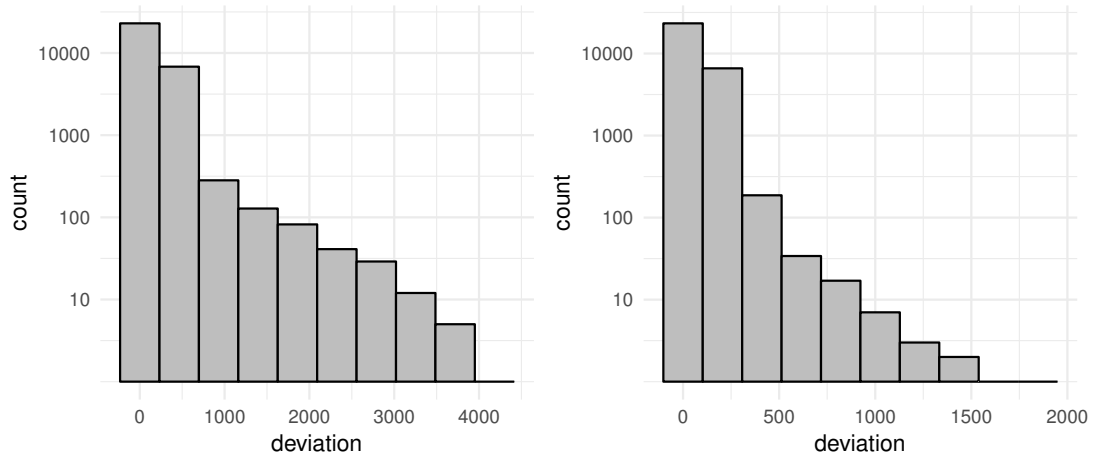
The procedure was scripted with *shell-scripts* that access the RESTful webservices of the controller and directly restarted the Docker containers on the remote machines.

6.3.3 Verification

To ensure the correctness of the implemented replication processes and the measured values, several verification processes were performed before and during the simulation:

- **Manuel observations**: The behavior of the replication strategies was observed prior to the final simulation runs based on detailed logging of all processes with simple test sets. These observations provide deep insights in the internal processes and their correct behavior.
- **System tests**: The simulation environment provides a test suite with approximately 100 system tests for the server and another approximately 100 system tests for the client which cover all replication processes and all processes performed by the controller. It ensures the correctness of the implementations and the compatibility of the Java and of the Ruby version.
- **Analytical examination**: The analytical examination in *Section 5.2* provided an estimation of the expected values. To ensure the correctness of the implementation, results of the simulation were compared with these estimated values.
- **Influence of network monitoring**: The simulation environment includes a toggle to disable the monitoring of bandwidth consumption. Prior to the final simulation runs, the simulation environment was tested with and without network monitoring to ensure that the monitoring mechanism has no influence on the measurement of latencies.
- **Warm-up phases**: The warm-up phases of the application servers and the message broker were observed within prior test runs. The results of these tests provided the basis for the warm-up phases described in *Subsection 6.3.2*.

- **Clock deviation**: The accuracy of clock synchronization was ensured during the simulation by logging of the deviation of the client-side clock and the server-side clock during all simulation runs. The results are presented in Figure 6.6. The absolute value of the mean deviation was 0.128ms.



(a) client-side clock (b) server-side clock

Figure 6.6: Deviation of clock synchronization in logarithmic scale (in microseconds)

[Wat15] stated that NTP provides only an accuracy of 1 to 100 milliseconds. It must be assumed that the better performance measured during the simulation is caused by the use of NTP in a local and isolated environment with short paths from the NTP server to the other components.

- **Delay of workload execution**: The deviation of the planned execution of operations to the real execution (*delay*) was logged during the final simulation. Figure 6.7 illustrates the deviation. It shows that the mean deviation was 0.16ms.

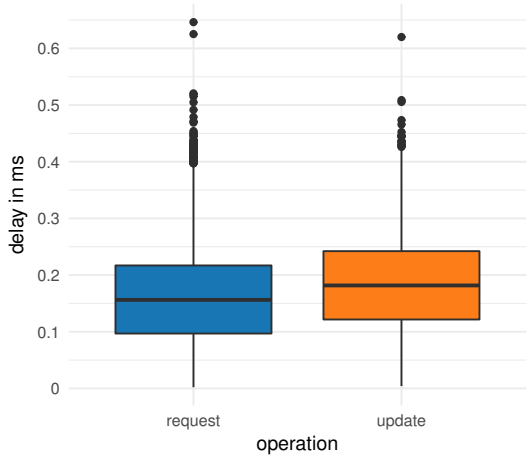


Figure 6.7: Delay of the execution of operation

6.4 Results

The data measured during the simulation was evaluated inside the database, the Angular-based user interface and further processed with R[14]. This chapter presents the results of the evaluation and their interpretation based on the quantitative indicators.

6.4.1 Client-Side Data Access Latency

Client-Side Data Access Latency was measured on the client (see *Subsection 6.1.4*). Table 6.9 shows the results of all simulation runs (in milliseconds).

	RS0	**RS1**	**RS2**	**RS3**	**RS4**	**RS5**	**RS6**	**RS7**
W1	25.04	24.77	5.50	0.01	0.01	2.43	0.01	0.01
W2	25.36	23.51	1.57	0.02	0.01	1.05	0.00	0.01
W3	23.70	22.82	0.38	0.01	0.01	0.36	0.00	0.00
W4	24.34	23.43	1.58	0.00	0.01	1.55	0.01	0.00
W5	-	-	-	0.01	0.02	-	0.01	0.00
W6	-	-	-	0.01	0.01	-	0.00	0.01

Table 6.9: Client-Side Data Access Latency

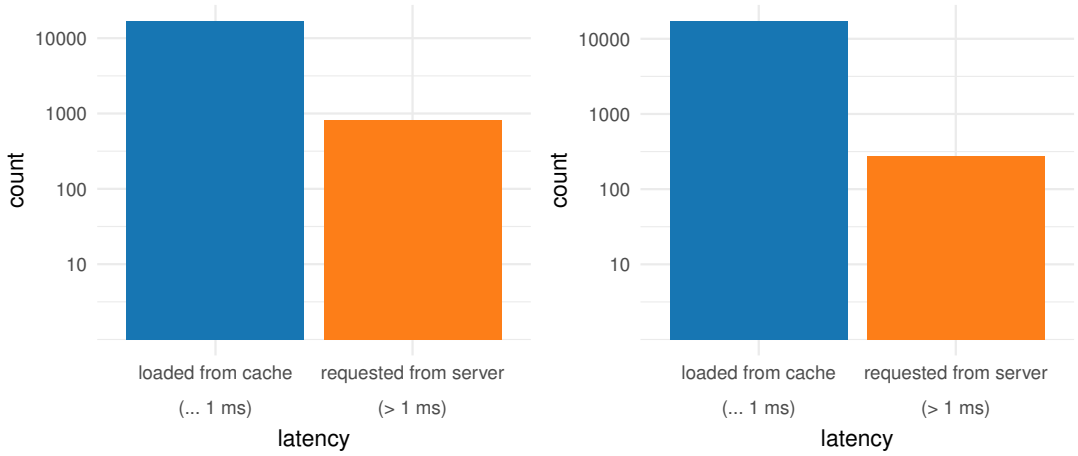
The data show that Client-Side Data Access Latency performed as examined in the analytical examination in *Subsection 5.2.3*. It also shows that this latency highly depends on weather data is read locally or has to be requested from the server. Different to what was expected during the analytical examination, there is less read latency for RS1 than for RS0. It was assumed that the access to the cache needs more time than the transfer of the contents of a data item. In reality, the results showed that read operations in the cache are faster than the data transfer.

Furthermore, the data show that latencies for replication strategies that access the server on every request (RS0, RS1) and the ones that use a full replica (RS3, RS4, RS6, RS7) are independent of the read/write ratio and the size of a change[15]. It is assumed that these latencies are always the same, independent of the workload.

The latency of replication strategies which use cache validation (RS2, RS5) depends on the existence of a data item in the cache. This dependency was already described in the analytical examination. Table 6.9 shows that read latency increases when data are changed more often and therefore must be requested more often from the server as well. Figure 6.8 illustrates how read latency is distributed for RS2 and RS5 and a Zipf-distributed request access pattern (W2).

[14] `https://www.r-project.org/` (last checked 26.03.2018)

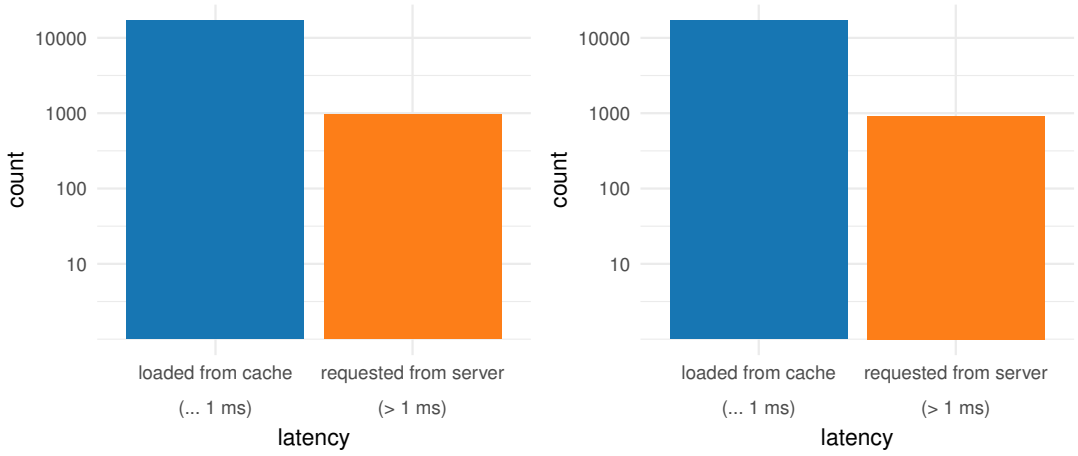
[15] The deviation of the measured values for RS0 and RS1 is attributed to fluctuations in network latency between client and server.



(a) RS2 (b) RS5

Figure 6.8: Distribution of read latency for a Zipf-distributed request access pattern

The better performance of RS5 compared to RS2 can be attributed to the worse level of consistency of RS5: the cache is invalidated less often and therefore data has to be requested less often as well. RS2 invalidates its cache for every new version and (depending on the access pattern) has to request each version from the server as well. RS5 invalidates only once per minute and therefore only has to request the most recent version after the invalidation. This difference is particularly pronounced with a Zipf-distributed access pattern, because a small part of the data items changes very often. Figure 6.9 shows the latencies for a uniform distributed request access pattern where the difference is less pronounced.



(a) RS2 (b) RS5

Figure 6.9: Distribution of read latency for a uniformly-distributed request access pattern

6.4.2 Server-Side Data Update Latency

Server-Side Data Update Latency was measured on the server (see *Subsection 6.1.4*). Table 6.10 shows the results of all simulation runs (in milliseconds).

	RS0	**RS1**	**RS2**	**RS3**	**RS4**	**RS5**	**RS6**	**RS7**
W1	16.12	15.92	21.15	20.48	20.88	26.04	25.10	26.66
W2	15.55	15.43	20.29	21.13	20.83	22.43	23.31	23.86
W3	15.14	14.26	23.66	24.88	23.42	23.59	27.81	27.36
W4	15.65	15.46	20.55	21.09	20.98	22.45	22.88	24.19
W5	-	-	-	20.82	20.76	-	23.42	21.95
W6	-	-	-	20.70	21.04	-	23.46	23.15

Table 6.10: Server-Side Data Update Latency

Replications strategies that use no notification mechanism (RS0, RS1) perform updates with the smallest latency. These replication strategies only perform local write operations and it was therefore already expected in the analytical examination in *Subsection 5.2.4* that they will out-perform the others. Notifications over messaging (RS2, RS3, RS4) and the filing of changes in a web feed (RS5, RS6, RS7) increase latency. Different to the results of the analytical examination, updates for Poll-Based replication strategies take a little longer than for Event-Driven replication strategies. It was expected that the communication to the message broker for client notifications takes longer than filing a change as an event in the web feed. In reality, it turned out the other way around.

6.4.3 Bandwidth Consumption

Bandwidth consumption was monitored at four independent points. The particular points of monitoring and the calculation of total bandwidth consumption were presented in *Subsection 6.1.4*. Table 6.11 shows the results of all simulation runs (in KB).

	RS0	**RS1**	**RS2**	**RS3**	**RS4**	**RS5**	**RS6**	**RS7**
W1	41,281	26,376	20,124	21,424	21,424	7,855	12,367	12,338
W2	41,280	23,248	4,784	4,517	4,517	2,645	2,972	2,908
W3	41,277	22,275	671	441	441	807	591	593
W4	41,298	23,294	4,829	4,517	4,517	3,449	2,976	2,909
W5	-	-	-	4,517	2,670	-	2,908	1,337
W6	-	-	-	4,517	3,491	-	2,905	2,020

Table 6.11: Bandwidth Consumption

Table 6.11 shows that most data is transferred when no replication is used (RS0). It also shows that bandwidth consumption for Event-Driven and Poll-Based replication strategies highly depend on the number of server-side update operations. A detailed evaluation is done based on the characteristics of the replications strategies.

Event-Driven versus Poll-Based

Event-Driven replication strategies transfer more data than Poll-Based replication strategies. This is due to the implementation of Event-Driven replication strategies with messaging for which data is transmitted two times: from the server to the message broker and from the message broker to the client. Table 6.12 illustrates the data transferred over particular communication channels (in KB)[16].

		RS3			**RS4**	
	RS0	**RS1**	**RS2**	**RS3**	**RS4**	**RS5**
W1	6,181	15,242	21,424	6,181	15,242	21,424
W2	1,307	3,210	4,517	1,307	3,210	4,517
W3	132	309	441	132	309	441
W4	1,307	3,210	4,517	1,307	3,210	4,517
W5	1,307	3,210	4,517	383	2,286	2,670
W6	1,307	3,210	4,517	794	2,697	3,491

Table 6.12: Comparison of bandwidth consumption for Event-Driven and Poll-Based replication strategies

The table indicates that the messages sent from the message broker to the client are larger than the messages sent from the server to the message broker on which they are based on. This observation is attributed the specific behavior of the AMQP protocol and was not further analyzed.

The observation that Poll-Based replication strategies transfer less data than Event-Driven replication strategies can be considered a special case of this simulation and the chosen parameters. Poll-Based replication strategies were optimized for minimal bandwidth consumption and a comparison is therefore not representative. It is assumed that Poll-Based replication strategies would transfer as much data as Event-Driven replication strategies (by whom the consumed bandwidth is independent of other parameters) or even more.

[16] The data are results of twelve independent simulation runs. It is conspicuous that the data transfer of simulation runs that use the same number of updates is constant.

Full State versus State Difference

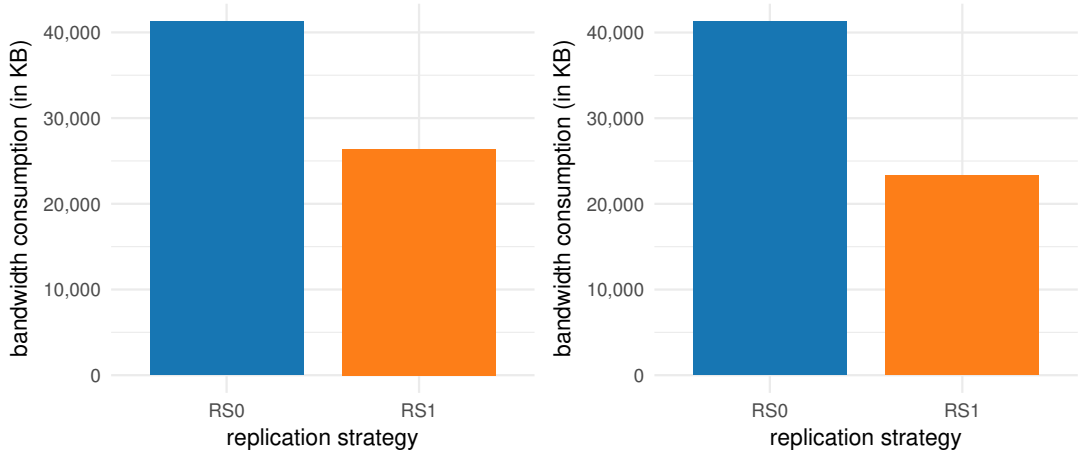
The transmission of state differences decreases the transferred data compared to the transmission of the full state. The decrease is proportionally lower than the change of content within a data item. This observation is attributed to the overhead of the used communication protocols and the metadata of the replication process (e.g. from AMQP and Atom) which is independent of the size of the content. Table 6.13 shows the percentage of change in data and the changes in the transferred data (in KB).

	Changes	RS3	RS4	%	RS3	RS4	%
W2	100%	4,517	4,517	100.00	2,972	2,908	97.85
W6	50%	4,517	3,491	77.29	2,905	2,020	69.54
W5	10%	4,517	2,670	59.11	2,908	1,337	45.98

Table 6.13: Comparison of bandwidth consumption for change distribution by full state or state differences

Cache Validation

Cache Validation (RS1) requires a request to the server on every client-side data access. The response contains only data when the data item changes in the server-side data store. This leads to a decrease of the consumed bandwidth compared to the use of RS0 from approximately 41MB to approximately 25MB. Figure 6.10 shows the relation of the transferred data when RS0 and RS1 are used with W1 and W3.



(a) write-intensive workload (W1) (b) write-intensive workload (W2)

Figure 6.10: Impact of the use of a cache on bandwidth consumption

Figure 6.11 shows that the bandwidth consumption varies for particular read/write ratios from approximately 22MB to approximately 26MB. It was assumed that the variation of the read/write ratio from 1/4 to 1/199 would influence bandwidth consumption even more.

Therefore, it must be concluded that most bandwidth is consumed by the fundamental communication and the content of the data item viz. the "real" data (1000 Bytes) only represents a small part of the total bandwidth consumption.

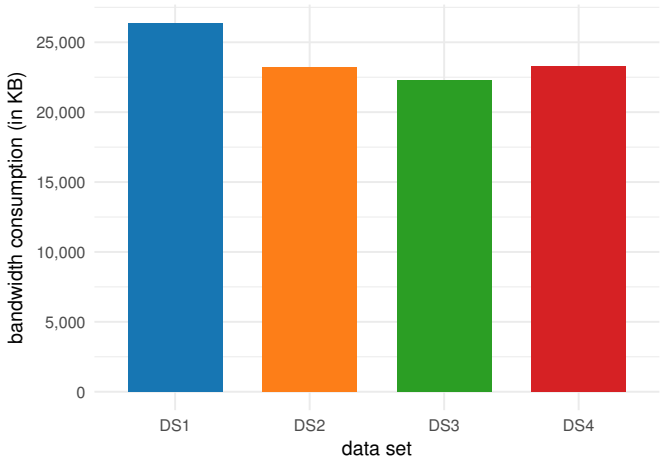


Figure 6.11: Bandwidth consumption of RS1 with different workloads

6.4.4 Consistency

Consistency was measured as the time it takes from a server-side change until the notification of the client (see *Subsection 6.1.4*). Table 6.14 shows the results of all simulation runs (in milliseconds).

	RS0	**RS1**	**RS2**	**RS3**	**RS4**	**RS5**	**RS6**	**RS7**
W1	-	-	16	16	16	31,709	31,705	31,697
W2	-	-	16	16	16	30,533	30,525	30,379
W3	-	-	19	20	18	26,009	27,032	26,504
W4	-	-	16	17	17	30,469	30,479	30,351
W5	-	-	-	16	16	-	30,403	30,440
W6	-	-	-	16	16	-	30,400	30,405

Table 6.14: Time until a client was notified about a server-side change

The data shows that the duration of inconsistencies mainly depends on the used notification mechanism, which corresponds to the results of the analytical examination. Event-Driven replication strategies notify the client on average in less than 20ms whereas the notification for Poll-Based replication strategies takes on average about 30 seconds. This timespan depends on the interval in which the client reads the web feed (which was set to one minute in the simulation). The time until a client is notified with Poll-Based

replication strategies takes between some milliseconds (if the change was performed directly before the web feed is read) and 60 seconds (if the change was performed directly after the last time the web feed was read). Figure 6.12 shows the times measured for each update operation for RS6 and W1.

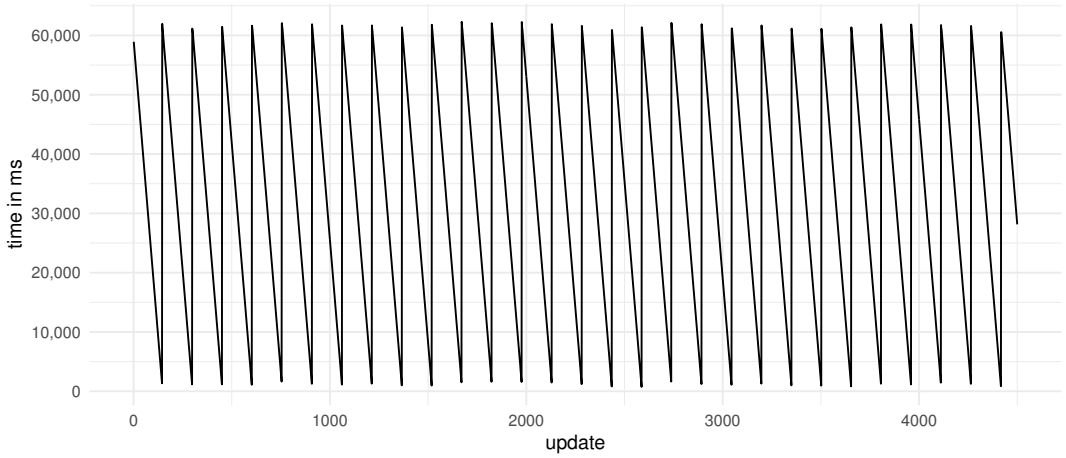


Figure 6.12: Consistency per update operation for RS6 (with W1)

Furthermore, Table 6.14 also shows that the measured times are independent of the read/write ratio and the size of a change. It is therefore assumed that these times are always the same, independent of the scenario. The deviation for RS3 is attributed to the small number of change and the small number of data points. This is also evident in Figure 6.13 which shows the times measured for each update operation for RS6 and W3.

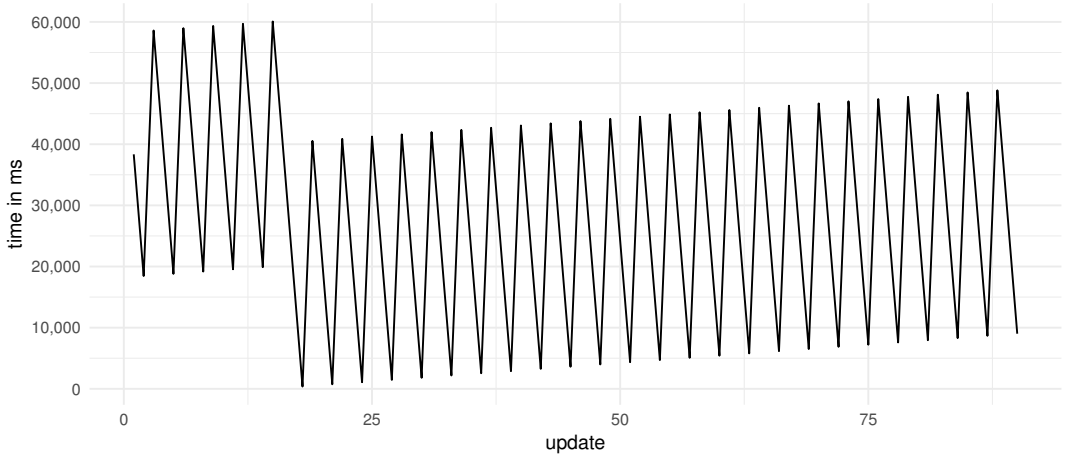


Figure 6.13: Consistency per update operation for RS6 with a workload with few update operations (W3)

Figure 6.13 also shows a gap in the curve. This is attributed to the fact that a little more than three update operations were executed per synchronization and the point in time were an update operation was performed changed. Update operations were executed approximately, but less than every 20 seconds whereas the synchronization job ran every 60 seconds. Before the gap, update operations were performed directly before the synchronization. After the gap, update operations were performed directly after the synchronization.

6.4.5 Summary

A final grading of the simulation results is presented in Table 6.15.

	RS0	**RS1**	**RS2**	**RS3**	**RS4**	**RS5**	**RS6**	**RS7**
Client-Side Data Access Latency	–	-	+	++	++	+	++	++
Server-Side Data Update Latency	++	++	-	-	-	-	-	-
Bandwidth Consumption	–	–	+	+	+	(+)	(+)	(+)
Consistency	++	++	+	+	+	-	-	-

Table 6.15: Grading of the replication strategies based on the results of the simulation

Although efforts were made to find realistic values for the parameters to base the simulation on realistic assumptions, it must be assumed that the validity of the grading is limited to the described assumptions and the chosen parameter values. This applies in particular for the grading of the bandwidth consumption for Poll-Based replication strategies (RS5, RS6, RS7) because they were optimized for the given workload.

The next chapter contains the final evaluation of the replication strategies. It also provides an evaluation of the dependencies and trade-offs between the measured values of particular quantitative indicators.

CHAPTER 7

Evaluation

The simulation shows that each replication strategy is advantageous for particular quantitative indicators, independent of the workload used. Results for different workloads show that the measured values vary within certain parameters but the overall ranking remains constant. Figure 7.1 presents the results of the simulation[1]. The size of a bubble indicates the transferred data. The color of a bubble the implemented consistency model.

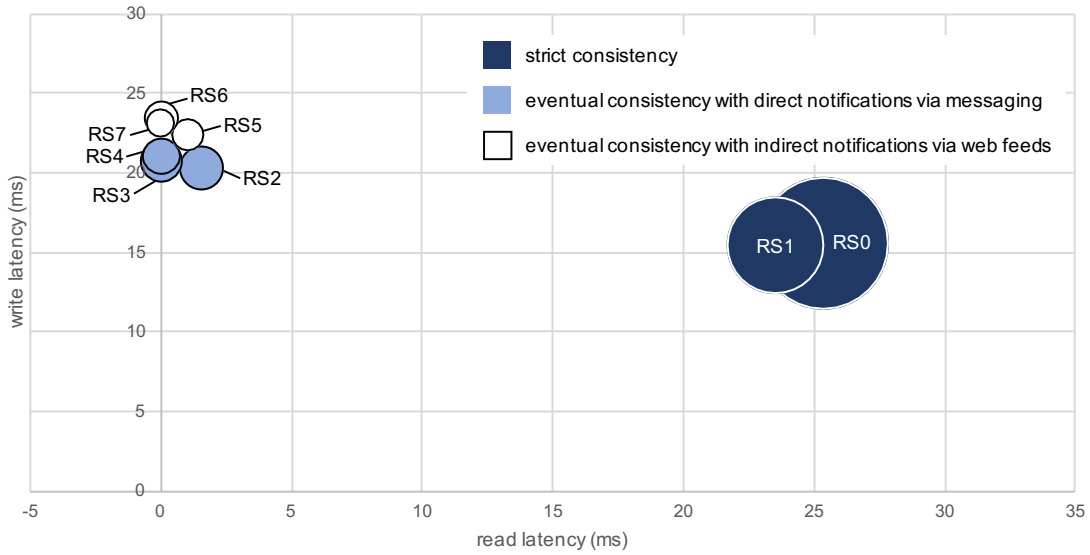


Figure 7.1: Results of the simulation

[1] Results of simulation runs with workload W5 are illustrated in the figure. Because RS0, RS1, RS2 and RS5 were not executed with this workload, the results of W1 are used. The workloads differ only in the percentage of content changed which does not affect the results for these replication strategies.

Figure 7.2 considers only eventually consistent replication strategies to highlight their behavior.

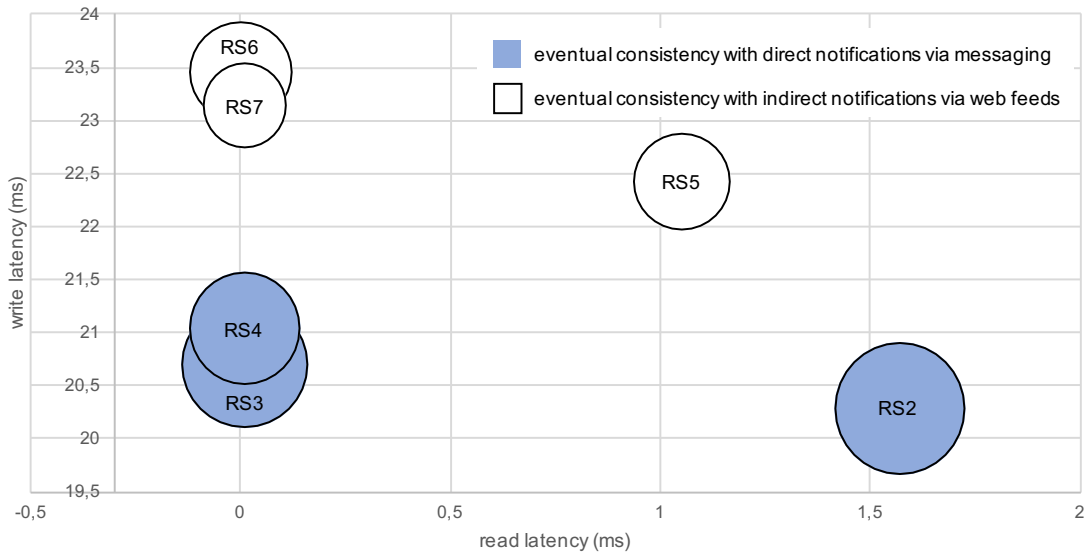


Figure 7.2: Results of the simulation for eventually consistent replication strategies

The figures indicate that there exist trade-offs between particular quantitative parameters. These trade-offs are described in detail in *Section 7.1*. In addition to the quantitative indicators, the analytical evaluation examined qualitative indicators based on [ISO-25010]. *Section 7.2* summarizes the findings for quantitative and qualitative indicators based on the general characteristics of replication processes. The concluding section summarizes all findings and presents an evaluation of the replication strategies.

7.1 Trade-offs

The results of the simulation indicate three trade-offs between the quantitative parameters. These trade-offs are discussed in the subsequent subsections.

7.1.1 Read Latency to Write Latency

There is a trade-off between read latency and write latency. The simulation showed that strictly consistent replication strategies have a high read latency and a low write latency while eventually consistent replication strategies have a low read latency and a high write latency. This is attributed to the implementation of the replication strategies: strictly consistent strategies include a request to the server on every access of a data item (which takes a lot of time) while eventually consistent strategies use an asynchronous approach. It is assumed that replication with synchronous notifications would also ensure a low read latency with an even higher write latency (because the server has to wait until all clients

are successfully notified) while ensuring strict consistency. It can be concluded that there is a trade-off between read latency and write latency, depending on the notification mechanism used and independent of the implemented consistency model.

7.1.2 Consistency to Read Latency

[Ada12] proposes that there is a trade-off between consistency and latency in IT systems without network partitioning. The simulation indicates the same. Replication strategies that provide strict consistency (RS0, RS1) have a higher read latency.

7.1.3 Consistency to Bandwidth Consumption

The results of the simulation indicate a trade-off between consistency and the consumed bandwidth (strict latency leads to a higher bandwidth consumption). This trade-off is conspicuous for Poll-Based replication strategies where the time of inconsistencies can be controlled by the read interval of the client. Shorter read intervals therefore directly lead to more bandwidth consumption.

7.2 Evaluation based on Replication Characteristics

The start of the thesis was the description of universal characteristics of replication processes, described in *Section 2.1*. These characteristics formed the basis for the selection of the concrete replication strategies. The findings of the analytical evaluation and the simulation can be attributed to these characteristics. The following subsections summarize findings for specific characteristics. Additionally, the use of synchronous and asynchronous notification mechanisms is discussed in *Subsection 7.2.4*.

7.2.1 Change Distribution

Replication strategies which distribute changes by state differences transfer less data than replication strategies that distribute the full state. The amount of saved transfer data is less than the actual change within the data item. The disadvantage of these replication strategies is the increased complexity of the implementation. They are therefore mainly advantageous in environments in which the network bandwidth is limited or if data items are very large.

The use of invalidation notifications was only examined in combination with caches. The evaluation of replication strategies that use caches is performed in *Subsection 7.2.3*.

7.2.2 Replica Reaction

Server-driven notifications (used for Event-Driven replication strategies) and periodic polling (used for Poll-Based replication strategies) ensure only eventual consistency. Server-driven notifications with messaging provide a better degree of consistency. The simulation showed that clients are notified in less than 20ms. Poll-Based replication

strategies are advantageous from a software architectural point of view because they need no central component and allow clients to act independently. However, the Atom-based implementation of Poll-Based replication strategies presented in this thesis cannot be recommended because of the insufficient availability of tools in the area of Atom-based web feeds. Therefore, the use of Event-Driven replication strategies is proposed over the use of Poll-Based replication strategies.

Polling on each request (used for Cache Invalidation) ensures strict consistency. As described in *Section 7.1*, strict consistency increases the latency of read operations on the client-side and of bandwidth consumption. The implementation of Cache Invalidation with RESTful webservices and ETags can easily be implemented in environments that already use RESTful webservices. It is therefore advantageous regarding implementation costs compared to the other replication strategies which require the introduction of complex technologies and tools.

7.2.3 Cache versus Full Replica

Replication strategies which use a full replica ensure faster read operations on the client-side because data is stored locally. This increases fault-tolerance and enables clients to perform queries over all data items locally as well. Replication strategies that use caches need less space on the local data storage[2]. The disadvantage of these replication strategies is the increased read latency when a data item has to be requested from the server which depends on the read/write ratio and the data access pattern. The simulation showed no differences in consistency of these two types of replication strategies. Caches are therefore mainly advantageous for environments where the client-side data storage is limited, or the total size of data is very large.

7.2.4 Synchronous versus Asynchronous

The use of asynchronous notification mechanisms enables loose coupling and was therefore used for replication strategies which use server-driven notifications and periodic polling. It also increases the fault-tolerance of the clients. Asynchronous communication generally decreases analyzability and tracing because the system state cannot be monitored by a single application but must be monitored from a system perspective.

7.3 Final Conclusions

The evaluation indicates that particular replication strategies are suited for the following cases of application:

1. **RS0 - No Replication** is easy to implement and therefore often used in IT systems. When there are no issues related to read latency or bandwidth consumption, it

[2] Especially when they use local invalidation processes (e.g. based on time-to-live) as well.

represents a sufficient solution and there is no need to introduce replication processes into the IT system.

2. **RS1 - Cache Validation** is an optimization of RS0 to decrease read latency and bandwidth consumption.
3. **RS2 - Event-Driven Cache Invalidation** is advantageous if the storage space on the client is limited[3].
4. **RS3 - Event-Driven Replication** seems to be the most suitable replication strategy for most loosely coupled IT systems if there are no specific requirements.
5. **RS4 - Event-Driven Delta Replication** is advantageous if the network bandwidth is limited.

RS5, RS6 and RS7 cannot be recommended because of the insufficient availability of tools in the area of Atom-based web feeds. Generally, RS5 behaves similar to RS2, RS6 behaves similar to RS3 and RS7 behaves similar to RS4. Therefore, they are suitable for the same cases of applications.

Figure 7.3 illustrates a process that can be used to find the appropriate replication strategy for a single application and use case. Because replication affects multiple components of an IT system, it must be assumed that replication processes are introduced for an entire system on the level of the system-wide architecture rather than for a single application. Changes in the system-wide architecture have to be aligned with all components in the system and their specific requirements. Furthermore, organizational aspects (like the know-how of the employees) have to be considered as well. Therefore, it is not possible to make a general recommendation. The evaluation presented in this thesis may be used as a supportive guideline to determine how replication can be implemented in an IT system.

[3] preferably in combination with local cache invalidation mechanisms

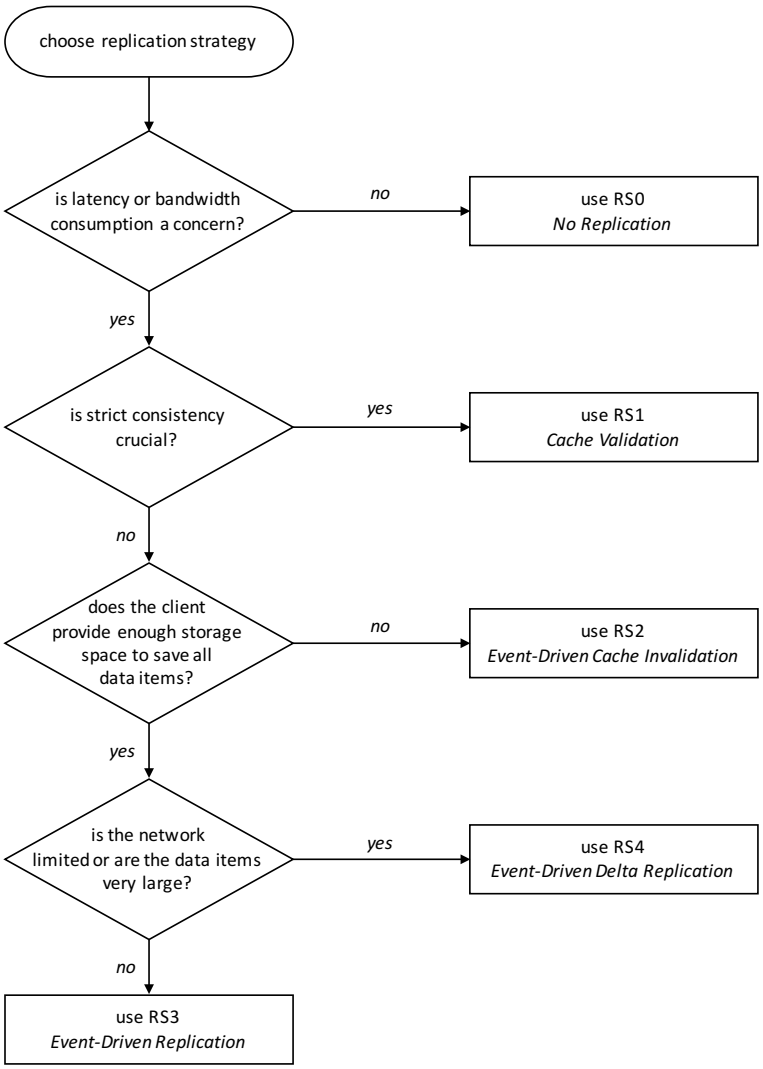


Figure 7.3: Process suggested to find the appropriate replication strategy for a single application and use case

CHAPTER 8

Summary

This thesis provided an evaluation of how replication can be performed in loosely coupled systems and described concrete processes to implement it. These processes and their general characteristics were examined in an analytical evaluation and in a simulation. To the authors knowledge, there is no comparable evaluation of replication in loosely coupled systems available. Thus, a general approach was used to cover a wide scope. Compared to a more specific and detailed approach for particular replication strategies or use cases, the chosen approach provides less precision in favor of an extensive overview.

8.1 Results

This thesis proposes the use of an Event-Driven Architecture to implement replication. It presented concrete implementations of replication strategies based on messaging (push-based) and Atom-based web feeds (pull-based). Furthermore, Cache Validation with RESTful webservices and ETags was used.

Concrete characteristics, benefits and downsides of the replications strategies was examined by the quality characteristics of the ISO 25010 standard and in a simulation with different workloads which represent typical scenarios. The simulation indicates that performance characteristics of the replication strategies (e.g. low read latency) are independent of the replication scenario.

The results are summarized and graded by the means of a comparison of the benefits and downsides of the replication strategies. A process to identify an appropriate replication strategy for a single use case was suggested during the evaluation. The work indicates that replication by push-based notification of the full state of a data item with a publish-subscribe message queue can be used for most use cases.

8.2 Future Work

In the course of the work, it turned out that there is little difference between the presented replication strategies and Event-Driven Architectures. Most replication strategies actually base on concepts of Event-Driven Architectures. The replication strategies proposed in this thesis use events from a technical perspective like `order.create` and `order.modify`. Event-Driven Architectures mostly focus on a business perspective [Fow06] and therefore use business events like `order` or `cancellation` which are able to implement the same processes. The replication strategies presented in this thesis can therefore be considered a special case of Event-Driven Architectures for the specific case of replication. The concept of an Event-Driven Architecture does not require the use of replication but it is assumed that many systems that implement this concept use replication as well. Research on the use of replication in loosely coupled systems and its use in Event-Driven Architectures is only briefly addressed in recent literature about Microservices [New15] [Wol15]. To the authors knowledge, there is no examination in this area available as well. Further research how replication is used in existing systems that implement an Event-Driven Architectures would therefore be desirable.

Our selection of replication strategies bases on a bottom-up approach which focuses on general characteristics of replication processes. The bottom-up approach did not consider technologies which use a totally different paradigm (e.g. Apache Kafka[1] which uses Event-Sourcing [Fow17]). An evaluation based on a top-down approach would fill the gap. This requires a detailed survey of how replication is currently used in existing loosely coupled IT systems as well.

The general approach used in this thesis required the use of a custom simulation environment which lead to several general assumptions on the concrete implementation of replication strategies and the workload used in the simulation. Therefore, this thesis provides only an indication of the behavior of the presented replication strategies in real systems. Additionally, long-time analysis and indicators in the area of maintainability could not be covered or were only examined analytically. Evaluations of the use of the presented replication strategies or similar replication processes in real IT systems under real workload would provide further insides into the topic.

[1] `https://kafka.apache.org/` (last checked 26.03.2018)

List of Figures

List of Tables

Bibliography

[ATG] Oracle. Repository guide - cache invalidation service. *https://docs.oracle.com/cd/E41069_01/Platform.11-0/ATGRepositoryGuide/html/s1018cacheinvalidationservice01.html, [last checked 26.03.2018]*, 2014.

[Ada02] Lada A Adamic and Bernardo A Huberman. Zipf's law and the internet. *Glottometrics*, 3(1):143–150, 2002.

[Ada12] Daniel Abadi. Consistency tradeoffs in modern distributed database system design: Cap is only part of the story. *Computer*, 45(2):37–42, 2012.

[All10] Subbu Allamaraju. *Restful web services cookbook: solutions for improving scalability and simplicity.* " O'Reilly Media, Inc.", 2010.

[Ami02] Yair Amir and Ciprian Tutu. From total order to database replication. In *Distributed Computing Systems, 2002. Proceedings. 22nd International Conference on*, pages 494–503. IEEE, 2002.

[Amz02] Cristiana Amza, Emmanuel Cecchet, Anupam Chanda, Alan L Cox, Sameh Elnikety, Romer Gil, Julie Marguerite, Karthick Rajamani, and Willy Zwaenepoel. Specification and implementation of dynamic web site benchmarks. In *5th Workshop on Workload Characterization*, number LABOS-CONF-2005-016, 2002.

[Bae97] Michael Baentsch, Lothar Baum, Georg Molter, Steffen Rothkugel, and Peter Sturm. Enhancing the web's infrastructure: From caching to replication. *IEEE Internet Computing*, 1(2):18–27, 1997.

[Bah11] Arshdeep Bahga and Vijay Krishna Madisetti. Synthetic workload generation for cloud computing applications. *Journal of Software Engineering and Applications*, 4(07):396, 2011.

[Bai12] Peter Bailis, Shivaram Venkataraman, Michael J Franklin, Joseph M Hellerstein, and Ion Stoica. Probabilistically bounded staleness for practical partial quorums. *Proceedings of the VLDB Endowment*, 5(8):776–787, 2012.

[Ban97] Gaurav Banga, Fred Douglis, Michael Rabinovich, et al. Optimistic deltas for www latency reduction. In *Proc. 1997 USENIX Technical Conference, Anaheim, CA*, pages 289–303, 1997.

[Ber13] David Bermbach, Liang Zhao, and Sherif Sakr. Towards comprehensive measurement of consistency guarantees for cloud-hosted data storage services. In *TPCTC*, pages 32–47. Springer, 2013.

[Ber13b] David Bermbach and Jörn Kuhlenkamp. Consistency in distributed storage systems. In *Networked Systems*, pages 175–189. Springer, 2013.

[Ber14] David Bermbach and Stefan Tai. Benchmarking eventual consistency: Lessons learned from long-term experimental studies. In *Cloud Engineering (IC2E), 2014 IEEE International Conference on*, pages 47–56. IEEE, 2014.

[Ber14b] D. Bermbach. *Benchmarking, Consistency, Distributed Database Management Systems, Distributed Systems, Eventual Consistency:*. KIT Scientific Publishing, 2014.

[Bli03] Johann Blieberger, Bernd Burgstaller, and Bernhard Scholz. Busy wait analysis. In *International Conference on Reliable Software Technologies*, pages 142–152. Springer, 2003.

[Bor05] Behzad Bordbar and Kyriakos Anastasakis. Mda and analysis of web applications. In *International Conference on Trends in Enterprise Application Architecture*, pages 44–55. Springer, 2005.

[Bre00] Eric Brewer. Towards robust distributed systems. In *PODC*, volume 7, 2000.

[Bre12] Eric Brewer. Cap twelve years later: How the "rules" have changed. *Computer*, 45(2):23–29, 2012.

[Bre99] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web caching and zipf-like distributions: Evidence and implications. In *INFOCOM'99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 1, pages 126–134. IEEE, 1999.

[Bul14] Frederik Bülthoff and Maria Maleshkova. Restful or restless–current state of today's top web apis. In *European Semantic Web Conference*, pages 64–74. Springer, 2014.

[Bus02] Mudashiru Busari and Carey Williamson. Prowgen: a synthetic workload generation tool for simulation evaluation of web proxy caches. *Computer Networks*, 38(6):779–794, 2002.

[Bus96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. Pattern-oriented software architecture volume 1. 1996.

[CDA] Jez Humble. Continous delivery architecture. *https://continuousdelivery.com/implementing/architecture/, [last checked 26.03.2018]*.

[Cal10] Tom Callahan, Mark Allman, and Vern Paxson. A longitudinal view of http traffic. In *International Conference on Passive and Active Network Measurement*, pages 222–231. Springer, 2010.

[Cal16] Maria Carla Calzarossa, Luisa Massari, and Daniele Tessera. Workload characterization: A survey revisited. *ACM Computing Surveys (CSUR)*, 48(3):48, 2016.

[Cao98] Pei Cao and Chengjie Liu. Maintaining strong cache consistency in the world wide web. *IEEE Transactions on Computers*, 47(4):445–457, 1998.

[Coo10] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154. ACM, 2010.

[Cou05] George F Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed systems: concepts and design*. pearson education, 2005.

[Dai12] Robert Daigneau. *Service Design Patterns: fundamental design solutions for SOAP/WSDL and restful Web Services*. Addison-Wesley, 2011.

[Dha17] Ajinkya Dhamnaskar. Rabbitmq und events. *https://ajinkya-dhamnaskar.github.io/2017/04/06/event-based-replication.html, [last checked 26.03.2018]*, 2017.

[EDA] Inc. Gartner. Event-driven architecture (eda) in the gartner it glossary. *https://www.gartner.com/it-glossary/eda-event-driven-architecture, [last checked 26.03.2018]*.

[Eug03] Patrick Th Eugster, Pascal A Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM computing surveys (CSUR)*, 35(2):114–131, 2003.

[Fei15] Dror G Feitelson. *Workload modeling for computer systems performance evaluation*. Cambridge University Press, 2015.

[Fie00] Roy T Fielding and Richard N Taylor. *Architectural styles and the design of network-based software architectures*, volume 7. University of California, Irvine Doctoral dissertation, 2000.

[Flo09] Daniela Florescu and Donald Kossmann. Rethinking cost and performance of database systems. *ACM Sigmod Record*, 38(1):43–48, 2009.

[Fow02] Martin Fowler. *Patterns of enterprise application architecture*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[Fow06] Martin Fowler. Focusing on events. *https://martinfowler.com/eaaDev/EventNarrative.html, [last checked 26.03.2018]*, 2006.

[Fow08] Martin Fowler. Test double. *https://martinfowler.com/bliki/TestDouble.html, [last checked 26.03.2018]*, 2008.

[Fow11] Martin Fowler. Tolerant reader. *https://martinfowler.com/bliki/TolerantReader.html, [last checked 26.03.2018]*, 2011.

[Fow12] Martin Fowler. Test pyramid. *https://martinfowler.com/bliki/TestPyramid.html, [last checked 26.03.2018]*, 2012.

[Fow14] Martin Fowler. Microservices - a definition of this new architectural term. *https://martinfowler.com/articles/microservices.html, [last checked 26.03.2018]*, 2014.

[Fow17] Martin Fowler. What do you mean by "event-driven"? *https://martinfowler.com/articles/201701-event-driven.html, [last checked 26.03.2018]*, 2017.

[Fow18] Martin Fowler. Integration test. *https://martinfowler.com/bliki/IntegrationTest.html, [last checked 26.03.2018]*, 2018.

[Gar03] Daniel F García and Javier García. Tpc-w e-commerce benchmark evaluation. *Computer*, 36(2):42–48, 2003.

[Gie15] Oliver Gierke and Eberhard Wolff. Integration von microservices – rest vs. messaging. *https://jaxenter.de/microservices-rest-vs-messaging-29875, [last checked 26.03.2018]*, 2015.

[Gil02] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *Acm Sigact News*, 33(2):51–59, 2002.

[Gil07] Phillipa Gill, Martin Arlitt, Zongpeng Li, and Anirban Mahanti. Youtube traffic characterization: a view from the edge. In *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*, pages 15–28. ACM, 2007.

[Gra92] Jim Gray. *Benchmark handbook: for database and transaction processing systems*. Morgan Kaufmann Publishers Inc., 1992.

[Gwe96] James Gwertzman and Margo I Seltzer. World wide web cache consistency. In *USENIX annual technical conference*, pages 141–152, 1996.

[Hel07] Pat Helland. Life beyond distributed transactions: an apostate's opinion. In *CIDR*, volume 2007, pages 132–141, 2007.

[Hoh04] Gregor Hohpe and Bobby Woolf. *Enterprise integration patterns: Designing, building, and deploying messaging solutions.* Addison-Wesley Professional, 2004.

[ISO-25010] Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models. Standard, International Organization for Standardization, Geneva, CH, March 2011.

[Jai90] Raj Jain. *The art of computer systems performance analysis: techniques for experimental design, measurement, simulation, and modeling.* John Wiley & Sons, 1990.

[Jeo12] Myeongjae Jeon, Youngjae Kim, Jeaho Hwang, Joonwon Lee, and Euiseong Seo. Workload characterization and performance implications of large-scale blog servers. *ACM Transactions on the Web (TWEB)*, 6(4):16, 2012.

[Kar07] Firat Kart, Louise E Moser, and P Michael Melliar-Smith. Reliable data distribution and consistent data replication using the atom syndication technology. In *International Conference on Internet Computing*, pages 124–132, 2007.

[Lev01] Mark Levene, José Borges, and George Loizou. Zipf's law for web surfers. *Knowledge and Information Systems*, 3(1):120–129, 2001.

[Lou02] Thanasis Loukopoulos, Ishfaq Ahmad, and Dimitris Papadias. An overview of data replication on the internet. In *Parallel Architectures, Algorithms and Networks, 2002. I-SPAN'02. Proceedings. International Symposium on*, pages 31–36. IEEE, 2002.

[Men00] Daniel Menascé, Virgílio Almeida, Rudolf Riedi, Flávia Ribeiro, Rodrigo Fonseca, and Wagner Meira Jr. In search of invariants for e-business workloads. In *Proceedings of the 2nd ACM conference on Electronic commerce*, pages 56–65. ACM, 2000.

[Nah02] Eric Nahum. Deconstructing specweb99. In *Proceedings of 7th International Workshop on Web Content Caching and Distribution*, 2002.

[New15] Sam Newman. *Building microservices: designing fine-grained systems.* "O'Reilly Media, Inc.", 2015.

[Ora] Oracle. Java authentication and authorization service (jaas) reference guide. *https://docs.oracle.com/javase/7/docs/technotes/guides/security/jaas/JAASRefGuide.html, [last checked 26.03.2018].*

[Pat11] Swapnil Patil, Milo Polte, Kai Ren, Wittawat Tantisiriroj, Lin Xiao, Julio López, Garth Gibson, Adam Fuchs, and Billie Rinaldi. Ycsb++: benchmarking and performance debugging advanced features in scalable table stores. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, page 9. ACM, 2011.

[Pie01] Guillaume Pierre, Ihor Kuz, Maarten Van Steen, and Andrew S. Tanenbaum. Differentiated strategies for replicating web documents. *Computer Communications*, 24(2):232–240, 2001.

[Pie02] Guillaume Pierre, Maarten Van Steen, and Andrew S Tanenbaum. Dynamically selecting optimal distribution strategies for web documents. *IEEE Transactions on Computers*, 51(6):637–651, 2002.

[RFC-2818] E. Rescorla. HTTP Over TLS. RFC 2818, RFC Editor, May 2000.

[RFC-3629] F. Yergeau. UTF-8, a transformation format of ISO 10646. RFC 3629, RFC Editor, November 2003.

[RFC-4287] M. Nottingham and R. Sayre. The Atom Syndication Format. RFC 4287, RFC Editor, December 2005.

[RFC-4422] A. Melnikov and K. Zeilenga. Simple Authentication and Security Layer (SASL). RFC 4422, RFC Editor, June 2006.

[RFC-5023] J. Gregorio and B. de hOra. The Atom Publishing Protocol. RFC 5023, RFC Editor, October 2007.

[RFC-5246] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246, RFC Editor, August 2008.

[RFC-5905] D. Mills, U. Delaware, J. Martin, J. Burbank, and W. Kasch. Network Time Protocol Version 4: Protocol and Algorithms Specification. RFC 5905, RFC Editor, June 2010.

[RFC-7234] R. Fielding, M. Nottingham, and J. Reschke. Hypertext Transfer Protocol (HTTP/1.1): Caching. RFC 7234, RFC Editor, June 2014.

[RFC-7235] R. Fielding and J. Reschke. Hypertext Transfer Protocol (HTTP/1.1): Authentication. RFC 7235, RFC Editor, June 2014.

[Ren12] Dominik Renzel, Patrick Schlebusch, and Ralf Klamma. Today's top "restful" services and why they are not restful. *Web Information Systems Engineering-WISE 2012*, pages 354–367, 2012.

[Ric07] Leonard Richardson and Sam Ruby. *RESTful web services*. "O'Reilly Media, Inc.", 2007.

[Ric13] Leonard Richardson, Mike Amundsen, and Sam Ruby. *RESTful Web APIs: Services for a Changing World.* " O'Reilly Media, Inc.", 2013.

[Ric15] Chris Richardson. Event-driven data management for microservices. *https://www.nginx.com/blog/event-driven-data-management-microservices/, [last checked 26.03.2018].*

[Ric17] Chris Richardson. Pattern: Monolithic architecture. *http://microservices.io/patterns/monolithic.html, [last checked 26.03.2018].*

[Ric17b] Chris Richardson. Pattern: Shared database. *http://microservices.io/patterns/data/shared-database.html, [last checked 26.03.2018].*

[Rob06] Ian Robinson. Consumer-driven contracts: A service evolution pattern. *https://martinfowler.com/articles/consumerDrivenContracts.html, [last checked 26.03.2018]*, 2006.

[SCS] Self contained systems. *http://scs-architecture.org/index.html, [last checked 26.03.2018].*

[SOA] Inc. Gartner. Service-oriented architecture (soa) in the gartner it glossary. *https://www.gartner.com/it-glossary/service-oriented-architecture-soa/, [last checked 26.03.2018].*

[San15] Yamen Sader. Events & microservices. *https://de.slideshare.net/YamenSader/events-microservices, [last checked 26.03.2018]*, 2015.

[Sch06] Bianca Schroeder, Adam Wierman, and Mor Harchol-Balter. Open versus closed: A cautionary tale. In *Nsdi*, volume 6, pages 18–18, 2006.

[Siv07] Swaminathan Sivasubramanian, Guillaume Pierre, Maarten Van Steen, and Gustavo Alonso. Analysis of caching and replication strategies for web applications. *IEEE Internet Computing*, 11(1), 2007.

[Smi00] Wayne D Smith. Tpc-w: Benchmarking an ecommerce solution, 2000.

[Ste15] Guido Steinacker. On monoliths and microservices. *https://dev.otto.de/2015/09/30/on-monoliths-and-microservices/, [last checked 26.03.2018]*, 2015.

[Tan07] Andrew S Tanenbaum and Maarten Van Steen. *Distributed systems: principles and paradigms.* Prentice-Hall, 2007.

[Val09] Mohammad Hadi Valipour, Bavar AmirZafari, Khashayar Niki Maleki, and Negin Daneshpour. A brief survey of software architecture concepts and service oriented architecture. In *Computer Science and Information Technology, 2009. ICCSIT 2009. 2nd IEEE International Conference on*, pages 34–38. IEEE, 2009.

[Vog07] Werner Vogels. Data access patterns in the amazon. com technology platform. In *Proceedings of the 33rd international conference on Very large data bases*, pages 1–1. VLDB Endowment, 2007.

[Vog09] Werner Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, 2009.

[Wad11] Hiroshi Wada, Alan Fekete, Liang Zhao, Kevin Lee, and Anna Liu. Data consistency properties and the trade-offs in commercial cloud storage: the consumers' perspective. In *CIDR*, volume 11, pages 134–143, 2011.

[Wat15] Steve T Watt, Shankar Achanta, Hamza Abubakari, Eric Sagen, Zafer Korkmaz, and Husam Ahmed. Understanding and applying precision time protocol. In *Smart Grid (SASG), 2015 Saudi Arabia*, pages 1–7. IEEE, 2015.

[Web10] Jim Webber, Savas Parastatidis, and Ian Robinson. *REST in practice: Hypermedia and systems architecture.* " O'Reilly Media, Inc.", 2010.

[Wil05] Adepele Williams, Martin Arlitt, Carey Williamson, and Ken Barker. Web workload characterization: Ten years later. In *Web content delivery*, pages 3–21. Springer, 2005.

[Wol15] Eberhard Wolff. *Microservices: Grundlagen flexibler Softwarearchitekturen.* Dpunkt. verlag, 2015.

[Xu13] Haiyun Xu, Jeroen Heijmans, and Joost Visser. A practical model for rating software security. In *Software Security and Reliability-Companion (SERE-C), 2013 IEEE 7th International Conference on*, pages 231–232. IEEE, 2013.

[YCSB] Oracle. Ycsb wiki: Core workloads. `https://github.com/brianfrankcooper/YCSB/wiki/Core-Workloads`, *[last checked 26.03.2018].*

[Yin99] Jian Yin, Lorenzo Alvisi, Mike Dahlin, and Calvin Lin. Hierarchicalcachecons istency ina wa n. In *Proceedings of the 1999 Usenix Symposium on Internet Technologies and Systems (USITS'99)*, 1999.

[Yu99] Haobo Yu, Lee Breslau, and Scott Shenker. A scalable web cache consistency architecture. In *ACM SIGCOMM Computer Communication Review*, volume 29, pages 163–174. ACM, 1999.

[Zip32] George Kingsley Zipf. Selected studies of the principle of relative frequency in language. 1932.

[Zip49] G.K. Zipf. Human behaviour and the principle of least effort. 1949.