Technische Universität München

Department of Informatics

Master's Thesis in Informatics

Efficient Storage Backends for IoT Data

Florian Georg Kreitmair

Technische Universität München

Department of Informatics

Master's Thesis in Informatics

Efficient Storage Backends for IoT Data

Effiziente Speicherung von IoT-Daten

Author	Florian Georg Kreitmair
Supervisor	Prof. Dr.-Ing. Georg Carle
Advisor	Dr. Marc-Oliver Pahl, Stefan Liebald, Dr. Cyrille Artho
Date	May 15, 2018

Informatik VIII
Chair for Network Architectures and Services

I confirm that this thesis is my own work and I have documented all sources and material used.

Garching b. München, May 15, 2018

Signature

Abstract

Storing information is at the core of almost every Internet-of-Things (IoT) middleware, which poses challenges relating to scalability, fault-tolerance and correctness. This is even more relevant for a subset of IoT platform architectures that use a database abstraction to connect IoT components and treat data that represents physical state, control output, or inferred information in a uniform manner. At the same time, new database products aim to address these challenges through means of distribution while retaining semantics similar to existing relational databases. This study analysed the requirements of a stateful IoT middleware on its database backend in order to identify viable candidates. A middleware product was adapted to store its data on two backend implementations based on the databases CockroachDB and Infinispan, which make use of different concepts and algorithms. To leverage their transactional capabilities, the frontend interface of the middleware was supplemented with transactional access methods. Finally, the performance of the implementations was measured and compared with a set of application-oriented benchmarks. The results show a considerable trade-off between consistency, transactional safety, performance, and fault-tolerance and suggest that IoT middleware should make a distinction between the processing of input data aggregation and decision-making coordination tasks.

Sammanfattning

Att lagra information är en central uppgift i nästan varje Internet-of-Things (IoT) middleware. Dessa system rörar mot skalbarhet, feltolerans och korrekthet. Det är ännu mer relevant för en delmängd av IoT-plattformsarkitekturer som förbinder IoT komponenter genom en databasabstraktion och behandlar data som representerar fysiskt tillstånd, styrningsdata eller avledad information på ett enhetligt sätt och har därmed höga krav på konsistens och performans. Samtidigt finns det nu nya databasprodukter som adresserar dessa utmaningar genom distribution, men behåller samtidigt semantiken av välkända relationsdatabaser. Denna studien analyserar kraven av en stateful IoT middleware på databasbackenden och försöker identifiera tillämpliga kandidater. En middleware-produkt anpassas sedan för att lagra data genom två backend-implementeringar som baserar på databaser CockroachDB och Infinispan vilka som använder olika koncept och algoritmer. För att utnyttja deras transaktionsmöjligheter kompletteras frontend-gränssnittet med transaktionala metoder. Slutligen mäts implementeringars prestation och jämförs med varandra genom en applikationsorienterade benchmark. Resultaten visar ett betydande avvägning mellan konsistens, transaktionssäkerhet och performans och tyda på att middlewaren skulle skilja mellan bearbetning av indata och besultsfattande koordinationsprocesser.

Zusammenfassung

Die Speicherung von Information ist eine zentrale Aufgabe jeder Internet-of-Things (IoT) Middleware, was Herausforderungen an Skalierbarkeit, Fehlertoleranz und Korrektheit stellt. Dies ist insbesondere für eine Teilmenge von IoT Plattform-Architekturen relevant, die eine Datenbank-Abstraktion verwenden um Echtzeitinformation zum physikalische Zustand, Steuerungsdaten oder abgeleitetes Wissen über eine einheitliche Schnittstelle zur Verfügung zu stellen und darüber hinaus hohe Anforderungen an Konsistenz und Performanz stellen. Diese Herausforderungen werden auch von vielen jüngeren Datenbank-Entwicklungen addressiert, die Daten verteilt speichern und dabei versuchen die Semantik von herkömmlichen relationalen Datenbanken zu wahren. Diese Studie ermittelt die Anforderungen einer IoT-Middleware an ihr Datenbankbackend und versucht geeignete Kandidaten zu identifizieren. Das Middleware-Produkt DS2OS wird dann für zwei der identifizierten Datenbanken (Infinispan und CockroachDB) angepasst. Zur Ausnutzung von deren Transaktionsfähigkeit wird das Frontend der Middleware zusätzlich mit transaktionalen Zugriffsmethoden versehen. Schließlich werden die Implmentierungen mit anwendungsnahen Benchmarks vermessen und miteinander verglichen. Die Ergebnisse zeigen eine deutlichen Zielkonflikt zwischen Konsistenz, Transaktionssicherheit und Performanz und legen nahe, dass eine IoT-Middleware bei der Datenverarbeitung zwischen der Aggregation von Sensordaten und Entscheidungsprozessen zur Koordination von Ausgabehandlungen unterscheiden sollte.

Contents

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 State of the Art . . . 3
- 1.3 Goal . . . 3
- 1.4 Outline . . . 4

2 Analysis 5
- 2.1 Intelligent Buildings and Smart Spaces . . . 5
  - 2.1.1 Conventional Building Automation . . . 5
  - 2.1.2 The Vision . . . 6
  - 2.1.3 Related work . . . 7
- 2.2 DS2OS . . . 9
  - 2.2.1 Design and Objectives . . . 10
  - 2.2.2 Developer-centred view of DS2OS . . . 11

3 Databases for Smart Spaces 19
- 3.1 Requirements . . . 19
  - 3.1.1 Data meta-model and structure . . . 19
  - 3.1.2 Interface . . . 22
  - 3.1.3 Storage Location . . . 22
  - 3.1.4 Partitioning (Sharding) . . . 24
  - 3.1.5 Replication . . . 24
  - 3.1.6 Concurrency control . . . 25
  - 3.1.7 CAP Trade-Off . . . 28
  - 3.1.8 Resource Consumption . . . 29
  - 3.1.9 Triggers & Stored Procedures . . . 30
  - 3.1.10 Related work . . . 31
- 3.2 Available distributed datastores . . . 32
  - 3.2.1 Zookeeper . . . 34
  - 3.2.2 CouchDB . . . 34
  - 3.2.3 Cassandra . . . 35
  - 3.2.4 HBase . . . 35

3.2.5 VoltDB . . . 36
3.2.6 CockroachDB . . . 36
3.2.7 Riak . . . 37
3.2.8 Scalaris . . . 37
3.2.9 Voldemort . . . 37
3.2.10 Infinispan . . . 38
3.2.11 Hazelcast . . . 38
3.2.12 Ignite . . . 39
3.2.13 Geode . . . 39
3.2.14 Conclusion . . . 39

4 Implementation 41
4.1 Architectural Integration . . . 41
4.1.1 Transaction Interface . . . 41
4.1.2 Version History . . . 43
4.1.3 Subscription and Notifications . . . 47
4.2 Back-end Implementation . . . 48
4.2.1 CockroachDB . . . 51
4.2.2 Infinispan . . . 52
4.3 Summary . . . 53

5 Evaluation 55
5.1 Test Model . . . 56
5.2 Measurable properties . . . 57
5.2.1 Information dissemination latency . . . 57
5.2.2 Transaction success rate . . . 58
5.2.3 Service-level latency . . . 58
5.2.4 Throughput . . . 60
5.3 Specific dataflow-oriented tests . . . 60
5.3.1 Aggregation Chain Length . . . 61
5.3.2 Coordination Complexity . . . 62
5.3.3 Coordination Scale . . . 63
5.3.4 Sensor Data Size . . . 64
5.3.5 Aggregation Chain Length Heterogeneity . . . 64
5.3.6 Indirect Control . . . 65
5.4 Load tests . . . 66
5.4.1 Load generation . . . 66
5.4.2 Latency . . . 67
5.4.3 Throughput . . . 68
5.4.4 Fault tolerance . . . 68
5.5 Threats to Validity . . . 69
5.6 Conclusion . . . 70

6 Conclusion 73
6.1 Differences & Trade-offs 73
6.2 Recommendations for Implementation 74

7 Future work 77

Bibliography 79

List of Figures

2.1 Context model example for a multicolour lamp. 12

4.1 Operations expicitly associated with a transaction object 42
4.2 One transaction environment per connector 42
4.3 One transaction environment per connector 43
4.4 Archiving the version history within the database that keeps the VSL data 45
4.5 Archiving state updates outside of the VSL state database in parallel . . 46
4.6 Archiving state updates before transaction commit 47
4.7 Archiving state updates after transaction commit 48
4.8 Database tables of the CockroachDB backend 51
4.9 Classes used as value types of the Infinispan backend 52

5.1 Information flow 57
5.2 Aggregation Chain Length Topology 61
5.3 Time between sensor output and actuator read of a causally related event dependent on aggregation chain length 62
5.4 Coordination Complexity Topology 62
5.5 Time for the execution of a coordination transaction dependent on the containing number of reads (from KI services) and writes (to actuators). 63
5.6 Coordination Scale Topology 63
5.7 Time for the execution of a coordination service transaction in scenarios varying in the number of coordination services with identical access patterns 64
5.8 Aggregation Chain Length Heterogenity Topology 65
5.9 Indirect Control Topology 66
5.10 Time for the execution of a control service transaction and throughput of transactions. 67
5.11 Latency Comparison 72
5.12 Non-transactional throughput 72

List of Tables

3.1 Requirement Matrix 33

Chapter 1

Introduction

1.1 Motivation

Modern buildings and infrastructures can contain a vast amount of sensors and remote controlled actuators. In a terminal building at an international airport, for example, there are hundreds of automatic doors, lights, heating devices, escalators, security cameras, light barriers, RFID scanners and other sorts of devices. These continuously emit new data or require control input, while the procedures to process data and to determine control input are complex.

Consider an automatic sliding door as an illustrative example: First and foremost such a door must be safe to use – it must not crush a person who is walking through. On the other hand it must not open when it is not supposed to, in order to isolate security-critical areas where only authorised personnel is allowed to pass, or to keep distinct groups of travellers, such as domestic and international ones, apart from each other. In yet another case, it might be vital that it opens, for example with a manual override in an emergency evacuation situation. An automatic door also emits information: it can count the number of people who pass through in both directions, signal if it is currently in an open or closed state, and trigger an alert when the emergency release button has been pressed. To align its own state with higher-level coordination policies, the door must communicate and interact with a network of distributed services. One such a service could, for instance, estimate the total number of people within an area at a given time by counting the traffic that passes through the door. This value can then be made available to other coordination processes; for example, it can be taken into account to dynamically adapt the number of security lines in response to passenger traffic. It can also be used to limit the number of people in an area for safety reasons, or to move baggage carts, tracked by RFID tags, to areas with increased demand. In the long run, this data, aggregated over time, is also valuable for airport shop owners, advertisers, or planners of terminal extensions.

Many of these coordination procedures can in principle be carried out by human operators on control panels that receive the measurements and give instructions, then configuring devices according to human reasoning and judgement. However, the sheer amount of information to process, the complexity, and the high expenses of the human operators makes this unfeasible, while the infrastructure to collect and transmit the measurement data and control information is nevertheless necessary. This makes it economical to automatise the procedures to a high degree.

The task is not trivial: Physical devices come from different vendors and have incompatible interfaces. Even if they are standardised, they remain heterogeneous, as a range of standards exists for various specialised applications – for instance, the DMX protocol for lighting installations. There are also implementations that are already integrated at the device-level, such as radiators that come with an integrated thermostat. In addition, some control functionality has to remain tightly integrated at the physical device because it is especially time- or safety-critical. To return to the door example, this might be the control procedure to unlock the door when the emergency button is pressed. It would be unsafe to fully rely on the availability of a human or non-human coordinator and a network that exactly in such a situation is prone to damage. Therefore, this behaviour is already hard-wired in door units sold on the market and works autonomously without the help of a back-end infrastructure. These circumstances make such devices complex to model, as they are not simply sensors and actuators any more; instead, a model has to incorporate a notion of autonomous behaviour as well. For example, by regarding a door as a service that provides functionality like changing the opening policy, reading the current state, or subscribing to alerts and people counter updates.

The dynamics involved are a further challenging point. The infrastructure is subject to change: New physical components might be added or removed, or might simply fail at times. And so must control policies changed from time to time: It should be simple to switch sections of the building into a standby mode when they are closed for the public, to switch subsystems to a manual operation mode, or to adapt heating control parameters in order to save energy. This point also becomes relevant when the same technology is used in small-scale deployments, which is becoming more common as the technology is getting more affordable. In a residential setting, also called smart home, it is more difficult to plan and manage the IoT system professionally; instead, it is important that the devices are compatible with each other easy to use by the occupants. It implies that the devices themselves must be able to coordinate their topology, discover resources and adapt to the environment. This points to a clear advantage of an adaptive, dynamic system over a hard-wired implementation where changes in hardware or policy entail changing physical infrastructure.

1.2 State of the Art

To account for these challenges, a number of solutions, often referred to as middleware, have been developed. These are not necessarily limited to the context of building automation, but to the wider internet of things as the challenges are similar in other cyber-physical infrastructures. Their key concept is to provide abstractions that allow programmers to access the heterogeneous physical space through homogeneous, well-defined interfaces, thus hiding their complexity where it is irrelevant to the programmer.

Being stateful distributed systems in an asynchronous network, IoT middleware is subject to the inherent conceptual constraints that these systems experience. Messages between distributed components are transmitted unreliably and out-of-order and it is not possible to detect failures of remote components reliably and consistently. It is difficult to abstract away from these phenomena and provide an interface that hides the distributed aspect and makes guarantees about safety, reliability and availability at the same time. Some of these challenges have in parts been overcome by recent developments in distributed database architecture. Modern products of this kind, like Google Spanner or Scalaris claim to offer a high level of availability and scalability while retaining consistency. Because of statefulness, a database is part of most IoT middleware architectures, it is very likely that basic quality features can be improved by making use of these modern developments.

DS2OS is an example of a distributed middleware product that aims to address the outlined challenges. At the core, it employs a blackboard-like approach towards sharing information: Services write their state to a global database abstraction and use the same for querying other services' current or past state. The information is described semantically by context models which allows to discover services by their type. In addition there is a subscription feature for push-based processing and a remote method invocation feature for lazy processing of data. This makes the middleware especially flexible. Yet further improvements are possible: It is desirable that middleware also satisfies a number of quality requirements to the best extent possible. A malfunction, such as the application of a wrong control rule, can result in physical and financial damage and have safety and security implications. Also, unavailability or a slow response time might transgress safety and security requirements or result in bad user experience in the better case.

1.3 Goal

This thesis explores concepts from recent developments in distributed databases that try to overcome some of the problems. In particular transactional safety, availability, and scalability are properties that might be improved. This goal can be split up further into the following research questions:

1. What are requirements on the datastore with respect to to smart space applications, and which carry priority under the existence of trade-offs?
2. What are suitable database products, how can they be integrated in DS2OS architecture and which quantitative properties are to expect these combinations?
3. What is the quantitative performance of DS2OS with the distributed database back-end in a realistic scenario?

In order to accomplish this, I surveyed the architectural design and features of distributed databases, the possible ways to integrate them into DS2OS, and a number of available database products. For two promising database products, I also developed connectors to use these to store DS2OS real-time data and conducted a few benchmarks.

1.4 Outline

Chapter 2 contains background information on smart spaces and the Internet-of-Things that form the area of application of this work and DS2OS which is the middleware framework used as a basis for further research. Design difference among databases, their properties, and features are explained in Chapter 3, followed by a survey on available database products. Chapter 4 contains explanations about the implementation of Infinispan and CockroachDB connectors for DS2OS. These are quantitatively evaluated with a test harness and set of benchmarks in the following Chapter 5. Chapter 6 sums up the results and suggests how to use them to improve future smart space middleware.

Chapter 2

Analysis

2.1 Intelligent Buildings and Smart Spaces

2.1.1 Conventional Building Automation

One area of application for the Internet of Things is building automation. This is not a new concept. It started out with small, focused subsystems such as automated elevators, access control and alert systems which are already around for decades. Later developments integrate subsystems on a building level, automate operation, and provide interfaces for manual monitoring and control. Another important step in development was to connect the systems to the internet which allows to carry out these tasks remotely and across buildings. The main rationale for these conventional automation techniques are to increase the reliability, reduce operating costs, enhance the productivity of operational personnel, protect people and equipment, and provide information for building management [1].

The most common automation processes in these systems are control functions, which can either be realised as closed or open loops. More than 80 percent are proportional, integral and derivative (PID) control functions, a particular algorithm [1]. It takes a measurement from the environment, for example the room temperature and a configured set point. The output, so in the example the heater in the room, is then controlled in proportion to the difference between the observed and desired temperature. Summing up the error between both values over time compensates for constant disturbance (e.g., heat losses through the exterior wall) that cannot instantly be observed. Furthermore, a derivative function of the error signal makes the control function react faster when there is short-term disturbance. So for example, after a window has been opened the heating should be turned up significantly for some time to compensate the loss, even if the drop in temperature was not that substantial.

At the implementation side, there are a number number of network protocols available and in use to query and trigger physical components. Wang [1] mentions BACnet, LonWorks, Modbus, PROFIBUS and EIB/KNX. Along with those special protocols exist for certain subsystems, for example DMX and DALI for lighting installations. All those have in common that they mostly focus on the communication between components, but not integration, orchestration, and management issues. Interoperability is provided by gateways that translate between protocols. Often they provide service-oriented interfaces (CORBA, DCOM, SOAP) that allow higher level integration.

2.1.2 The Vision

While the described building automation technology fulfils the goals described in 2.1.1, it does not allow for the amount of integration of distributed devices that is possible. Pervasive computing scenarios assume that in the future many or most building and household items will have advanced sensing and computation capabilities and will interact in some form with the persons in their vicinity.

Smart houses can for example feature sensors such as touch-sensitive floor panelling (smart floor), infrared and ultrasound person and object detection, smart power outlets, smart kitchen devices, a smart mailbox, as well as multimodal interfaces, such as displays, gesture recognition, smart mirrors, and smart phones that seamlessly integrate and offer additional value like social-distant dining or care for the elderly [2]. Smart classrooms can seamlessly combine on-site with remote education and enhance the learning experience [3].

These scenarios add additional complexity in terms of heterogeneity, scale, and dynamics that cannot adequately be addressed by old fashioned building automation systems. Similar challenges can also be found in other application areas of cyber-physical systems, and a number of projects aim to develop middleware and frameworks to mitigate them. These offer various abstractions and interoperable interfaces to link together distributed physical components and applications [4, 5].

It is also to expect that the previously mentioned claim that building automation logic mainly consists of feedback loops of continuous variables, no longer holds true. Additional advanced reasoning and analytical capabilities are demanded. Discrete variables, like the number and identity of people in a room play an increasing role in the coordination logic. Also, complex reasoning tasks or aggregation of information over time and sources is becoming more common. Since this data is part of the nonphysical state of the IoT system, measurement and reasoning errors can accumulate over time and lead to unwanted action. At the same time the topology of devices and rules in an IoT network is becoming more dynamic through the integration of mobile devices brought in by people and vehicles.

It is furthermore expected expected that devices will become active participants in the environment and processes in which they are deployed, rather than passive sources and sinks of information [6].

These changes require powerful frameworks that are able to accommodate the various needs of the applications and provide interoperability despite heterogeneity.

2.1.3 Related work

There are a number of surveys that compared IoT middleware products in terms of their features, capabilites and interfaces. A very throughout one was published by Razzaque et al. [5]. The paper names and describes important properties of IoT, distinguished by IoT infrastructure- and application-related ones. From these it deduces functional and non-functional requirements of an IoT middleware. The core contribution are the description of a classification of core concepts that middleware architectures provide as an abstraction to their users, namely event, service-oriented, virtual machine, agent-based, tuple space, database, and application-specific approaches. Event-based middleware uses messages or streams to couple components together. Service-oriented middleware adapts the service orientation paradigm. VM-based middleware provides a platform for the secure execution of IoT applications by virtualisation of the infrastructure. Agent-based middleware provides location transparency and recoverability by managing the state of autonomous applications. Tuple-space middleware overcomes concurrency issues by assigning each to a unique owner at any time. Database-oriented middleware provides querying and processing functionality for sensor data. Application-oriented middleware enables new features for a particular application. This framework is then used to give an overview of middleware research projects.

With regard to data storage and processing, researchers have already looked at the conceptual challenges imposed by the introduction and evolution of the IoT. In particular Cooper et al. [7] have identified a number of challenges that the Internet of Things poses for database management, namely

- Size, Scale and Indexing
- Query Languages
- Process Modelling and Transactions
- Heterogeneity and Integration
- Time Series Aggregation
- Archiving

The issues arise because of the fact that data generated by the IoT is typically vast, distributed and unstructured. These characteristics are fulfilled by the applications that

DS2OS aims to support. Thus the mentioned issues arise when designing an appropriate database backend, and have to be addressed. The paper concludes that the problems are not trivial and deserve closer examination, but does not provide possible solutions.

Other researchers propose specific solutions. Li et al. describe a storage system for IoT data [8]. They assume a unidirectional flow of data from sets of sensor objects, thus disregarding the decision-making, control and actuation aspects of IoT systems, which stands in contrast to DS2OS' design. Instead they put the focus on spatial and temporal context and efficient querying of object data based on these features. For these requirements they propose that the ideal database system favours performance over consistency, is schema-less, sharded to allow scalability, and has a uniform interface for access which also provides distribution transparency. The requirements on a database in this work here are different since the database is also used to provide a layer for coordination between sensors, actuators and services, reflecting object state rather than only historical sensor readings, and thus requires consistency and transaction support. In addition there is no particular support for indexing and querying spatial context, which relaxes querying requirements.

Ma et al. propose a conceptual model for various data processing and management steps which occur in the IoT and conducted a survey on approaches and projects for each of these [9]. Their reference model distinguishes between the functional aspects data cleansing, semantic enrichment through event processing methods, data persistence and querying, and application integration through middleware. Put into this model, with the middleware product given, this work focuses on data persistence and access (storage and analysis layer), which covers exchange, storage, compression and mining. In this category, the main challenges lie in semantic expression of the data and scalable processing frameworks. The role that data plays for coordination and control of devices, and how this is reflected in the persistence layer, are in this work here important aspects, which the theoretical framework does unfortunately not cover in detail.

Jiang et al. propose a storage architecture for IoT data that leverages the idea of distinguishing structured from unstructured data [10]. These are stored in separate databases – a key-value store for unstructured and a relational database for structured data. The platform offers a query interface through a REST webservice that can operate and even execute join queries on both, and provides namespaces for multi-tenancy. Unfortunately the authors forgave the opportunity to leverage the transaction capabilities of the relational database and pass this feature on to the outer interface, although both read and write operations are accessible through it.

A number of previous research evaluated databases in hindsight to their suitability for IoT applications or propose storage architectures for IoT related data. It is important to note though, that these evaluations are always tied to a specific scenario and application at a specific place in a specific IoT framework. As already pointed out, the diversity among IoT middleware architectures varies considerably and so do use cases for databases.

Performance measurements of databases with IoT data were conducted conducted by Copie et al. with Riak and MongoDB [11]. They investigated the impact of sharding with small cluster sizes (1 and 2 nodes) on the latency of operations. While they did not describe the exact workload that they used, the scope only captures sensor data. They did also not explore the advantages and disadvantages of the two databases in terms of their qualitative features. This is one of few performance benchmarks of databases that specifically targets an IoT workload. Despite that, the kind of data is very different from the one that DS2OS produces, since latter mixes up the sensor readings with derived and internal state, and control commands.

A similar study was undertaken by Phan et al. [12]. They compared the performance of read and write operations with sensor data between MySQL, MongoDB, CouchDB, and Redis. They did however only measure their performance in a non-distributed database setup, with a variable number of concurrent local readers and writers. They concluded that it depends on the type of data and access patterns to decide which database product is the most performant choice. CouchDB showed low performance in general, while MySQL and Redis achieved much better results.

MySQL and MongoDB were also compared in a benchmark by Paethong et al. [13]. They evaluated them with special regard towards power consumption on low-end devices, in particular the Raspberry Pi, which is often used in low-end IoT deployments. To mitigate the storage constraints of those devices, the setup uses sharding, which is implemented on top of the database layer and is thus similar to the architecture of the middleware analysed in this work. Apart from the power consumption they measured the latency for batches of various database operations on scalar values. The results showed that both databases perform better on normal x86 computers, although at the cost of higher energy consumption.

Fatima et al. compared the performance of MySQL, VoltDB and MongoDB, notably databases with different designs, with IoT sensor data [14]. The setup did unfortunately not include distribution of the data, although this is an important distinguishing aspect between the three test candidates. The results showed a significantly better performance for VoltDB, which is not surprising since VoltDB uses in memory storage.

2.2 DS2OS

DS2OS is a middleware for IoT applications and the software that the implementation part of this work is based on. It is the outcome of a research project that has been documented in the PhD thesis of Marc-Oliver Pahl [15]. The thesis describes its motivation, design considerations and implementation in detail. The project's primary objective is a real-world-usable orchestration service that allows developers of IoT applications to access diverse physical and virtual objects and combine their functionality with automated coordination services. This section shall summarise the important points of its design, analyse the how the current implementation handles data, and which semantic properties the database interface provides. It shall also identify aspects that impose requirements or limitations on possible changes and improvements to the database layer.

2.2.1 Design and Objectives

DS2OS is described as “Smart Space Software Orchestration”, which is in turn defined as the management of Smart Devices in order to create a Smart Space [15]. Smart Devices are physical devices that are linked with a computing node and provide remote communication via a network. Formerly found mostly in professional settings, they are increasingly available on the consumer market. Their capabilities can only be leveraged when a multitude of such devices are linked together to allow more complex behaviour. Unfortunately this remains challenging due to non-standardised or incompatible interfaces. DS2OS, among other IoT middleware, seeks to overcome this by providing a flexible, uniform access layer that makes this heterogeneity transparent for developers. To accomplish this the original dissertation presents a number of components and requirements of a middleware. On a functional level these are the following (summarised):

- Context models provide at the same time semantic description for objects and syntactic description for their interfaces and aid towards interoperability through access transparency, standardisation, and extendability
- A dynamic service oriented architecture that fosters encapsulation, abstraction and re-usability of modules
- A resource discovery service to provide location and migration transparency and allow applications to dynamically look-up resources by type
- Flexible communication methods that allow push- or pull-based, persistent or volatile, and exclusive or shared interaction between services and devices
- Multi-user support and authorisation

Apart from that ease-of-use, scalability, availability and security have been identified as important non-functional attributes and are incorporated into DS2OS' design.

DS2OS is realised as a distributed peer-to-peer system with group communication. Peers, also called knowledge agents (KA), discover each other by means of IP broadcasts, limiting deployment to infrastructures that are fully connected in an IP subnet and thus are co-located at one site or connected trough a Virtual Private Network (VPN). The peers gossip group membership to each other, so that eventually each peer knows about the existence and address of every other peer. When peers communicate directly with each other, their communication is encrypted with a peer-specific private key by SSL with mutual authentication. Communication between multiple peers is enabled by network multicasts which are symmetrically encrypted with group key that is negotiated through gossiping.

2.2.2 Developer-centred view of DS2OS

Developers who want to use the DS2OS middleware must encapsulate the logic of their application in a so-called service which acts as a client to the middleware. This concept applies regardless of the kind of desired functionality of the application. The services can communicate with each other through the Virtual State Layer (VSL). The VSL is a global state storage that has an API similar to a database. Basically, services can read and write values to data locations identified by addresses. They can also subscribe to be notified when a value changes and make remote procedure calls through the same API.

Developers are encouraged to reflect all the state of their services on the database interface, so that the services themselves can easily be terminated and restarted withut losing their state.

The data in the VSL is described by composable and extendable context models that are shared throughout the deployment. Services can easily discover resources by searching for a given type. Each service has one such context model associated to itself in order to describe its properties and interfaces.

2.2.2.1 Service classes

There are various use cases in regard to which functionality shall be implemented. These are described in the following and map to a service class as described in [15].

1. Control an actuator, such as a motor, light, or valve setting. In this case the developer would provide the physical component and hardware to control it. Linking the actuator to DS2OS happens with a so-called adaptation service. Those services typically take a desired value as input, adapt the physical component state and write the resulting value to a different tuple serving as input to other services which are interested in the current state of the component.
2. Provide sensor data, for example from a light, pressure or temperature sensor. In DS2OS terminology those services are also referred to as adaptation services. They take the sensor reading periodically, transform it into meaningful data (such as the temperature in degrees Celsius) and write it to the VSL.

3. Implement coordination functionality, for example set a valve of a radiator dependent of a temperature measurement. In DS2OS terminology they fall into the orchestration service class. Typical control procedures in building automation follow an open- or closed loop.

4. Inferring aggregate information, for example energy consumption over a day.

5. Reason about circumstances, for example estimate the number of people in a room by light barrier measurements, the utilisation of the WiFi access point, noise levels etc. This might involve techniques from statistics or machine learning.

The service classes are not a construct that must be explicitly denoted in a service implementation. It rather serves as a framework to classify use cases and provide recommendations for their implementation. This is in so far relevant as it should be demonstrated that services of the identified kinds can access the database efficiently.

2.2.2.2 Context models

Context models describe the data that is stored in the VSL. The original dissertation suggests to understand them “as templates for representing properties of the physical world in the virtual world” [15]. Technically they represent a schema that describes the data and interface, like WSDL does for web services and XSD for XML. The description shall be both syntactic and semantic, as it is the case in object oriented programming. DS2OS requires that each service has exactly one equally-named context model, that describes its data and interface. The XML files can be stored at a Central Model Repository, which is a server with a well known address, at a site-local mirror of that repository, locally in the file-system, or a combination thereof. The models can be cached as they are not supposed to change. An central goal the DS2OS project is to create an infrastructure of repositories for distributing context meta-data publicly. But this task is still in progress. Figure 2.1 shows a context model for a multi-colour lamp as an example.

```
<rgbLamp type="/basic/composed">
  <brightness type="/basic/number" restrictions="minimumValue=’0’,maximumValue=’1’">0</
    brightness>
  <colour type="/basic/text" restrictions="regularExpression=’^#([0-9a-f]{3}){1,2}$’">#
    ffffff</colour>
</rgbLamp>
```

Figure 2.1: Context model example for a multicolour lamp.

2.2.2.3 Addresses and Namespaces

Addresses are used to identify and access tuples in the VSL. The addresses are denoted in the well-known URL format, with slashes as separation between identifiers. They contain the location of a service, the service identifier and the path within the hierarchical structure as defined by the service's respective context model. An address that is valid within the DS2OS system has to start with the agent identifier, followed by the service identifier, and then the location within the service's node hierarchy, e.g., `/agent1/lamp5/brightness`. On top of that, virtual nodes can be registered.

2.2.2.4 Composition

VSL context models can be a scalar value, such as a string or integer, or contain of list of subnodes thus forming a hierarchical structure. Each node further has a type indicating which values are valid. Basic types are text, list, and number, but additional types can be defined through restrictions expressed by regular expressions or the minimum and maximum allowed value. Furthermore, every context model itself forms a type, which can be referenced by other nodes through its model name. Additionally, existing models can be extended with additional subnodes through inheritance of a parent node, in a manner similar to inheritance in object oriented programming.

2.2.2.5 Access Control

VSL nodes have two fields to store read and write permission respectively. The possible values are either a list of the identities of authorised services, an asterisk as wildcard indicating unrestricted access, or an empty value meaning that access is strictly private. The permission to use subscription and locking methods on a node is also restricted by these two values.

2.2.2.6 Connector interface

Services provide their own runtime environment. They communicate with the VSL by obtaining an instance of the Interface `org.ds2os.vsl.core.VslConnector`, which handles network communication with the knowledge agent. An interesting feature of DS2OS is that various mechanism of process coordination are bundled in the same interface. There is the register-type pattern where clients access the same datastore, event-based coupling with publishing and subscribing to information, and finally a message-passing approach where some remote code is executed on request. As mentioned these mechanisms are used through the same interface and are thus transparent to the client.

Register-based communication—Register-based communication means that clients that want to interact, access the same datastore and namespace concurrently. In this case the datastore is the VSL. Clients can read and write values by calling `VslNode VslConnector .get(String address)` or `void VslConnector.set(String address, VslNode knowledge)` respectively. In both cases, the `address` refers to the VSL address, and the `VslNode` to the value that is written or read.

Event-based communication—Event-based communication means that there are publishers which release information, and subscribers which are interested in it. An event broker then forwards information to interested parties. DS2OS provides the `void Connector.subscribe(String address, VslSubscriber subscriber)` method which allows to register subscriptions on a VSL address. The publishers release information as before via the `set`-method. The subscriber has to provide a callback method that is wrapped in the `VslSubscriber` class. This method is then called by the knowledge agent whenever new data is available for the registered address. A peculiarity is that the client is only informed – the updated value is not delivered and has to be requested explicitly by the client through the `get` method. The notification provides the exact address where the update occurred, which can be a child of the subscribed address. It should be mentioned that it does not include the version number, so there is no guarantee that the subsequent request for the value returns exactly the one that lead to the notification being issued. By the semantics of the database interface it is guaranteed that it is the latest though. Publishers can also issue notifications for its virtual nodes, which are described in the following section.

Method-Invocation-based communication—Clients can register so-called Virtual Nodes which serve like conventional VSL nodes with the difference that they can be inserted into the VSL dynamically and have no persistent data. To insert such a node, a client can call `void Connector.registerVirtualNode(String address, VslVirtualNodeHandler virtualNodeHandler)`. The programmer has to provide an implementation of `VslVirtualNodeHandler`, which consists of callback methods for `get`, `set`, `subscribe` and `unsubscribe`. These are then executed whenever another client issues a respective operation on the registered address.

This feature constitutes a form of remote method invocation (RMI). In combination with other functionallity of DS2OS, like context models and resource location, this becomes a service oriented architecture (SOA) [15].

2.2.2.7 Data management

As noted before, DS2OS provides two interfaces for communication with clients. One is the VSL, the other the context model store that describes the VSL data structures semantically. But from an implementation perspective there are also other pieces of information that are necessary to provide the interface in its current design and is implicitly. Examples for that are neighbour (connection) information, locks, subscriptions for tuples etc. Those also represent state in technical sense and special consideration must be taken on how to manage this state in a distributed system.

In the current implementation every service client is associated to exactly one knowledge agent. This knowledge agent manages the VSL state associated to that client all by itself. The task also includes keeping subscriptions and locks. Since the URL to access those data items always includes the name of the knowledge agent, it is implicitly clear which host to contact in order to register a subscription or to set a lock for a particular tuple. This association is lost when there is one global database (=state) abstraction. It can of course be used to store the data, but it is not immediately clear which host would be responsible to send out notifications for existing subscriptions or who has to check whether a VSL operation violates a safety constraint.

VSL data is stored locally at the knowledge agent with which resources are associated. Data can also be accessed remotely; in that case the KA would extract the location of the remote KA from the VSL address and route the operation call there. The data is persisted in a federated HSQLDB database, which is a file-based, embedded Java DBMS that offers SQL support through JDBC and ACID-compliant transactions. New data is only appended to the database, so if there is an update to an existing tuple, a new one with an incremented version identifier is created and stored. Old versions can be explicitly queried through the VSL API, but otherwise always the latest value is returned. Each KA also keeps a local structure that stores the locks and information to rollback incomplete transactions in memory. From an external perspective, the VSL itself can be regarded as a sharded distributed datastore.

Consistency—Since DS2OS' interface resembles that of a simple database, it is interesting for application developers to know which consistency guarantees it provides. In this context, it should be mentioned that the term consistency can be used for different things in connection to distributed databases. What they have in common is the idea that concurrent or dependent operations should lead to a more or less deterministic outcome that does not violate defined safety constraints. We commonly distinguish between transaction consistency and the consistency of a replicated register. A transaction is a sequence or a partially ordered set of operations on a number of database elements. The other refers to the consensus of a single database element that is replicated among a number of nodes.

The conventional implementation provides atomic consistency on a record (VSL node) level since there is no replication of VSL data: Each node is uniquely assigned to one KA and only stored in a local database there. This database system itself orders operations and executes them atomically, which means that there is a single point of truth for each

VSL node and a deterministic sequence of state changes for each local database. Due to the atomicity guarantee, there is a specific point in time at which the state of a DB changes. This implies that there is a natural global sequence in which state changes occur. Another fact that is worth noticing is that the clients communicate with the KAs via a blocking API. Therefor requests cannot get reordered by the network on a client level which means that they all see the same sequence. Operations are also strictly executed at some point in time between the request and the delivery of the result. Together these properties satisfy linearisable (or atomic) consistency as defined in [16]. Linearisable consistency also entails causal consistency. That is means that if a state change A causes a client to commit a write B and thus another state change, any process able to observe B can also observe A.

Transactions—Another, but different important safety feature are transactions. These are sets of operations that have to be executed in a defined order and in isolation of other concurrent transactions. DS2OS does not provide an idiomatic transaction feature. Instead there is a simple locking mechanism. A service that wants to isolate a set of operations from conflicting ones has to set and release the locks itself. This is done by acquiring an exclusive lock (shared locks are not part of the specification) on a VSL node tree before it is accessed for a read or write operation. After the operations have been carried out, all the locks either have to be committed or rolled back. As committing only a subset can lead to inconsistency in the form of lost updates, it is important that either all or none are committed. Since there is no atomic commit operation, the client itself has to make sure of this behaviour. It also has to ensure that locks are held long enough, because if the client does not release them within a pre-defined period of time, DS2OS does that automatically.

Also, if a lock cannot be acquired because of a concurrent transaction, it is up to the client to decide how to proceed. It can either wait for the lock by calling the lock command again after some time, or abort the transaction. If several clients apply the former behaviour then there is a risk of deadlocks. The latter alternative in contrast is prone to livelocks. These are the only alternatives since DS2OS has neither a mechanism to abort transactions and release their locks from the outside, nor an algorithm to detect deadlocks.

It is clear that the lock interface does not provide carefree transaction support with the usual interface. A similar behaviour can be achieved with the provided locking methods but they should be used very carefully. To ensure isolation, service clients must adhere to a locking protocol (like 2-phase locking, 2PL). Atomicity can only be guaranteed if neither the client nor the network fails during the commit phase, and no locks are implicitly removed from the client due to timeouts. It is wise to implement services so that they can compensate the effects in case of a failure. Given these shortcomings the locking interface is hardly suitable for widespread use of transactions, e.g., for decision-making services that check if a certain premiss is fulfilled and apply a resulting action. The question arises if they are always necessary and meant for widespread use. The dissertation [15] motivates the locking feature with the need to set configuration data consistently. Such a task would only be carried out occasionally as a result of administrator intervention, in contrast to operational coordination services that would access data more regularly. Due to the tree-shaped structure of the VSL it is possible to lock a complete KA, so that for most interventions only one lock has to be acquired. In this case the lock commit operation is atomic.

To support ACID compliant transactions, the locking API needs at least a command that commits a set of locks atomically rather than a single one. Even better because simpler to use would be an interface call that marks a block of read and write operations as transactional and implicitly acquires locks for all accessed VSL nodes, thus hiding the details of the concurrency control algorithm from the developer. Additionally, a way to define program behaviour if the transaction is aborted by external means (e.g., by raising an exception) would allow to transfer transaction scheduling to the database. This is relevant when transaction support offered by various distributed DBMS products shall be used. These typically feature the outlined interaction pattern, originally conceptualised by Jim Gray [17], and require changes to the VSL API for compatibility.

Meta information—DS2OS associates various pieces of meta-information to each VSL tuple. Some of that information is determined statically by the context model (schema) that defines the structure of the data, some is set dynamically.

- Type information
- Latest version
- Reader identifiers
- Writer identifiers
- Subscribers
- Lock state and owner
- Update timestamp
- Caching property

The context-models that describe the VSL structure semantically are expressed in XML and stored in a central location. They specify the data types, access rights and default values of data and can be composed to express complex semantic structures.

It is part of the design of DS2OS (reference) that data of different nature is accessed with the same methods. Other models do make a distinction that is not necessarily transparent to the application developer. The reference model by Abu-Elkheir et. al. [18]

for example suggests dedicated data stores for metadata and object catalogues, for structured and unstructured data and for aggregation and query layers. It also suggests to distinguish between temporal and modal data. The former continuously update sensor readings, while the latter do not. This is in so far relevant as various types of data have different access patterns, such as the update frequency, the proportion of read and write operations and the number of concurrent writers. They might also have varying consistency requirements.

Chapter 3

Databases for Smart Spaces

The current approach of federated data storage in DS2OS is not yet optimal, because it lacks tolerance to failures and support for transactions. Scalability is in so far provided as the dataset can easily be partitioned by assigning services to other knowledge agents. The obvious alternative, using a centralised database, lacks this property, and is thus clearly not an option.

In this chapter, available datastores shall be evaluated in regard to their suitability to a decentralised state storage for IoT. The approach is to first describe the variations of architecture and features among distributed databases, decide if they are beneficial to the application, and then find candidates of database products that satisfy the requirements.

3.1 Requirements

3.1.1 Data meta-model and structure

Among databases, there are different ways how the data is conceptually described and physically organised and indexed. This is very closely related to the types of queries that a database supports, their performance and versatility, and flexibility in general and can thus render databases more or less suitable for certain applications. Thus, this dimension is of particular importance, and therefore most commonly used to classify database systems. The common data models are:

Key-value is the simplest one. It is basically a unidirectional mapping between a key of a certain type to content of a certain type. Typically strings or hashes are used as keys, and the value can be arbitrary binary code or a document format like JSON or XML. Typically it is only possible to query based on a key, sometimes also on key ranges. Some key value stores are marketed as grids or caches (e.g., Hazelcast, Infinispan) and some also provide additional indexing capability (e.g., Riak, Infinispan).

Document -oriented databases store semi-structured XML or JSON data. They are similar to key-value stores in that data is identified by a key, but in contrast to them the meta-format of the value is known to the database and can thus be interpreted. This allows the database to index the value in order to offer more advanced querying support. The internal representation can vary, but is typically suited to hierarchical structures.

Relational is the most established for persistent storage. They store data in tables which are described by a defined schema. Databases of this kind typically offer a rich set of features, including support for transactions, advanced query support including joins between datasets, aggregate functions and secondary indices, all commonly provided through a SQL interface. Some of these features can be costly to implement and execute in a distributed environment [19], though there is a class of recent software projects, so called "NewSQL" databases that aim to implement this functionality in a distributed database [20]. Physically, the data can be organised either in rows or in columns. The latter are designed to use less space for sparse data and perform better for queries that concern few columns but many records, which is common in analytic workloads.

Wide-column stores, sometimes also called "extensible record" stores, share properties with the three so far mentioned. They share with the key-value model that data is identified by a key, and with the relational model that data is organised in tables. The important difference to the relational model is that the schema of columns is not defined statically, and no space for empty columns is reserved. A row can instead be seen as another map of keys to values, making it a two-dimensional key-value store, though only the first level is sharded over the network so that there is some control on which data ends up together on one network node. They can be seen as a special case of the document-oriented model, restricted to only flat documents.

Graph databases store data that represents a graph, so nodes and edges. There is no general standard how those data items are represented, and how the graph data is organised physically. Graph databases emphasise associations between records and offer query support for graph traversal.

For storage, DS2OS data imposes relatively few requirements on the data model. The VSL data has a hierarchical structure and can be written as XML data, which would suggest a document model for adequate representation.

However, it is also possible to represent it as a mapping from a path to a leaf node within that hierarchy. The version identifier can be either part of the path or the value contains a list of the most recent versions. Metadata can be stored along with the node content. In this model, querying a subtree would constitute a range query on VSL paths. So to get the full data of `/ka/srv/`, a query from `/ka/srv/` to `/ka/srv/~` would return the relevant tuples. It should be noted that not all key-value stores support this kind of query. Alternatively all possible paths can be computed from the context model, though this would result in a much higher number of queries.

The same behaviour can be achieved with wide-column stores, by taking the URL that identifies a service as the key for a row and the internal structure of service data as the column. This way all the service data is stored on one physical node and can be retrieved in one package or with given granularity.

Since the VSL address is the only way to access data in the VSL, there are no requirements on join query support or secondary indexing capabilities, thus a key/value store is sufficient, although document-oriented, relational or wide-column data models work as well. Generally speaking, the models are somewhat similar on a conceptual level in that they can easily be mapped onto each other. A document store can be mapped to the wide-column model, by taking making the document identifier the row key, and the path within the document the column key. A wide-column store can be represented in the key/value model by concatenating row and value keys. A transactional wide-column or key/value store can be extended with indexing and schema capabilities to yield a relational database, as the internal architecture of CockroachDB demonstrates. Lastly, a a relational table with a primary key and one other column behaves like the key/value model.

R-1 New data, consisting of a character sequence (VSL address) as key and arbitrary binary data or a character sequence as payload, can be inserted into the database.

R-2 A single record can be retrieved by its key.

R-3 A written record can be updated with a new value.

R-4 Hierarchies of VSL nodes, beginning with a prefix of knowledge agent and service identifier (such as "/ka1/service2/") can be fetched efficiently in a bulk read.

R-5 To ensure R-4, the database has to maintain locality between records with a common VSL node prefix. This can either happen by a 2-dimensional data model as provided by a wide-column store, or lexical ordering of keys for sharding and disk storage in all other databases.

R-6 To ensure R-4, the database forthermore has to provide a bulk read or range query operation for keys that start with a common VSL node prefix, or a read operation for a full row in the wide-column model.

3.1.2 Interface

Databases are either embedded, and interact with the client application by direct API calls, or they communicate via a network protocol. If they use the network, the database manufacturer provides libraries that hide the details of the interaction under a transparent API, which either allows direct access to particular database operations or encapsulates a query language. Relational databases are typically access with SQL or a derivative language. In Java, SQL interfaces are usually made available through the standardised JDBC API. Key-value stores do typically not provide an SQL interface, as they allow only simple read, write and update queries.

The database interface API also handles security-related functionality, such as authenticating the client at the database server, and encrypting the connection. As confidentiality and integrity are important design considerations of DS2OS, both features have to be supported.

R-7 The database has to support a blocking or non-blocking Java API, that supports the database operations subsequently identified as required.

R-8 The database supports a SQL-based API, which is accessible through JDBC. This requirement is an alternative for R-7.

R-9 The database offers functionality to authenticate a client prior to creating a session. This is to maintain confidentiality and integrity of the VSL data.

R-10 The connection between database and client is encrypted. This is to ensure that transmitted data cannot be read by an unauthorised third party (maintining confidentiality) or altered (maintaining integrity). It is not necessary if the database is embedded.

3.1.3 Storage Location

There are two possible ways on how databases physically access their data: Either they place it in memory, or on secondary storage devices like hard disks or solid state (flash) memory.

Memory storage has the major advantage of very low latency in comparison to harddisks. It provides fast fine-grained (random) access through memory addresses. Thus the order in which data is stored and its fragmentation is much less a consideration as compared to disk storage and databases using this type of persistence achieve lower latency and higher throughput for random access operations. On the downside main memory is much expensive and it is volatile. The data does not survive system restarts or crashes. To allow recovery it must be shadowed to non-volatile storage (e.g., through an write-ahead log), or replicated.

Conventional disk storage is cheap, non-volatile and provides high throughput for sequential read and writes. In contrast to main memory it has to be accessed indirectly through a filesystem abstraction provided by the operating system. Filesystems are typically not optimised to store many separate items with only small size, so they have to be combined in a file with an internal structure for retrieval. Physically, harddisks only provide block-level access, which means that are several kilobytes of data that are fetched or written as a piece, so that there is no fine-grained random access. Also they are relatively bulky regardless of their capacity and have a high energy consumption, which can limit their suitability for embedded applications. In contrast to that, flash solid state storage does not have theses shortcomings and on top of that a lower read latency. However this comes at the price of considerably lower write performance.

For the database developer the choice between main memory, flash or disk storage has implications regarding the optimal combination of indexing and concurrency control algorithms. For the user of a database product, the decision between both is all about the trade-off between high throughput and low latency versus price and extensibility, so all questions of quantity. A main memory database has lower latency for OLTP-style access patterns than a conventional database with comparable characteristics [21].

Furthermore, volatile storage usually means that a fail-stop model is assumed (e.g., Scalaris), where a node that is considered to have crashed is removed permanently from the network overlay. Non-volatile storage in contrast does not lose persisted data and allows to be taken online again (e.g., CockroachDB). The decision between either does not limit the durability of the data, as this can also be provided through replication.

It has to be mentioned that the classification between both types is not so clear in practice. Modern databases may provide a choice between both, or even combine them transparently. Sometimes the module to actually persist data can be changed or reconfigured. Operating systems also cache open files in memory to decrease latency and extend the main memory with swap files stored on a harddisk. Lastly solid state storage can be used like disk storage through a filesystem but provides random access and low latency like main memory.

Operational smart space workload as it is modelled by DS2OS, involves many small pieces of data that are updated frequently. This would carefully hint towards main memory or flash storage. However as mentioned the decision rationale is a quantitative one that has to take into account the current state of hardware development and prices. Therefore this is rather a soft requirement, or a point to take into consideration, than a criterion for exclusion. An optimal candidate would provide both options to choose from or combine them transparently to leverage their respective advantages.

R-11 The database maintains durability of write operations.

R-12 The database leverages the latency advantages of memory either by transparent caching or complete in-memory storage.

3.1.4 Partitioning (Sharding)

Partitioning or sharding means that a dataset is divided up and stored at different locations. The technique is used to enable scalability through a distribution of database load on multiple nodes and can be combined with replication.

Highlighy scalable, and mostly automatic partitioning of data is the distinctive feature of modern distributed datastores.

R-13 The database must shard data automatically and homogenuously across the cluster.

R-14 The database must support functionality to rebalance the distribution when nodes are added or removed.

To locate and access sharded data, there are in principle two options: One is to use a Distributed Hash Table (DHT) and route the communication on a peer-to-peer basis through the resulting overlay topology. There are a few such algorithms available [22], but obviously the most popular for distributed databases are algorithms reminiscent of Chord [23], as used by Cassandra, Amazon Dynamo, Riak, Voldemort and Scalaris. These locate data on a self-healing ring-shaped overlay structure, with shortcuts to route requests to the target quickly. Advantages of this concept are hiegh scalability due to the routing mechanism and availability due to the lack of a single point of failure, and the self-healing capabilities. The other option is a directory which maps keys of the data to its destination address. This approach has the advantage that after address resolution, the node which holds the data can be contacted directly, thus without the routing overhead of a DHT. The directory can either be kept in a central location, as it is for instance practised with Apache Hadoop (prior to version 2) where the directory resides on the NameNode [24], or replicated to achieve an improvement of scalability and availability.

3.1.5 Replication

Replication describes that the same data item is stored on multiple nodes. This can happen for various reasons, namely for reducing read or write latency, sometimes to improve scalability in read-heavy workloads, but foremost to improve availability when nodes fail. To achieve the latter, replication is the only viable option and therefore a necessary requirement for a smart space database. If the objective lies instead on achieving performance gains, replication it is often referred to as caching.

Replication poses challenges for consistency. As mentioned, linearisabe consistency is desirable, which requires that every read operation must return the value of the most recent confirmed write. Distributed databases can achieve this through a number of different protocols. Their key mechanism is always to achieve an ordered sequence of state changes. This can happen by routing all write operations through a leader which ultimately decides on a write sequence. In this case there is the additional problem of deterministically electing a leader and recognise reliably when it fails, so that the database stays available. Alternatively, there are algorithms which write or read from all nodes instead, or use quorums. [25] provides details on relevant available algorithms. The key takeaway from there is that there is no algorithm which satisfies all the previously mentioned properties. To read the definite latest value from any replica, a precedent write operation has to update all replicas before it can be considered successful, leading to additional roundtrips and time required to discover probable node failures. Alternatively, quorums for both read and write operations can be used, but these also come with an overhead compared to non-replicated storage. Ultimately this is implied by the CAP theorem.

R-15 The database must replicate data with a configurable redundancy transparently.

3.1.6 Concurrency control

Many applications that access and manipulate the state of a database want that the effect of their action is internally consistent. A frequently employed example is that of a withdrawal of a certain amount of money from a bank account. For that, a process reads the account balance from the database, subtracts the amount and writes the result back to the database. It is important that the updates on the record correspond to the sum over the actual flows of money, otherwise there is a breach of integrity and the customer ends up with more or less money than there should be on their account. The straightforward procedure should typically accomplish this, but there are issues when another concurrent process accesses the account record. It could be that two processes read the balance at the same time and then carry out their updates. Then only one withdrawal action will be effective. This is also called a “lost update” [26]. It could also happen, that a different process adds money, and increases the balance. A withdrawal process reads the new value, and completes its action by writing back the new amount. After that the first process wants to abort the top-up. This situation is called a “dirty read” [26]. In another situation a process has to read the value twice for some reason, but due to an interfering withdrawal the amount has changed in between. This “unrepeatable read” is undesirable [26] in many cases, though not in every one. For this reason, some database products allow to opt in or out from some guarantees.

Actions that demand consistency can occur in building automation systems as well. For example to switch on a certain cooling system, the cooling device and a number of fans to distribute chilled air in the building have to be switched on together. It wouldn’t make sense if a concurrent control process would switch off either the cooler or some fans in the middle of this. In more severe cases, safety constraints could be violated, for example when a robot that is only allowed to operate when a door is locked and the door and the robot are not accessed transactionally.

To avoid the described situations, related operations should be combined to a transaction. Transactions are sets or sequences of operations that are executed in isolation of each other. That means that when they are executed, a scheduler puts them in a partial order with conflicting transactions (Conflict serialisability, CSR). It may also abort transactions and roll the database back to the original state. The client that issues a transaction has to be aware that, depending on the scheduling algorithm, its transaction can either be aborted midway, or that it has to wait until locks are released. Transactions can involve multiple different records, but are typically constricted to an instance of a database system, so state-changing side effects on systems outside of the database have to be compensated by the client when its transaction aborts. The database system has to include algorithms that order the execution of transactions, make sure that they don't interfere with each other, and that changes of succeeding transactions are atomically applied.

There are two classes of algorithms that can be employed to achieve isolation and atomicity [27]. Pessimistic algorithms take action (delay or abort a transaction) as soon as there is an indication of a conflict. Optimistic algorithms wait until a transaction wants to commit and only then they check for conflicts and decide which one will eventually succeed. Subsequently, the changes applied by a succeeding transaction must be stored durably and those of aborted transactions must be recovered to their previous state. On top of that, in a replicated or sharded database, it must be ensured that all involved database nodes execute the transaction consistently, meaning that either all or none decides to commit and carry out the resulting state changes to their dataset. This is a consensus problem and there are algorithms, namely Two Phase Commit Protocol (2PC) and Paxos that solve this in a distributed setting [28]. The 2PC algorithm uses a transaction manager to coordinate that asks participants in the transaction to make sure that they are able to commit, decides to commit if there is a consensus that the commit is possible and – if it is – then requests the participants to execute the commit. This procedure is vulnerable to node failures. In particular if the transaction manager fails during one of the phases, it cannot proceed.

It is a debatable question whether Smart Space Orchestration needs support for strong ACID-compliant transactions. Algorithms like Paxos and 2PC are tedious to implement and come with a considerable messaging overhead. Some database products such as for example Cassandra or Dynamo promise to achieve higher performance by relaxing consistency guarantees. And this might actually be sufficient for many scenarios. In a building automation setting, coordination services that have more than one input or output, are typically open or closed loop control circuits. For example they might adapt the setting of a valve that supplies a radiator according to a temperature measurement, a target temperature and the current valve setting. In this scenario it is less of a concern if the input parameters are the latest available ones or whether the last temperature measurement happened before or after the last valve adjustment, but more so if the input parameters are recent in a chronological sense. Independent from that there is the need to change system parameters and configuration data in a reliable way, as previously recognised. However those are typically rare, don't involve a high number of competing writers and limited to a certain subtree, so that a simple local locks might actually suffice.

Given the obvious performance drawback of distributed transactions, they should only be implemented if there is sufficient need for them. We can provide the following arguments in favour of this:

Achieving statelessness. .Guidelines for DS2OS service developers explicitly want services to store their state in the VSL, so that they can be stopped, restarted or migrated without causing safety or consistency concerns. Since they are imperative programs, they have both variable state (current values of variables) and program state (current line and stack trace), so if real statelessness shall be achieved, both have to be persisted. Doing that for variable state is intuitive for the programmer; they simply operate with the VSL API's get and set calls instead of local variables. For program state, this is not so easy, as it requires to change the design of a service, to achieve something like a state machine, that in each service iteration reads the program state from the VSL, executes an operation, and saves the new program state, but also with this technique, it is impossible to make service executions atomic. If there is support for transactions, there is an elegant solution to this problem: The entire code of a sequential service execution is wrapped in a transaction. This makes all the state changes that it does on the VSL database atomic, so that it actually becomes stateless and can be restarted at any point.

Guaranteeing safety properties. Certain parameters must always appear in a consistent state or otherwise a major problem occurs. Consider for example, a pump that is connected to a pipe with valve. When the pump is switched on, the valve must be opened too, or otherwise the pump would overheat or the pipe burst. And when the pump is switched off the valve must be closed to prevent an undesirable counterflow. It is obvious that a service accessing these two devices must always access both, and keep their state consistent. This cannot be guaranteed in a non-transactional system, since when a concurrent access occurs, both services' operations interleave in arbitrary order. This problem can be mitigated by allowing only one service private access to the system of valves and pump, which would then take configuration parameters and set the state of the devices accordingly. This approach is possible only as long as there are clear boundaries between the systems, with no shared components – in the example of the pump and valve, if a valve is connected to several pumps, all of these must be controlled by the same process. If the network of devices is very large, this can lead to a performance bottleneck. Take for example all doors in an airport terminal building. These belong to different subsections and fulfil different purposes, but are subject to the same global constraint to keep various groups of people (e.g., with or without security clearance, staff/passengers, arrival/departure) separate from each other. If subsections shall be used flexibly (e.g., to accommodate various airlines' requirements or use sections for either domestic or international flights), all the doors involved must be controlled by the same service to prevent accidental missconfiguration due to concurrent allocation. Similar constraints also exist for non-physical properties, for example network parameters,that would render a device unreachable if set in an inconsistent manner.[1]

Accurate reasoning. In addition to requirements of state-changing operations as in the previous example, inconsistent reads can also be problematic, in particular for reasoning services. Those query the state of a number of system parameters, before calculating a conclusion that is then returned as the output. As an example, consider a service that queries the measurements of flow meters and valve settings in a network of pipes in order to detect if there is a leakage. It is important that while the measurements are taken, no other process can interfere and switch – for example – a valve, as this could potentially lead to a wrong conclusion. To prevent this, however, it is necessary that the all involved processing steps form a transaction together, which is unfeasible in reality. An easier alternative is to track causality between service inputs and outputs, so that services realise if they have read data that has been influenced by different versions of a sensor reading. This can be accomplished for example by vector clocks [29, 30] and does not require transactions.

Given that transaction support is evidently difficult to implement on top of an existing datastore and comes with additional communication overhead between client and database servers, the database itself should provide support for it.

R-16 The database provides support for ACID-compliant transactions, makes it available through the API and allows R-1, R-2, R-3 and R-4 to be executed in a transactional context.

3.1.7 CAP Trade-Off

In 2000 Eric Brewer presented the CAP conjecture [31] which claims that it is impossible to combine three properties, namely consistency, availability and partition tolerance in any stateful distributed system without perfect failure detection. The claim has been formalised and proven subsequently [32]. Consistency in this context suggests

[1]Example given in [15]

linearisability of operations, meaning that all client processes of the distributed system observe the same sequence of state changes, and upon request, are always served with the latest available state. Availability means that any request, read and update operations, sent to any non-failed node of the systems will be served with a response after arbitrary time. Partitions refer to network failures that make communication between groups of nodes impossible; a partition-tolerant database would allow to execute operations in each of the partitions. In the original talk, examples for each combination of two out of theses three properties were mentioned. The practical relevance can easily be overestimated, because it does not make predictions about more interesting properties, like latency, transactions and tolerance of node failures [33].

Linearisable consistency is a property that the original implementation of DS2OS provides, and that many applications rely upon. Furthermore it is a requirement for implementing distributed mutual exclusion and thus pessimistic concurrency control techniques which provide transaction support. It's therefore definitely worth to keep.

Regarding the remaining two properties, they are of minor relevance. In terms of availability it is mainly important to keep the latency low and to tolerate node failures. Partition tolerance would be worth considering in two cases. First, if we would look at an IoT system that is distributed over multiple sites, or even globally, though operates on a common database. DS2OS has the capability of addressing remote deployments, but this feature is not transparent to the application layer and thus does not require strict availability guarantees. The second interesting case would be a middleware that remains fully functional when parts of the local system have been split apart from the rest. This might be relevant in practical scenarios (sabotage, fires etc.), but guarantees of that sort would also require deeper changes of DS2OS architecture, so that for example application layer services are replicated and can be scheduled automatically to ensure continuous operation. Therefore it is safe to assume that partition-tolerance guarantee is not required. The interesting property is again how many database node failure can be tolerated.

R-17 The database has to maintain linearisable consistency (read always returns value of latest write operation) across replicas.

3.1.8 Resource Consumption

A further important point is the database's consumption of system resources. Examples for resource consumption are CPU, memory, storage utilisation and network bandwidth, and due to the distributed nature also the number of computing nodes. These influence the hardware requirements, costs of operation, and thus constrain the the area of economically viable deployments. Like the question of storage method, this issue rather entails a goal for optimisation than a hard requirement. Generally the goal is to achieve a lower resource consumption. The total consumption can, for each system resource, roughly be divided up into a fixed base footprint and a load-dependent variable component. In the end the hardware has to be dimensioned large enough so that it can handle the maximum expected load. However, it is important that the chosen product is also able to scale down and work resource-efficient in scenarios with consistently low load. DS2OS is not designed to be deployed in a multi-tenancy setting across organisations, so low-traffic deployments are common. This stands in contrast to the fact that conventionally, distributed datastores have been designed to address scalability issues that arise from needs global, integrated storage of mass data, for example in the context of software as a service products or large social networks. They are not necessarily designed to perform efficient in smaller settings, which makes the minimum resource requirements an important point to consider.

However, it is difficult to give a hard quantitative threshold the maximum resource consumption, since it is mainly a question about economics. Lower requirements on the hardware and less consumption of electricity leads to lower costs of operation and increases the margin to make the deployment of Smart Space Orchestration economically feasible. As performance of hardware increases and its cost decreases continuously, it is subject to change over time, so the current situation will soon be outdated. Therefore database candidates should be evaluated in hindsight of memory, disk and CPU utilisation and the results should be discussed, but it is impossible to give an absolute limit.

3.1.9 Triggers & Stored Procedures

Many databases allow to execute user-defined queries when an event occurs, for example when data has been inserted, updated or after a transaction commits. They typically only notify the code, but do not allow it to change the outcome of the action that lead to the notification. Triggers can be used to enforce application-level integrity constraints, so when data is changed, other records are affected automatically. In DS2OS a scenario where this would provide beneficial is archiving of the version history and to notify subscribers about changes. For either, a procedure must be invoked when a record is updated. For versioning it is important that this happens in a transactional context in order to maintain the consistency of the archive. This is not required to implement notifications, but the invoked procedures must be allowed to contain side-effects, so that they can inform subscribers.

R-18 It is possible to invoke custom code (stored procedure) when a write operation occurs. This is only required to either implement versioning or notifications inside the database.

R-19 Stored procedures can execute completely within the transactional context of the write operations that triggered them. This is important to maintain consistency when there are concurrent writes or when transactions with write operations are aborted.

R-20 Stored procedures are able to send messages over the network outside of the database system. This allows to implement notification functionality.

These three requirements are optional, as they also can be implemented in the code of the knowledge agents at the cost of additional roundtrips.

3.1.10 Related work

Corbellini et al. conducted a fairly recent survey on scalable, sharded NoSQL datastores, comparing products on a qualitative level by data model, persistence, replication, sharding, consistency, API, query methods and implementation language [34], which are dimensions that are highly relevant for this work. Apart from that they give an overview over theoretical concepts and implementation techniques. The authors note the high diversity in all of the investigated aspects, concluding that the choice of a database must be suited to its application, that in some cases hybrid approaches with a combination of different databases behind an integration layer are needed, and that benchmarking results vary a lot by the test characteristics and thus need to designed according to the application.

Unfortunately a study that compares the performance of transactional features of NewSQL database products could not be found.

Stonebraker et al. published a guideline of of 10 rules with justification to chose an appropriate distributed data storage architecture [35]. The rules cover aspects of the implementation of a distributed database as well as choice and usage in an application context. The following of them are found relevant for this work:

- Look for shared-nothing scalability: This refers to the horizontal scaling (sharding) capability in opposition to vertical scaling where some resources (e.g., disk or memory) are shared between database nodes. The databases that are considered for evaluation all apply this principle.
- Plan to carefully leverage main memory databases: The authors suggest to use a main memory database, as operations otherwise involve multiple disk seeks due to, locking, logging and the actual data access.
- High availability and automatic recovery are essential for SO scalability
- Don't try to build ACID consistency yourself: The authors state that it is very inefficient to write own protocols to maintain transaction isolation. This justifies listing transactional capabilities of the database system itself as a requirement.

- Open source gives you more control over your future: The rule states reasons in favour of open source software, which is a general requirement for compatibility with the DS2OS project.

3.2 Available distributed datastores

This section presents a selection of available distributed datastores, their features, characteristics and some of their internal architecture. Table 3.1 indicates which of the databases fulfils the described requirements. The selection of database candidates is based on the following properties:

Open license This includes free software (GPL, FGPL etc.), as well as business-friendly open source licensed (Apache, BSD etc.) and dual-licensed (service or extra features commercially available from developer) software products. This constraint has to be made to be compatible to DS2OS and its philosophy of supporting an open maker-culture.

Documentation Documentation is needed to evaluate if the product fullfils the required features quantitatively. In the optimal case it also includes descriptions on the internal architecture, extendability and scientific literature with comparisons, verification, and performance studies.

Aims & Goals The database should make distribution a fundamental choice of its design, meaning both replication to improve fault-tolerance and latency, and sharding/partitioning with automatic placement and re-balancing for scalability. The latter should happen automatically and transparently and not only be an add-on feature as it is the case with many older RDBMS. This excludes local in-memory databases like Redis or RocksDB, but also databases that use replication merely to achieve higher availability, without sharding. This is in line with the research topic.

Data model Graph-oriented data organisation has not been identified as having any particular advantage over the other models for our use case. Thus products from this category do not need to be considered.

Nowadays, the number of available of distributed datastores is vast; and thus it is impossible to list and compare all of them. The selection is supposed to be both representative of the available technologies, and to include all candidates that seem to be especially promising for this particular task by their marketing.

Many of the available distributed databases were modelled after proprietary prototypes developed by internet companies which faced scalability requirements that could not be met by any existing product. Amazon came up with Dynamo, a key-value store with high availability and partition tolerance, and sharding by consistent hashing. This

	Operations						Interface				Storage		Distribution			Consistency		Triggers		
	R-1 Insert	R-2 Read	R-3 Update	R-4 Querying on key prefix	R-5 Controlable locality	R-6 Bulk reads	R-7 API in Java	R-8 JDBC connector	R-9 Authentication	R-10 Encryption	R-11 Durable writes	R-12 In-memory data access	R-13 Sharding	R-14 Automatic rebalancing	R-15 Replication	R-16 ACID transactions	R-17 Linearizable consistency	R-18 Triggers	R-19 Atomic stored procedures	R-20 Procedures with side effects
Zookeeper	X	X	X	X	X	X	X	-	X	X	X	X	-	-	X	-	X	X	-	X
CouchDB	X	X	X	-	-	-	X	-	X	X	X	-	X	-	X	-	-	?	-	-
Cassandra	X	X	X	X	X	X	X	X	X	X	X	-	X	X	X	-	X[a]	X	-	X
HBase	X	X	X	X	X	X	X	-	X	X	X	-	X	X	X	-	-	X	-	X
VoltDB	X	X	X	X	X	X	-	X	X	X	-	X	X	X	X	X[b]	X	X	X	-
CockroachDB	X	X	X	X	X	X	-	X	X	X	X	-	X	X	X	X	X	-	-	-
Infinspan	X	X	X	?	X	X[c]	X	X	X	X	X	X	X	X	X	X	?	X	?	X
Riak	X	X	X	-	-	X[d]	X	-	X	X	X	X	X	X	X	-	X[a]	X	X	X
Scalaris	X	X	X	-	X	X[nt]	X	-	-	-	-	X	X	X	X	X	X	-	-	-
Voldemort	X	X	X	x	X	X[c]	X	-	-	-	X	X	X	X	X	-	-	-	-	-
Hazelcast	X	X	X	-	-	X[c]	X	-	X	X	X	X	X	X	X	X	X	?	?	?
Ignite	X	X	X	-	-	X[c]	X	X	-	X	X	X	X[e]	X	X[e]	X	X	X	X	X

[a] It can be specified to read/update records from/to a quorum of nodes.
[b] Only availabile for stored procedures, without side effects.
[c] Only by list of keys, no support for range queries.
[d] Via secondary indices.
[nt] No support for transactions.
[e] Either sharding or replication, no comination of both.
[?] Project documentation, user guide, publications do not give a clear answer.

Table 3.1: Requirement Matrix

design inspired Riak, Voldemort and Cassandra. Google developed the Google File System and Bigtable, which serve as the prototypes for the Hadoop File System (HDFS) and HBase. Google also developed Spanner which leverages timestamps as a way to order commits in a transactional system. This principle is adopted by CockroachDB.

3.2.1 Zookeeper

Zookeeper is a coordination service for distributed applications [36]. Like DS2OS it provides the interface of a simple hierarchical datastore with read and write operations, but its primary purpose is not to store large quantities of data. It is rather meant to provide strong guarantees in terms of linearisability, availability and fault tolerance, which can then be used to build other coordination primitives upon, for instance group membership, locking, barriers, or configuration management. This is reflected in its characteristics: All the data is fully replicated to all of the clients and kept in memory, while all write operations are forwarded to one leader that decides on a total order. The data is not partitioned among nodes as it is not designed for high scalability, but rather fault tolerance and consistency. In terms of fault-tolerance, Zookeeper is capable of handling $\left\lfloor \frac{n}{2} \right\rfloor$ concurrent node failures with n being the total number of nodes. Quite obviously Zookeeper is not a viable candidate for building the VSL state layer upon, not only because of limited scalability, but also because there is no native support for transactions. However, it can be used to implement concurrency control mechanisms which add transaction support for other non-transactional stateful distributed systems or to distribute metadata in a reliable way. Clients can register "watches" on records that act like one-time subscriptions, informing the client that data has changed.

3.2.2 CouchDB

CouchDB is a document-oriented database which focuses on web technologies to simplify integration in modern web-based applications. In line with this, data is represented as JSON documents, accessed through a REST interface and queries and views are expressed as JavaScript code. CouchDB supports both replication and automatic partitioning of data; but it is not possible to reorder or increase the number of shards during runtime [37].

In the CAP-model, CouchDB achieves availability and partition tolerance while relaxing consistency. In other words, it allows one to continue reading and writing data even in the case of a "split-brain" scenario – when parts of the network have been disconnected [38]. Between these autonomous segments, conflicting writes can occur. Although CouchDB will not drop any data, resolving the conflicts is up to the user. While partition tolerance is an interesting property for systems that need resilience in the case of major damage or are globally distributed, insufficient consistency guarantees and no support for transactions exclude it as a candidate.

3.2.3 Cassandra

Cassandra [39] is a wide-column store that employs consistent hashing to distribute keys among the nodes and to route requests. It offers access through a SQL-like interface and supports secondary indices on single columns, although without fulltext indicing or joins. By default there is no guarranteed consistency. It is however possible to explicitly specify quorums of nodes for select and update statements, in order to avoid conflicts. There is thus also no support for ACID-compliant transactions, but updates are guaranteed to execute atomically on row level. It is also possible to make an update or insert operation dependent on a condition on the affected row, which is checked atomically (compare-and-swap). Additionally, batches of operations can be configured to execute in isolation. While it is possible to get transaction support by implementing a concurrency control protocol on top of these, it is discouraged, since there is is no performance gain as compared to dedicated ACID-compliant databases and the implementation effort is considerable [35].

On the upside, [40] shows that Cassandra achieves constantly low read and write latency even in high load scenarios. The scalability is almost linear. However, since transaction support is missing, it is not suitable for our purpose.

3.2.4 HBase

HBase [41] is another candidate from the wide-column family of databases. It leverages the Hadoop Distributed Filesystem (HDFS) for distributing data. In contrast to Cassandra, its interface is relatively bare: There is no SQL abstraction, but only an API in native Java, also available as a web service with various serialisation formats. All data is generally represented as byte arrays and there is no support for secondary indices either. Columns can be grouped into column families which are then physically stored together. This is relevant for performance optimisations. Reads and Updates are all routed through a unique server per shard (region server) which orders requests and thus ensures consistency on shard level. Batches of operations are not guaranteed to execute in isolation or even in the order that they were specified. There is however, like in Cassandra, an atomic check-and-mutate operation that allows one to tie an update or insert to a precondition on the affected row. It is still impossible to build transactions with ACID semantics on top of it, since there is no way of specifying atomic operations that affect more than one row (like atomic batches in Cassandra).

Similar to Cassandra, it was demonstrated that HBase scales well and has low latency for read and write operations [40]. But the missing transaction support likewise disqualifies it for our purpose.

3.2.5 VoltDB

VoltDB falls into the “NewSQL” category. Those are databases that aim to provide the well-known functionality and transaction capabilities of legacy relational databases, in combination with a distributed design that achieves higher scalability and availability through partitioning and replication [42]. On top of that, VoltDB places all its data in memory, and entertains the idea to move code that requires transactional access closer to the database, in the form of stored procedures. These are executed transactionally, and can be repeated if they fail. This is unfortunately the only way to access VoltDB data transactionally. There is no equivalent to the well-known sequential communication between the database and the client within a marked transaction context, which imposes the limitation that transactions cannot have side-effects. Although this is generally desirable, it constrains the functionality, especially in the context of a cyber-physical system, and is more difficult to integrate since transactional service code would have to be compiled to a VoltDB stored procedure, marshalled and shipped to the database for execution.

3.2.6 CockroachDB

CockroachDB [43] is like VoltDB a member of the “NewSQL” movement, inspired by the design of Google’s Spanner [44], thus mimicking the functionality of classic transactional RDBMS, with additional scalability and high availability through global distribution. Unlike VoltDB, it stores its data on disk, employing RocksDB as a backend for local storage. It also retains the more generic form of transaction support which allows user-defined client-side code to communicate with the database server, rather than scheduling a user-defined function to execute on the server. This functionality is accessible through a SQL interface.

Internally CockroachDB uses Raft to achieve row-level consistency among the replicas and MVCC to ensure transaction isolation. Total ordering of version requires an accurate time source. To obtain it, CockroachDB employs an algorithm that combines a time source from synchronisation algorithms of physical clocks with that from a logical Lamport clock [45]. Spanner in contrast uses technologically advanced hardware, namely atomic clocks and GPS receivers, to achieve this.

3.2.7 Riak

Riak [46] is a key-value datastore. Inspired by Amazon Dynamo [47], it uses consistent hashing to distribute data on a ring structure and to route requests. It allows to plug in various storage back-ends that place data either on disk or in memory.

Riak has a sort of secondary indexing capability, but since it is a key-value store, it cannot create indices automatically. Instead the programmer must manually attach index labels to each persisted record. On top of generic payload as values, Riak also supports maps, sets and counters. These have semantics of conflict-free replicated datatypes (CRDTs) [48], so will eventually converge to a value without losing data. Riak does not provide strong consistency, but tracks versions of records with vector clocks and resolves conflicts on detection either with last-writer-wins semantics or return multiple values, thus leaving conflict resolution up to the user. This behaviour is comparable to CouchDB or Amazon Dynamo. If there are higher requirements on consistency, Riak allows the user to globally configure strong consistency, for all operations, while sacrifying availability guarantees. In this mode, Riak additionally provides a conditional modification operation, like to Cassandra and HBase.

3.2.8 Scalaris

In contrast to most of the other datastores presented in this section, Scalaris is a research project, not a commercial product. It aims to provide a scalable distributed key-value store with strong transactional semantics [49]. It is entirely based on peer-to-peer messaging, employing the gossiping protocol T-Man to build a ring-structured overlay [50], an algorithm derived from Chord to provide routing [51], and Paxos for optimistic concurrency control [52]. Unfortunately there is no support for versioning, encryption, authentication or triggers which limits its use for real-world applications a little. There is also no option to submit queries for more than one record at once in a transactional way, which would make querying VSL node trees inefficient.

3.2.9 Voldemort

Voldemort is a key-value store, modelled after Amazon Dynamo [47]. Thus, it also shares some characteristics with Riak. It shards data according to a configurable hash function. Data locality can be preserved if a hash function is provided that maps a key to an integer which then determines the storage location. For VSL addresses, this could be a hash of the substring determining the knowledge agent or the knowledge agent and the service identifier, to preserve locality among those. Like Riak, Voldemort versions all its data with vector clocks. This feature is not hidden from the database user and can be used to resolve conflicts manually. These can occur in the event of partitions for which there are no further mechanisms to prevent, making Voldemort an AP store in the CAP model, like CouchDB and Riak in their standard configuration. Unlike for these two, there is also no option to change this behaviour. Obviously, there is no support for transactions or conditional update operations.

3.2.10 Infinispan

Infinispan [53] and similar products (Hazelcast, Ignite, and Geode) are marketed as "distributed caches" or "data grids". Data grids aim at improving the performance of computation workload [54]. For that reason they incorporate functionality to transfer and distribute computation workload to the servers where the data resides, to store partial results, and to make these results available to other processes. Irrespective of that, managing distributed data is the core concept, making Infinispan and consorts likewise distributed databases with key-value semantics.

Infinispan distributes data like other key-value stores on a DHT with a ring topology, using consistent hashing [55]. Both key and value can be arbitrary Java objects, as long as they can be marshalled into binary data. By default all data is stored in-memory, but optional persistence to disk can be configured. Infinispan can be used as a cache with an optional automatic retention policy and with configurable levels of replication and sharding. It furthermore provides the possibility to index and query data by incorporating Apache Lucene for full-text indexing. Infinispan incorporates support for distributed transactions, employing the 2-phase commit protocol. The user can chose between an optimistic and a pessimistic mode.

Infinispan can either run as a Java library within the runtime environment of the application that accesses the data or independently as a standalone server.

3.2.11 Hazelcast

Hazelcast is conceptually similar to Infinispan, Ignite, and Geode. Like those, it functions as a distributed key-value store for Java objects. Additionally, it can also store data structures that are different from the key-value map, such as lists, sets, queues, locks and counters and it can also be used as an event broker. This makes it a more versatile product. Hazelcast locates sharded data by using a global, replicated partitioning table.

In a recent study, it was shown that Infinispan outperforms Hazelcast for simple operations with key-value semantics [56].

3.2.12 Ignite

Apache Ignite is similar to Hazelcast. It stores Java or .NET objects in memory, with an option to persistence to disk and provides additional datatypes like queues and sets. Ignite provides an optional SQL interface, though this is currently without the support for transactions, which are only provided for the native key-value API. These are implemented using the 2-phase commit protocol. Developers of Ingite claim that it remains functional when the transaction manager or a node that stores affected data crashes [57].

3.2.13 Geode

Apache Geode is another instance of key-value stores that closely resemble Infinispan. Geode supports a subset of SQL for querying data. Notably, support for joins is unavailable for sharded datasets. It allows both replicated and sharded distribution of data, but has only rudimentary support for a combination of both (additional master-slave replication for a sharded dataset).

3.2.14 Conclusion

For further analysis, we selected CockroachDB (version 1.0) and Infinispan (version 9.1). Both cover the requirements and allow to compare different approaches in regard to modelling, integration, and performance relating to their architectural differences. They are also representative for their product class: CockroachDB is a “NewSQL” database that aims to provide a more scalable and fault-tolerant version of a relational database and Ininispan is an “In-memory data grid” that aims towards a tight integration into the application and bears similarity to Hazelcast, Geode, and Ignite.

It should be noted at this point, that distributed databases are evolving rapidly and the development of many products is still ongoing. During the time that this work was conducted, two new versions (1.1 and 2.0) of CockroachDB were released, which contain new features and are claimed to bring significant performance improvements. Another promising candidate also inspired by Spanner, TiDB [58], was only released for production in October 2017 (version 1.0).

Chapter 4

Implementation

After surveying candidates and concepts among databases, this chapter first explores how the VSL data model of DS2OS can be integrated into the databases, how the semantics of subscriptions, version history, and virtual nodes can be retained, and how the VSL interface has to be changed to support transactions. The second part then provides details about the implementation of CockroachDB and Infinispan backends to achieve this goal.

4.1 Architectural Integration

There are several challenges involved in integrating a distributed datastore into DS2OS that mostly arise from the fact that it allows to interact with KOR data in various different ways. Data can either be queried explicitly or the client registers a subscription and is informed whenever data is available. Then, there is a distinction between virtual and persistent (non-virtual) data, and it is not statically known in which of these two categories a node identified by an address falls, since it can be registered as either of them.

In contrast to these versatile ways of interaction, databases usually – and that includes most of the reviewed instances – only provide support for a querying mode of access; that is reading and writing values. Database triggers is a feature that cannot be considered a standard among NoSQL databases and if it is supported, its behaviour in relation to transactions is often not explicitly documented, not to mention controllable.

4.1.1 Transaction Interface

As outlined before, the user-side API of DS2OS does not have an idiomatic transaction interface that hides details of concurrency control procedures away from the user. This is a feature that is often provided by databases, which hide the details of the concurrency control. Typically, a database user first starts a transaction, executes a sequence of operations and then either commits or rolls back the transaction. The database confirms if the operations have been carried out sucessfully or informs if the transaction has been aborted. To schedule conflicting transactions, the database may either delay an operation or abort a transaction, which can happen immediately or when it is committed. Some conflicts cannot be resolved through delaying because they mutually depend on their progress. These cases have to be detected and resolved by the database scheduler.

To bring this behaviour to DS2OS the Connector interface has to be changed. Two possible alternatives on how such an interface could look like exist:

```
org.ds2os.vsl.core.VslConnector conn;

try {
   Transaction trans = conn.createTransaction();
   VslNode lamp1 = trans.get("/agent2/light1/isOn");
   trans.set("/agent2/light2/isOn", lamp1);
   trans.commit();
   // or: trans.rollback();
} catch (TransactionAbortedException e) {
   // handle transaction abort
}
```

Figure 4.1: Operations expicitly associated with a transaction object

```
org.ds2os.vsl.core.VslConnector conn;

try {
   conn.beginTransaction();
   VslNode lamp1 = conn.get("/agent2/light1/isOn");
   conn.set("/agent2/light2/isOn", lamp1);
   conn.commitTransaction();
   // or: conn.rollbackTransaction();
} catch (TransactionAbortedException e) {
   // handle transaction abort
}
```

Figure 4.2: One transaction environment per connector

In 4.1 the user creates a transaction object first. They then use that handle to add operations and commit or abort the transaction. In 4.2 there is only one possible transaction at a time and the user uses the `VslConnector` object to issue operations. The former is more generic and has the advantage that multiple transactions can co-exist at the same time, while also allowing non-transactional operations when a transaction is active. The latter is closer to the original interface, but there is only one transaction at a time. It should be noted that 4.1 can cause implementation issues, as not all databases themselves have interfaces that allow multiple concurrent transactions within a session. An example for this are all SQL interfaces; they resemble alternative 4.2.

The interface has to inform the user on the success of a transaction. This can happen eagerly, when a operation leads to a conflict and thus an abort of the transaction, or lazily, when the transaction is committed. In the former case an exception has to be thrown. The latter could also allow to return a boolean value for the `commitTransaction()` call that would indicate if the transaction was successfull, as in the listing:

```
org.ds2os.vsl.core.VslConnector conn;

conn.beginTransaction();
VslNode lamp1 = conn.get("/agent2/light1/isOn");
conn.set("/agent2/light2/isOn", lamp1);
if(conn.commitTransaction()) {
   // transaction was successful
else {
   // handle transaction abort
}
```

Figure 4.3: One transaction environment per connector

Preferrably, the new interface will be backwards compatible. First this implies that the get and set methods of `VslConnector` should not throw any new exceptions. This is given when they are instead explicitly called for a transaction handle as in 4.1, or if the success is only checked on commit as in 4.3. Secondly, it is desirable if the existing lock, commit and abort methods still work and integrate with the transaction interface. To accomplish that, a possible solution is to initialise an implicit transaction whenever a lock method is called, and commit it when an acquired lock is committed; otherwise abort. Algorithm 1 describes the procedure in detail.

4.1.2 Version History

DS2OS records all versions of a VSL tuple and allows access to it with incremental version numbers, e.g., `/agent/service/data/15`. This version number has to be updated whenever a field is updated, and the old value has to be stored. If, when and in what order a new value is decided uppon, is in the end up to the database and its concurrency control mechanism. The database might either already have functionality to keep records of the version history and make it accessible, or that behaviour has to be implemented externally.

One way of implementing this behaviour is to store the current version number along with the value of a VSL address, in the value-part of a key/value store, or a column in a

Algorithm 1 Locking shim for transactional connector

```
conn                                              ▷ Transactional VSL connector interface
$l \leftarrow \emptyset$                          ▷ Mapping of lock address to transaction ID
$h \leftarrow \emptyset$                          ▷ Mapping of lock address to lock handler

procedure GET($addr$)
    if $\exists(address, tid) \in l$, $address$ prefix of $addr$ then
        try $conn.get(addr, tid)$
        on TransactionException $handleAbort(address)$
    else
        $conn.get(addr)$
    end if
end procedure

procedure SET($addr, data$)
    if $\exists(address, tid) \in l$, $address$ prefix of $addr$ then
        try $conn.set(addr, data, tid)$
        on TransactionException $handleAbort(address)$
    else
        $conn.set(addr, data)$
    end if
end procedure

procedure LOCKSUBTREE($addr, handler$)
    $tid \leftarrow conn.beginTransaction()$
    $l \leftarrow l \cup \{(addr, tid)\}$
    $h \leftarrow h \cup \{(addr, handler)\}$
    $handler.lockAcquired(addr)$
end procedure

procedure COMMITSUBTREE($addr$)
    for $\forall(addr, tid) \in l$ do
        $conn.commitTransaction(tid)$
    end for
end procedure

procedure ROLLBACKSUBTREE($addr$)
    for $\forall(addr, tid) \in l$ do
        $conn.rollbackTransaction(tid)$
    end for
end procedure

procedure HANDLEABORT($address$)
    for $\forall(addr, handler) \in h$ do
        $handler.lockExpired(address)$
    end for
    $l \leftarrow \{(addr, tid) \in l, addr \neq address\}$
    $h \leftarrow \{(addr, handler) \in l, addr \neq address\}$
end procedure
```

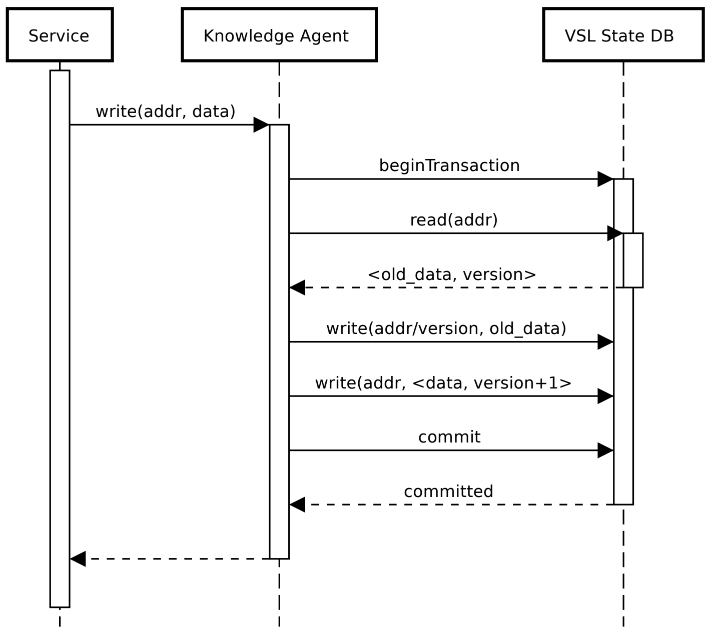


Figure 4.4: Archiving the version history within the database that keeps the VSL data

database table. When a VSL tuple is updated, the version number has to be read, and the old record has to be copied and archived. Then the version can be incremented and the new value written to the database record. To ensure consistency, this whole procedure has to happen within a transaction (see Figure 4.4). This is also the approach that is taken in the current implementation. There it is not much of a performance problem as all of that happens locally. In a distributed setting however, this will likely involve data partitioned and replicated to multiple servers between which the transaction has to be coordinated. This can lead to a bad performance even for simple write operations that are committed outside of a transaction from the user API. The problem can parially be resolved when it is ensured that the current version and version history of a record are stored in the same partition. Furthermore database triggers can be used to move the coordination of the procedure to the database layer and avoid unnecessary rountrips between the knowledge agent responsible database server.

Alternatively, a record containing the number of the latest version can be kept as a database record. This can then be used to reference the latest value indirectly. Unfortunately it involves more than one database operation for both VSL reads and writes. Thus they also have to be wrapped in a transaction.

Another alternative approach is to seperate the version history and the current state of data, and not involve the VSL database in tracking the versions. Instead the numbering and archiving is handled by a seperate append-only log system. This can either be a time series database or an event stream system with storage capabilities like Apache Kafka. The important requirements are just that it can agree on a total order and number the

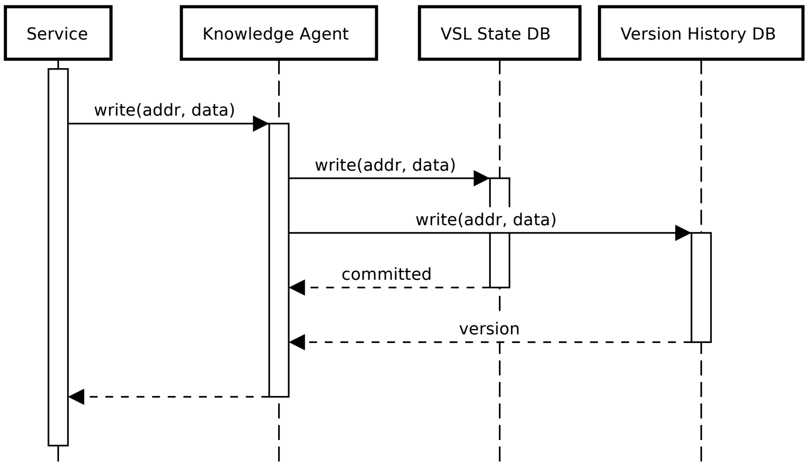


Figure 4.5: Archiving state updates outside of the VSL state database in parallel

versions and that it allows random access to these version numbers. Apart from that, there are no more consistency requirements as data is never changed after appending. It can be replicated to achieve performance benefits arbitrarily many times without worrying for coherency.

When the VSL state is altered, one of the components that is involved in the update has to forward the new state to the version archive. This can be done by the database that keeps the VSL state, if it supports this feature, or by the knowledge agent whose service changed a VSL tuple. In the latter case, there are several alternatives as to when this update happens: When a the client issues a write (Figure 4.5), when the client called to commit the transaction that contains it (Figure 4.6), or when the commit is confirmed by the database (Figure 4.7). In the former two cases it's guaranteed that the new values end up in the version history database, but it may also contain some uncomitted versions in the end. In the other there is a small risk that the knowledge agent crashes and that the values are ommitted from the version history. The second option, waiting until the client commits the transaction, allows to enforce the version order in the history by delaying committing the VSL transaction until the version history database has confirmed the append. In all cases, reading historic versioned data is unproblematic for consistency guarantees, because it can never change.

Lastly, many databases keep old versions of their data by themselves. This has mostly to do with the multi-version concurrency control (MVCC) mechanism that many of these deploy, and to avoid seek time in disk storage. Despite the different primary intention, some allow the user to access old versions as a feature as well, though not necessarily with the same semantics as DS2OS versions.

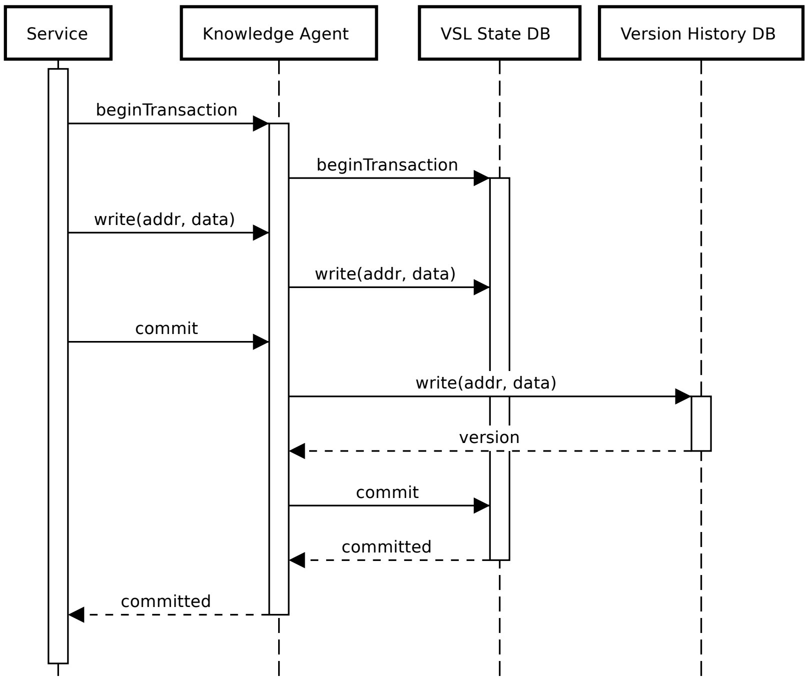


Figure 4.6: Archiving state updates before transaction commit

4.1.3 Subscription and Notifications

As mentioned before, services can subscribe to VSL addresses and are informed when their values change. The subscriptions are so far stored at the knowledge agent that is responsible for the service that owns the data. Since this knowledge agent processes all writes to that data, it is guaranteed that all interested parties are informed correctly as long as no messages are lost or the knowledge agent crashes. This mechanism is not possible anymore if all knowledge agents access the distributed datastore directly. Instead, any knowledge agent via which a write is executed must make sure that all subscribers are served. That also means that there must be one single location where the list of subscribers are kept. There are three straightforward ways to implement this:

1. Store the list of subscribers in the distributed database along with the value. The writing KA must carry out the actual notifications by itself.

2. Store the subscriptions at the knowledge agent to which the VSL address is associated, as it is practiced in the original version and let this knowledge agent send out the notifications. Any other KA that carries out a write needs to inform this KA.

3. Use a separate message broker, such as Apache Kafka for handling the subscriptions.

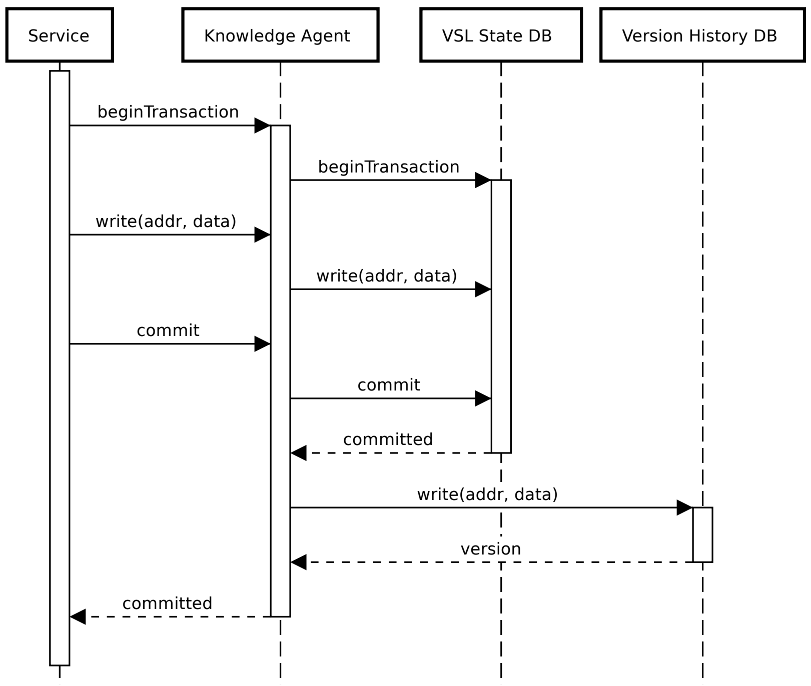


Figure 4.7: Archiving state updates after transaction commit

The semantics of the notifications are also a point to consider. Unless there are agent failures, subscribers are informed exactly once per write access and at most once when the owning KA fails. Currently there is no guarantee on the delivery time and order of notifications and that the value read as a cause of a notification is the value whose write operation triggered the notification being sent out. This means that additional notifications for an update would not cause any harm, which is insofar relevant as it simplifies notification handling within transactions. Since writes that are issued within a transaction block can be rolled back, sending out notifications would have to be delayed until they are committed.

4.2 Back-end Implementation

In the implementation phase, we took the existing DS2OS codebase and applied the necessary changes. The service-side and internal APIs were extended by transactional capabilities. For this, the more generic version with explicit transaction identifiers was chosen. This is because on the logical level of interaction between knowledge agents and service connectors, there is no concept of a session, so requests from service connectors are stateless. The transaction identifiers are valid within one connection between a client and a knowledge agent and can be obtained by starting a transaction with `String beginTransaction()`. Then, transactional access can be performed through the `VslNode` `get(address, parameters, transactionID)` and `void set(address, knowledge, transactionID)` methods with the respective transaction identifier. For the original DS2OS implementation with a federated backend, a connector shim mimics the behaviour of the transaction API for compatibility (see listing 2). It implements a simple 2-phase locking algorithm on top of the old locking functions, although without guaranteeing consistency in the case of a KA failure or deadlock detection.

Two more implementations of this API connect either as Infinispan and CockroachDB as database backends. Since Infinispan is designed to execute embedded in the Java Virtual Machine together with the application that uses it, the connector accesses it directly through the provided API. For CockroachDB it uses the compatible Postgres JDBC driver to communicate with the database server over a SSL-secured TCP protocol.

The subscription and virtual node querying functionality were changed as little as possible in order to preserve their original semantics: Reads and writes to virtual nodes are forwarded to the knowledge agent where the node is registered after it has been detected that the node is virtual by a set flag in the database. The routing layer was changed to check if a node is virtual or persistent and to direct requests for persistent data directly at the database instead of a remote knowledge agent. The downside of this approach is that it requires an additional database access. To mitigate this, both modes of access are executed in parallel for get and set operations: To check for a virtual node, the request is sent through the responsible remote knowledge agent; to access persistent data, the request is forwarded to the distributed database. When the data at the destination does not match the access method, so when the database record indicates, that the queried node is virtual or the Virtual Node Manager does not find a corresponding Virtual Node Handler, they return a failure result. Whichever of these two operations fist returns a valid result is chosen. This helps to speed up processing of virtual nodes, so that accessing virtual nodes still has the same latency as the old implementation.

Notifications are implemented as suggested by alternative 2 in section 4.1.3: The subscriptions are stored at the knowledge agent with which a VSL address is associated, like in the original version. The disadvantage of this approach is a higher delay in notification because of the additional round-trip. Whenever a persistent node's data is updated, the knowledge agent where the access happens informs the knowledge agent where the accessed service is registered which can then send out the notifications. This happens irrespective of whether the write access happens in a transaction or not and may thus result in additional notifications in the case of aborted or rolled-back transactions or too early notifications for which the notified party cannot yet see the results.

Algorithm 2 2-phase locking shim

```
$conn$                                                        ▷ VSL connector interface
$l \leftarrow \emptyset$                                      ▷ Locks per transaction
$c \leftarrow \emptyset$                                      ▷ Completed transactions

procedure GET($addr, tid$)
    if $tid \in c$ then
        return
    else if $\exists\langle tid, lock\rangle \in l.\exists x.\ (lock \circ x) = addr$ then   ▷ Already holds lock
        return $conn.get(addr)$
    else
        $conn.lockSubtree\ (addr, \lambda\_.notify\ (var))$
        $wait(var)$                                           ▷ Wait for lock
        $l \leftarrow l \cup \langle tid, addr\rangle$
        return GET($addr, tid$)
    end if
end procedure

procedure SET($addr, data, tid$)
    if $tid \in c$ then
        return
    else if $\exists\langle tid, lock\rangle \in l.\exists x.\ (lock \circ x) = addr$ then   ▷ Already holds lock
        return $conn.set(addr, data)$
    else
        $conn.lockSubtree\ (addr, \lambda\_.notify\ (var))$
        $wait(var)$                                           ▷ Wait for lock
        $l \leftarrow l \cup \langle tid, addr\rangle$
        return SET($addr, data, tid$)
    end if
end procedure

procedure BEGIN
    return $random\ tid$
end procedure

procedure COMMIT($tid$)
    $c \leftarrow c \cup tid$
    for $\forall\langle tid, lock\rangle \in l$ do
        conn.commitSubtree(lock)
        $l \leftarrow l \backslash \langle tid, lock\rangle$
    end for
end procedure

procedure ROLLBACK($tid$)
    $c \leftarrow c \cup tid$
    for $\forall\langle tid, lock\rangle \in l$ do
        conn.rollbackSubtree(lock)
        $l \leftarrow l \backslash \langle tid, lock\rangle$
    end for
end procedure
```

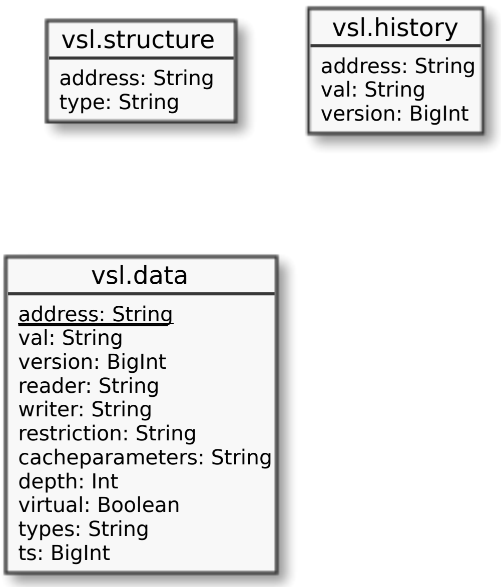


Figure 4.8: Database tables of the CockroachDB backend

4.2.1 CockroachDB

For the implementation with the CockroachDB backend, data is spread across three database tables: structure, data, and history (see Figure 4.8). All current data along with structural information is stored in the data table. The records in that table have a field depth that indicates the length of the address, or in other words slashes in the address string, which helps to query subtrees of a specific length. VSL data is queried trough an equality expression on the address. When a node tree of a depth other than zero is queried, this has to be a regular expression, which has bad performance in CockroachDB because a fulltext index feature has not yet been released (it has been announced for a future version). The structure table contains pairs of an address and a type and forms an index for the type search functionality. The history table contains versions of the records present in the data table, marked with a version number, but without the structural information. For every write, the record in the respective data table is inserted into the history table.

The interface code uses a connection pool to enable multiple threads to access the database concurrently. The connections itself are encrypted with SSL and mutual authentication through a client key. For transaction processing, pairs of the transaction identifier and a corresponding active JDBC connection object are kept in a map and used whenever a transactional get or set operation is requested.

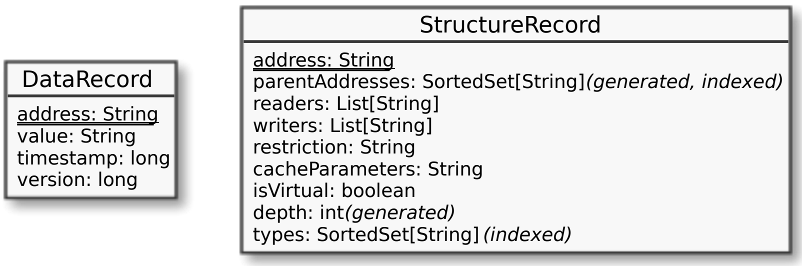


Figure 4.9: Classes used as value types of the Infinispan backend

4.2.2 Infinispan

The Infinispan backend uses two collections, which are maps of a VSL address to Java objects of a specific types, to store the VSL data. One is named vsl-structure, the other data. The values are Java objects as displayed in Figure 4.9. The structure table is marked as non-transactional and asynchronously replicated, so it is shared with all knowledge agents to decrease access latency. It contains all the metadata that is associated with a VSL node. Furthermore the node address is broken up into its parent addresses (prefixes) which are stored in a list structure within the structure object. This structure is indexed by the integrated Lucene search engine to query for subtrees of the VSL structure faster. Also, the type field is marked to be indexed for the type search. The other table, vsl-data contains the values and timestamp of the current version. This table is marked as transactional and synchronously distributed. That means for every write the database makes sure that the operation is transmitted to at least three replicas to guarantee durability. The Infinispan backend does not use separate collections for current and historical data, instead the old data is appended to the data collection with the version as suffix to the key.

The connector takes the same approach to transaction handling as the CockroachDB one, with a map of transaction identifiers and database handles. Unfortunately, the Infinispan API has the idiosyncrasy that transaction sessions are tied to the executing thread. This might simplify the development of applications in many cases, but in this one it is rather cumbersome since within the knowledge agent it is not defined which threads are used and reused to submit database requests. To work around this limitation, the approach was to create a new thread on each transaction start which wraps the database API handle and executes all transactional requests for the respective transaction identifier. The communication with the calling thread happens through variables and low-level synchronisation primitives (wait and notify), so that there should not be a significant decline in performance.

4.3 Summary

Both backend implementations have the same interface: They store the VSL structure and current payload data and archive its history. They are able to mark virtual nodes and have an optional transactional mode for get and set operations. Independent of the chosen backend, subscriptions, notifications on update, and requests to Virtual Nodes are always forwarded to the KA responsible for the affected VSL address.

Among the two databases themselves, there are a number of architectural differences: Infinispan runs within the Java Virtual Machine of the KA, CockroachDB as an independent server, that is accessed via a local TCP socket. Also, the storage location differs. Infinispan uses pure in-memory storage and will not attempt to restore any data if it crashes. In contrast, CockroachDB uses a RocksDB backend which stores data on disk and is able to reuse the dataset after a crash or restart. CockroachDB maintains consistency using a consensus protocol, while Infinispan only makes a best effort of updating records on the reachable nodes. These differences are important to consider when the backends' performance is compared.

Chapter 5

Evaluation

For benchmarking, two test setups were created. One consists of several small scenarios with a few services and aim to analyse specific properties. The other contains scenarios with a large number of randomly generated services and is meant for simulating high traffic scenarios. The tests and measurements were carried out at the ilab computing facility at the Chair for Network Architectures and Services at TUM. The hardware setup was a cluster with 6 nodes, each equipped with 4 core CPU (Intel Xeon E3-1265L v2) and 16 GiB memory. The nodes were connected through a switched 1000 MBit/s Ethernet network, with a simulated transmission delay (2ms + 1ms normal dist.) and packet loss (0.1%, 75% dependent on predecessor). This should represent a deployment within a company site where the network is shared with other applications.

For generating test traffic and taking measurements, a test harness was implemented in Scala. It executes an embedded knowledge agent with a configurable database connector, runs concurrent load tests according to a provided test model or specific test for a particular property (microtests), and takes measurements of performance metrics. The measurements are asynchronously written to local storage through a buffer in order to avoid putting additional load on the network or delaying test execution.

It is likely that performance testing will reveal a trade-off between throughput and latency on one side and consistency and availability on the other. The expectation is that CockroachDB will perform worse in terms of latency and throughput in comparison with Infinispan and the existing HSQLDB backend. CockroachDB offers the transaction semantics of a conventional relational database, but in a distributed environment and with higher availability and scalability, which requires more sophisticated scheduling algorithms, quorums and more messaging between nodes. Infinispan in contrast doesn't enforce quorums and uses a simpler 2-phase commit protocol for distributed transaction coordination. The expectation here is that its non-transactional operation latency will compete with the HSQLDB backend. The throughput might be slightly lower because of the overhead for replication.

5.1 Test Model

Generally, the clients of an IoT system can be classified into different groups. Some services are linked to a physical component and their state reflect their state in the real world while others perform purely logical tasks. And some produce data while others only consume it. The following categories classify services with regard to their data access patterns and role in information transformation:

Sensors These continuously produce measurements and write it to their own VSL address. The data is scalar, thus the node can be flat and the write access is does not need to be wrapped in a transaction.

Complex devices The restriction on single values and unidirectional data flow is too simple for many devices found in real building automation installations. Some devices incorporate both sensor and actuator functionality and possibly also their own logic (e.g., automatic sliding door). These devices generally access data confined to their own VSL node tree, but can access several parts of it at once, wrapped in a transaction.

Knowledge inference services These services aggregate or enrich information from sensors, complex devices, and other knowledge inference services. On each iteration, a knowledge inference service reads data from different VSL nodes, performs computations, and writes the result to its own VSL node. A real-world knowledge inference service would perform computations which can be as simple as a unit conversion but also a more complex reasoning or machine learning task. The service has to make sure that it reads recent data.

Coordination services Coordination nodes implement automatic control loops and provide coordination functionality between the input and output sides of an IoT network. In a typical iteration, a coordination process first reads data from various nodes, determines the control output, and then writes the results to other nodes. These nodes are outside of the service's own VSL address space. A stateful coordination service might also have its own private state stored in its VSL node, but this would typically only be accessed by itself. All operations carried out within a control iteration should be wrapped in a transaction in order to enforce policies imposed by the control logic.

Control services Control services manage and control hardware (sets of actuators and complex devices), and act themselves on control inputs. The difference to coordination services is that they do not collect data from other services themselves, but rather act upon changes of their own VSL node, representing control input. The execution of such services can be wrapped in a transaction, as control functions might impose consistency requirements. For load tests this class of service is not necessary, as its access behaviour is equivalent to a coordination service with one read access.

Actuators Actuators continuously consume data from their own VSL address. Representing devices, they would adjust their physical state to the data that was read from the VSL. The value stored there is scalar and the read accesses occur outside of transactions.

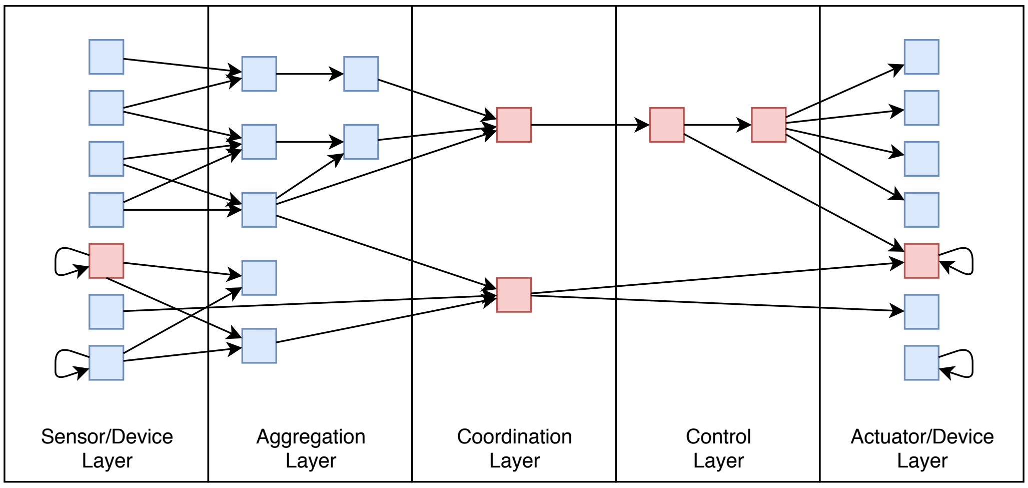


Figure 5.1: Example topology of information flow across an IoT network, boxes indicate services, red ones those performing transactions

These service classes typically form chains of information flowing from sensors and complex devices to actuators and again complex devices (e.g., Figure 5.1).

5.2 Measurable properties

5.2.1 Information dissemination latency

It is desirable that the smart space reacts to changes quickly. This property can be expressed as the difference between the time of writing a sensor reading and the time at which an actuator responds to it. Of course, an actuator action can be the causal result of readings from several different sensors and coordination actions. In these cases, the newest sensor reading is relevant as it is the event which finally triggered the chain of updates that lead to the actuator action, while the other may not directly have caused an intermediate knowledge inference or coordination service to emit a new value. Secondly, it is also relevant to see if any stale data is present; that is when an actuator read returns causally related sensor data for which a newer version is available. This is not avoidable unless a whole chain from sensor to actuator is put in one distinct transaction, which is only possible if there is only one intermediary service. For tracking the causal relationships of transmitted information, vector clocks can be used.

5.2.2 Transaction success rate

Devices, coordination and control services, in some cases possibly also knowledge inference services, can use transactions to apply consistent changes of VSL state. If those transactions are aborted by the middleware, they have to be attempted again, affecting the overall reactiveness of the smart space. Conflicts can only entirely be avoided if transactions are scheduled in a strict sequential order, without concurrency, or if conflicts are detected in advance. Both methods are infeasible for distributed databases, as the first affects scalability and fault tolerance and the second entails a semantic restriction. Thus, conflicts between transactions can possibly occur, but it is up to the concurrency control algorithms of the database to handle them effectively and decide which transactions will be cancelled, delayed, or transparently resubmitted. From the user perspective, it affects the rate at which transactions are accepted and the time it takes to complete transactions.

It should be mentioned though, that workloads which are likely to cause deadlocks, and thus the inevitable abort of a transaction, are unlikely to be encountered in realistic workloads. It would require that several sensors access the same data in a non-uniform manner, e.g., by randomising over the sequence of write access to a set of output addresses. This cannot be ruled out, but it is also not the norm. Therefore, the tests do not contain workload that deliberately causes deadlocks.

5.2.3 Service-level latency

From the perspective of the developer of coordination services, an important performance property is the time it takes to execute a database operation. This includes reading and writing of nodes of varying complexity on virtual and non-virtual nodes, executing type searches, and initiating and committing transactions. A low latency for these is obviously desirable because the service process can typically not progress while waiting for data to arrive. As a result of high latency, the overall service performance degrades, affecting safety and user satisfaction.

In addition to that, a predictable distribution of latency is desirable. In particular, an especially long response time should be avoided – even if this happens rarely – since it stalls the accessing service and might annoy user or have safety or security implications.

Internally, the expected latency is composed of several factors: First, there is the delay for transmission between the service client and the knowledge agent and between the knowledge agent and the database node. These are more or less constant for each submitted operation and dependent on network characteristics known to a potential user who wants to deploy the middleware. It is independent on the choice of the database backend or type of operation; thus it is fair to avoid network transmission between services, KA, and database to obtain more meaningful results.

Another component is time where a database node is waiting for system resources to become available. This can be access to the disk, to a thread pool, or other local resources that are shared between concurrent processes that execute the database operation. This component is sensitive to the load put on a database node and limits the total amount of operations that the database can handle in parallel.

A further component of latency is communication between database nodes. This is specific to distributed databases and thus of particular relevance. Since data might be placed an a remote node, there is a further transmission delay. On top of that the concensus protocols needed to ensure consistency and fault tolerance require additional communication for most operations. This can cost several round trip times in addition to processing delays. CockroachDB, which relies on hybrid timestamps for transaction commit ordering also requires additional time to compensate for synchronisation imprecision. In the old approach with the federated database backend, the inter-node communication (which is actually inter-KA communication) is only a single roundrip and there is an upper boundary. Additionally database nodes sometimes have to transmit batches of data between them to handle failover and rebalancing. This, however, happens asynchronously in the background and should not affect query processing.

Lastly, processing of an operation can be delayed because it has to access a database record that has already been locked by concurrent operations. This is specifically the case when a locking concurrency control algorithm is used to implement transactions. It is expected that the total lock waiting time increases with the number and size of transactions and the overall time a client needs to complete a transaction. This relationship is not linear since the waiting time feeds again into the time a transaction is holding other locks. Thus long waiting times have the potential to delay operation.

Overall, the following measurements are needed:

- Latency of every read or write operation, together with information if the operation was part of a transaction sequence or not and the size of the returned/submitted VSL node tree.

- Latency of every whole transaction sequence, from the begin to the end of the commit operation or to the forced end of the transaction, together with information on the success and the number of reads and writes within it.

5.2.4 Throughput

The throughput refers to the number of operations that the database handles per time. The maximum amount of this metric cannot be measured directly, but has to be observed indirectly by executing very high load scenarios and measure how the latency of operations respond. This maximum throughput is the result of a bottleneck of a system resource.

5.3 Specific dataflow-oriented tests

Considering the whole network that is controlled by the middleware, an important property is the time that is takes for information to disseminate along the control chain. So after a sensor emitted a new value, it is the total amount of time that it takes knowledge aggregation and coordination services to read and process that information and to update the relevant actuators.

This property is best tested in small test scenarios that model a complete flow of information. The scenarios consist of sets of services of a particular class, which exchange data through the VSL. In the graphical representation (e.g., Figure 5.2) they are denoted as a box in the charts. An arch between boxes indicates, that each service of the target group is connected to each service of the origin and consumes all its data. All services emit data containing physical timestamps and vector clocks as payload, so that it can be determined which input is causally related to which output. Coordination between services is handled by subscriptions, so for example knowledge inference services subscribe to a sensor value and poll it whenever it receives an update notification. Most scenarios include several instances of a service of a specific type. This is to simulate some amount of concurrency between data access, which can affect the performance especially of transactional access, and also to obtain a better distribution of the data over database nodes.

Unfortunately the existing implementation of subscriptions and group communication encountered scalability and concurrency issues in a clustered setup. For that reason, it was impossible to distribute the participating services on different nodes, so they all had to be placed on one KA. This is not a big issue for testing the Infinispan and CockroachDB backends since these use a distributed hashtable to distributed data among their database nodes and the results concerning database access are just as meaningful. When used with the original backend with the federated HSQLDB database however, this would mean that all data is co-located on one node which would produce incomparable results – which are only meaningful insofar as they represent a baseline for comparison to a non-distributed setup. Thus only the measurements with the Infinispan and CockroachDB backends are interpreted here. The test setup for the federated backend, including the transaction shim (Algorithm 2, remains fully functional and can be used for future evaluations.

5.3.1 Aggregation Chain Length

The first scenario measures the impact of differences in the number of layers of knowledge inference services. It is relatively obvious that a longer chain will lead to a longer time until an action occurs. To determine how big this effect is quantitatively, the number of theses aggregation chain lengths is a variable parameter of the scenario.

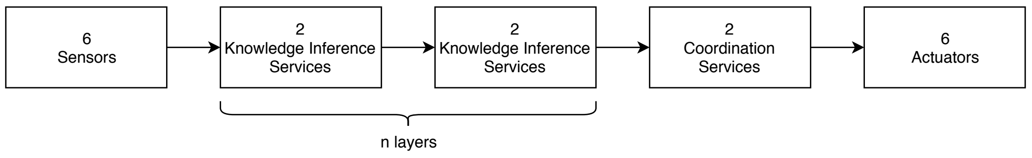
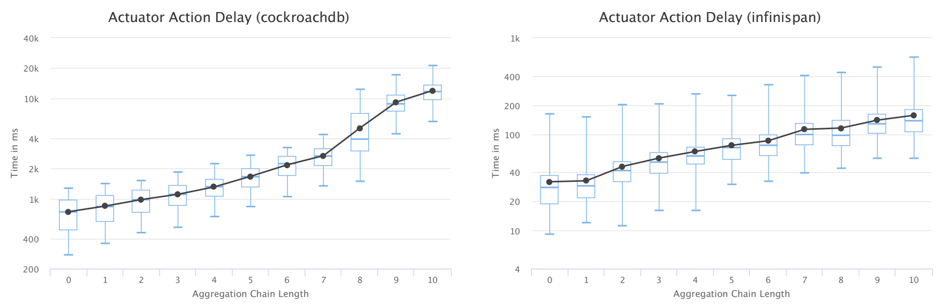

6 Sensors
2 Knowledge Inference Services
2 Knowledge Inference Services
2 Coordination Services
6 Actuators
n layers

Figure 5.2: Topology of the Aggregation Chain Length Scenario. *n* indicates the number of aggregation layers (services that consume and process the values of preceding knowledge inference services or directly form sensors).

The primary expectation of this test is that the aggregation chain increasing in length affects the time it takes for information to travel from sensors to actuators in a more or less linear manner. This expectation holds true for the Infinispan backend, where it takes on average 33 ms when no knowledge inference service is involved, up to 160 ms when 10 layers consisting of 2 knowledge inference services are chained. In CockroachDB this increase also exists with higher delays ranging from 800 ms to 11 s. Also, the increase seems to be slightly overproportionate to the chain length, and measurements of the read and write latencies suggest also an increase in both transactional and non-transactional database operations, which explain this relationship. These effects were not present in the measurements with the Infinispan backend where operation latencies show no significant increase.

Interestingly, the length of the chain affects the latency of transactional operations in CockrochDB, which is in so far unexpected as knowledge inference services are in this test non-transactional and access, with the exception of the last one in the chain, only records that are independent of the transactional coordination services. The likely explaination of this is that the push-based linking of aggregation steps which consist of several single knowledge inference nodes increases the frequency of notifications and therefore thriggers the execution of following processing steps more often. The increase in transaction duration is then explained by the fact that the concurrent execution of transactions degrades their performance in CockroachDB as also demonstrated by the Coordination Scale scenario.

The increase in notification output can in fact be observed in this simulation, noticeable in the count of coordination service transactions which increased from 226 with a chain



(a) CockroachDB (b) Infinispan

Figure 5.3: Time between sensor output and actuator read of a causally related event dependent on aggregation chain length

length of 0 up to 491 with 10 knowledge inference steps. This is something that should be kept in mind when chaining services. However, since notifications in DS2OS are by design decoupled from access to the value of the subscribed address, an explosion in the number of following read accesses can be prevented if the services are implemented properly and thus do not issue read requests when a request on the same address is still executing.

5.3.2 Coordination Complexity

In this scenario, the number of knowledge inference services and actuators is variable. Both classes of services are connected to coordination services which perform transactional reads on knowledge inference services and writes on actuators. The number of these services and thus the number of reads and writes that the coordination services perform within one transaction will determine the time that it takes for this transaction to complete. Since there are two such coordination services, their transactions will compete with each other and dependent on the concurrency control capabilities of the underlying database leading to delays.

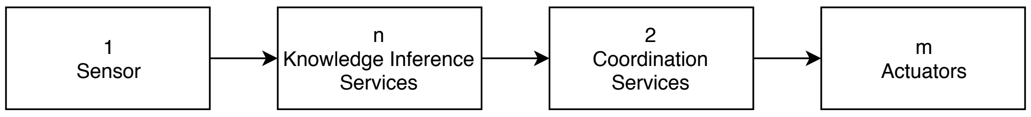
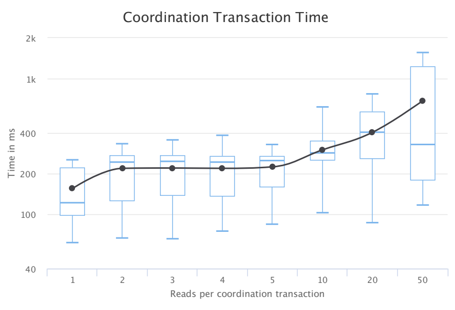
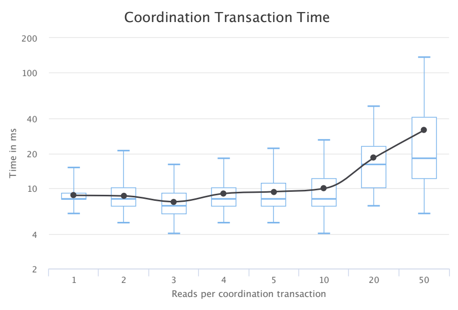


Figure 5.4: Topology of the Coordination Complexity Scenario. *n* indicates the number of Knowledge Inference services that form the input and *m* the number of Actuators that form the output of the coordination services.

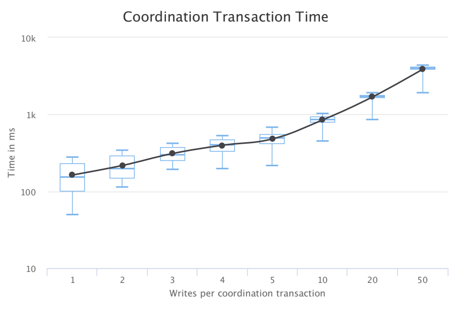
Both Infinispan and CockroachDB are able to handle this task well. With increasing numbers of operations, the time to complete transactions did not increase overproportionally. A less than asymptotically linear increase cannot be achieved since there always is network communication between the service and the database between each



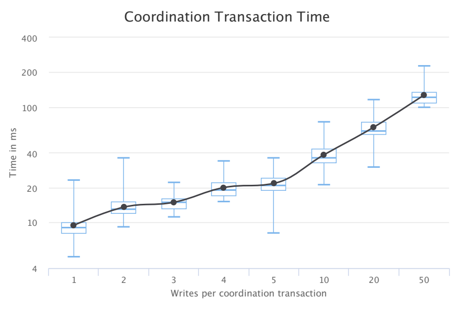
(a) CockroachDB



(b) Infinispan



(c) CockroachDB



(d) Infinispan

Figure 5.5: Time for the execution of a coordination transaction dependent on the containing number of reads (from KI services) and writes (to actuators).

operation. Reads have a lower impact than writes, which is analogue to the latencies of non-transactional counterparts.

5.3.3 Coordination Scale

The impact of such competition between transactions can further be investigated by increasing the number of homogeneous coordination services. The expectation is that services that concurrently access the same data transactionally, produce conflicts, which leads to transaction aborts and/or longer delays depending on the databases' transaction schedulers. This shall be evaluated in this scenario, where the number of coordination services is the variable.

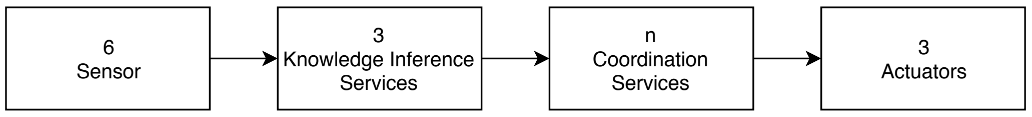
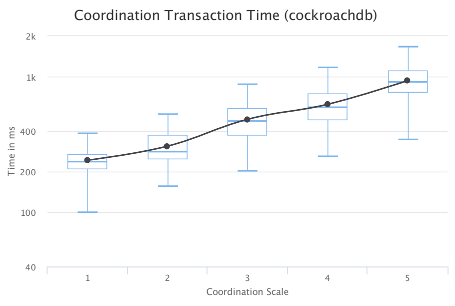
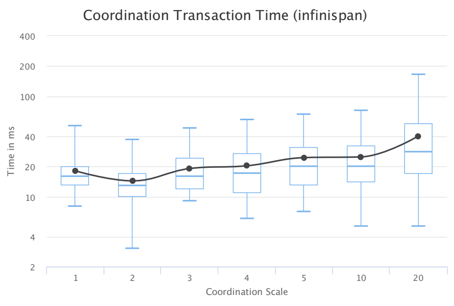


Figure 5.6: Topology of the Coordination Scale Scenario. *n* indicates the number of coordination services that concurrently access the same data.



(a) CockroachDB



(b) Infinispan

Figure 5.7: Time for the execution of a coordination service transaction in scenarios varying in the number of coordination services with identical access patterns

Unsurprisingly, this had an influence on transaction latency. The CockroachDB backend could not schedule more this workload for more than 5 parallel transactions and duration increased to times longer than the test execution (5 minutes).

5.3.4 Sensor Data Size

This scenario should test how the databases cope with varying sizes of data. To do this, the model incorporates sensors that emit relatively small amounts of data (only the vector clock) and a second set of sensors with variable payload size. It is expected that the latency for read and write operations increases with increasing payload size, since it requires more time to transmit and store the data.

With both backends, there was no significant difference, only non-transactional accesses with the Infinispan backend was slightly faster with lower payload. Overall the results suggest, that the databases can cope with data size that is in the order of a few hundred kilobytes and that the delay introduced by it is negligible.

5.3.5 Aggregation Chain Length Heterogeneity

In this scenario chains of knowledge inference services of different length are simulated. This is to evaluate if different versions of a sensor value are causally related the control output for the actuators at the end. This can happen when the same data is processed in two different tasks whose output is merged again afterwards and if one of these processes is slower than the other, e.g., because it incorporates more steps as in this scenario.

No such event occurred in tests with either backend with chain lengths up to 20 and a sensor update frequency of 5 seconds. Despite this, it is still possible that such event happen, since transactions cannot cover processing steps over multiple services.

Processing tasks that are vulnerable to inconsistent results have to take own precautions to prevent this. That can be to add a vector clock in the payload and delay producing an output when input data from another KI service is causally related to an older version of a sensor value.

Apart from that, the performance metrics with both the Infinispan and CockroachDB backends were similar to those of the Aggregation Chain Length scenario, which is apart from the additional parallel knowledge inference layer identical.

5.3.6 Indirect Control

This scenario investigates the impact of serially linked transactional services, which is common when control services are used to manage access to devices. This pattern involves a lot of transactions which access the same data. Thus there is a risk that the transactions affect each other, leading delays or aborts.

This apprehension did not play out with the Infinispan backend. There, the time that the control services need to execute a transaction remained constant. CockroachDB however seemed to have problems with this kind of load pattern. As the chain of services become longer, the transaction throughput drastically decreased and only very few of them were carried out, while others seemed to experience very long delays, so that only few sensor data updates actually made it through and lead to an actuator action. From a chain length of 4, significantly less transactions were carried out within the 5 minutes of test execution and from 5 layers upwards, the test scenario got completely stuck in the sense no actuator access was recorded anymore.

A potential problem that was also noticed in the Aggregation Chain Length scenario is, that push-based chaining of services through subscriptions leads to an exponential increase of service action. In this scenario this is even more dangerous as transactional operations are expensive and might congest the scheduler. Services developers must consider this.

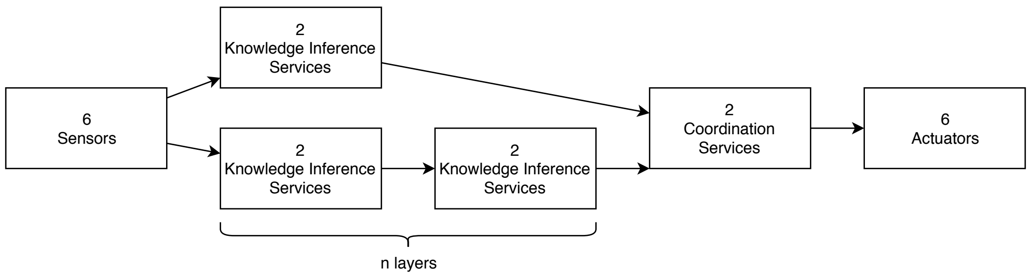


Figure 5.8: Topology of the Aggregation Chain Length Heterogenity Scenario. n indicates the number of aggregation layers (services that consume and process the values of preceding knowledge inference services or directly form sensors).

5.4 Load tests

Load tests incorporate a mixture of traffic that is assumed to be representative of a real scenario. The properties of interest are here how much load the database can handle and how that throughput affects the latency of database access operations. In contrast to the previously described small test scenarios, using subscriptions for service coordination has been omitted, since this would put additional load on the network and KA processes and affect the VSL access rate of services and makes it harder to control, while it is not a feature that accesses the database directly. Instead, services carry out their work at fixed intervals.

5.4.1 Load generation

When it comes to determine the total amount of load that a smart space would put on the middleware, there are essentially two factors: First, the amount of hardware, that is the sensors, actuators and other devices, that directly access data mostly in a nontransactional fashion. This number grows with building size and over time as technology progresses and smart devices become more ubiquitous. Here it is foremost important that the IoT middleware is capable of handling the traffic present in a smart space of a certain scale. Second, the amount of intelligence that is implemented to make the hardware form a smart space. This translates to the amount of coordination and knowledge inference services, the frequency and the amount of records that they access. Installing additional coordination services comes at little price for the end-user, in contrast to installing additional hardware. So it is expected that users will install more such services until they exceed the technical capabilities of the IoT middleware. So, apart from basic coordination and monitoring tasks which implement necessary core functionality, the scale of intelligent services will also be determined by capabilities that the IoT middleware offers. Thus, a high performance of coordination services is desirable, but it is hardly possible to come up with a specific baseline.

The test scenario takes this into account by six parameters:

- Amount of sensors
- Amount of actuators

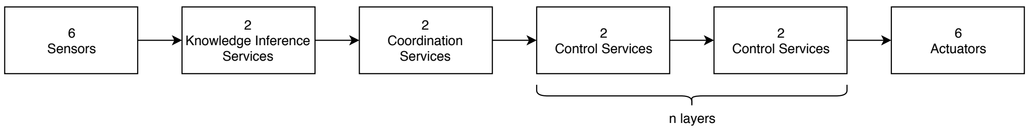
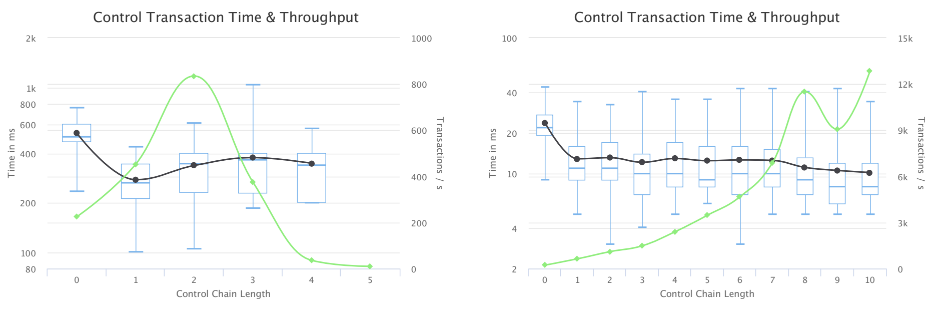


Figure 5.9: Topology of the Indirect Scenario. *n* indicates the number of layers of control services that execute a control function transactionally).



(a) CockroachDB (b) Infinispan

Figure 5.10: Time for the execution of a control service transaction and throughput of transactions.

- Amount of transactional complex devices
- A coordination frequency factor that determines how often coordination and knowledge inference services poll data from sensors, complex devices and other services
- A coordination complexity factor that determines how many other nodes coordination and knowledge inference services poll in each iteration
- A coordination quantitiy factor that determines how many coordination and knowledge inference services exist in the network

5.4.2 Latency

Overall, the Infinispan backend allowed had the lowest latency for reads. This is likely due to in-memory storage and replication – since every record is stored on three nodes, remote access can be avoided in one out of two cases. The federated HSQLDB backend had the highest write performance. This is very likely the result of the absence of replication which eliminates any remote access to complete a write operation.

CockroachDB showed significantly higher latency, for both reads and writes, but writes took especially long. Most likely this is due to the consensus protocol that CockroachDB uses in order to maintain consistency and the usage of disk-based storage.

There was no considerable difference in latency between non-transactional and transactional operations. This is no surprise since also non-transactional operations are affected by concurrency control methods.

Figure 5.11 shows a comparison of the latency in one execution of the test setup with 500 sensors, 100 actuators, 100 devices, 70 coordination services each performing 5.6 operations per transaction on average every 14.2 seconds on average.

The measurements show that scenarios with a high number of physical infrastructure (5k sensors, 5k actuators, 1k complex devices) congest the CockroachDB and HSQLDB backends, resulting in very long latency above or close to 1 second, which is which is prohibitively long for applications that require fast reactions or direct user interaction.

5.4.3 Throughput

For conducting the throughput measurements, a test model consisting of 1000 sensors, 1000 actuators and 1000 coordination service instances was used. Then the frequency of their VSl access was varied to yield a specific load of reads, writes, mixed traffic and transactions. Infinispan generally achieved the best results. With the described setup of 6 hosts, it could handle more than 10k actuator reads per second. It had similar write performance as the federated HSQLDB backend, which is remarkable as Infinispan replicates data to three different hosts. Also, in processing transactions it outperformed the other candidates with 900 transactions per second, themselves containing 5 reads and writes each. There also did not seem to be a notable difference in transactional and non-transactional access to the data in terms of throughput. CockroachDB and the original database backend performed considerably worse. CockroachDB did only occasionally complete any transaction successfully (the maximum was 2); HSQLDB with the concurrency control shim peaked at 23 transactions per second, which is less than with equivalent non-transactional traffic. The CockroachDB backend had a low write throughput of 120 writes per second while write traffic also reduced read throughput to about 130.

5.4.4 Fault tolerance

DS2OS is itself not designed to be fault-transparent, since it is not completely location-transparent. VSL addresses that are located on a failed knowledge agent simply become unavailable. For more advanced fault tolerance properties, services must also be replicated and made location-transparent which is currently not the case and out of scope of this work. Thus improvements of fault tolerance are confined to the storage layer. However the new design with a distributed database should be at least as good in terms of fault-tolerance as the old one. That includes that healthy parts of the network are not affected when some nodes fail and that their data stays accessible. With a distributed database this is not necessarily the case as the crashed nodes can also contain data associated to otherwise healthy knowledge agents.

To test failure tolerance, the Aggregation Chain Length scenario was taken. After 90 seconds of test execution, a number of database nodes were shut down.

CockroachDB could tolerate up to 2 node failures without a measurable decline in the performance metrics.

5.5 Threats to Validity

There are some factors that might interfere with the validity and generalisability of the obtained results. Most importantly, the tests are subject to the design decisions made when modeling the Virtual State Layer. There is a variety of different options, as outlined in Chapter 4 and these affect the performance. Especially the hierachical structure of the VSL, allowing operations on entire subtrees, is something that is difficult to model in a way so that queries can be performed efficiently. Some approaches to accomplish it will have better performance for small subtrees or focused queries, while others scale better for bigger subtrees. To mitigate this, for both backends a similar approach was chosen by indexing the subtree column, so that the results are comparable. The load test model contains a mixture of VSL node trees of various sizes in order to add some variance in this regard.

Network latency has been modelled statically, but it might vary from depoyment to deployment. It definitely has an effect on operation latencies, especially if a lot of communication is necessary to complete a request. The one millisecond that has been set as the average transmission delay is a rather pessimistic estimation representative for one-site deployments.

A property that has not been addressed in the quantitative evaluation is horizontal scalability, meaning the ability of the database to increase throughput by execution on a higher number of nodes. That the databases posses this ability is relatively clear, since both of them avoid the usage of centralised resources and try to spread all data out by sharding and replication mechanisms. Generally these mechanisms do not scale linearly. For the Chord algorithm, for example, the number of hops required to reach a certain node increases logarithmically. The effects will, however, only become noticeable when the cluster size increases significantly, not only by a few more nodes, which was not achieveable with the available hardware. However, the setup is sufficiently big enough to test whether the databases are able to execute efficiently in distributed settings in principle. Since that is the case, there is a high probability that as they work well on six nodes, they will also do so on 50 nodes, which could be considered a rather large deployment already.

As pointed out before, the databases have a number of important differences, notably execution environment, data model, consistency guarantees, and storage location. This makes it difficult to attribute the measured differences in the performance metrics to any of these differences. The choice of two such very different products is in so far justified as both chosen databases are each representatives for a class of available software for which similar products can be found, employing similar approaches and offering comparable features.

To overcome variance in measurements, all tests have been carried out long enough to obtain a large sample of several hundred measurements for the specific scenarios and several thousand for the load tests. The average and median measured values were relatively stable. The 99th percentile was naturally subject to higher levels of variance but should still be a good estimate of the maximum expected delay in a specific configuration.

5.6 Conclusion

Overall, Infinispan had significantly better performance than CockroachDB. There are several likely reasons for this. Most notably, Infinispan offers lower consistency guarantees than CockroachDB does. The latter uses a consensus protocol to ensure single value consistency, which requires multiple roundtrips between a majority of replicas for any write access. In Infinispan, in contrast, there is one leader that coordinates the writes, so only one node which has the replicas must be contacted to obtain the data. In addition to that, since the number of replicas versus the number of nodes in total is high in the test setup, the probability that the requested data is locally available is relatively high, and contacting a remote node can be avoided. This approach becomes problematic when a leader is slow and is suspected failed by a subset of the other nodes or if there is a network partition. In these cases inconsistency might occur. CockroachDB's longer response time can be seen as the price for this guarantee. Additionally CockroachDB guarantees sequential transaction isolation and high availability of the transaction manager.

Another reason is that CockroachDB always stores data on disk. This applies to transaction management data as well. And, since every operation can possibly interfere with a concurrently running transaction, locks have to be read or written as well. CockroachDB's durability guarantees also require that a log of all operations is persisted, so that every operation can include several accesses of the filesystem. Infinispan in contrast stores all its data in main memory which is considerably faster.

Finally, Infinispan is a key-value store which allows the programmer to optimise the structure and indexes for the expected access patterns. This was done in the connector implementation. So, since the consistency of structural information of the VSL is not so critical[1], it is marked as asynchronously replicated. Subsequences of VSL addresses are marked to be indexed, which speeds up querying for subtrees. These are two things that are not possible in the relational interface provided by CockroachDB. Its strong consistency guarantees apply to all its data and there is no option to exclude some parts and it is not possible to use set types and use indexes on these. On top of that, the

[1]What could happen is that the type search returns a service that no longer exists or that structural information (e.g., access permission) are old. These are relatively static because given by the context model, which are cached anyway.

SQL interface comes with some overhead too, requiring parsing of the SQL statement, serialisation of the transmitted data to a binary format, and generation of a query plan, even for very simple operations.

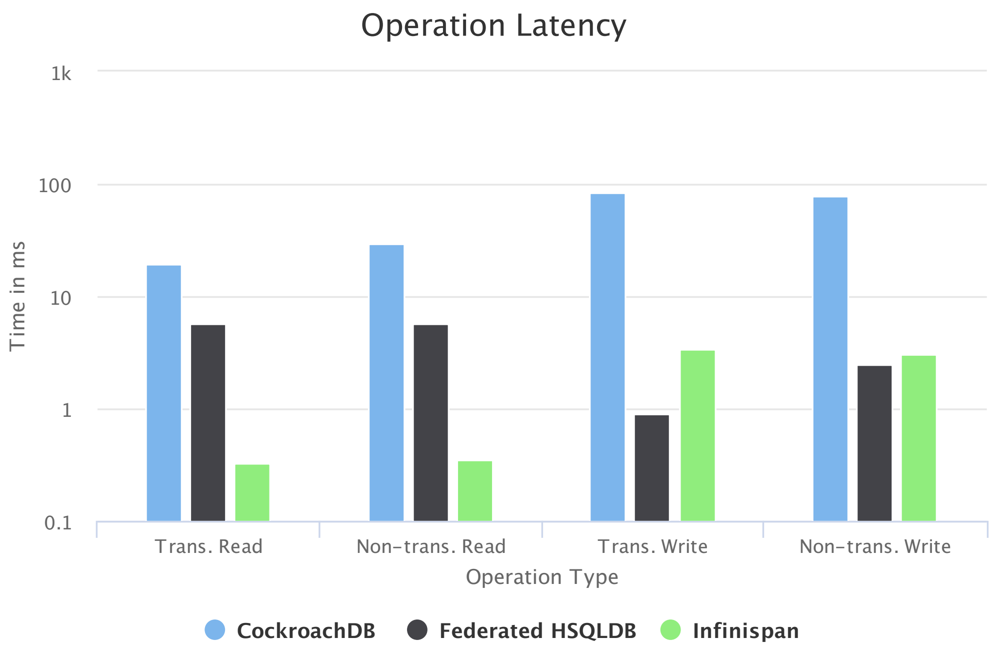
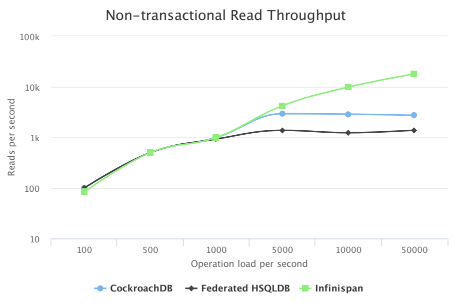
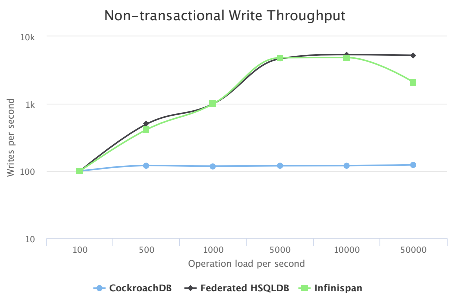


Figure 5.11: Comparison of latencies in a test setup (sensors=500, actuators=100, complex devices=100 complexity factor=0.04, frequency factor=0.7, quantity factor=0.1)



(a) Read Throughput



(b) Write Throughput

Figure 5.12: Plots of simulated load against successfully executed operations, the amount that a backend can handle flattens at some point, determining its maximum throughput

Chapter 6

Conclusion

In this work, we evaluated how distributed databases can be used to store operational IoT data efficiently and how to use them to address concurrency issues through a transaction abstraction.

To accomplish this, the key requirements on a database are scalability which is obtained by automatic partitioning, availability which can be achieved through replication, querying options for hierarchical structures or for prefixes of strings, and security features. Of lesser important are the employed data model and querying and indexing support beyond the capabilities mentioned.

Those requirements are satisfied by a number of available products, although the integration of these can look very different, as demonstrated with CockroachDB and Infinispan connector implementations.

6.1 Differences & Trade-offs

The Infinspan backend had a low latency and high throughput. CockroachDB fared an order of magnitude worse in both. This can be explained in the richer set of features, that CockroachDB offers, in particular modelling, indexing, and query features, the persistence on disk, the strong semantics, and an excellent capability to deal with node churn. A major advantage is also that it can be used like any other SQL database. Infinispan does not offer such comfort. Indexes must explicitly defined, advanced queries are impossible, and the consistency guarantees are lower. In return the developer gets the chance to apply own optimisation, which has been done, for instance, by replicating VSL structure data. The result is a much lower latency and higher throughput. Unfortunately it also comes at the price of fault-tolerance: Under churn, some VSL data was not accessible anymore.

The benchmark results highlight a trade-off between performance, consistency and fault-tolerance, which was expected. Record consistency requires coordination of reads and writes, while fault-tolerance requires that the protocol must not block when a node fails and that failures are detected reliably. Only sophisticated algorithms like Raft of Paxos are able to guarantee that. The same applies to concurrency control methods, where transaction records and recovery data must be stored consistently and the peers must agree on the status of locks and order of operations. Infinispan provides an approximation of this, although it is not perfect. But obviously, the performance benefits are significant. This makes choosing between one and the other difficult: The need for transactional safety does definitely exist, as pointed out earlier, on the other hand real-time performance is also a desirable property.

There is a further trade-off between failure tolerance and measured latency. As observed with the CockroachDB backend, the latency of operations can get so high, that it becomes a problem for availability. In a realtime system, long response times are not any more tolerable than actual failure to serve a request, as it was observed with the Infinispan backend under churn.

6.2 Recommendations for Implementation

The generic design of the DS2OS interface also leaves many decisions concerning service coordination and data access up to the service developers. It is up to them whether to use pull-based communication with get operations or push-based communication with notifications, coordination by persistent state or remote method invocation through Virtual Nodes, whether to use transactional access or not, where to locate a service and which execution model to use. These decisions have significant performance implications as shown in the benchmarking and affect the execution of other services as well (e.g., through locks). The basis for such a task has already been laid in the original publication and in this thesis, out of the necessity to express meaningful patterns of service interaction for a realistic test setup:

- Classification into data-generating sensor services, aggregating, knowledge inference services, decision-making coordination and management services, executing actuator services, and complex devices.
- Clocks attached to sensor data and inferred information to obtain time-consistent knowledge.
- Allowing knowledge inference services to keep private state that results from a series of input data and is thus recomputable.
- Keeping coordination and management services stateless and wrapping their decision-making process into a transaction in order to obtain flexible re-scheduling.

- Notification-driven processing of data to obtain a push-based flow of information. Services that access the same data repeatedly should cache it and only read it again after a new notification has been received.

The overall topology should have the shape of a two-sided funnel: On both sides, high number of information sources and actuators as sinks, a smaller number of inference services that process the raw data and infer knowledge, and a smaller number of decision-making services and coordination tasks in between. This way, it is also ensured that the services can execute efficiently. These building blocks are sufficient to build acyclic real-time building control. Templates or libraries can be provided to reduce the workload of implementing these interaction patterns. In the ideal case, the user specifies the input sources and output and provides a function that maps one to the other while the template code handles transactions, subscriptions and other details of interaction.

Chapter 7

Future work

Given the results, the choice is currently between using a fast database backend that has availability and consistency shortcomings in edge cases or a perfectly consistent and available but slow one. The ultimate answer might be to drop the idea of a homogeneous interface that treats all data equally and and instead enable costlier features and properties only where they are really needed. This is possible because services have very different needs in regard to data storage and suggests a hybrid approach, where a consistent and available database is used for the parts that require it, and a faster one for the rest.

For sensor readings, the order of values and their age counts, requirements that are already met with eventual consistency guarantees. Historical data is even easier to manage from a consistency viewpoint, as it never changes. Instead, it requires sophisticated indexing and access methods to query for aggregate information and to make use of it, which is something that is not addressed in the current design of DS2OS at all. For inferred knowledge, causality is an important aspect, but atomicity is not, so using transactions might simply be an overkill. They only become important when it comes to implementing decision rules in order to avoid conflicts and to ensure consistent atomic access to multiple output devices. Also, for data that stems from services which mainly produce data, such as sensors and knowedge aggregators, there is typically only this one writer. For these kinds of data, alternatives exist: Event brokering systems like Kafka can preserve causality order and met realtime requirements and might be suitable for sensor-processing of information. Scalable, non-transactional storages with indexing capabilities and a selection of adapters for analytical software, like for instance Cassandra or Riak, are better suited to historical data. For complex devices that encapsulate their own logic, exclusive access on its own data combined with message-based interaction patterns for all external access might be the more suitable choice, which is something that DS2OS already provides. Transactional capabilities are needed for data that fulfils a coordinating function, rather than represent a measured or inferred fact. This is the case for actuators, coordination, and control services, which write data in order to cause some change and are therefore subject to conflicts.

A middleware product that separates these kinds of data must be aware of the nature of services and their access behaviour on data. This means the uniform interface that treats every service alike and provides the same access methods to them is not possible. Instead there have to be separate interfaces for each service class or the nature of data is described in the associated context model, so that the middleware can make the decision where to place it. The advantage of such a hybrid architecture is that it is faster and at the same time safe for parts where transactional state changes are needed.

Another improvement that comes up is decouple the services from the knowledge agents. So far, services are not location-transparent – every service's VSL address contains the identifier of the knowledge agents where it is registered. With the federated database, this is necessary for routing access requests to the right location. With a distributed database the requests can be served by any knowledge agent, making it unnecessary. The adaptation requires some changes to the DS2OS codebase, since some parts rely on extracting the agent identifiers from VSL addresses. The benefit is full location transparency. This enables load-balancing and automatic fail-over for knowledge agent failures for services and improves availability. For transactional and stateless services, another possibility is also to decouple them from their execution environment. Since they can be stopped and restarted at any point, they can be replicated and migrated between physical nodes and their lifecycle can be managed automatically, providing higher availability and tolerance to failures.

Bibliography

[1] S. Wang, *Intelligent buildings and building automation.* Routledge, 2009.

[2] S. Helal, W. Mann, H. El-Zabadani, J. King, Y. Kaddoura, and E. Jansen, "The gator tech smart house: A programmable pervasive space," *Computer*, vol. 38, no. 3, pp. 50–60, 2005.

[3] Y. Shi, W. Xie, G. Xu, R. Shi, E. Chen, Y. Mao, and F. Liu, "The smart classroom: merging technologies for seamless tele-education," *IEEE Pervasive Computing*, vol. 2, no. 2, pp. 47–55, 2003.

[4] A. H. Ngu, M. Gutierrez, V. Metsis, S. Nepal, and Q. Z. Sheng, "Iot middleware: A survey on issues and enabling technologies," *IEEE Internet of Things Journal*, vol. 4, no. 1, pp. 1–20, 2017.

[5] M. A. Razzaque, M. Milojevic-Jevric, A. Palade, and S. Clarke, "Middleware for internet of things: a survey," *IEEE Internet of Things Journal*, vol. 3, no. 1, pp. 70–95, 2016.

[6] O. Vermesan, P. Friess, P. Guillemin, S. Gusmeroli, H. Sundmaeker, A. Bassi, I. S. Jubert, M. Mazura, M. Harrison, M. Eisenhauer *et al.*, "Internet of things strategic research roadmap," *Internet of Things-Global Technological and Societal Trends*, vol. 1, pp. 9–52, 2011.

[7] C. Joshua and J. Anne, "Challenges for database management in the internet of things [j]," *IETE Technical Review (Institution of Electronics and Telecommunication Engineers, India)*, vol. 26, no. 5, pp. 320–324, 2009.

[8] T. Li, Y. Liu, Y. Tian, S. Shen, and W. Mao, "A storage solution for massive iot data based on nosql," in *Green Computing and Communications (GreenCom), 2012 IEEE International Conference on.* IEEE, 2012, pp. 50–57.

[9] M. Ma, P. Wang, and C.-H. Chu, "Data management for internet of things: challenges, approaches and opportunities," in *Green Computing and Communications (GreenCom), 2013 IEEE and Internet of Things (iThings/CPSCom), IEEE International Conference on and IEEE Cyber, Physical and Social Computing.* IEEE, 2013, pp. 1144–1151.

[10] L. Jiang, L. Da Xu, H. Cai, Z. Jiang, F. Bu, and B. Xu, “An iot-oriented data storage framework in cloud computing platform,” *IEEE Transactions on Industrial Informatics*, vol. 10, no. 2, pp. 1443–1451, 2014.

[11] A. Copie, T.-F. Fortis, and V. I. Munteanu, “Benchmarking cloud databases for the requirements of the internet of things,” in *Information Technology Interfaces (ITI), Proceedings of the ITI 2013 35th International Conference on.* IEEE, 2013, pp. 77–82.

[12] T. A. M. Phan, J. K. Nurminen, and M. Di Francesco, “Cloud databases for internet-of-things data,” in *Internet of Things (iThings), 2014 IEEE International Conference on, and Green Computing and Communications (GreenCom), IEEE and Cyber, Physical and Social Computing (CPSCom), IEEE.* IEEE, 2014, pp. 117–124.

[13] P. Paethong, M. Sato, and M. Namiki, “Low-power distributed nosql database for iot middleware,” in *Student Project Conference (ICT-ISPC), 2016 Fifth ICT International.* IEEE, 2016, pp. 158–161.

[14] H. Fatima and K. Wasnik, “Comparison of sql, nosql and newsql databases for internet of things,” in *Bombay Section Symposium (IBSS), 2016 IEEE.* IEEE, 2016, pp. 1–6.

[15] M.-O. Pahl, “Distributed smart space orchestration,” Dissertation, Technische Universität München, München, 2014.

[16] M. P. Herlihy and J. M. Wing, “Linearizability: A correctness condition for concurrent objects,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 12, no. 3, pp. 463–492, 1990.

[17] J. Gray and A. Reuter, *Transaction processing: concepts and techniques.* Elsevier, 1992.

[18] M. Abu-Elkheir, M. Hayajneh, and N. A. Ali, “Data management for the internet of things: Design primitives and solution,” *Sensors*, vol. 13, no. 11, pp. 15 582–15 612, 2013.

[19] R. Cattell, “Scalable sql and nosql data stores,” *Acm Sigmod Record*, vol. 39, no. 4, pp. 12–27, 2011.

[20] M. Stonebraker, “Newsql: An alternative to nosql and old sql for new oltp apps,” *Communications of the ACM. Retrieved*, pp. 07–06, 2012.

[21] F. Raja, M. Rahgozar, N. Razavi, and M. Siadaty, “A comparative study of main memory databases and disk-resident databases,” in *TRANSACTIONS ON ENGINEERING, COMPUTING AND TECHNOLOGY.* Citeseer, 2006.

[22] E. K. Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim, “A survey and comparison of peer-to-peer overlay network schemes,” *IEEE Communications Surveys & Tutorials*, vol. 7, no. 2, pp. 72–93, 2005.

[23] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan, “Chord: a scalable peer-to-peer lookup protocol for internet applications,” *IEEE/ACM Transactions on Networking (TON)*, vol. 11, no. 1, pp. 17–32, 2003.

[24] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, “The hadoop distributed file system,” in *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on.* Ieee, 2010, pp. 1–10.

[25] C. Cachin, R. Guerraoui, and L. Rodrigues, *Introduction to reliable and secure distributed programming.* Springer Science & Business Media, 2011.

[26] W. Kim, B.-J. Choi, E.-K. Hong, S.-K. Kim, and D. Lee, “A taxonomy of dirty data,” *Data mining and knowledge discovery*, vol. 7, no. 1, pp. 81–99, 2003.

[27] G. Weikum and G. Vossen, *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery.* Elsevier, 2001.

[28] J. Gray and L. Lamport, “Consensus on transaction commit,” *ACM Transactions on Database Systems (TODS)*, vol. 31, no. 1, pp. 133–160, 2006.

[29] C. J. Fidge, “Timestamps in message-passing systems that preserve the partial ordering,” 1987.

[30] F. Mattern *et al.*, “Virtual time and global states of distributed systems,” *Parallel and Distributed Algorithms*, vol. 1, no. 23, pp. 215–226, 1989.

[31] E. A. Brewer, “Towards robust distributed systems,” in *PODC*, vol. 7, 2000.

[32] S. Gilbert and N. Lynch, “Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services,” *Acm Sigact News*, vol. 33, no. 2, pp. 51–59, 2002.

[33] D. Abadi, “Consistency tradeoffs in modern distributed database system design: Cap is only part of the story,” *Computer*, vol. 45, no. 2, pp. 37–42, 2012.

[34] A. Corbellini, C. Mateos, A. Zunino, D. Godoy, and S. Schiaffino, “Persisting big-data: The nosql landscape,” *Information Systems*, vol. 63, pp. 1–23, 2017.

[35] M. Stonebraker and R. Cattell, “10 rules for scalable performance in’simple operation’datastores,” *Communications of the ACM*, vol. 54, no. 6, pp. 72–80, 2011.

[36] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, “Zookeeper: Wait-free coordination for internet-scale systems.” in *USENIX annual technical conference*, vol. 8. Boston, MA, USA, 2010, p. 9.

[37] A. S. Foundation. (2017) Apache couchdb 2.1 documentation. [Online]. Available: http://docs.couchdb.org/en/2.1.0/index.html

[38] B. Holt, *Scaling CouchDB : replication, clustering, and administration*, 1st ed. Sebastopol, Calif.: O'Reilly Media, c2011, elektronische Ressource. [Online]. Available: http://proquest.safaribooksonline.com/9781449304942

[39] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[40] B. G. Tudorica and C. Bucur, "A comparison between several nosql databases with comments and notes," in *Roedunet International Conference (RoEduNet), 2011 10th*. IEEE, 2011, pp. 1–5.

[41] M. N. Vora, "Hadoop-hbase for large-scale data," in *Proceedings of 2011 International Conference on Computer Science and Network Technology*, vol. 1, Dec 2011, pp. 601–605.

[42] M. Stonebraker and A. Weisberg, "The voltdb main memory dbms." *IEEE Data Eng. Bull.*, vol. 36, no. 2, pp. 21–27, 2013.

[43] C. Labs. (2017) Cockroachdb docs – architecture. [Online]. Available: https://www.cockroachlabs.com/docs/stable/architecture/overview.html

[44] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild *et al.*, "Spanner: Google's globally distributed database," *ACM Transactions on Computer Systems (TOCS)*, vol. 31, no. 3, p. 8, 2013.

[45] M. Demirbas, M. Leone, B. Avva, D. Madeppa, and S. Kulkarni, "Logical physical clocks and consistent snapshots in globally distributed databases," 2014.

[46] B. Technologies. (2017) Riak kv docs – concepts. [Online]. Available: http://docs.basho.com/riak/kv/2.2.3/learn/concepts/

[47] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: amazon's highly available key-value store," in *ACM SIGOPS operating systems review*, vol. 41, no. 6. ACM, 2007, pp. 205–220.

[48] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "Conflict-free replicated data types," in *Symposium on Self-Stabilizing Systems*. Springer, 2011, pp. 386–400.

[49] T. Schütt, F. Schintke, and A. Reinefeld, "Scalaris: reliable transactional p2p key/-value store," in *Proceedings of the 7th ACM SIGPLAN workshop on ERLANG*. ACM, 2008, pp. 41–48.

[50] M. Jelasity, A. Montresor, and O. Babaoglu, "T-man: Gossip-based fast overlay topology construction," *Computer networks*, vol. 53, no. 13, pp. 2321–2339, 2009.

[51] T. Schütt, F. Schintke, and A. Reinefeld, "Chord#: Structured overlay network for non-uniform load-distribution," 2005.

[52] F. Schintke, A. Reinefeld, S. Haridi, and T. Schütt, "Enhanced paxos commit for transactions on dhts," in *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on.* IEEE, 2010, pp. 448–454.

[53] T. I. community. (2018) Infinispan 9.1 user guide. [Online]. Available: https://docs.jboss.org/infinispan/9.1/pdf/user_guide.pdf

[54] A. Chervenak, I. Foster, C. Kesselman, C. Salisbury, and S. Tuecke, "The data grid: Towards an architecture for the distributed management and analysis of large scientific datasets," *Journal of network and computer applications*, vol. 23, no. 3, pp. 187–200, 2000.

[55] W. R. d. Santos, *Infinispan data grid platform definitive guide.* Packt Publ., 2015.

[56] H. Salhi, F. Odeh, R. Nasser, and A. Taweel, "Open source in-memory data grid systems: Benchmarking hazelcast and infinispan," in *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering.* ACM, 2017, pp. 163–164.

[57] D. Setrakyan. (2014) Two-phase-commit for distributed in-memory caches. [Online]. Available: http://gridgain.blogspot.de/2014/09/two-phase-commit-for-distributed-in.html

[58] P. Inc. (2018) Tidb documentation. [Online]. Available: http://download.pingcap.org/tidb-manual-en.pdf?v=1526301490