Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit Nr. 186

Control-plane Consistency in Software-defined Networking: Distributed Controller Synchronization using the ISIS^2 Toolkit

Jan Strauß

Course of Study:	Softwaretechnik
Examiner:	Prof. Dr. rer. nat. Dr. h. c. Kurt Rothermel
Supervisor:	M. Sc. Thomas Kohler
Commenced:	2014-11-06
Completed:	2014-05-08
CR-Classification:	C.2.3, C.2.4

Abstract

Software-defined Networking (SDN) is a recent approach in computer networks to ease the network administration by separating the control-plane and the data-plane. The data-plane only forwards packets according to certain rules specified by the control-plane. The control-plane, implemented by a software called controller, determines the forwarding rules based on a global view of the network.

In order to increase fault tolerance and to eliminate a possible performance bottleneck, the controller can be distributed. The synchronization of the data that holds the global view is conventionally realized using distributed key-value stores offering a fixed consistency semantic, not respecting the heterogeneous consistency requirements of the data items in controller state. The virtual synchrony model, an alternative approach to the commonly used state machine replication method, offers a more flexible solution that can result in higher performance when certain assumptions on the data kept in controller state can be made.

In this thesis a distributed controller based on OpenDaylight, a state-of-the-art SDN controller and the ISIS^2 library, that implements the virtual synchrony model, is proposed. The modular architecture of the proposed controller and the usage of a platform independent data model allows to extend or replace parts of the system. The implementation of the distributed controller is described and the macro and micro performance is evaluated with benchmarks.

Kurzfassung

Software-defined Networking (SDN) ist ein aktueller Ansatz zu Computernetzwerken, der die Netzwerkadministration vereinfacht, in dem die Kontrollschicht von der Weiterleitungsschicht getrennt wird. Die Weiterleitungsschicht ist nur für das Weiterleiten von Paketen nach Regeln zuständig, die von der Kontrollschicht festgelegt werden. Die Kontrollschicht, die von einer Controller genannten Software implementiert wird, legt die Weiterleitungsregeln, basierend auf einer globalen Sicht auf das Netzwerk, fest.

Um die Ausfallsicherheit zu erhöhen und um einen möglichen Leitungsengpass zu eliminieren kann der Controller verteilt werden. Die Synchronisation zwischen den Controllern wird herkömmlicherweise mithilfe von verteilten Key-Value Stores realisiert, die nur eine feste Konsistenzeigenschaft anbieten, was die heterogenen Konsistenzansprüche der Daten im Controllerzustand nicht berücksichtigt. Das Virtual Synchrony Modell, ein alternativer Ansatz zu der üblichen State-Machine Replication Methode, bietet eine flexiblere Lösung die zu höherer Leistung führen kann, wenn bestimmte Annahmen über die Daten im Controllerzustand gemacht werden können.

Diese Arbeit stellt einen verteilten Controller basierend auf OpenDaylight, einem aktuellen SDN Controller und ISIS^2, einer Bibliothek die das Virtual Synchrony Modell umsetzt, vor. Die modulare Architektur des vorgestellten Controllers und die Verwendung eines plattformunabhänigen Datenmodells erlauben es, das System zu erweitern oder Komponenten zu ersetzen. Die Implementierung des verteilten Controllers wird beschrieben und die Komponenten und Gesamtleistung wird durch Benchmark-Tests ausgewertet.

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

Today's computer networks struggle with the ever-increasing demand for dynamic and flexible use-cases. This is caused by the fact that networks historically were seen in a more static context. Today data-centers with thousands of virtual hosts need to reconfigure constantly and modern backbone networks must adapt to ever-changing package flows. Software-defined networking (SDN) is a popular approach to overcome the limitations of conventional networks.

SDN allows network administrators to change the behavior of the network more easily and with greater flexibility. It achieves this by separating the actual packet forwarding, which is based on a set of rules, from the creation of these rules. These tasks are closely coupled in traditional networks and the possibility to change how those rules are determined is limited. The forwarding is done in the so called data-plane, while the rules are decided upon in the control-plane. The job of the data-plane can still be performed by the highly optimized hardware used today, so the performance of SDN networks is in principle not worse than in traditional networks. The control-plane however is realized by a program called controller, running on a conventional server. The controller has a complete view of the network in contrast to the limited local view of conventional network elements. It offers applications, that realize network functions (e.g. routing) and that are build on top of the controller, an abstraction of the network. By working with a global view and a higher-level abstraction the development of applications generating forwarding rule sets for the data-plane is greatly simplified.

But concentrating the control of a whole network in a single instance leads to high risks of failure and in consequence the controller is distributed. Several distributed controllers have been proposed (see chapter 4), using different distribution patterns and synchronization techniques. However all of those solutions rely on a single consistency model, that is often very conservative. The data, controller instances need to keep synchronized, however is heterogeneous regarding its consistency requirements.

The virtual synchrony (VS) model offers a set of different consistency levels, allowing to respect strict consistency when needed, while also allowing to use faster methods for data items that have lower consistency requirements. ISIS^2 is a library that implements VS.

In this thesis a distributed SDN controller based on a state-of-the-art controller and the ISIS^2 library is proposed, implemented and evaluated.

1.1 Thesis Organisation

The remaining parts of the thesis are organized as follows:

Chapter 2 provides background information about SDN, Key-Value stores, their scientific background, virtual synchrony, ISIS^2 and their relation respectively.

Chapter 3 outlines the motivation behind this thesis.

Chapter 4 gives an overview of related work on controller distribution.

Chapter 5 describes the proposed architecture of the distributed controller as well as the system and data models used.

Chapter 6 presents the implementation, used software and gives rationales about design decisions.

Chapter 7 contains the evaluation of the proposed system. Micro benchmarks of system components and the datastore are presented, as well as an evaluation of the whole system using different realistic scenarios.

Chapter 8 gives a conclusion about the presented work and lists possible starting points for future work.

2 Background

In this chapter a brief description of Software-defined Networking and a overview over key-value stores, their inner workings and scientific background is given.

2.1 Software-defined Networking (SDN)

Software-defined Networking [1] is a recent approach to ease network administration and accelerate network research. Originally started as research project at Stanford University, that eventually resulted in OpenFlow (see 2.1.1), SDN has gained momentum since and is considered a key enabler for advancements in data center networking, driven by the need for a data center network that is able to adopt to the ever changing requirements that are inherent in the rise of cloud computing.

Today's network nodes are highly specialized systems. The packet forwarding is performed by highly optimized hardware, minimizing the delay and maximizing the throughput of switches and routers. Beside the forwarding, these systems also run multiple protocols to determine the forwarding rules while respecting routing constraints, react to topology changes and to support additional functionality, all closely coupled with the forwarding. While this solution has worked well so far, the increasing demand to support features like quality of service guarantees, host mobility or isolation of traffic was answered by packing more and more ad hoc protocols into the network elements, increasing the complexity of those systems. Often those additional features are implemented in a vendor specific and proprietary way, leading to vendor lock in.

SDN promises to overcome this by providing a clean way to add new functionality to networks, while being vendor neutral and preserving the benefits of optimized forwarding hardware. The key principle behind SDN is the separation of the **data-plane** and the **control-plane**. The data-plane is responsible for forwarding packets according to a set of forwarding rules, while the control-plane is responsible to determine these rules. The data-plane consists of the network nodes performing their core task of packet-forwarding according to a set of rules they received from the control-plane. The control-plane and thus the decision making is moved from the networking nodes to a separate entity, called controller. The controller is a software system, hence the name software-defined networking, connected to all data-plane elements in the network. It communicates with the data-plane over the **southbound** interface of the controller, using a dedicated SDN protocol (OpenFlow, see 2.1.1, being the most prominent).

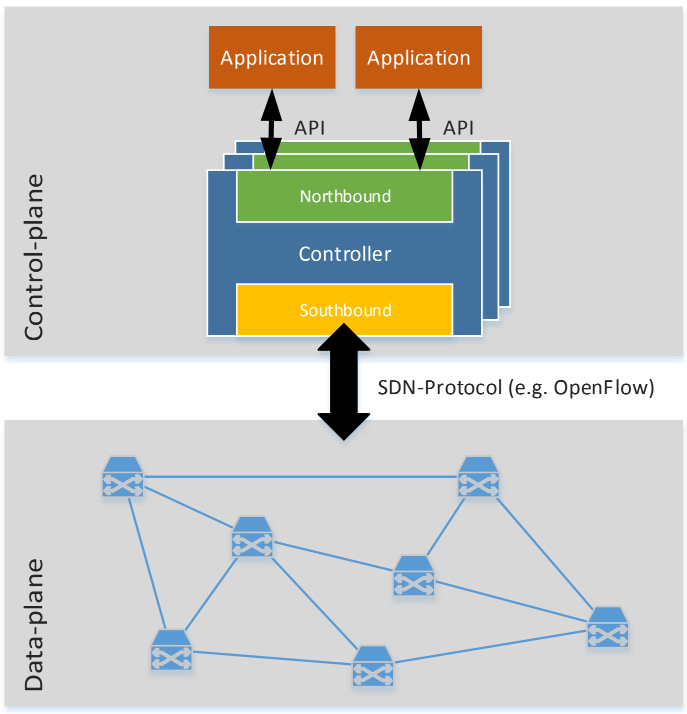


Figure 2.1: SDN architecture. The data-plane contains the network elements performing the packet-forwarding. In the control-plane the controller(s) communicate with the data-plane elements using a SDN-protocol and offer applications realizing network functionality an API.

The controller has a global view of the network, receives network events from the data-plane and offers an unified abstraction of the network to applications build on top of the controller API, called the **northbound** interface. These applications will implement network functions like routing or load balancing. The implementation of such functionality is eased by using the high-level abstraction the controller API offers and by the global view.

The typical architecture of a SDN environment is shown in figure 2.1. In the figure it is suggested that there are multiple controllers. This is often the case as moving the control of the whole network into a single instance is a great risk as it employs a single point of failure. To overcome this, multiple controllers are used, acting as a single logical controller while being physically distributed.

2.1.1 OpenFlow

OpenFlow [2] is the most prominent protocol that enables SDN and is maintained by the OpenNetworkFoundation. OpenFlow specifies the communication between the data-plane elements and the controller southbound interface. Network nodes that support OpenFlow typically connect to the controller over a separate control network. The switches or routers in the data-plane offer an abstraction called flow table. This table contains a pair of filters and actions. Incoming packets are checked against the filters and if they match a filter the actions defined for that table entry are performed. The filters can, among others, contain switch ports, MAC and IP addresses or TCP/UDP ports. The actions range from dropping the packet over sending it over multiple ports to modifying the packet header. If a packet does not match any filters, it is forwarded to the controller, that can then check if new rules must the installed and can inject the (modified) packet at any node in the data-plane network. The controller further can request statistics from the network elements to obtain load information.

2.2 Key-Value Stores, Synchronization Mechanisms and ISIS^2

As mentioned SDN controllers are distributed in order to increase fault tolerance by eliminating a single point of failure. To ensure the distributed controllers are in a consistent state, synchronization between the controllers is needed. A popular way to achieve this is to keep the state information of the controllers in a data-structure that ensures all controllers have the same state and state-transitions are performed in a consistent way. A Key-value store is such a distributed data-structure and will be used by in the proposed system. To understand how the key-value store of the proposed system differs from conventional ones, the different mechanisms used to keep the state consistent between the peers are discussed and ISIS^2 is presented, the library used to implement the key-value store of the proposed system.

2.2.1 Key-Value Stores

Traditionally RDBMS are the tool to use when storing data, as they provide a consistent, flexible way to store or query data and to aggregate information from the data, all backed by a mathematically well defined model. In order to make applications relying on such databases fault-tolerant, databases are replicated. By this distribution, they became subject to the CAP-theorem [3] which states that out of consistency (the replicas have the same data), availability (requests to the database are answered) and partition tolerance (the database can continue to operate if messages are lost or processes can't communicate) only two can be fulfilled simultaneously. RDBMS have strong consistency guarantees and as availability for most database applications, partition tolerance is neglected. The growing trend to scale-out systems and the recent cloud computing development raised the risk for partitions and such an event may render the whole system inoperable. As a result so called no-SQL databases experienced growing popularity. They relax the consistency guarantees in order to be more partition tolerant and to increase the availability. These databases also drop the verbose relational model to provide more flexible or specialized data models.

One kind of no-SQL databases are so called key-value stores. Key-value stores have an interface similar to the map data-structure (also called dictionary). Operations performed against a key-value store also behave like the map data-structre, the synchronization between the replicas is performed transparently[1]. By simplifying the complexity of the data model compared to SQL databases, it is also easier to distribute the data and keep it consistent. Examples of popular key-value stores are memcached [4], etcd [5], redis [6], hazelcast [7] or infinispan [8].

2.2.2 Sate Machine Replication

Distributed fault tolerant systems like key-value stores can be realized/described in an abstract way as state machine replication [9]. The instances are viewed as (identical) deterministic state machines, all initially in the same start state (e.g. empty). Client requests (e.g. put or get) are viewed as inputs to the state machines that cause state transitions and an output to be generated by every instance. Since all (non-faulty) instances will arrive at the same state and produce the same output if given the same sequence of inputs, the ordering of inputs is the critical step in order to successfully build a consistent distributed system. One way to solve the ordering problem is to derive a causal ordering of the inputs, as described in [10]. Another way is to have the instances to agree on the order using consensus protocols which are described in 2.2.3.

In order to achieve fault tolerance, the outputs produced by the instances, after an input is processed, will be checked to see if the majority of instances returned the same value. Instances that did return a output different to the one returned by the majority, or did not respond in time, must be considered faulty. In order to tolerate f failures, at least $2f + 1$ instances are required [11]. This ensures that at least one of the non-faulty processes will have to participate in the next request handling and will ensure no inconsistent value will be agreed upon.

This simple model is extended by implementations to support changes in the set of processes.

2.2.3 Consensus Protocols

Reaching consensus, which means having the instances of a distributed system, typically called processes, to agree on a value, is a fundamental problem of distributed computing. There

[1]Some key-value stores are not transparent as conflicts are exposed when performing either writes or reads and the user has to resolve them

are several ways to solve this problem. The solutions differ in their assumptions about the environment and resistance to different failure models. Generally a consensus protocol is considered correct if and only if the following properties hold true [11][12]:

Agreement All processes that participate in a consensus protocol agree on the same value (may be weakened to majority of instances)

Validity The value that is agreed upon must have been proposed by one of the processes

Termination Every (non failed) process will eventually decide on a value

The protocols distinguish between different roles a participant can take. However a process might act as multiple roles simultaneously.

The protocols described will, if at all, tolerate the fail-stop fault semantic. This means a process that fails will simply stop operating and not send any messages after it crashed. Another model is the byzantine fault model, where failed processes can still send messages with arbitrary values. Tolerating byzantine faults is way harder and requires further actions not discussed here.

Two-phase Commit

The simplest protocol that allows instances of a distributed system to achieve consensus is called two-phase commit (2PC) [13, Chapter 7]. As the name suggests it consists of two phases: In the first phase a coordinator, the process that starts the protocol, proposes a value to all other processes and waits for their responses. If all responses are positive, the coordinator contacts everyone again and informs them that consensus was reached. The processes will then consider the proposed value final (and commit it or pass it to a state machine) and the protocol terminates. If no consensus has been reached, the coordinator also contacts everyone but tells them that no consensus was reached and the participating processes will take no further action.

The obvious problem with this solution is that even if only a single node fails, the protocol will either lead to inconsistencies or will not make progress and thus violating the correctness properties discussed for consensus protocols.

Three-phase Commit

The three-phase commit protocol [14] solves some of the shortcomings of the 2PC protocol. An additional phase is placed between the two phases of the 2PC protocol, the so called prepare phase. So after the first phase is completed successfully, the coordinator will first send prepare messages to the participating processes and then waits for confirmation that everyone received a prepare message before starting phase three, which is just like the second phase of the 2PC protocol.

By adding the prepare message, every instance will know the outcome of the first phase before actually committing. This allows to successfully terminate the protocol if the coordinator or any participant crashes. Another instance might take over the role after a timeout and ask the other instances what state they are in. If a node responds that it has committed, the backup coordinator can assume that all processes have confirmed to be at least in the prepared state, as the old coordinator would not have send a commit message to one of the processes and can therefore tell the other processes to commit as well. With the same argumentation a crash of a process after it committed can be tolerated.

However this protocol can still fail in presence of network partitions or faults other than the fail-stop model.

Paxos

Paxos, originally described in [15] is the most prominent solution for the consensus problem. It is however generally considered hard to understand and to implement. An attempt to make it more understandable can be found in [16]. Paxos remains correct even if network partitions occur and is resilient to the characteristics of asynchronous networks (e.g. packet loss or delay), contrary to the 2PC and 3PC protocols. Paxos can make progress as long as less or equal than f instances out of $2f + 1$ instances fail.

Paxos distinguishes the roles of **proposers**, **acceptors** and **learners**. A proposer can create a `proposal` with a number N. N is a sequence number, a pair of (n, id), where n is a natural number the proposing instance monotonically increases for each proposal and id is a unique identifier of the instance. This sequence number allows for a total ordering of the proposals. The proposal is send to the acceptors.

Acceptors respond to proposals either with a `promise` or with a `reject` message. It decides which action to take by comparing the proposal number of the received proposal with the highest proposal number it has received N_{pmax}. If the proposal number of the received proposal (N) is higher than N_{pmax}, the acceptor will set N_{pmax} to N and send a promise back to the proposer to never accept any proposal with a number lower than N. The promise will also include the proposal with the highest number that was accepted (if present). If N is less than N_{pmax}, a reject message is sent back including N_{pmax}.

If the proposer receives a promise from any majority, it will choose the value to propose. To do so it will choose either any value if none of the promises included a proposal, or the value of the proposal with the highest number from the received promises. An `accept` message is send to the acceptors with the proposal N_a and value V_a. If no majority of promises was received, the proposal failed and the proposer may start again.

If an acceptor receives an accept message A, it will accept the proposed value V_A if N_A is greater than N_{pmax}. If a proposal was accepted by an acceptor, it will store it and send an `accepted` message including the proposal number and value to all learners and the proposer.

If a learner or proposer receives accepted messages for the same proposal from a majority of acceptors, it can consider the value final and terminate.

Usually there is only one proposer, but a situation with more than one might arise when a proposer crashes, a new one steps in and the first proposer recovers later. Multiple proposers can lead to a situation where they outbid each other with higher proposal numbers before a proposal is accepted by a majority. In this situation the protocol will not terminate until one of the proposers is finally successful or one of the proposers steps down. The acceptors have to keep N_{pmax} and the accepted proposal on disk so that after a crash they can recover without causing paxos to yield incorrect results. The writing to disk has a significant impact on performance.

The text above describes the basic single-paxos variant of the protocol. In typical use-cases more than one value must be agreed upon which leads to the multi-paxos protocol that supports multiple values to be agreed upon consecutively. In [17] the challenges of implementing paxos for production use are described.

Viewstamped Replication

Viewstamped replication (VR), originally described in [18] and later revised and extended in [19] was developed approximately in the same time as paxos, with both inventors having no knowledge of each other. While solving the same problem and having the same constraints, the protocols differ in the way they reach consensus. [20] provides a detailed comparison of Paxos and Viewstamped replication. VR realizes log-replication rather than simply agreeing on a single value. It uses three protocols to solve request processing, primary selection and recovery of replicas after they failed.

VR defines three roles: a **client**, a **replica** and the **primary**. Clients only send requests to the primary which is also a replica. Replicas have state that includes information about the other replicas, its own identity, the view-number, the status it is currently in (either normal, view-change or recovering), the last operation number received, a log containing all operations, the commit-number which is the request number of the operation most recently committed and a client table keeping the request number, and result of the request if present for each client. The identity of the primary can be derived from the view-number.

The normal protocol is triggered by a client sending a `request` to the primary. The request contains the id of the client, a request number and the operation to perform. The primary checks if the request number is higher than the one stored in its client-table. If the request number matches the number stored in the client table, the `result` is sent again. If the request number is higher, the primary increases the operation number, updates the client table and sends a `prepare` message to the other replicas containing the original client request, the operation number, the view number and the commit number. When a replica receives a prepare message, it will only process it if its log contains entries for all prepare messages with a lower operation number and if its current state is normal. A replica might request missing entries from other replicas using state transfer. If the log is up to date it will perform the same steps the primary did except sending out prepare messages, but will send a `prepareOK` message back to the primary. When the primary received a majority of prepareOK messages, the operation is considered final (committed), the primary will increase the commit number to the operation number, will send a response to the client and add the response to the client table. Replicas will commit operations when they receive new prepare messages with a commit number higher than their own or with a separate `commit` message if no new client request is received in time. The replicas will only commit if their log is complete up to the operation they learned to be committed by the primary and all previous operations have been committed. After committing, the replica will perform the same actions the primary performed, except sending a response to the client.

If the primary fails, the replicas will notice this by using timeouts and trigger a view change. Once a replica detects the failure of the primary, it will advance its view number, change its status and send `startViewChange` messages to the other replicas containing the new view number and the id of the sending replica. When a replica receives a startViewChange message it will send a `doViewChange` message to the replica that should be the primary based on the view number received. This message includes the new view number, its own log, the last view number where the replica had a normal state, its operation number and the commit number. When the replica that is determined to be the new primary received a majority of doViewChange messages, it will update its view-number, replace its log, commit number and operation number with the most up-to-date received, change its state back to normal and send `startView` messages containing the new log, operation number, view number and commit number to the other replicas. They will then update their local information to the received state and commit outstanding operations, which the primary will as well. The replicas will send prepareOK messages to the primary for the not committed operations in the log.

A crashed or replaced replica can participate in the normal operation after it went through a recovery phase. To perform the recovery, the replica will set its state to recovering, send `recovery` messages to all replicas including its id and a unique identifier. Other replicas will only answer if their state is normal. The answer is a `recoveryResponse` message including the view number and the identifier from the recovery message. If the primary receives a recovery message, it will further include its log and the commit and operation number in the recoveryResponse message. The recovering node waits to receive a majority of recoveryResponse messages. Only if the primary of the newest view it learned from has send a response it will update its local values to the data from the primary and change its state to normal. Otherwise the recovery attempt failed and the recovering node can try again with a different unique identifier.

A drawback of VR is that messages can become fairly large as the whole log is transmitted in a number of message types. This can be overcome by either using techniques to shorten the log or by keeping the committed log on disk. The protocol can be further optimized to support local reads at the primary or, if staleness of data doesn't violate client constraints, at the replicas. In [19] a way to support dynamic changes of the cluster is given.

Raft

Raft [21] is a consensus protocol that was developed as an alternative to Paxos that is easier to understand (according to a survey carried out by the authors) and use, while having comparable performance and proven correctness. Raft shares many concepts with VR, but concentrates more functionality in the **leader**, one of the three roles along **follower** and **candidate**. Normal request handling is similar to VR: Only the leader receives requests and tells the followers which updates they should append to their replicated log. Once the majority of the followers have confirmed that they have received the update, the leader considers the value committed and tells the followers to commit the update as well. The difference mostly lies in the way the protocols handle leadership change in case of leader/primary failure: Raft specifies that a follower will become a candidate if it does not receive a message (heartbeat or update message) within a certain timeout. The candidate will send vote requests to the other nodes and, if it has the majority of votes, will become the new leader. However not every node can become leader, nodes will only vote for a candidate if the candidates log is at least as up to date as the voters. This is done by comparing the id and term, a concept similar to view in VR, of the last entry of the log, both numbers are included in the vote requests. If no candidate received the majority of the votes, the election process is repeated until a leader is found. The voter restriction ensures that no information is lost during leader change, in VR this is done by transferring the logs to the designated primary, which will then choose the most recent.

Raft supports membership changes and, like paxos and VR, is resilient to network partitions. However Raft assumes that the nodes statically know each other node in the cluster and membership changes require manual intervention.

2.2.4 Virtual Synchrony

Virtual synchrony (VS) [22] is an alternative model to state machine replication. The key observation behind virtual synchrony is that applications often don't require the strict semantics of state machine replication. In this case, messages can be delivered in less restrictive orders, allowing higher performance, and yet the replicated process sees the same synchronous series of events, therefore the name virtual synchrony. VS defines an abstraction called process group. A process group consists of processes that have mutual state which can be modified by every member by sending messages to the group members. Processes may join or leave a process group at any time. Joining processes will receive the current state via state transfer. The process group provides its members with consistent information about group membership. Events in VS, be it messages or membership changes (joins, leaves, crashes or recoveries) are delivered in the same order to processes, respecting the observation described above. VS does not require to run a consensus protocol to perform an operation, the sender will retry to deliver the message until a timeout is triggered, which will then trigger a membership change. If the failed note did not crash but was unable to communicate with the group because of a network partition and tries to rejoin the group, it will be forced to restart first. This allows to perform read operations against the local copy which is a huge performance gain. The difficulty, that comes with the advantages VS offers, is that the developer building an application employing the VS model must choose a messaging primitive that satisfies the requirements of the application, as failure to do so can lead to inconsistencies in the mutual state.

2.2.5 ISIS^2

ISIS^2 [23] is a C# library written by Ken Birman, the main contributor to virtual synchrony. It offers a broad feature set aimed to ease the development of distributed applications for cloud computing. It implements the concepts of virtual synchrony: Process groups can exchange messages using primitives with different semantics. ISIS^2 includes the following send methods:

SafeSend equates Paxos, with optional disk logging

OrderedSend offers a total ordering of the messages but no durability guarantees

CausalSend corresponds to causal ordering as defined by [10]

Send reliable messages, FIFO ordered per sender

RawSend unreliable datagram messages

As mentioned in the description of VS it is in the developers responsibility to choose the right send primitive for the particular use-case. In the use-case of a key-value store for a distributed SDN controller, it is important to consider the implications of the SDN model. For example a SDN switch is only connected to a single controller at a time. Therefore this controller will be the only one to receive updates from this switch and can thus be considered the owner of data items concerning this switch. When an owner can be defined for an item, meaning updates will only come from this process, the **send** send primitive can be used, as FIFO-ordering per sender is sufficient to achieve consistent update ordering as there will only be one sender. This observation holds true for normal operation, but since switches will change the controller they are connected to in case of controller failure, a more strict send-primitive has to be used to keep the mutual state consistent if a controller fails. For data items like links that concern multiple switches that might be connected to different controllers, total ordering is needed since updates can come from multiple controllers. Apart from the data properties, performance should also be considered when choosing the send primitive. In [24] as well as in chapter 7, a performance comparison of the send primitives is given.

In addition the these communication primitives, ISIS^2 supports state transfer between nodes by employing snapshotting. This allows for dynamic group membership changes, as joining nodes will be requesting a snapshot of the state from the group members, which will take a snapshot and transfer it to the joining node. ISIS^2 also offers a Distributed Hash Table (DTH) which supports sharding. Sharding means that only a part of the complete state is kept on every replica, reducing the size of the state every replica has to keep but requires remote reads. Further a distributed locking API is included.

3 Motivation

One of the main advantages of software-defined networking is the global view provided to the control applications by the controller. Many networking problems are easier to solve when the complete network situation is provided, compared to operating on the limited local view of a node. For example, the routing can be based on Dijkstra's shortest path algorithm instead of using link-state or distance-vector techniques as in a fully distributed approach. Also being able to program against a high level abstraction and in a higher level programming language compared to writing low level firmware further eases the development of applications implementing network functionality and allows for more sophisticated control logic.

While the advantages of the global view and the centralized model are clear, concentrating all critical network control logic in a central instance leads to high risks of failure. Scalability issues might arise, considering the growing size of today's networks. So by making it easier to solve a series of problems, SDN leads to another problem: It employs a single point of failure for the whole network. It further depicts a possible performance bottleneck considering the growing scale of networks. However, both, scalability and fault tolerance, are problems that can be solved by distributing the controller. This distribution contradicts the centralized aspect imposed by SDN, but by having the controller instances operate on the same state, the control plane remains logically centralized but is physically distributed.

Sharing state between instances in a distributed system is far from trivial and an active research area. The famous CAP-Theorem forces systems trying to share state between instances to make trade-offs between properties (data consistency, availability and partition tolerance). Most available tools for state synchronization only offer a single type of synchronization. Distributed controllers that have been built (see chapter 4) on top of these tools have inherited the fixed semantics for state synchronization offered by those solutions. However, the data items a controller instance holds differ in their requirement for consistency, constituting a degree of freedom and leaving room for optimization. For example, it is important to keep the topology information consistent, but typically the load statistics of switches are considered less important and thus can be kept less consistent.

The ISIS^2 library is an exception from the tools providing synchronization functionality, as it doesn't only offer a single semantic. It offers a wide range of consistency levels to systems built on top of it by employing the virtual synchrony model with multiple send primitives discussed in 2.2.4. Virtual synchrony allows to exploit the fact that switches are only connected to one 'master' controller, so updates to the data concerning this switch will only be done by a single controller. This allows to use a less strict ordering of update messages for this data, As only FIFO ordering for a single sender is needed, and not a global ordering.

The contribution of this thesis is the implementation of a distributed controller built on top of ISIS[2] and OpenDaylight, a widely used, state-of-the-art SDN controller. The proposed controller provides the possibility to use different consistency levels for the different kinds of data kept in controller state. Further a unified middleware layer is provided, supporting the extensibility of the system. The proposed system is evaluated in both, micro and macro benchmarks.

4 Related Work

The need to distribute the control-plane has been recognized in literature and several proposes were made:

Hyperflow [25] was among the first proposals for a distributed controller. A Hyperflow instance has a global view on the network, but only directly controls a local subset of network nodes. The instances communicate over a publish/subscribe system build on top of WheelFS [26], a distributed file system.

Kandoo [27] proposes a hierarchical distribution pattern that provides scaleability rather than fault-tolerance or consistency. Kandoo seperates controllers into two tiers: The bottom tier and the root tier. The bottom tier controllers have a limited number of switches connected to them and are not interconnected, but report their view to the root controller. The main assumption the authors of Kandoo make, is that most of the events generated by the network can be handled by the bottom controllers with their limited local view, shielding the root controller, that has a global view, from the majority of events as bottom controllers only forward events they can't handle to the root controller.

Onix [28] offers applications built on top of it a view on the network they call Network information base (NIB). Applications can add data to the NIB and choose between two datastores: a transactional, persistent database with a SQL-frontend that is consistent but has low performance and an eventual consistent Distributed-Hash-Map with higher performance but no consistency guarantees.

The authors of [29] argue that consistency is a desirable property of controller state, as shown in [30] and that the low performance of the onix datastore is not a general property of such systems, but rather implementation specific for the onix datastore. A prototype based on BFT-SMaRt [31], a byzantine fault-tolerant state machine replication library, is presented and evaluated.

In [32] ElastiCon is proposed, a system to dynamically distribute the load between controllers. Controller instances report load measurements and based on the reported load, new instances are started, switches moved between controllers or controllers are stopped. To preserve a consistent view between the instances, a distributed data store is used, implemented on top of Hazelcast.

All of these Systems provide a single consistency semantic, except Onix. Onix however only offers a choice between two extremes, forcing the use of the slow transactional datastore if the

DHT is no feasible choice. As shown in [29], it it possible to build a consistent datastore for SDN controllers with sufficient performance. A controller using the ISIS^2 library and exploiting the range of consistency semantics offered should be able to provide the same performance, if not better.

5 Distributed Controller Datastore

In this chapter the architecture and system model of the proposed distributed controller datastore are described.

5.1 System Model

The System consists of two modules: The controller and its datastore. Together they form one controller instance. Multiple controller instances form a controller cluster. The controller cluster controls multiple data-plane switches. Each switch can be connected to multiple controller instances, but only one of those connections is active (the master connection) at any given time. The other connections act as fallback if the master controller fails. Controller instances store their state in their local datastore and use the information in the datastore to react on network events. The datastores of the instances in the cluster will be synchronized using the ISIS^2 library. As discussed earlier, the controllers can perform read operations against their local datastore, ISIS^2 is only used to distribute the write operations between the instances. The architecture is illustrated in Figure 5.1.

The controller instances operate with arbitrary processing speeds and communicate over a (separate, non data-plane) network that possibly duplicates, reorders or drops messages. The message delay has no upper bound and the network might experience partitions. The cluster membership is not static and controllers can join or leave at any time. Controller instances are assumed to fail by crashing. A controller instance has failed if any of the modules (controller or datastore) failed.

5.1.1 Scope Limitations

The controller, used as basis of the system, OpenDaylight, is a large (above 300,000 lines of code[1]) and complex system. This thesis is practically a feasibility study and as it was hard to predict what problems might arise, the scope was limited on selected parts of the controller, namely (with datastores of the module):

[1]measured with cloc [33]

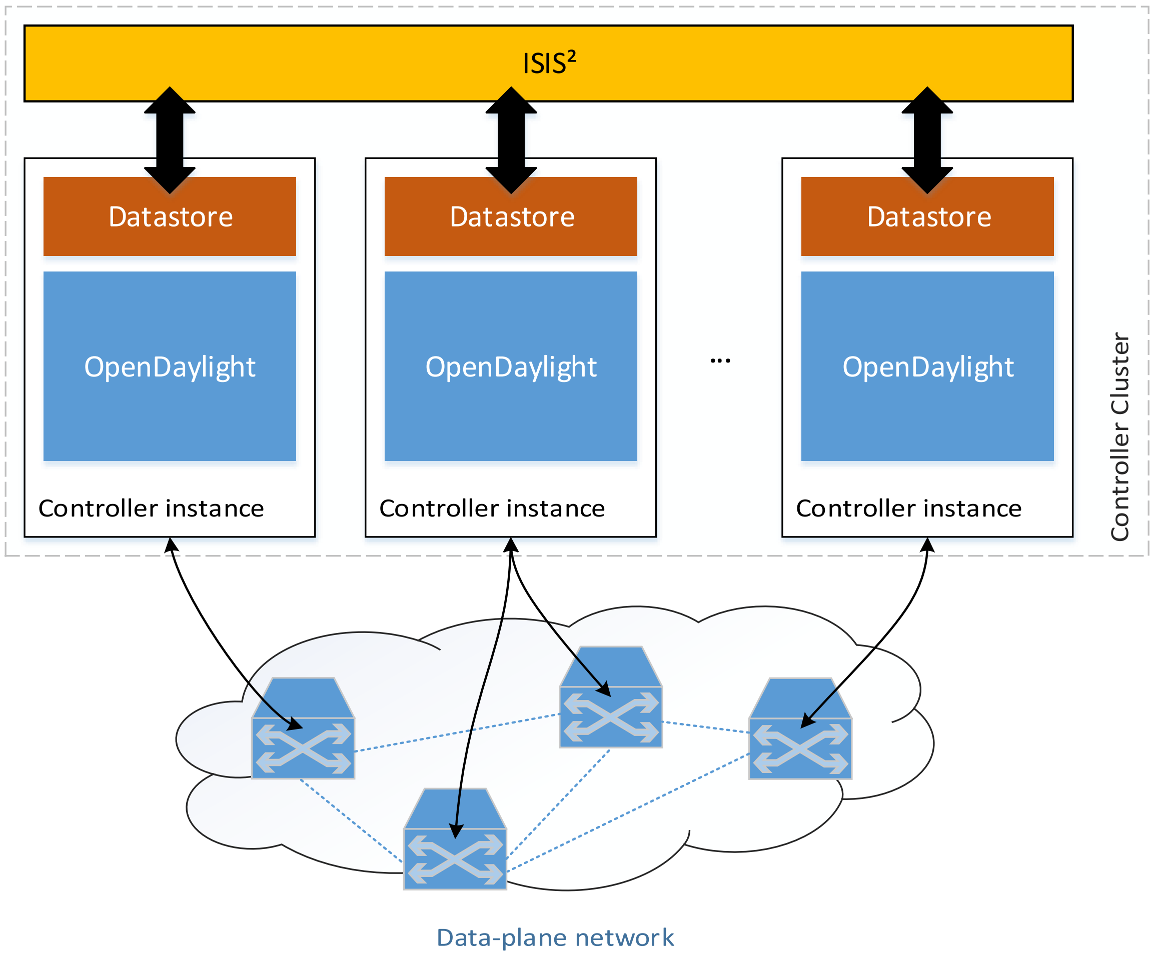


Figure 5.1: System model. The Data-plane nodes are connected to a controller instance. The controllers operate on the same state, using a local copy, synchronized by ISIS^2. Fallback connections from network nodes to their backup controllers are not shown.

TopologyManager edgesDB, nodeConnectorsDB, hostsDB

SwitchManager nodeProps, nodeConnectorProps, nodeConnectorNames

HostTracker activeHosts, inactiveHosts

These modules together provide a complete view of the dataplane network and are used heavily by the *simpleforwaring* module of OpenDaylight, which provides reactive routing functionality by listening to ARP messages and when a new host is discovered, installing forwarding rules pointing to the host on all network nodes. As these functions are essential to a controller, evaluation of these parts should provide enough information to decide whether ISIS^2 is a feasible building block for a distributed controller datastore. Since other modules could easily be integrated into the existing system, complete coverage of all controller modules would be doable with reasonable effort (see 6.4).

5.2 Abstractions

The datastore provides access to multiple predefined statically typed key-value stores with a simple Java map compatible interface. Every controller instance can issue operations on the key-value stores. The read operations are, as already mentioned, performed against the local copy. The write operations are synchronized by ISIS^2. Operations may have parameters and a return value. They are invoked in a synchronous manner, so the calling thread will block until the operation is completed. The datastore can handle concurrent calls but will process them serially.

Another feature of the datastore is that instances will be notified of remote changes. This functionality is required by OpenDaylight and called update aware. In order to make an entity of the controller update aware for a certain key-value store, the entity must implement an interface and register themselves with the datastore.

Each store ensures that "write-before-read" semantics are respected which means that a read operation following a write operation by the same controller will respect the write operation.

The supported operations are listed below, as well as the update aware functionality.

5.2.1 Operations

The datastore supports the following operations on the key-value stores. Keytype and valuetype are placeholders for the actual types used in the individual key-value store. Table 5.1 contains the key-value stores and their data types.

LIST `(void) → list<pair<keytype,valuetype>>`
    The list operation returns a list of all key-value pairs stored in the datastore.

SIZE `(void)` → `int`
: Returns the number of entries(key-value pairs) in the datastore.

GET `(keytype KEY)` → `valuetype`
: Returns the value stored under the given key. If no key was found, `null` is returned.

PUT `(keytype KEY, valuetype VALUE)` → `valuetype`
: Stores the given value under the given key. If there was already a value stored for this key, it is returned, otherwise null is returned.

REMOVE `(keytype KEY)` → `valuetype`
: Removes the given key and the value stored for it from the datastore. If the key was in the datastore, the value is returned.

CONTAINS `(keytype KEY)` → `boolean`
: Returns a boolean indicating if the given key is in the datastore.

CLEAR `(void)` → `void`
: removes all entries from the datastore.

`LIST`, `SIZE`, `GET` and `CONTAINS` are read operations, `PUT`, `REMOVE` and `CLEAR` are write operations.

5.2.2 Update Aware

Entities that use a datastore can request to be notified if the contents of a datastore change. The interested entities must provide the following interface:

ENTRY_CREATED `(keytype KEY, boolean LOCAL)`

ENTRY_UPDATED `(keytype KEY, valuetype VALUE, boolean LOCAL)`

ENTRY_REMOVED `(keytype KEY, boolean LOCAL)`

the `LOCAL` flag is true if the change to the datastore was triggered by the local replica, false otherwise. When an `ADD/UPDATE/REMOVE` to a store occurs, the entity interested in changes will be notified over the corresponding method. The method will be called after the update was applied to the local datastore copy. If the update was triggered locally, the update aware method may be invoked before the call to the operation will return.

Listing 5.1 Excerpt from the data type definitions of the datastore using Protobuf. The Edge data type uses the NodeConnector data type which uses the node data type

```
...
message Node {
   required string type = 1;
   required string id = 2;
}
...
message NodeConnector {
   required string type = 1;
   required string id = 2;
   required Node node = 3;
}
...
message Edge {
   required NodeConnector head = 1;
   required NodeConnector tail = 2;
}
...
```

5.3 Data Model

Both main building blocks of the system require a strongly typed data model. Since Java and C# types are not compatible, a language neutral definition of the data is needed. Protocol Buffers (Protobuf) [34], a serialization mechanism developed by Google was used to define the data model. Protobuf data types are defined in an own language from which a compiler produces language specific code. More rationale behind the decision to use Protobuf is given in 6. Listing 5.1 shows how the edge data type is defined using previously defined data types. The data types are close representations of the data types used by OpenDaylight, which eases conversion between the two representations. Most key-value stores have lists as their valuetype (see table 5.1). Lists can be elegantly modeled using protobuf by using different field rules. The fields of data types can either be **required**, **optional** or **repeated**. **Required** fields must be set and cannot be null, **optional** fields may be null and **repeated** fields can contain zero to n values.

Component	Datastore	Key-Type	Value-Type
TopologyManager	edges	Edge	Property*
TopologyManager	nodeConnectors	NodeConnector	Property*
TopologyManager	hosts	NodeConnector	HostWithProperties*
SwitchManager	nodeProperties	Node	Property*
SwitchManager	nodeConnectorProperties	NodeConnector	Property*
SwitchManager	nodeConnectorNames	Node	NamedNodeConnector*
HostTracker	activeHosts	HostID	HostNodeConnector
HostTracker	inactiveHosts	NodeConnector	HostNodeConnector

Table 5.1: Data types used by the key-value stores. Types annotated with * are lists.

6 Implementation

The system is based on OpenDaylight [35], a state of the art SDN-controller and the ISIS^2 library. An overview over OpenDaylight is given in 6.1.1 and more information about ISIS^2 is provided in 2.2.5 and 6.1.4. The overall architecture of the system is pictured in Figure 6.1. A client-server model is used for the communication between the controller and its datastore. The datastore server is written in C# and uses ISIS^2 for synchronization with other instances. The client is integrated into the OpenDaylight platform, providing access to the datastore to controller modules.

Since OpenDaylight is a Java application and ISIS^2 is written in C#, there are some points to consider implementing a system based on both applications:

- Separation of concerns.
  As there are two distinct processes, one major design decision is about what functionality will live in what part of the system. Different features (e.g. sharding based on data locality) have been considered and as a result the communication protocol between the two subsystems is quite verbose and the local state copy is kept in the C# subsystem. This certainly isn't beneficial to performance but eases integration into ISIS^2.
- The target environment of the system is a linux server.
  To be able to run the C# module of the system, Mono [36] is used. Mono provides an open source implementation of the .NET runtime and a C# compiler. However using Mono comes with performance drawbacks compared to native execution on a Windows platform (see 7 and below).
- The communication between the Java runtime and the C# .net platform.
  In almost every deployment scenario both processes would run on the same machine, so an inter process communication (IPC) based communication channel would be preferable. In order to have a simple abstraction of IPC, ZMQ REQ/REP Sockets over the ZMQ-ICP transport was chosen. During evaluation stability issues arised on the C# side when using the IPC transport[1]. As ZMQ does not offer another communication channel for IPC and changing the system to use another IPC mechanism wasn't possible in a timely manner, the TCP transport of ZMQ was used as fallback. While there is a performance

[1]ZMQ internally uses UNIX domain sockets for IPC[37], technically not supported by Mono/C#. However C# implementations based on the C++ library still offer the IPC transport.

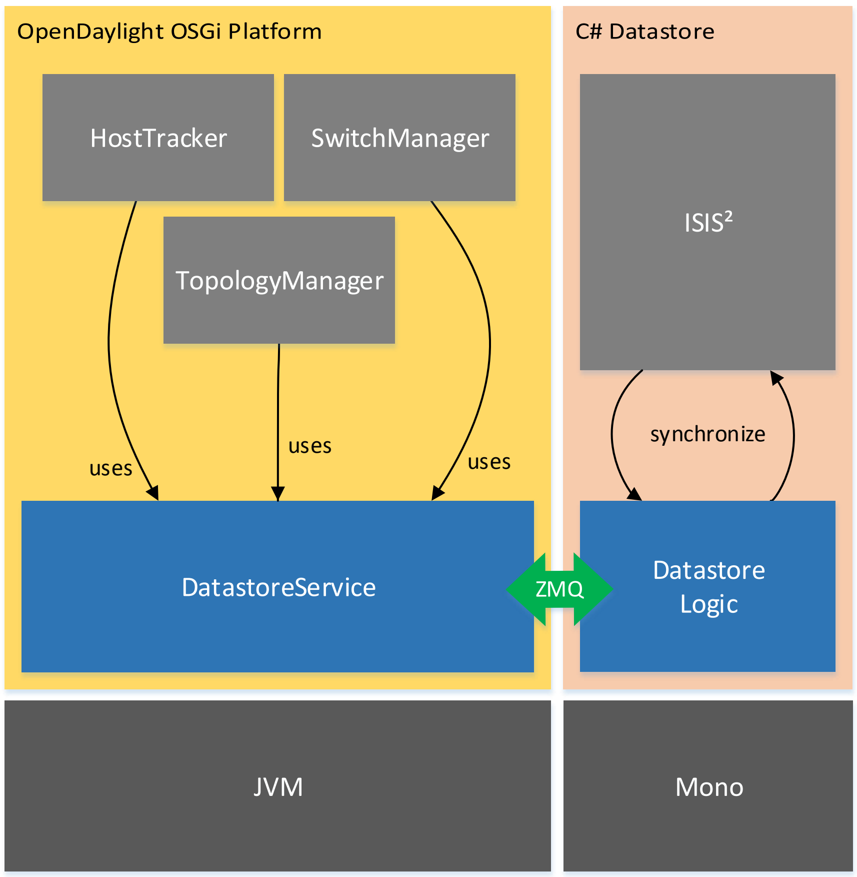


Figure 6.1: System overview

penalty, there are still enough capacities so this poses no bottleneck (see 7.2.1). ZMQ is discussed in more detail in 6.1.2.

- An efficient way to serialize data between Java and C# is needed.
  Google Protocol Buffers (protobuf) [34] was chosen as serialization solution as it is simple to use, offers good performance and has bindings available for almost all programming languages. Also ISIS^2 can be configured to use the protobuf-net library (a C# protobuf language binding) [38] for its internal serialization.

The rest of this chapter will give an overview of the software used to build the system, a more detailed description of the functioning of the modules and will discuss the model driven aspects and the extensibility of the system.

6.1 Used Software

6.1.1 OpenDaylight

OpenDaylight is written in Java and uses the OSGi-framework, which is a platform to write modular software systems. OpenDaylight consists of multiple bundles (modules), each offering specific services to other bundles and requiring others to function. The OSGi platform takes care of bundle dependencies during start up and allows to start, stop, restart or update bundles at runtime. OpenDaylight consists of different layers, each offering a different level of abstraction of the underlying network. The "lowest" layer, the southbound interfaces, are bundles that provide connectivity with certain network elements, with the most prominent being the OpenFlow bundle. The next layer is the service abstraction layer (SAL), which unifies the interfaces from/to the southbound plugins and provides a common data model for the upper modules. On top of the SAL there are the bundles that implement the "controller business logic" and provide basic functionality like a topology abstraction, switch statistics, host tracking and flow management. In the stock release of OpenDaylight 1.0, a bundle called simpleforwarding is included which uses these functions to provide rudimentary reactive routing functionality for the connected network. In addition to the possibility to write own bundles directly against the services of the "business logic" bundles, there is another layer called northbound, which offers a RESTful API to applications implementing additional network functionality. In Figure 6.2 an overview of the OpenDaylight controller is given. Version 1.0, called Hydrogen of OpenDaylight was used.

To use OpenDaylight as basis for the proposed system, a new bundle is added to the OSGi platform, containing a service that provides access to the distributed key-value stores. This bundle is discussed in detail in 6.2.1. In addition, some of the bundles were altered to use this service.

6.1.2 ZMQ

ZeroMQ (also called ØMQ or ZMQ) [40] is a high performance messaging library. It provides an abstraction based on classic sockets. These sockets can use one of the supported messaging patterns request-reply, publish/subscribe or pipelines and support different underlying transportation mechanisms. While the core library is written in C++, ZMQ is available for a wide range of languages, either by a library written completely in the targeted language or by providing a wrapper for the native library. For the Java side of the system the JZMQ implementation v3.1.0 was used, which is a wrapper around libzmq. On the C# side, several implementations were tested (Castle.ZMQ, NetMQ and clrzmq4). The only library that supported the IPC transport is Castle.ZMQ however, as mentioned earlier, turned out to be unstable and the NetMQ library version 3.3.0.11 is now used over the TCP transport.

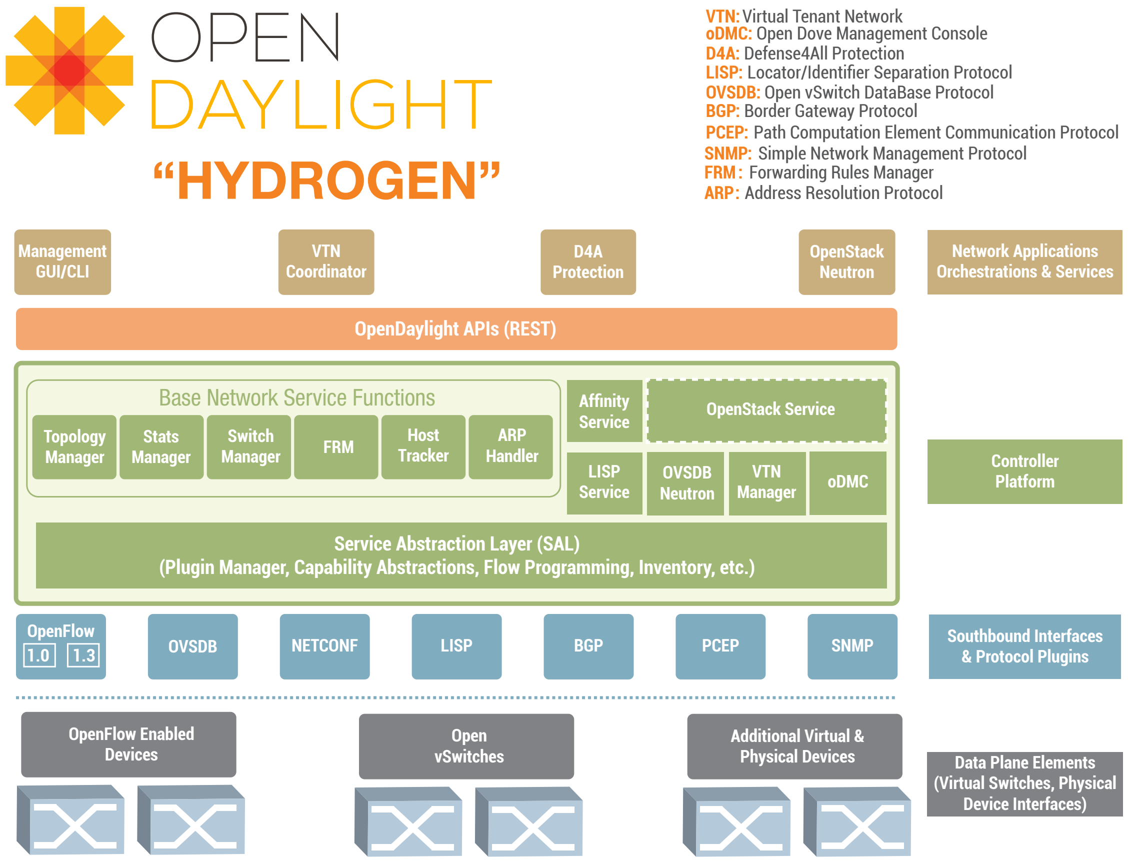

OPEN
DAYLIGHT
"HYDROGEN"
VTN: Virtual Tenant Network
oDMC: Open Dove Management Console
D4A: Defense4All Protection
LISP: Locator/Identifier Separation Protocol
OVSDB: Open vSwitch DataBase Protocol
BGP: Border Gateway Protocol
PCEP: Path Computation Element Communication Protocol
SNMP: Simple Network Management Protocol
FRM: Forwarding Rules Manager
ARP: Address Resolution Protocol
Management GUI/CLI
VTN Coordinator
D4A Protection
OpenStack Neutron
Network Applications Orchestrations & Services
OpenDaylight APIs (REST)
Base Network Service Functions
Topology Manager
Stats Manager
Switch Manager
FRM
Host Tracker
ARP Handler
Affinity Service
OpenStack Service
LISP Service
OVSDB Neutron
VTN Manager
oDMC
Controller Platform
Service Abstraction Layer (SAL)
(Plugin Manager, Capability Abstractions, Flow Programming, Inventory, etc.)
OpenFlow 1.0 1.3
OVSDB
NETCONF
LISP
BGP
PCEP
SNMP
Southbound Interfaces & Protocol Plugins
OpenFlow Enabled Devices
Open vSwitches
Additional Virtual & Physical Devices
Data Plane Elements (Virtual Switches, Physical Device Interfaces)

Figure 6.2: OpenDaylight 1.0 Hydrogen architectural overview [39]

As already mentioned, the system uses the ZMQ request-reply pattern. This pattern uses two socket types, the ZMQ-REQ socket and its counterpart, the ZMQ-REP socket. Together, they provide a synchronous, message based communication abstraction. A REQ-socket is only allowed to send a message if no previous messages have been sent or after it received a message. The REP-Socket may only send a message after it received one. So both sockets have two non-error states: send and receive. In the receive state a socket may only receive a message, but not send one. In the send state a socket may send a message, but not receive one. The difference between a a REQ and a REP socket is that the REQ socket initially is in the send state and the REP socket in the receive state. Messages are byte-arrays, that are converted into the language specific representation by the respective library.

6.1.3 Protocol Buffers (Protobuf)

As discussed earlier, Google's Protocol Buffers, or short Protobuf, is used for defining the data model. Protobuf is further used to define the communication messages the two subsystems exchange and RPCs are defined on top of those messages, also using Protobuf.

From the RPC definitions the Protobuf compiler generates a service class, containing the defined RPCs. This service class uses a RPCChannel that must be implemented seperatly, to send and receive the messages. The RPCChannel implementation is built on top of ZMQ. This RPC abstraction is only used on the Java side, as the C# side will simply wait for a request, handle it and send the answer back. On the Java side however, this abstraction greatly simplified the implementation of the datastore abstractions build on top of it.

On the Java side the default binding generation included in the stock-compiler version 2.6.1 was used and on the C# side the protobuf-net library version 2.0.0.668 was used. This library differs from the other implementations available for C#, as it provides a serialization functionality based on attributes, a functionality similar to annotations in Java. A class can be prepared for serialization by using attributes that protobuf-net uses for serialization of the class. Protobuf-net also includes a compiler that generates annotated classes from a .proto file. Alternative libraries are available, providing a model more similar to the Java library, but since ISIS^2 can be configured to use protobuf-net as serialization mechanism, protobuf-net was chosen.

6.1.4 ISIS^2

As the general characteristics of ISIS^2 are discussed in 2.2.5, the technical details will be shown here. Version 2.2.1962 of ISIS^2 was used. ISIS^2 is distributed in source code as a single .cs-file, with around 36,000 lines of code[2]. ISIS^2 offers an abstraction on group communication with different consistency semantics. The API of ISIS^2 is similar to a RPC system: On a group handle operations and their parameters are defined as methods (callbacks), that ISIS^2 will invoke when an instance issues a call to such an operation. Triggering these operations can be done with one of the different ordering/consistency semantics offered by ISIS^2.

ISIS^2 itself is highly asynchronous and uses multiple threads internally. When building a system against the ISIS^2 library, one must be aware that ISIS^2 will call the callback methods from its own threads, so any data structure that is accessed in these callbacks must be thread-safe or locking must be used.

[2]measured with cloc [33]

ISIS^2 has its own serialization functionality built-in, but this functionality is quite limited (e.g. empty lists can't be serialized). It is, as mentioned earlier, however possible to use the protobuf-net library to do the serialization.

The selected serialization method is used for the group communication as well as the state transfer functionality of ISIS^2. This allows instances to join already running clusters and to receive the current state from another instance and to apply it to its local data structures. The state transfer is done via snapshotting, and methods similar to the callbacks must be provided for taking and applying snapshots of the local state.

On top of this basic group communication ISIS^2 offers a distributed hash table (DHT), that supports sharding. It was checked if the DHT would be a valid choice for the implementation of the datastore, but the lack of a LIST operation and the fact that ISIS^2 only offers a single DHT per group would force the use of aggregate keys to distinct between the datastores and type conversions as the DHT is not typed. So the datastore was implemented on top of the group communication feature as it is more flexible and better fits the use-case.

ISIS^2 offers other advanced features such as out-of-band state transfer to speed up state transfer involving large state, a querying API similar to the send primitives that can be used to query state from other instances using C#s LINQ and Map-Reduce like processes can be realized with ISIS^2 as has aggregation mechanics built in.

6.2 Modules

6.2.1 DCDS OSGi Bundle

OpenDaylight already has a clustering feature built in that is based on infinispan. The clustering feature is wrapped in a bundle called clustering.services. This bundle exposes an interface that allows to create Key-Value stores, called Caches by infinispan. These caches implement the ConcurrentHashMap interface of the Java standard library. Every bundle that contains state that should be replicated over the cluster uses these caches, so replacing the clustering.service bundle with a implementation that would redirect calls to the ISIS^2 datastore would be an elegant solution. However the interface of the clustering.service is voluminous and interacts heavily with the OpenDaylight container functionality, which won't be used by the proposed system. Therefore, instead of replacing the clustering.service bundle, a new bundle called DistributedControllerDataStore (DCDS) is added to the OSGi platform and the bundles using the new service are modified to use this bundle instead of the built in cluster service. The modifications are minimal and boil down to importing the service and replacing the call to retrieve the map from the DCDS instead of retrieving it from the built in service. Listing 6.1 shows an example on how a bundle was modified.

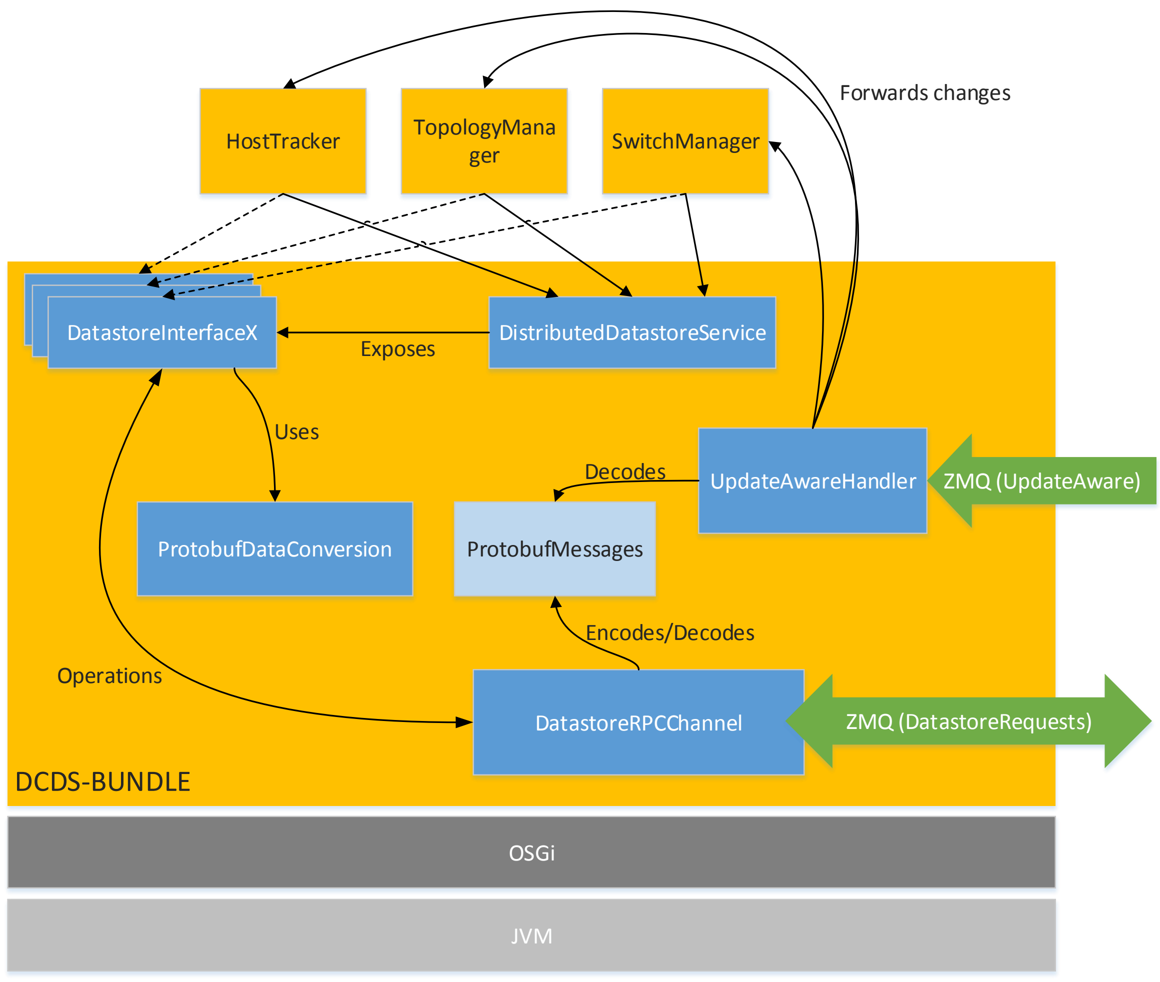


Figure 6.3: Inner structure of the DCDS bundle. The service exposes access to the datastores via interfaces that will perform operations by forwarding them to the C# server using the DatastoreRPCChannel. The update aware handler will do up calls to interested listeners when the contents of a datastore have changed.

Listing 6.1 Modifications to the TopologyManger class in order to use the distributed datastore

```
// original code:
this.edgesDB = (ConcurrentMap<Edge, Set<Property>>)
    this.clusterContainerService.createCache(TOPOEDGESDB,
    EnumSet.of(IClusterServices.cacheMode.TRANSACTIONAL));
// modified code:
this.edgesDB = this.datastoreService.getTopologyManagerEdgesDatastore();
```

Listing 6.2 Implementation of the containsKey-Method in the TopologyManagerEdgesDatastore

```java
@Override
public synchronized boolean containsKey(Object key) {
      if (key instanceof Edge) {
            Edge edge = (Edge) key;

            try {
                  TopologyManagerEdgeDatastoreRequest request = [...]
                  TopologyManagerEdgeDatastoreReply reply =
                      service.topologyManagerEdgeDatastore(request);
                  return reply.getResult();
            } catch (ServiceException e) {
                  throw new IllegalStateException(e);
            }
      } else {
            return false;
      }
}
```

Datastores

The Key-Value stores returned by the DCDS implement the ConcurrentHashMap interface, but all invocations are forwarded to the C# server. To do this, the method parameters are converted into their protobuf representation, a request object is created, serialized and sent to the C# server using the DatastoreRpcService which internally uses the DatastoreRPCChannel. The methods of the service will block until the response arrives. When the response arrives it is first parsed as a Protobuf-message. Then the value returned by the operation, if present, is converted back into the format OpenDaylight internally uses and the call will return. Listing 6.2 shows how a ConcurrentHashMap method is implemented and uses the DatastoreRpcService.

DatastoreRpcChannel

The DatastoreRpcChannel class implements the BlockingRpcChannel interface from the protobuf library. It is used by the DatastoreRpcService that is generated from the .proto file by the protobuf compiler as abstract communication gateway. It provides a communication channel built on top of the ZMQ request-reply pattern.

UpdateAwareDatastores

Two of the bundles that are modified to use the DCDS bundle, implement the IUpdateAware interface from the clustering-service bundle. These classes want to be notified if another instance added, changed or removed data from one of the Key-Value stores they use. As this functionality is heavily incorporated into how the bundles internally work, the DCDS bundle also needs to provide this functionality. In order to do so, a second ZMQ-Socket is created, but instead of a REQ-Socket a REP-Socket is used. A new thread is started to listen on this socket and when a remote change occurs, this thread will call the relevant method of the IUpdateAware interface. The C# server will connect to this socket and forward all relevant events over this communication channel.

6.2.2 C# server

The C# server holds the local copy of the datastore, will handle client requests from the local controller and integrates with the ISIS^2 library. The structure of the C# server is shown in Figure 6.4. It is connected to the local controller over two ZMQ sockets. One socket is a REP-Socket used to receive and respond to controller requests. The other Socket is a REQ-Socket used to "push" UpdateAware notifications to the controller. The REP-Socket will forward incoming requests to the RequestHander after they have been parsed. The RequestHandler will then decide to either answer the response from the local datastore copy if the request is a read operation, or if it is a write operation, to update all instances, including itself, via ISIS^2 by calling one of the send primitives on the group handle, for example
`group.OrderedSend(TOPO_EDGE + REMOVE, key);`
This call will return instantly and not wait for the operation to be confirmed by the other instances. A call to `group.flush()` should, according to the ISIS^2 manual, do exactly this. However, it will not wait for the callbacks to actually be completed and this leads to problems when an instance adds a new entry and then reads are performed on the local copy, not yet containing the added value. To prevent this from happening, the thread handling the request will wait until the local callback has returned, ensuring the reads will see the write. The callbacks will update the local copy according to the operation/received data and will trigger the UpdateAwareSender to inform the local controller of the update. In a similar fashion to the callback methods, methods used for state transfer are needed. One method is used to take a snapshot of the local copy and another must be provided to apply the snapshotted data. Listing 6.3 shows how these callbacks are implemented for one of the key-value stores.

6.3 Model Driven Aspects

In order to make the system more extensible and to support multiple programming languages, model driven methods should be integrated in the development process of the system. While most of the system is not model driven, the specification of the data model and communication protocol using Protobuf and the automated generation of the language specific bindings is a model driven aspect of the system. This could have been taken further by automating the generation of the code needed in order to get a functional key-value store, however the work

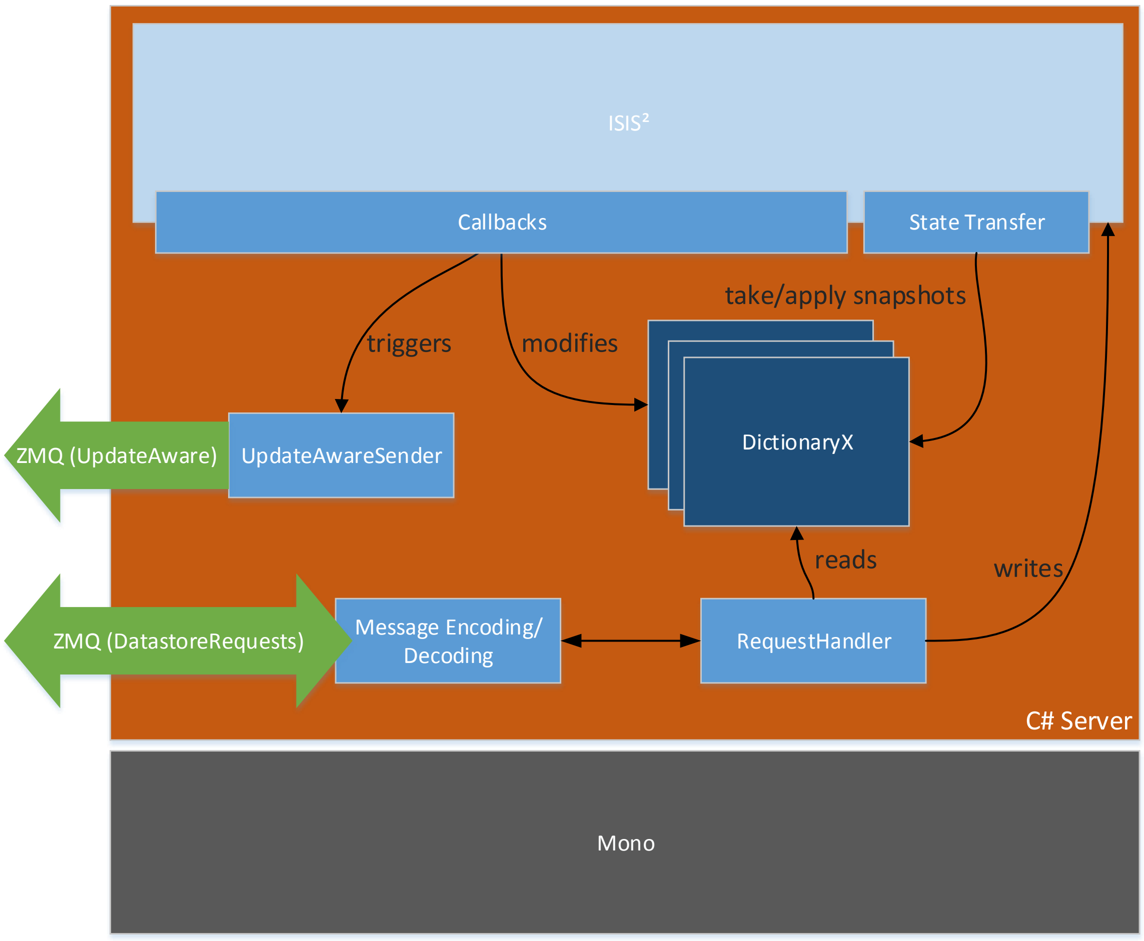


Figure 6.4: Inner structure of the C# server. Operations from the local controller will be handled by the requestHandler. Read operations are performed on the local data, writes will be performed over ISIS^2. ISIS^2 will use the provided callbacks to modify the local copy, which will also trigger the UpdateAwareSender to inform the controller of modifications. The state transfer methods are used by ISIS^2 to initialize a new member in the cluster.

Listing 6.3 Simplified representation of the initialization of ISIS[2]. The System is started, then the data types are registered and a group handle is created. The supported operations and checkpoint handling code must be registered before joining the group

```
IsisSystem.Start();
registerTypes();
var group = new Group("dcds");
group.Handlers[TOPO_EDGE + REMOVE] += (Action<Edge>)delegate(Edge key)
{
   TopologyManagerEdgeDatastore.Remove(key);
};
group.Handlers[TOPO_EDGE + PUT] += (Action<Edge, PropertyList>)delegate(Edge key, PropertyList
     value)
{
   if(TopologyManagerEdgeDatastore.ContainsKey(key))
   {
      TopologyManagerEdgeDatastore[key] = value;
   } else
   {
      TopologyManagerEdgeDatastore.Add(key, value);
   }

};
group.LoadChkpt +=
     (Action<List<TopologyManagerHostDatastoreEntry>>)delegate(List<TopologyManagerHostDatastoreEntry>
     state) {
   foreach(var entry in state){
      TopologyManagerHostDatastore.Add(entry.key, entry.value);
   }
};
group.MakeChkpt += (Isis.ChkptMaker)delegate(View view) {
   group.SendChkpt(TopologyManagerHostDatastore.Select(e => new
        TopologyManagerHostDatastoreEntry(e.Key, e.Value)).ToList());
};
group.Join();
```

needed to get such a model driven process running would most likely exceed the advantages of automated code generation.

6.4 Extensibility

As discussed in 5.1.1, The implemented system is currently not distributing the whole data held by a controller. Other bundles could be integrated with reasonable expense. The data model defined should cover most of the data types OpenDaylight internally uses, so extensions to the data model shouldn't be needed. The communication messages and RPC definitions on the other hand must be extended for the new key-value stores. Once this is done and the language bindings have been recompiled, on the C# side a new dictionary acting as the local copy must be added and some boilerplate code is needed to instruct ISIS^2 to synchronize the dictionary. On the Java side the service provided by the DCDS bundle must be extended to expose the new key-value store and the bundle meant to be modified must import and use the store provided by the DCDS service.

One of the requirements for the implementation was the possibility to integrate other datastore systems apart from ISIS^2 into the system. This is possible and can, depending on the implementation of the datastore, be done directly in the DCDS bundle by providing an alternative RPCChannel implementation, or by implementing the protocol used between the Java and C# components. As a proof of concept, an alternative RPCChannel backed by the etcd key-value store and its Java API could easily be integrated into the existing system.

Another requirement was the possibility to replace OpenDaylight with another controller implementation. While this is possible, it isn't as easy as replacing the datastore. The data model used is strongly adapted to the internal OpenDaylight data classes. Since controllers operate on the same abstractions, another controllers data model might fit the used data model, but even if this is the case, it would require quite some work to integrate.

7 Evaluation

Adding a datastore to an existing system like OpenDaylight might have noticable performance impact, especially considering that the datastore resides in a different process and all read and write operations include (de)serialization and transport.

To better understand the implications of the modifications introduced, the individual parts of the system as well as the complete system are analyzed in their behavior. The components are evaluated in tests similar to their use-case and the complete system is tested under two different scenarios using the different send primitives of ISIS^2 and by using the cbench benchmark.

Three individual components are evaluated: ZMQ transports are compared using messages per second as well as Protobuf message parsing. ISIS^2 is evaluated regarding the performance of the different send primitives and how they scale with process count. The complete system is evaluated through two scenarios that show the impact of the data-plane topology on system load and how the datastore behaves in these situations. Also cbench, a benchmark for controllers, is used to compare the performance of the proposed system with other controllers.

7.1 Test Environment

The evaluation tests, if not stated otherwise, were run on a xubuntu 14.04 x64 VM with 8GB RAM and 4 CPU cores allocated, using Vmware player 6.0.5. The VM hard-drive is stored on a SSD. The host system runs Windows 7 x64, has 16GB RAM and a 8 core Xeon e3-1230v3 CPU with 3.3GHz. Java x64 version 1.7u76 was used, and for the C# parts Mono 3.12.1.

To simulate the data-plane network, Mininet 2.1.0 [41] was used. Mininet is a network emulator that creates hosts, switches and links according to a topology defined by the user. The switches in the virtual network are emulated using open vSwitch, which support the OpenFlow protocol. The hosts are processes that only have access to their virtualized network interface.

7.2 System Components

The system component evaluations include a comparison of the TCP and IPC transports of ZMQ, the throughput of the Protobuf implementations used and how the different ISIS^2 send primitives scale with process count.

7.2.1 ZMQ-Transport

In order to decide (and to legitimate the use of TCP after the IPC transport appeared to be unstable) on a transport method for the ZMQ based communication between the DCDS bundle and the C# server, the throughput between the TCP and IPC transports were compared in a setup similar to the planned use-case. Over a timespan of 60 seconds it was measured how often the following task was performed: A pseudo-random (each iteration used the same seed) byte array of size $64 + rnd(0, 256)$ was created on a Java process, sent to a C# process that sent the same byte array back. The Java process then compared the results and started the next task. The measurement was repeated 50 times.

Figure 7.1 shows the result of the test. While the IPC transport has slightly better performance, its standard deviation was higher compared to the TCP transport (9033 ± 1211 for IPC compared to 8316 ± 688 when using TCP). During these tests both transports performed stable and no problems appeared. But, as mentioned earlier, during the tests of the complete system ZMQ leaded to crashes on the C# part of the system. Debugging this showed that the IPC transport seemed to cause this and after using the TCP transport, the system showed no such behavior. This resulted in TCP being used over IPC.

7.2.2 Protobuf Serialization

The serialization and deserialization of the data and message types defined using protobuf are critical to system performance as they are used frequently and on all communication channels (both ZMQ socket pairs and internally by ISIS^2) of the system.

In Figure 7.2 a performance comparison parsing a recorded message between used protobuf libraries is given. The C# protobuf-net libary was tested twice, once running on Mono and once native on the Windows host described in 7.1 to illustrate the performance impact of Mono. The test was repeated 50 times, each test run took 10 seconds. The message used was a LIST operation response for the TopologyManager-Edge datastore when containing 12 entries. The message has a size of 574 bytes which renders it one of the larger messages. The requests are normally around 100 bytes, as are most responses except the LIST responses that are naturally larger as they scale with the number of entries stored in the datastore the LIST operation is called on.

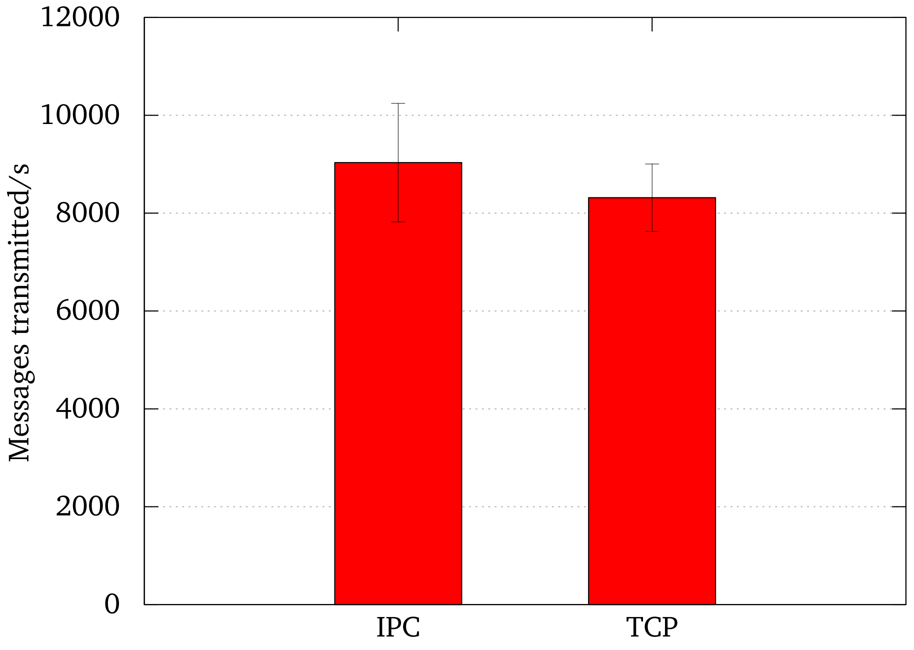


Figure 7.1: ZMQ messages mirrored per second. Comparison of the IPC and TCP transports of ZMQ using a Java client which sends messages to a C# server, mirroring the received messages back.

While the performance of the message parsing is higher than the performance of the ZMQ sockets and the parsing therefore doesn't depict a performance bottleneck, the difference between the Java and C# implementation are extreme, with the Java version performing 14 times faster than the C# code running on Mono. Attempts to increase the performance of the C# message parsing according to the available documentation didn't lead to improvements. Also the impact of Mono is drastic, as the same binary performed 4 to 5 times better running native on Windows.

7.2.3 ISIS^2

As ISIS^2 is one of the main building blocks of the system, the performance of the communication primitives offered by ISIS^2 are critical to system performance. In figure 7.3 the throughput in write operations to one of the key-value stores is shown using one active process and 0, 2, 4, 9 or 24 passive "receiver" processes. 7.3a shows the throughput when the local synchronization mechanism used to guarantee read-after-own-write consistency as described in 6.2.2 are

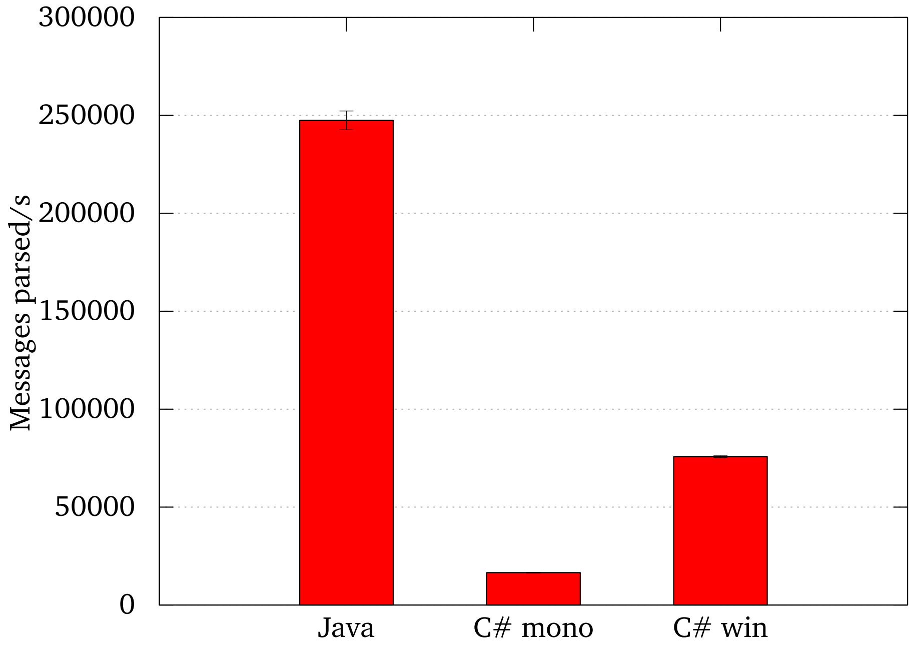


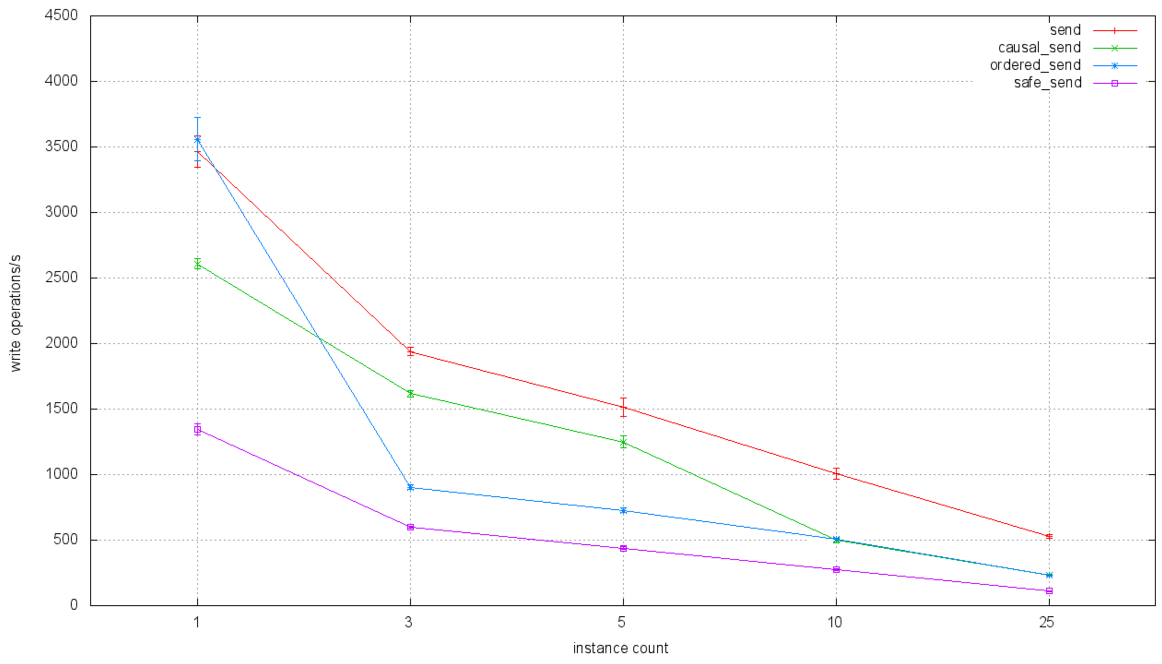
Figure 7.2: Comparison of throughput when parsing a message using the default Java Protobuf library and protobuf-net, running on Windows and Mono.

enabled and in 7.3b when they are disabled. An explanation for the high performance of the OrderedSend primitive when only one process is present can be found in this quote from page 40 of the ISIS^2 documentation(available at [42]):

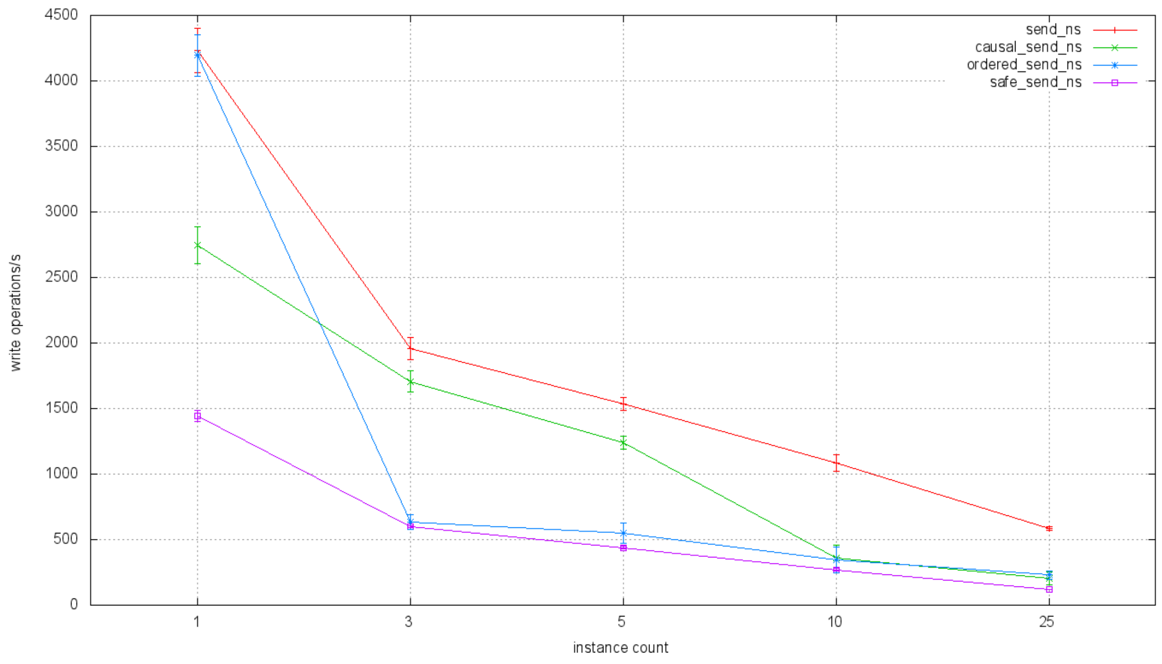
> OrderedSend is slower than Send except in one special case: the protocol optimizes itself in situations where the sender is the rank-zero member of the group and no other member has sent any ordered multicasts during the current view. It will run faster while this condition holds. As soon as some other member does use OrderedSend, the optimization shuts itself off, and the rank-zero member's multicasts then have the same cost as multicasts from any other member, until the next membership view change event.

Each test run lasted one minute and was repeated ten times. The processes were restarted after the first five runs. The data used to test the write performance was extracted from the message described in the Protobuf evaluation.

The difference in performance between the send primitives is as expected, with the Send primitive performing the best followed by CausalSend, OrderedSend and with SafeSend being



(a) Local synchronization enabled



(b) Local synchronization disabled

Figure 7.3: Performance of ISIS^2 send primitives for 1, 3, 5, 10 and 25 processes with and without the local synchronization mechanisms required by the C# server build on top of ISIS^2

the slowest. The impact of the local synchronization mechanism is not as severe as expected and vanishes with growing process count.

7.3 Complete System

In order to see if the proposed controller has enough performance to support realistic use-cases, three scenarios with different topologies and events were run. Since currently the datastore is limited to only contain topology information, The events are mostly limited on topology changes like link up/down or host add/remove. The Mininet functionality to issue a bidirectional ping between all host pairs is used frequently as this causes the routing bundle of OpenDaylight to generate new flow rules. To calculate the flow rules needed, it heavily queries the topology data stored in the datastore.

Additionally the proposed controller was compared to freely available controller implementations using the cbench benchmark.

7.3.1 Methodology

Each scenario was repeated three times for each send-primitive, topology and static ARP entries turned on/off which results in 86 total runs, each taking between 5 and 10 minutes. For each run, the following information was recorded: A UNIX-Timestamp indicating the start of the operation, The name of the datastore, the operation performed and the duration of the operation in nanoseconds. The measurements were performed in the RPC-Channel component of the DCDS bundle. The diagrams showing the results of the scenario runs are based on the recorded values.

7.3.2 Used Topologies

In both scenarios two different network topologies were used. The first topology, called fattree, represents a data center network with its hierarchical structure. The topology consists of 16 switches and 32 hosts arranged in a tree structure. Figure 7.4 shows the fattree topology used. The other Topology used is based on the X-WiN backbone of the DFN, the German Research Network (a Mininet topology script available at [43] was used). The Topology is available It consists of 58 switches and each having a host connected to it. This topology resembles a large WAN and the topology includes delay information for the links. The topology can be seen in Figure 7.5.

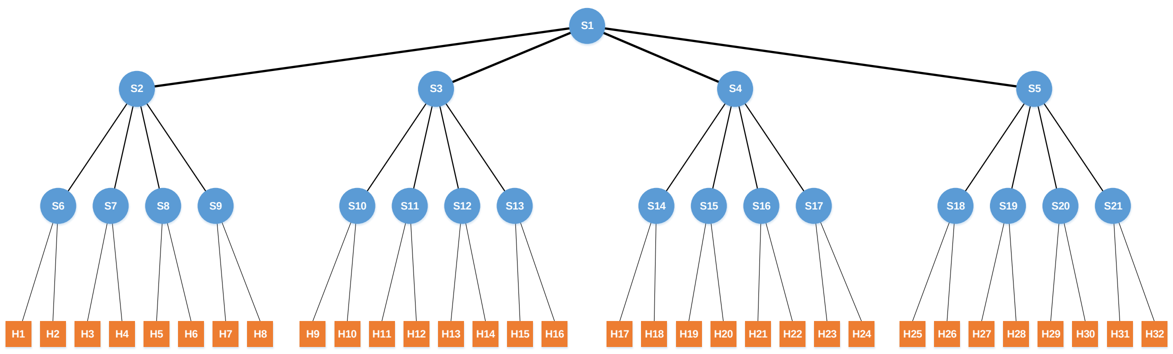


Figure 7.4: First test topology, called fattree. It consists of 16 switches and 32 hosts connected in a tree structure with growing link bandwidth towards the root.

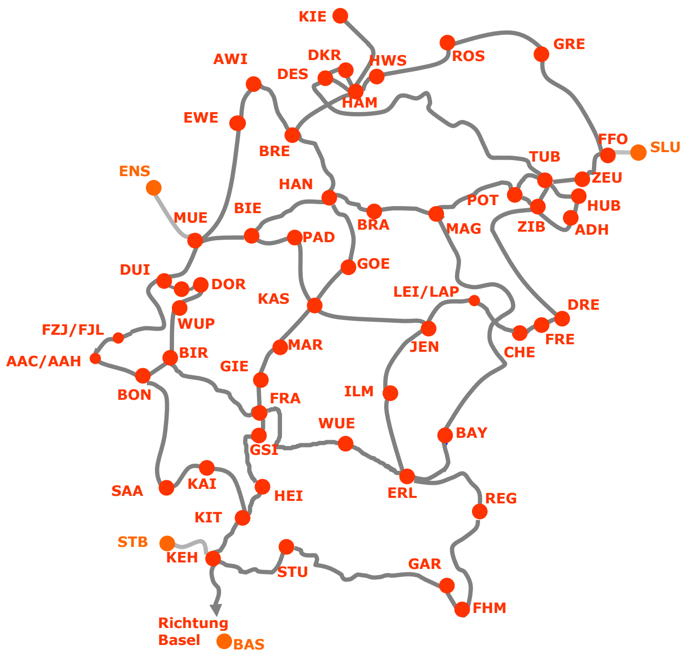


Figure 7.5: Second test topology, this topology is based on the DFN X-WiN, the backbone of the German Research Network. Extracted from [44]

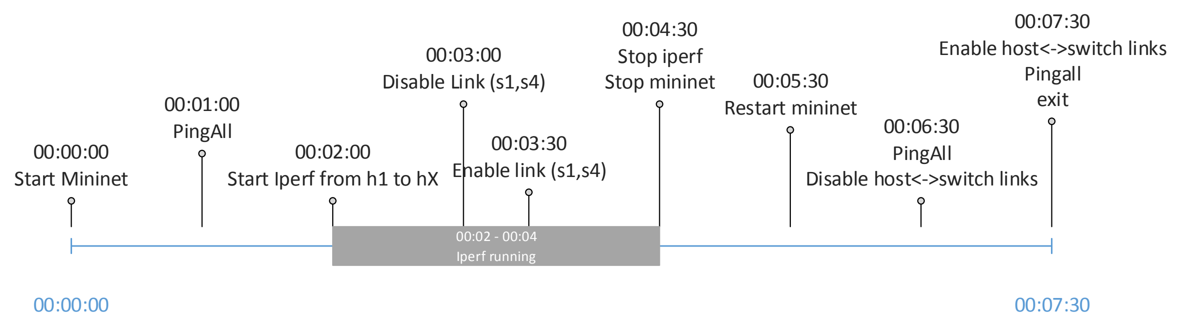


Figure 7.6: The sequence of events in scenario 1.

7.3.3 Scenario 1

The sequence of events specified for scenario 1 are visible in Figure 7.6.

In Figure 7.7 the operations each datastore performs are shown. It is clearly visible that the dominating factors are CONTAINS operations on the NodeConnectorDatastore as well as LIST operations on the NodeConnectorPropsDatastore. A investigation showed that both recquests are triggered by the same class in the simpleforwarding bundle. This class (SimpleBroadcastHandlerImpl) registers a listener on incoming packets from switches. If the received packet is a broadcast packet, it will iterate over the list of switch ports known (using the information from the NodeConnectorPropsDatastore) and, if the port is external (facing a host), send the packet out over this port. The check if the port is external causes the CONTAINS operation on the NodeConnectorDatastore, that only contains internal switch ports. If the broadcast functionality would be realized in a different way, most of the load the datastore experiences could be removed.

Figure 7.8 shows that enabling or disabling static ARP entries doesn't have an effect on the load the datastore experiences. The two data series are slightly offset, a effect that all test runs show. This is caused mostly by single pings failing, which leads to high timeouts to occur.

Figure 7.9 and 7.10 show the operations per second for all the send primitives and both topologies. It is visible that the send primitive doesn't have a major effect on the load the system experiences, a behavior that would begin to show when multiple controller instances are up. The topology on the other hand has a huge impact on the load. Another thing to notice is that during the wait steps of the scenario, the system load does not drop instantly but falls off slowly. If a larger wait interval would be used, the load would almost drop to zero, as the cause of the load during the wait periods, broadcast packets as discussed earlier, are slowly falling off in frequency.

In Figures 7.11 and 7.12 The duration distribution of all operations are shown. As expected the send primitive also does not have a major impact on operation duration when only one instance is present. A effect that is visible however is that the topology has an impact on operation duration. In the fattree topology, the second peak which mainly consists of LIST operations is around 0.2 ms while the X-WiN topology showed these operations around 0.5 ms.

7.3.4 Scenario 2

The sequence of events specified for scenario 2 are visible in Figure 7.13.

As the send primitive as well as the enabling/disabling of static ARP entries have no influence on the observed system behavior, for the second scenario only runs with the ordered send primitive and enabled static ARP entries are presented. In order to see that the load caused by broadcast packets falls off with time, the wait interval between test events is increased to 120 seconds. In this scenario the controller is started, then Mininet. A wait interval follows after which a ping all is issued. After the ping all and another wait interval, random switch-switch links are brought up and down for one minute. After this and another wait interval all switch-switch links are brought back up and after another wait interval, ping all is run again and then the scenario terminates.

Figures 7.14 and 7.15 show the operations per second observed for the second scenario on the fattree and X-WiN topologies. While in both runs the scenario events are clearly visible, the load the topologies produce differ, especially during and after the startup and ping all events. While the load during the random link state changes did not differ too much from the first to events in the X-WiN topology run, in the fattree run the load was significantly larger. This observation can again be explained with the impact of the broadcast packet handling and its scaling with topology size.

In Figures 7.16 and 7.17 the duration distribution observed for ordered send in the second scenario are shown. Again the impact of topology size on the speed of LIST operations can be observed. In the fattree duration histogram a small peak around 0.2 ms can be seen, most likely introduced by the increased number of PUT and REMOVE operations caused by the link state changes.

7.3.5 Cbench

The implementation was compared with current, freely available SDN controllers using the Cbench controller benchmark [45]. On the Cbench website it is described as follows:

> Cbench (controller benchmarker) is a program for testing OpenFlow controllers by generating packet-in events for new flows. Cbench emulates a bunch of switches which connect to a controller, send packet-in messages, and watch for flow-mods to get pushed down.

Except the DCDS win test run, where the controller was running on the windows host and Cbench on the guest VM, all tests were run on the test environment described in 7.1. The results are presented in Figure 7.18. Compared to OpenDaylight, the performance impact of the modifications are huge. However when compared with other controllers the performance of the proposed controller are within the range of most other controllers.

A comparison of the influence of instance count for distributed controllers, namely the proposed solution and OpenDaylight is given in Figure 7.19. At first sight the significant drop in performance OpenDaylight shows is surprising. While certain influences from side effects (every controller ran in a dedicated VM) can't be excluded, infinispan shows similar behavior for writes, documented in a benchmark on the official blog [46]. The proposed controller has way lower results, but they remain constant. It would be interesting to see how both controllers behave when tested with higher instance numbers. However, that remains a task for future work.

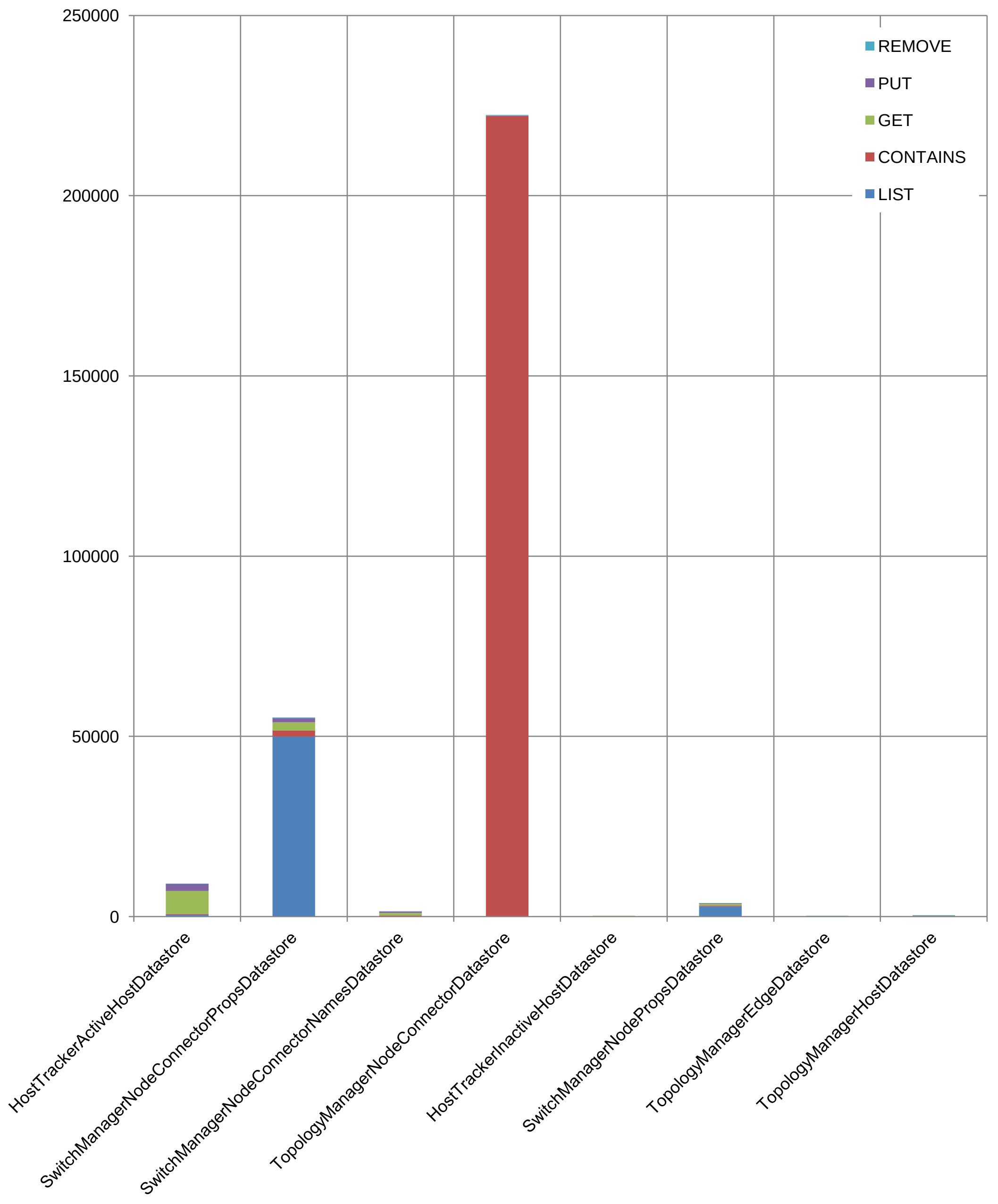


Figure 7.7: Operations per store. This diagram shows the distribution of operations per datastore and the type of the request for a run of scenario 1 with ordered send and static ARP entries. It is clearly visible that most of the requests were contains request on the NodeConnector datastore used by the Topology manager.

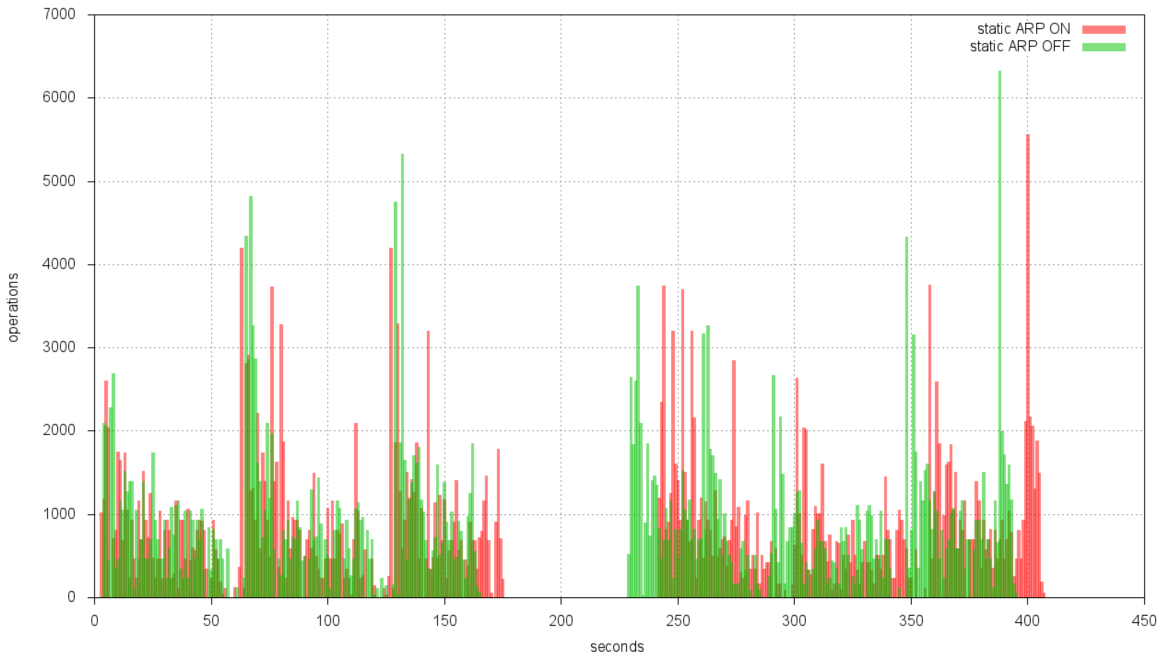


Figure 7.8: Comparison of a test run with static ARP entries enabled and a run with static ARP disabled. The observed differences show no significant influence of static ARP entries on the datastore load. The horizontal offset is caused by some of the pings (mininet has high timeouts for pings) failing.

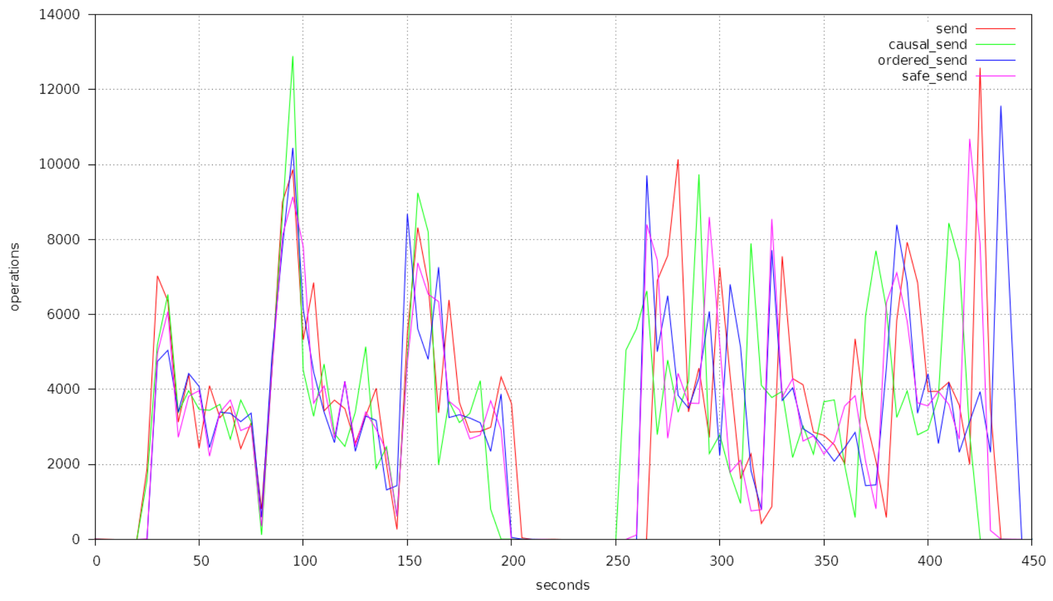


Figure 7.9: Timeline of scenario one running on the fattree topology.

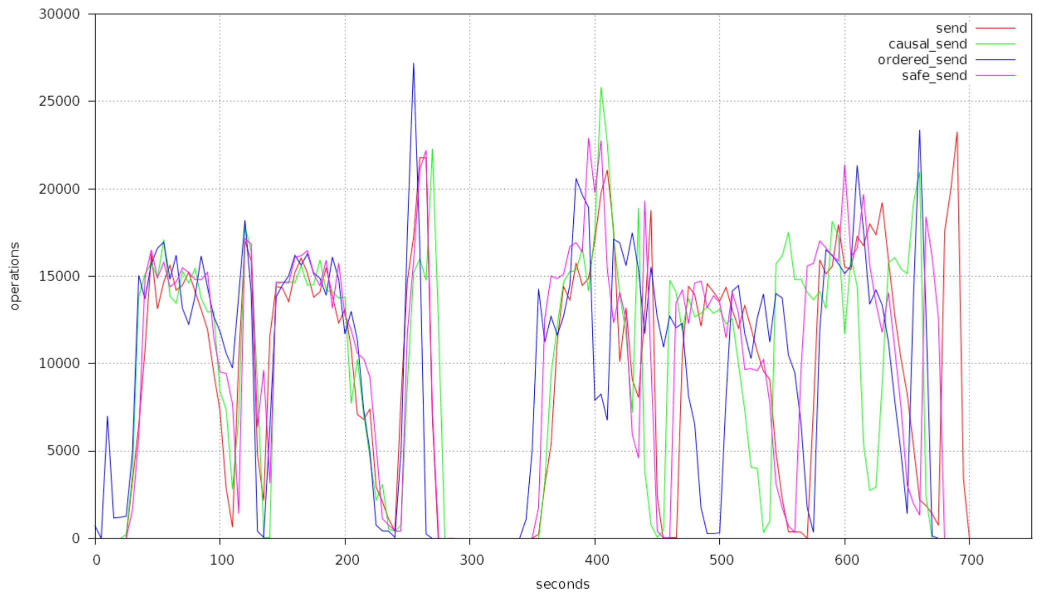


Figure 7.10: Timeline of scenario one running on the DFN X-WiN topology.

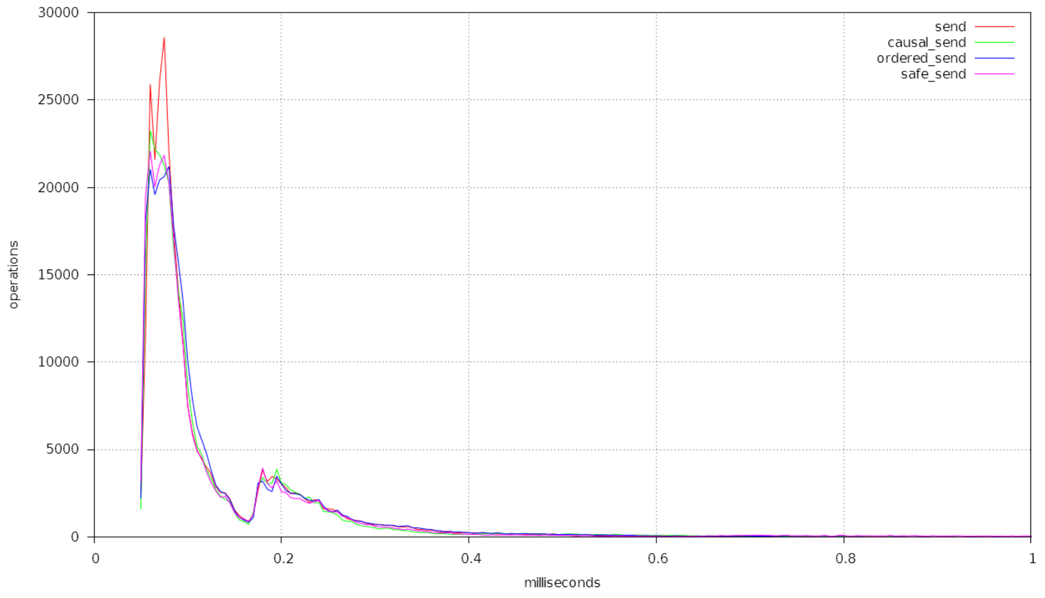


Figure 7.11: Histogram showing the duration of operations for the first scenario on the fattree topology.

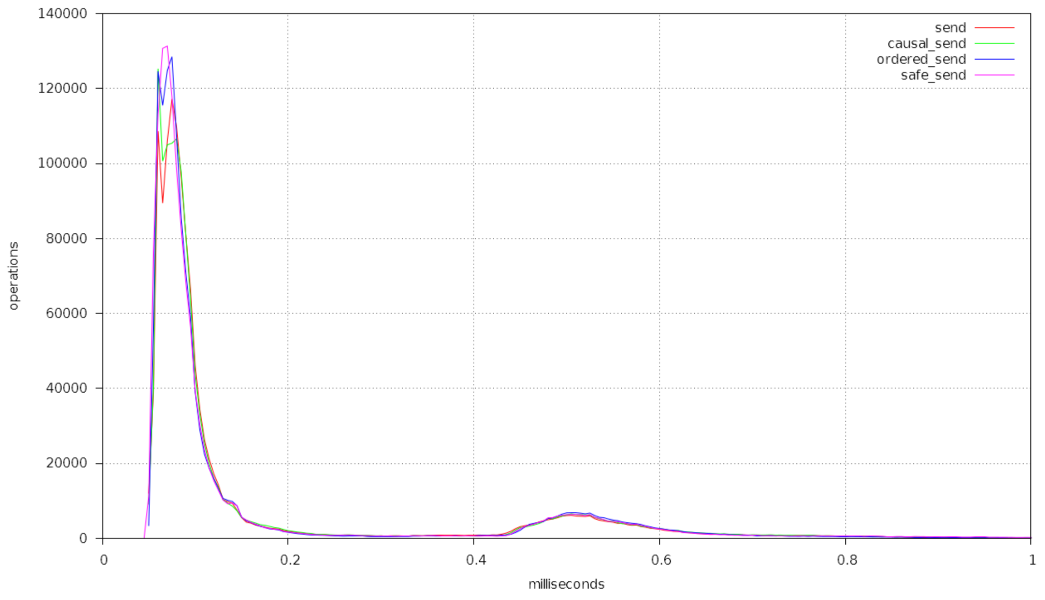


Figure 7.12: Histogram showing the duration of operations for the first scenario on the DFN X-WiN topology.

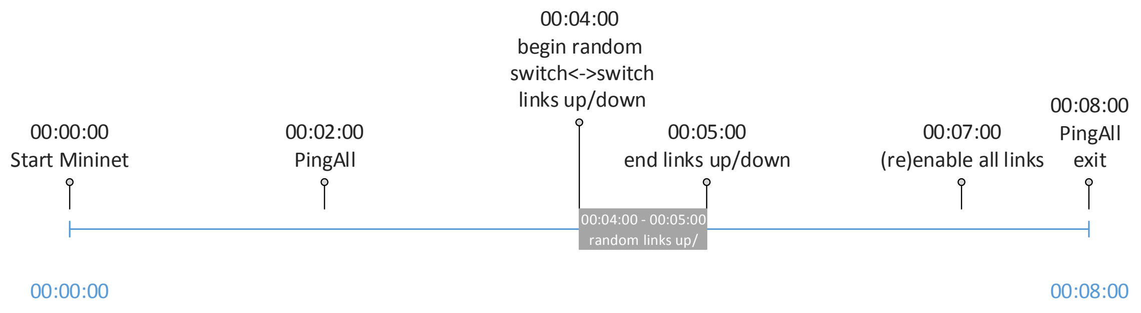


Figure 7.13: The sequence of events in scenario 2.

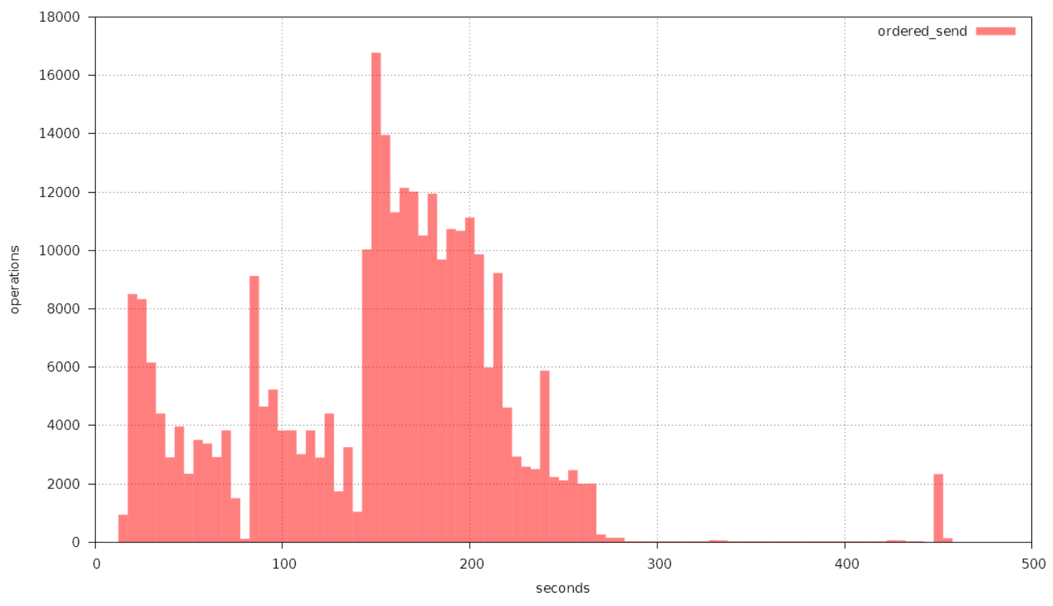


Figure 7.14: Timeline of scenario two running on the fattree topology.

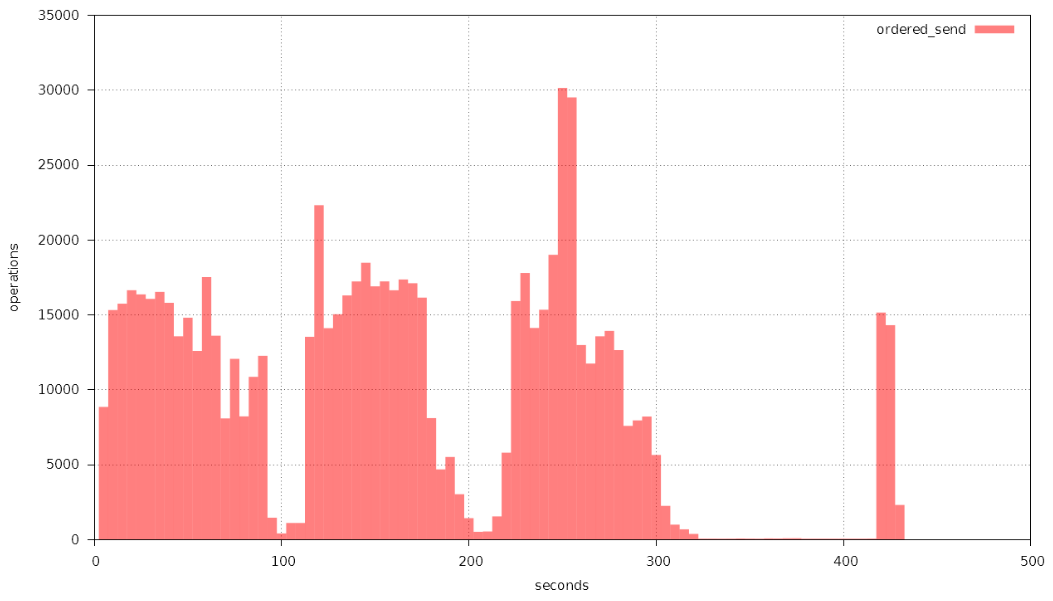


Figure 7.15: Timeline of scenario two running on the DFN X-WiN topology.

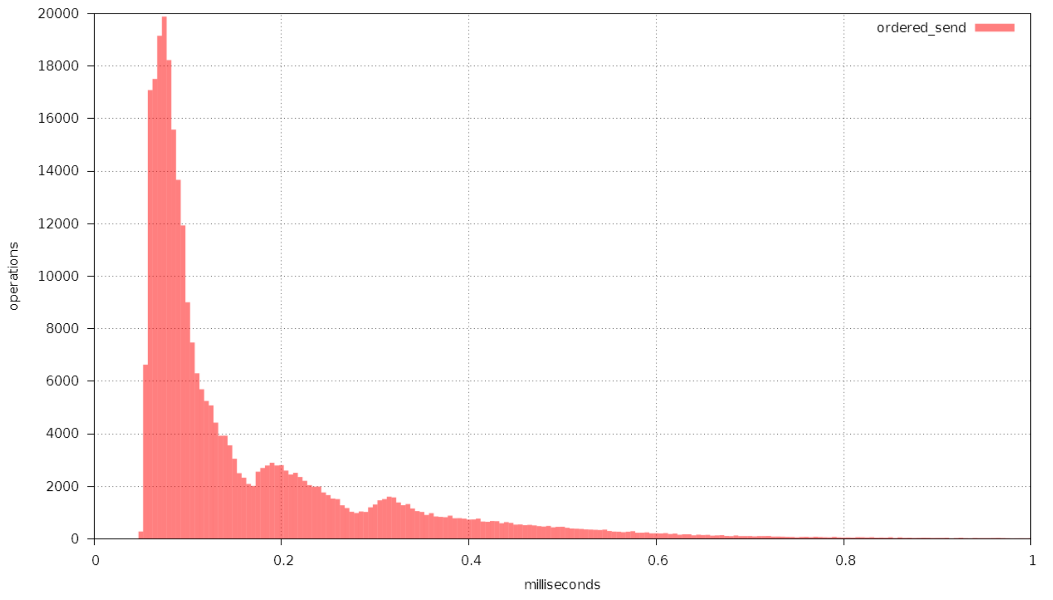


Figure 7.16: Histogram showing the duration of operations for the second scenario on the fattree topology.

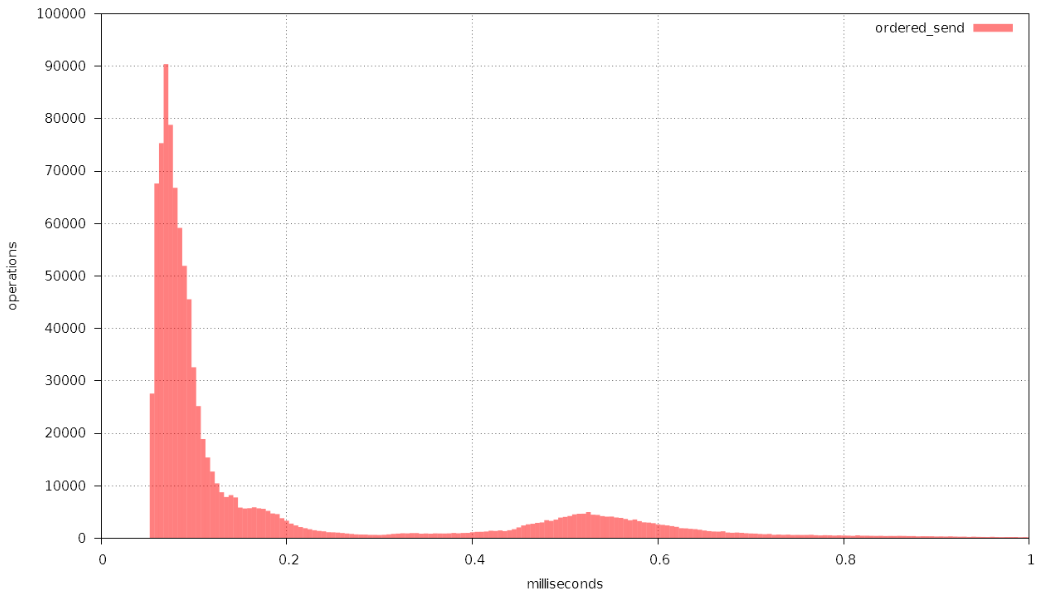


Figure 7.17: Histogram showing the duration of operations for the second scenario on the DFN X-WiN topology.

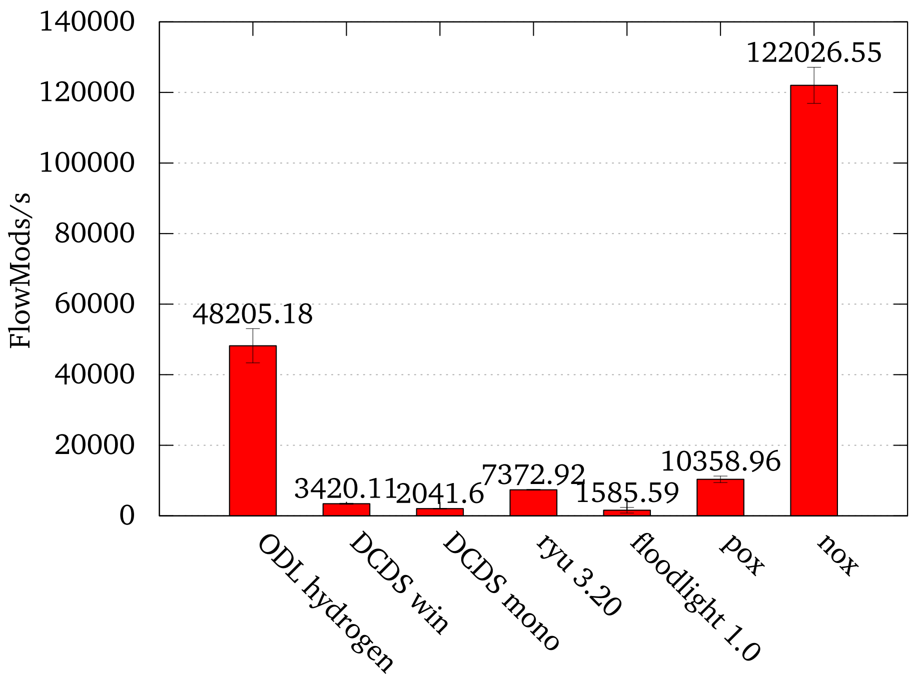


Figure 7.18: Comparison of the proposed solution with existing SDN controllers using the cbench benchmark.

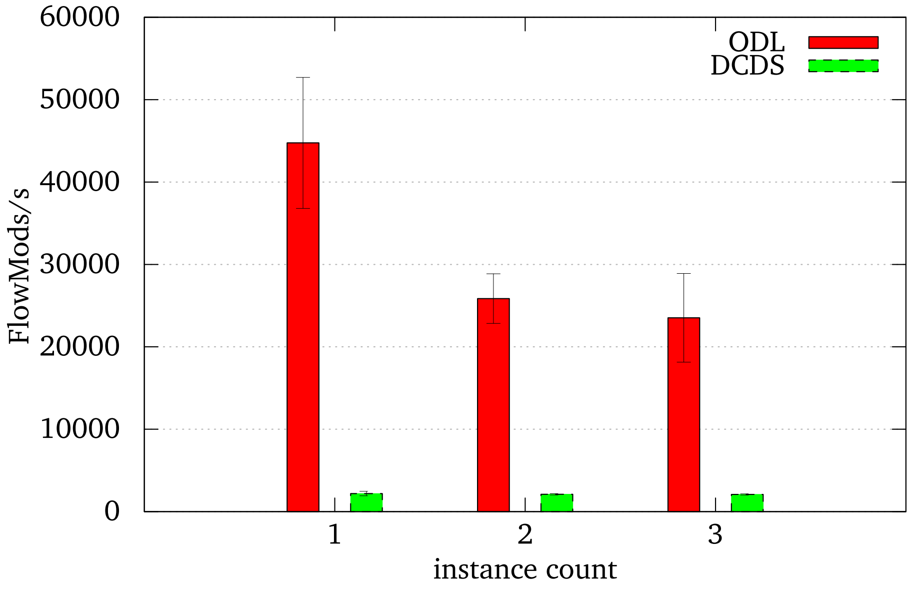


Figure 7.19: Comparison of the proposed solution with the OpenDaylight controller using the cbench benchmark and different numbers of instances.

8 Conclusion and Future Work

This thesis presented a distributed SDN controller design and implementation based on OpenDaylight and the ISIS^2 library. Using ISIS^2 to realize the synchronization mechanism of the datastore holding controller state, the benefits of the virtual synchrony model can be exploited, which enables higher synchronization performance when certain assumptions about the data and system can be made. This was shown to be true in the case of a SDN controller. The proposed system supports dynamic cluster changes at runtime.

Evaluation of the building blocks of the system showed acceptable performance and the evaluation of the whole system indicated that in real world use-cases the proposed implementation functions correctly and has adequate performance compared to other available (non-distributed) controllers. Since the evaluations were done on a virtual machine and with a virtual network, repeating the measurements in a more realistic, non virtual environment might yield different results.

One of the aspects that should be evaluated in more detail are the performance implications of Mono. While certain performance loss was excepted as a result of running on mono, the dimension observed exceeded the expectations. With Microsoft recently beginning to open sourcing the .net platform [47], this effect might mitigate in the near future.

A huge possibility for further optimization lies in the fact that currently the read operations are not fully exploiting the benefit of local-only reads introduced by the virtual synchrony model. If the local copy would be kept directly in the DCDS bundle and not on the C# side, this could significantly increase performance. However, this change would require substantial changes to the system architecture and the possibility to incorporate features like sharding would be aggravated. Also, by eliminating this handicap of the current system, the overhead introduced by the serialization and communication between two processes would be minimized as only the write operations would require those mechanisms.

Eliminating the general need for serialization between the datastore and controller altogether would further increase the system performance. However this would either require a controller to be build from scratch in the C# eco system, or to migrate the ISIS^2 library to Java. Both possibilities are complex and would require a significant amount of work and pose a possibility for future work.

Bibliography

[1] Open Networking Foundation, "Software-defined networking: The new norm for networks," April 2012. (Cited on page 13)

[2] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008. (Cited on page 15)

[3] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *SIGACT News*, vol. 33, no. 2, pp. 51–59, Jun. 2002. [Online]. Available: http://doi.acm.org/10.1145/564585.564601 (Cited on page 15)

[4] Dormando. Memchached. [Online]. Available: http://memcached.org/ (Cited on page 16)

[5] CoreOS. etcd. [Online]. Available: https://coreos.com/etcd/ (Cited on page 16)

[6] Redis. redis. [Online]. Available: http://redis.io/ (Cited on page 16)

[7] Hazelcast, Inc. hazelcast. [Online]. Available: http://hazelcast.com/ (Cited on page 16)

[8] Red Hat, Inc. Infinispan. [Online]. Available: http://infinispan.org/ (Cited on page 16)

[9] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Comput. Surv.*, vol. 22, no. 4, pp. 299–319, Dec. 1990. [Online]. Available: http://doi.acm.org/10.1145/98163.98167 (Cited on page 16)

[10] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978. (Cited on pages 16 and 22)

[11] ——, "Lower bounds for asynchronous consensus," in *Future Directions in Distributed Computing*. Springer, 2003, pp. 22–23. (Cited on pages 16 and 17)

[12] G. Bracha and S. Toueg, "Asynchronous consensus and broadcast protocols," *Journal of the ACM (JACM)*, vol. 32, no. 4, pp. 824–840, 1985. (Cited on page 17)

[13] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency control and recovery in database systems*. Addison-wesley New York, 1987, vol. 370. (Cited on page 17)

[14] D. Skeen and M. Stonebraker, “A formal model of crash recovery in a distributed system,” *Software Engineering, IEEE Transactions on*, no. 3, pp. 219–228, 1983. (Cited on page 17)

[15] L. Lamport, “The part-time parliament,” *ACM Transactions on Computer Systems (TOCS)*, vol. 16, no. 2, pp. 133–169, 1998. (Cited on page 18)

[16] ——, “Paxos made simple,” *ACM Sigact News*, vol. 32, no. 4, pp. 18–25, 2001. (Cited on page 18)

[17] T. D. Chandra, R. Griesemer, and J. Redstone, “Paxos made live: an engineering perspective,” in *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*. ACM, 2007, pp. 398–407. (Cited on page 19)

[18] B. M. Oki and B. H. Liskov, “Viewstamped replication: A new primary copy method to support highly-available distributed systems,” in *Proceedings of the seventh annual ACM Symposium on Principles of distributed computing*. ACM, 1988, pp. 8–17. (Cited on page 19)

[19] B. Liskov and J. Cowling, “Viewstamped replication revisited,” 2012. (Cited on pages 19 and 21)

[20] R. Van Renesse, N. Schiper, and F. B. Schneider, “Vive la différence: Paxos vs. viewstamped replication vs. zab,” 2013. (Cited on page 19)

[21] D. Ongaro and J. Ousterhout, “In search of an understandable consensus algorithm,” in *Proc. USENIX Annual Technical Conference*, 2014, pp. 305–320. (Cited on page 21)

[22] K. Birman and T. Joseph, *Exploiting virtual synchrony in distributed systems*. ACM, 1987, vol. 21, no. 5. (Cited on page 21)

[23] Ken Birman. Isis2 cloud computing library. [Online]. Available: https://isis2.codeplex.com/ (Cited on page 22)

[24] K. P. Birman, D. A. Freedman, Q. Huang, and P. Dowell, “Overcoming cap with consistent soft-state replication,” *Computer*, no. 2, pp. 50–58, 2011. (Cited on page 23)

[25] A. Tootoonchian and Y. Ganjali, “Hyperflow: A distributed control plane for openflow,” in *Proceedings of the 2010 internet network management conference on Research on enterprise networking*. USENIX Association, 2010, pp. 3–3. (Cited on page 27)

[26] J. Stribling, Y. Sovran, I. Zhang, X. Pretzer, J. Li, M. F. Kaashoek, and R. Morris, “Flexible, wide-area storage for distributed systems with wheelfs.” in *NSDI*, vol. 9, 2009, pp. 43–58. (Cited on page 27)

[27] S. Hassas Yeganeh and Y. Ganjali, “Kandoo: a framework for efficient and scalable offloading of control applications,” in *Proceedings of the first workshop on Hot topics in software defined networks*. ACM, 2012, pp. 19–24. (Cited on page 27)

[28] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama *et al.*, "Onix: A distributed control platform for large-scale production networks." in *OSDI*, vol. 10, 2010, pp. 1–6. (Cited on page 27)

[29] F. Botelho, F. Valente Ramos, D. Kreutz, and A. Bessani, "On the feasibility of a consistent and fault-tolerant data store for sdns," in *Software Defined Networks (EWSDN), 2013 Second European Workshop on*. IEEE, 2013, pp. 38–43. (Cited on pages 27 and 28)

[30] D. Levin, A. Wundsam, B. Heller, N. Handigol, and A. Feldmann, "Logically centralized?: state distribution trade-offs in software defined networks," in *Proceedings of the first workshop on Hot topics in software defined networks*. ACM, 2012, pp. 1–6. (Cited on page 27)

[31] A. Bessani, J. Sousa, and E. E. Alchieri, "State machine replication for the masses with bft-smart," in *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*. IEEE, 2014, pp. 355–362. (Cited on page 27)

[32] A. Dixit, F. Hao, S. Mukherjee, T. Lakshman, and R. Kompella, "Towards an elastic distributed sdn controller," in *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4. ACM, 2013, pp. 7–12. (Cited on page 27)

[33] Al Danial. Cloc count lines of code. [Online]. Available: http://cloc.sourceforge.net (Cited on pages 29 and 39)

[34] Google Inc. Protocol buffers. [Online]. Available: https://developers.google.com/protocol-buffers (Cited on pages 33 and 36)

[35] OpenDaylight project. Opendaylight. [Online]. Available: http://www.opendaylight.org (Cited on page 35)

[36] Mono Project. Mono. [Online]. Available: http://www.mono-project.com/ (Cited on page 35)

[37] iMatix Corporation. Zmq manual. [Online]. Available: http://api.zeromq.org/2-1:zmq-ipc (Cited on page 35)

[38] Marc Gravell. protobuf-net. [Online]. Available: https://code.google.com/p/protobuf-net (Cited on page 36)

[39] OpenDaylight project. Opendaylight resources collateral. [Online]. Available: http://www.opendaylight.org/resources/collateral (Cited on page 38)

[40] iMatix Corporation. Zmq. [Online]. Available: http://www.zeromq.org (Cited on page 37)

[41] Mininet Team. Mininet. [Online]. Available: http://mininet.org/ (Cited on page 47)

[42] Ken Birman. Isis documentation. [Online]. Available: https://www.codeplex.com/Download/AttachmentDownload.ashx?ProjectName=isis2&WorkItemId=63&FileAttachmentId=878983 (Cited on page 50)

[43] Professur für Informatik, KTR. Dfn x-win topology mininet script. [Online]. Available: https://github.com/uniba-ktr/assessing-mininet/blob/master/parser/topologies/Dfn.graphml-generated-Mininet-Topo.py (Cited on page 52)

[44] Deutsches Forschungsnetz. Dfn x-win topology. [Online]. Available: https://www.dfn.de/fileadmin/1Dienstleistungen/XWIN/Netzentwurf_20140110_Seite1.pdf (Cited on page 53)

[45] Rob Sherwood. Cbench: an open-flow controller benchmarker. [Online]. Available: http://www.openflow.org/wk/index.php/Oflops/ (Cited on page 55)

[46] Manik Surtani. Infinispan 4.0.0.final has landed! [Online]. Available: http://blog.infinispan.org/2010/02/infinispan-400final-has-landed.html (Cited on page 56)

[47] Dotnet Foundation. Dotnet. [Online]. Available: https://github.com/dotnet (Cited on page 67)

Declaration / Erklärung

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

place, date, signature
Ort, Datum, Unterschrift