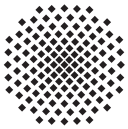

Universität Stuttgart

Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelor's Thesis Nr. 145

Elastic control of content-routing in OpenFlow

Alexander Kicherer

Course of Study:	Softwaretechnik
Examiner:	Prof. Dr. Kurt Rothermel
Supervisor:	M. Sc. Sukanya Bhowmik
Commenced:	2014-05-15
Completed:	2014-11-28
CR-Classification:	C.1.2, C.1.4, C.2.4

Abstract

Publish/subscribe systems are an essential part of many distributed systems for their components (publishers and subscribers) to communicate with each other. The use of content-based routing for more efficient bandwidth usage and to decrease the usage of other resources, led to systems developed are fairly efficient and fast for their architecture of creating an overlay network.
With the use of Software-defined Networking (SDN) and in-network filtering by its usage, the event forwarding efficiency and delays of publish/subscribe systems were further improved. Event forwarding in SDN based publish/subscribe systems using in-network filtering is already very good, but the processing of requests like (un)advertisements and (un)subscriptions needs to be scalable, too. In the current implementation requests are handled in a sequential manner, which is not scalable at all.
This thesis proposes a way to handle the computational part of request processing in a parallelised way with little computational overhead by taking advantage of the independency of the partitions of the event space and the corresponding data used for computation as well as resulting flow rule changes. By this the average waiting time until a request is processed is to be decreased and the general throughput of requests per time is to be increased. This goal is reached by independent computing of request parts based on partitions created by spatial indexing.
The evaluation was done with a multithreaded solution to show the impact of parallel computation of changes of flow rules on switches. The proposed approach to process requests in parallel shows the average waiting time of requests to drop up to to one fourth when using four threads on a machine with four cores to compute requests in parallel. This shows the possibility of large performance gains by parallelising request processing the proposed way.

Kurzfassung

Publish/Subscribe Systeme sind ein zentraler Teil vieler verteilter Systeme, um deren Komponenten (Publisher und Subscriber) miteinander kommunizieren zu lassen. Durch Inhaltsbasiertes Routing für effizientere Nutzung von Bandbreite und sonstiger Ressourcen wurden Systeme entwickelt, welche für ihre Overlay-Netzwerk-Architektur sehr effizient arbeiten.
Mit Software-defined Networking (SDN) und Filterung im Netzwerk durch dessen Nutzung wurde die Effizienz von Event Weiterleitungen und die dabei auftretende Verzögerungen weiter verbessert. Die Weiterleitung von Events in einem auf SDN basierenden Publish/Subscribe System mit der Nutzung von Filterung im Netzwerk ist schon sehr gut, das Verarbeiten von Anfragen wie (un)advertisements und (un)subscriptions sollte jedoch auch skalierbar sein. In der bisherigen Implementierung werden Anfragen sequentiell abgearbeitet, dies ist nicht skalierbar.
In dieser Arbeit wird ein Ansatz vorgestellt, wie der Berechnungsanteil von Anfragen mit wenig zusätzlichem Aufwand bewältigt werden kann. Dazu wird die Unabhängigkeit der Partitionen des Event-Raumes, deren zugehörige für Berechnungen relevante Daten und die zugehörigen berechneten Änderungen der Weiterleitungsregeln dafür genutzt. Dadurch soll die durchschnittliche Wartezeit bis eine Anfrage bearbeitet wird reduziert werden und der Durchsatz an Anfragen erhöht werden. Dieses Ziel wird erreicht indem unabhängig Anfragenteile parallel zueinander Abgearbeitet werden, basierend auf den durch Spatial Indexing entstandenen Partitionen.
Die Evaluierung wurde durch eine auf Multithreading basierende Lösung um die Auswirkungen von paralleler Berechnung der Änderungen von Weiterleitungsregeln der Switches im Netzwerk. Der vorgeschlagene Ansatz, Anfragen auf diese Art parallel zu bearbeiten zeigt, dass die durchschnittliche Wartezeit sich bei verwendung von vier Threads auf einem System mit vier Kernen auf bis zu ein Viertel reduziert. Dies zeigt die Möglichkeit großer Leistungssteigerung durch den Vorgeschlagenen Ansatz.

Contents

List of Figures

List of Algorithms

Introduction

With growing wide area networks and the increasing usage of distributed systems new technologies were developed for data and message distribution, such as event-notification systems, also called publish/subscribe systems. Since most distributed systems are based on asynchronity and middleware for communication among the distributed parts as well as referencial decoupling, publish/subscribe systems are useful for and widely used in distributed systems. Especially in the age of the internet of things with mobile systems as cars, environmental sensors or mobile phones sending information about their status, current position and/or their surrounding environment, but also for high frequency trading at the stock market, event-notification systems are crucial for the systems work, reliability and success.

Current publish/subscribe systems consist of broker servers in the network, handling and optionally filtering the messages of the event-notification system. These systems achieve their goal, but produce a high delay due to analysing and resending each message in the broker servers. Additionally, the brokers have only an abstract view of the network, so a message might be sent multiple times over the same link. This makes the broker-based system use the bandwidth of the individual links suboptimal. In addition, as a result of processing the forwarding of the messages at software layer, the broker servers reduce the throughput of messages to the brokers' computational capabilities and add additional delay to the messages time to arrive at the according subscribers. With Software Defined Networking there is a technology to erase the problems with high latency, suboptimal bandwidth usage and suboptimal throughput. Software Defined Networking enables a controlling instance to manage the forwarding rules (flow rules) on its switches, while the switches built for Software-defined Networking (SDN switches) are able to forward messages according to flow rules at line-rate. This enables a publish/subscribe system to filter the messages in the net at much better link usage, therefore having a much better throughput than broker based systems, whilst maintaining message forwarding at line-rate.
Now the bottleneck still is the control plane of the network. This control plane has to manage all flow rules of the entire network and compute and deploy new flow rules or recompute flow rules with every new publisher or subscriber joining or leaving. This process takes some time, so the throughput of subscribe, advertise, unsubscribe and unadvertise requests is limited if processed on a single machine. However, the ability to adapt fast to these changes in the network is crucial to the systems success, especially with modern mobile devices constantly changing their location in the network.

In order to solve this problem, the processing of requests has to be parallelised by distributing the processing to multiple controllers or distributing even at a more fine grained level. The aim of this thesis is to change an existing SDN-based publish/subscribe system

based on a single controller, to have the controller receiving requests of clients, forwarding them to multiple threads processing them. The resulting system is expected to scale better with increasing load while still maintaining a centralized view of the network. This thesis compares different request forwarding/distribution strategies to computation instances based on the throughput and processing time of the request handling. It evaluates the throughput of requests compared to the non-parallelised system. Further possibilities are mentioned in chapter 5 Future Work.

Thesis Organization

The structure of this thesis is as follows: The first chapter introduces and illustrates current systems and principles this thesis is based on. The second chapter is about the purpose of this thesis and the problems it is aimed to solve. Afterwards the possible approaches to solve these problems are illustrated and the following chapter introduces the implemented system. The fifth chapter shows the evaluations done with the system, and the last chapter contains the conclusion and further possibilities and ideas for future work.

Chapter 1

Background and problem statement

In order to understand this thesis, the key concepts and technologies it is based on must be known. First of all, the principles of Publish/Subscribe systems, then how Software-defined Networking works. In addition we need to know how to use Software-defined Networking for a high performance Publish/Subscribe system and what advantages and troubles come with its usage.

1.1 Principles of Publish/Subscribe

In a Publish/Subscribe system a *notification service* manages the propagation of information, so called *events* between information sending participants and participants wanting to receive a specific part of the information possibly sent by other participants. This information propagation happens without direct connections between the participants, with the publish/pubscribe system handling the message forwarding in an asynchronous and decoupled way. Each participant is either a *publisher* or *subscriber*, has only a connection to the notification service and is unaware of other participants in the system. A publisher tells the notification service what kind of events it will send to the notification service. After this, a publisher can send events to the notification service, being sure it will handle them correctly. When a publisher does not want to send the events any more, it can tell the notification service that it will stop sending events.
A subscriber tells the notification service in which range of the possible events it is interested in. After that, a subscriber will wait for the notification service to forward events to it. When a subscriber does not want to receive events any more, it can tell this the notification service. The notification service is the central part of the publish/subscribe system. It has to handle all requests of participants and forward all events of the publishers to the respectively interested subscribers. In common systems, the notification service is one or multiple servers called *brokers* managing the system and distributing events from publishers to subscribers. In the system this thesis is based on, another approach is used, which will be explained later in this section. Since the notification service is the central part of a publish/subscribe system, it defines all interactions of the participants with the notification service. There are several interactions, a publish/subscribe system bases on:

Advertise By advertising, a publisher can tell the notification service that it will send events, and what kind of events these are.

Unadvertise This action is used by publishers who have already advertised, to tell the notification service they will no longer send events.

Publish This action is used by a publisher whenever it sends a new event to the notification service.

Subscribe By Subscribing, a subscriber can tell the notification service that it wants to receive events and what kind of events these should be.

Unsubscribe This action is used by subscribers who have already subscribed, to tell the notification service they do not want to receive the events they had subscribed to any more.

Notify This action is the forwarding of an event by the notification service to a subscriber, The subscriber gets *notified* and the event it receives is called a *notification*.

1.1.1 Topic-based Publish/Subscribe

One goal of publish/subscribe systems is to avoid unnecessary forwarding of events to paths to subscribers not being interested in these particular events. By this, unnecessary path-usage and usage of processing power in switches, brokers, other network nodes and participants is reduced. One approach for publish/subscribe systems to do so, is to provide logical channels, so called *topics*. Each topic has its own identifier which makes it possible for participants to subscribe to all events of a particular topic or advertise on a topic. The system can distribute events according to the topic they belong to, but can not further distinguish by their content. There is no more fine-grained filtering of events possible by using this method.

1.1.2 Content-based Publish/Subscribe

So far the structure of events was of no importance for understanding, but for understanding how a content-based publish/subscribe system can handle events, we need to know their structure at least in an abstract way. An event e consists of its attributes, and therefore is a set of attribute-value pairs $< attr_i, value_i >$. So an event
$e = < attr_0, value_0; attr_1, value_1; ...; attr_n, value_n >$
can be filtered by its content by filtering the values of its attributes. An advertisement or a subscription therefore is a tuple of the publisher *pub* or subscriber *sub* and a filter function f defining the scope of the attributes of the events the publisher will publish or the subscriber is interested in. Publish/subscribe systems filtering by the content of events are called *content-based* publish/subscribe systems.

This thesis is based on a specific content space representation model using *spatial indexing* proposed in [1]. Seeing the attributes of events as *dimensions* provides an accurate representation of each event's location in the possible space of attributes, the so called *content space*. So every event can be located in a content space having one dimension per attribute of the event. Figure 1.1 shows an example of an event $e_1 = < A, 20; B, 50 >$ and $e_2 = < A, 75; B, 65 >$ in a two-dimensional event space with the dimensions A and B ranging from 0 to 100. Spatial

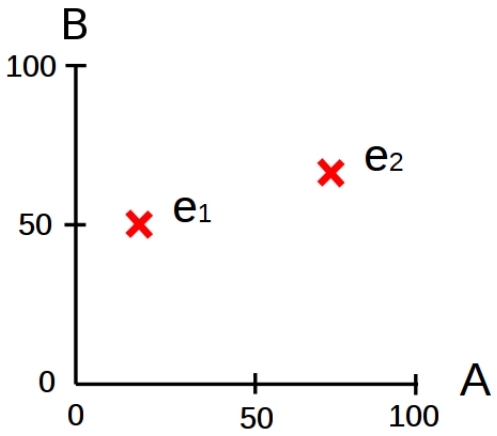

Figure 1.1: Two exemplary events e_1 and e_2 in the two dimensional event space of A and B

indexing partitions the content space by recursive binary decomposition, representing every sub-space by a binary string called *dz-expression*. An example of the decomposition of a two dimensional space can be seen in figure 1.2. By partitioning the content space this way, any degree of fine-grained partitioning can be achieved, only limited by the dimensions' resolutions or a maximum length of the dz-expression. This model is simple and useful, since a publish/subscribe system can use it as described in section 1.5.

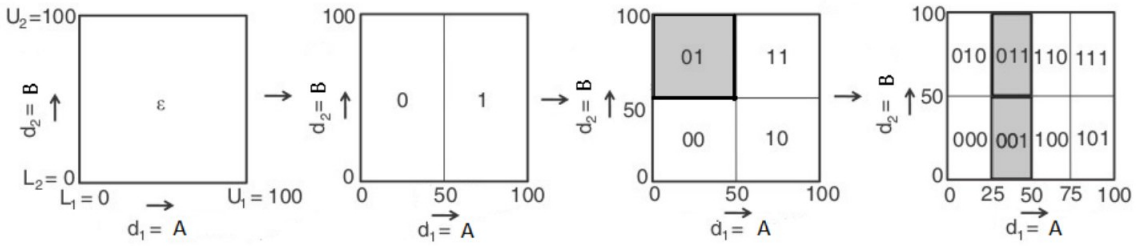

Figure 1.2: Partitioning of a two dimensional event space by spatial indexing with an increasing length of the used dz-expression, as described in [1]

1.2 Conventional Publish/Subscribe Systems

Common publish/subscribe systems like SIENA [2] and HERMES [3] implement overlay networks, which make clients connect to a server network consisting of so called broker servers. Some other implementations work without the use of broker servers, having the clients connect to each other but are not widely used. The components of this overlay network then manage requests and event forwarding at software level.

1.3 Optimizing Routing

Common publish/subscribe systems do have little to no knowledge of the underlying network they work on. This leads to inefficient link usage since clients do not send events to each other or to brokers using the shortest connection paths available. Brokers do not use the most efficient connections between them, either. To solve this problem, Tariq et al. [4] proposed algorithms for routing in overlay networks with the knowledge of the topology of the underlying network.

1.4 Software-defined Networking

With *software-defined networking* (SDN) a new network architecture emerged, separating the control plane from the data plane. Therefore the control logic can be somewhere else than on the devices in the network: namely on a controlling server, the *controller*, being connected to every switch. This controller and its behaviour can be customized and coupled with other applications to serve custom purposes. In contrast to classic approaches this enables the controller to run on a separate, powerful machine, route paths on its own and work with a full view of the whole network, rather than the distributed way used in common IP networks. Every SDN switch has one interface connected to the controller for receiving commands and reporting back, the so called *northbound interface*, being part of the control plane. All other interfaces of the switch are *southbound interfaces* connected to other SDN switches or host computers, making up the data plane network.
Most modern SDN switches have a specially designed memory (TCAM) for matching IPs and some other basic attributes of received packets against tens of thousands to hundreds of thousands of forwarding rules in a single CPU-cycle. This enables them to forward data packets at line-rate once the according forwarding rules (*flow rules*) are deployed on each SDN switch. So when there is a new route deployed in the network, there is only an initial delay for routing and deploying flow rules. After these initializing actions, packets on the deployed routes are forwarded at line rate without further delay.

1.5 Publish/Subscribe and In-network Filtering with the use of Software-defined Networking

The approach to realise publish/subscribe systems with broker servers spread all over a network is based on filtering of events at software layer. Thus each hop at a broker consumes a relatively large amount of time from receiving the packet until the packet is sent to the next node. Gagan Bihari Mishra [5], Koldehofe et al. [6] and [7] use spatial indexing to represent the content of an event in the target IP of a data packet containing said event. A part of the target IP address bits then contains the dz-expression representing the content of the event. By this, flow rules can be used for forwarding packets based on this representation of their content. This enables the switches to forward packets only on paths where there are subscribers interested in the containing events, with relatively few false positives. The amount of false positives is directly related to the length of the dz-expression, which is basically the bits used of an IP address for representing the events location in the event space. This method is called in-network filtering, since the network forwards according to the content which is represented by the IP address. This way events get filtered by the switches with every single forwarding job.

To ensure cycle free forwarding and guaranteed delivery of events to subscribers, [5] proposes to lay acyclic graphs on the network topology and use those for finding routes for forwarding data packets containing events. These acyclic graphs are created as trees, with the first publisher becoming the root node of the tree. This tree is then used for processing all requests in this partition of the event space covered by the publisher, represented by a dz-expression. If another publisher advertises for a larger subspace which covers one or more partitions for which trees are already existing, the affected subspace of this request is partitioned into multiple, adjacent ones, again represented by dz-expressions. The resulting set of partitions consists of the already existing, affected partitions, and new partitions. An example can be seen in figure 1.3 with a few routing trees existing for partitions, and the resulting partitions to be handled for a request covering a subspace that is only partially covered by a partition with an existing routing tree. Then the original request is handled as if it was multiple requests, each one advertising on a different one of those partitions.

After processing multiple advertisements, this results in many routing trees, each being used for routing events in the according partition. Additionally this does not only ensure cycle-free forwarding, but also that every event passes any link at most once. No event will be sent over a link multiple times which leads to more efficient link usage than routing through overlay networks.

1.6 Distributing Independent requests

Current publish/subscribe systems using overlay networks use distributed processing to get scalable event handling. The techniques used in the following systems can be used for this

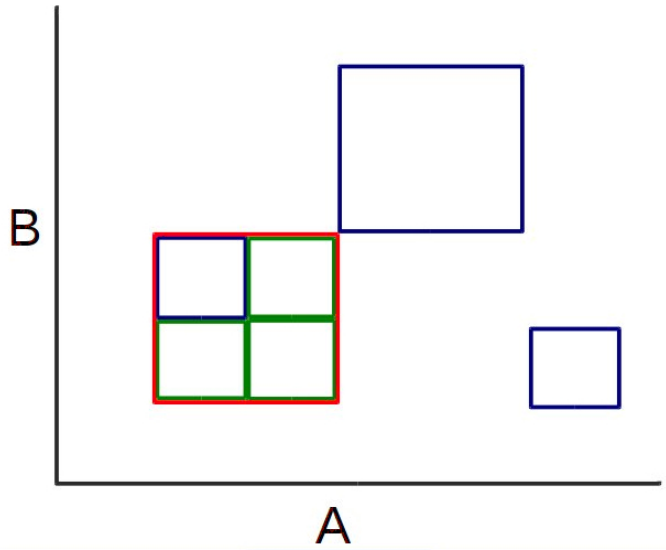

Figure 1.3: Subspace partitioning in a two dimensional event space with existing routing trees for some partitions (blue), request subspace (red) and created partitions for further routing trees (green) to handle the request.

thesis, too.

Bluedove The publish/subscribe system Bluedove [8] consists of dispatching servers and matching servers connected to participants via a physical network, creating an overlay network. All (un)advertisements, (un)subscriptions and events are sent by the participants to the dispatchers, which dispatch the events to the according matching servers. To be able distribute the matching of events against subscriptions, Bluedove uses partitioning of the content space. The content space gets divided into continuous, non-overlapping subspaces among which every matcher is responsible for matching events to subscriptions in one or more of these subspaces. According to each events content, it gets dispatched to a matcher being responsible for matching events in the subspace this particular event is located in. This matcher then matches the event to the subscriptions in this particular subspace and sends the event to all subscribers interested in it.

StreamHub Just like Bluedove, StreamHub [9] is a publish/subscribe system creating an overlay network with its participants and clusters subscriptions. In contrast to Bluedove, however, StreamHub uses pipelining and the three stages of "subscription partitioning" (dispatching to matching nodes), "publication filtering" (matching events to subscriptions) and "publication dispatching" (sending events to interested subscribers), each stage running on separate servers. This enables the system to be largely scalable, adapting to the load on each individual stage by adding servers to or removing servers from the stage.

1.7 Objective of this Thesis

Processing an (un)advertisement or (un)subscription request in the controlling instance of an SDN-based publish/subscribe system using spatial indexing for forwarding is depending on the flow rules already installed for previous requests. Such processing of a request can take much time when the network is very large or many subspaces of the content space (partitions) are affected by a single request. In addition, deployment of flow rules to switches takes additional time, especially since it includes the delay of the link between the controller and the switch. Processing requests in a sequential manner, partition for partition, and later deployment of the generated changes in flow rules to switches has a very limited scalability and results in overload with a sufficiently large network and amount of requests per time. This sequential processing has no ability to become reasonably scalable, but a publish/subscribe system needs to be scalable in order to serve its purpose in large networks and under heavy load. A positive side effect may be a possible decrease in the time until an advertisement or subscription is live, due to parallel processing having the chance to be faster than sequential processing.

Current approaches with the goal to make SDN-based publish/subscribe systems scalable relied on multiple controlling instances, each taking care of a part of the network and communicating with the controllers of all other subnets, see [10]. This distributes the processing load horizontally to multiple machines, but can not distribute the load of each subnet and has additional computational overhead due to synchronizing between the individual controllers. This thesis aims to compare different ways of parallelising processing of requests in publish/-subscribe systems based on SDN and spatial indexing while maintaining a global view of the network. It focuses on distributed computing of paths and flow rules, enabling vertical scalability of the system by taking advantage of additional resources available to a controlling instance.

Chapter 2

Possible Approaches

In order to process SDN subscription- and advertisement-requests in parallel, there are different approaches, each having it's own advantages and disadvantages. *Processing* in this context means *receiving and dispatching* requests to calculation instances, then *computing* the paths and generating an according new set of flow rules on a computation instance, and then *deploying* those changes onto the switches in the network using an SDN-controlling instance. This thesis aims on distributing of request processing in a publish/subscribe system based on spatial indexing, taking advantage of its properties. To remind: spatial indexing leads to non-overlapping partitions of the content space, as explained in chapter 1.1.2. The IDs of the resulting partitions are dz-expressions, which are used to identify a data packets location in the content space for forwarding purposes.
In order to be able to process requests or at least parts of requests in parallel, several data structures and operations have to be independent to the same operations on other partitions:

Network Topology The network topology is the same for all computation instances, it can not be altered by the system since it is a fundamental property of the network. It is only used for creating routing trees and gets updated only via changes in the network, which by now forces the system to start all over again and remove all data but the live advertisements, subscriptions and uncompleted requests.

Routing trees The partitions created by spatial indexing are independent (non overlapping) as described in 1.5, the network topology is read only and tree generation is done only inside each of those partitions. This enables tree generation to be done for each partition without affecting another partition's tree generation. It can work with a copy of the network topology, since if it changes, there will be a request to recompute all requests of this partition with an updated network topology, anyway. Routing bases on those generated trees, so it can be done independently on the tree of each partition to the routing on trees in other partitions.

Flow rule creation Flow rules are created according to the partition's routing tree and dz-expression, and even contain this dz-expression. So flow rules can be computed in one partition without interfering with flow rules created in a different partition. The changes to flow rules of a partition depend only on previous changes done to the flow rules of this particular partition.

(Un)Subscriptions/(Un)Advertisements Live advertisements and subscriptions need to be stored in the system to handle new requests. They can affect multiple partitions, but are never altered, only deleted when processing the corresponding unsubscription or unadvertisement. So when for every partition there is a set of the live advertisements and subscriptions affecting it, these sets can be separately stored and altered without interfering each other.

The dependency on the previously done changes to the data of a partition enforces that only one computation instance can work on the data of a partition at a time. But the independence of routing trees and flow rules to those of other partitions make it possible for up to one computation instance per partition to handle another part of a request in parallel, each affecting another partition. This leads to the ability of independent storing of the partition-specific data, i.e. the routing tree and flow rules. If copies of already processed and still active advertisements and subscriptions affecting the according partition, are stored together with the partition-specific data, only the network topology needs to be accessible for every computation instance. Since there are only read operations conducted on the network topology, synchronisation can be avoided by storing a copy of it for every partition or computation instance. It only needs to be updated when a change of the topology is handled.

These properties enable requests to be processed on every affected partition without interfering processing on other partitions, allowing parallel processing of requests on multiple partitions. To avoid unnecessary complex algorithms, the partitions for routing tree creation are initially defined, and the routing trees are created when processing the first advertisement request affecting this partition, using the switch the publisher is connected to as root node of the tree. To process requests on multiple instances, these instances need to synchronise or be synchronised on which instance processes requests on which partition either once, or all the time, depending on whether there is a fixed or dynamic responsibility of computation instances for partitions. This can be either a central controlling/dispatching instance or alternatively a distributed algorithm.
This is the central idea of this thesis for high-performance parallel processing of (un)subscription and (un)advertisement requests in a software-defined network. By parallel processing of requests on multiple affected trees the only needed synchronisation is to make sure there is at maximum one request processed at once on every partition. Some approaches may include a preprocessing to forward the requests according to the affected trees. These distributed responsibilities to independent parts of the events content space resemble the way, Bluedove distributes event matching responsibility to multiple matching servers, as described in 1.1.2.
Just like the processing of events in StreamHub, in this system requests can be processed in three stages: receiving and dispatching requests to computation instances, computing trees and flow rule changes, and deploying the resulted flow rule changes to the switches of the network. This facilitates to focus only on the middle part, with little dependencies to the other parts.

2.1 Approaches for Distributed Computing

Processing (un)advertisement- and (un)subscription-requests on multiple machines in a network can be done in different ways. The following section describes the different possible approaches of computing the according flow rules in a parallelised way.

Adaption of Multithreading with Centralised Memory The first approach is to have a central dispatcher containing a request queue and data storage manager, where every computation server receives requests and data from to be processed. After the processing is done, the altered data gets written back to the central storage and the computation server awaits new tasks.

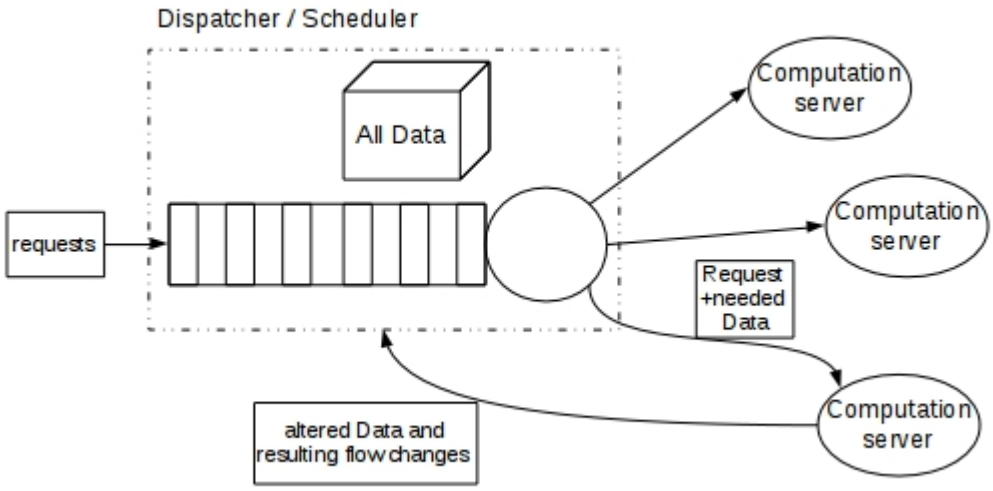

Figure 2.1: Distributed processing approach 1: adaption of typical multithreading

Fixed Distribution of Partition-specific Data with centralized Dispatcher The second approach is to have the data for processing requests distributed over multiple computation servers. The central dispatcher initializes the data on these servers and forwards the subscription and advertisement requests to the servers, either every request to every server or only to those servers holding the data for processing the particular request.

These two approaches can be merged to a third one, where the servers hold some of the previously used data in their caches and the dispatcher distributes the requests intelligently to

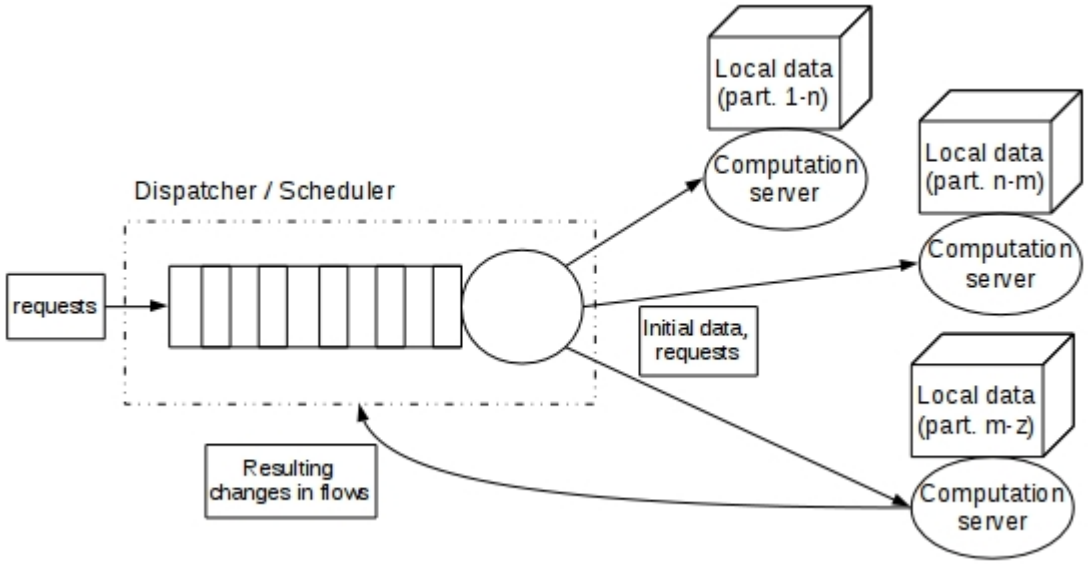

Figure 2.2: Distributed processing approach 2: disjunct data is held by the servers

those servers already holding the needed data. When data is not needed for further computations, or the computation server holding it is overloaded, it gets written back from the cache to the main storage for further use on other computation servers.

In addition, all these variants can be de-centralised with multiple SDN-controllers, see in Chapter 5 Future Work. This can be done by removing the central dispatcher, moving centralised data to all servers or a shared network cache or storage, and making all servers receive all requests. Depending on the system, additional synchronising servers on shared data if existing and which request processing might be needed.

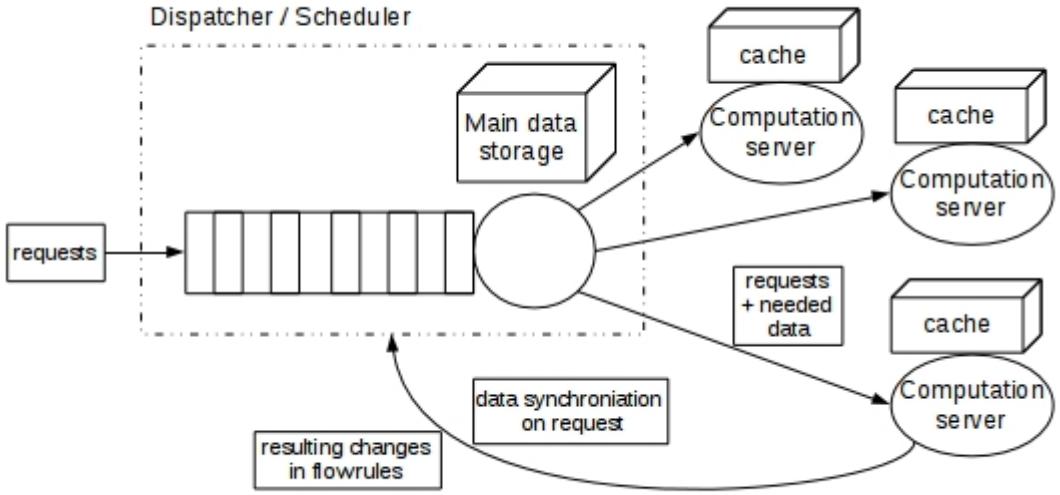

Figure 2.3: Distributed processing approach 3: merging of approaches 1 and 2

Chapter 3

Concept and Implementation

A simple approach was implemented by altering an existing implementation of the ideas of Gagan Bihari Mishra explained in 1.5. In this implementation, a standard Opendaylight [11] Controller with an additional plugin written in Java 1.7 realises the SDN-controller. In order to see possible effects without putting enormous effort in implementation resulting in multiple implementations of the system, the implemented solution is a single SDN-Controller which internally has multiple threads representing computation servers. This way, depending on the scheduling behaviour of the dispatcher, the Adaption of Multithreading and Fixed Distribution approaches can be simulated and evaluated against each other.
Initially the event space gets divided into multiple partitions by spatial indexing to create the partitions the computation threads will later work on. Then for every partition a waiting queue for requests awaiting computation, a set for storing the flow rules that are already deployed and a set for all subscriptions and one for all advertisements affecting this partition that are already live are initialized. Then the specified number of computation threads is started and the partitions are assigned to the computation threads.

If the system is in static assignment mode, a map is created for later looking up the responsible computation thread for each partition and every computation thread gets multiple partitions assigned by filling the map with tuples of partitions and computation thread identifier. When a request arrives, the dz-expression of its desired subspace is matched to the partitions existing in the system as shown in algorithm 1, and the request is put in the request queues of the partitions covered or partially covered by the request. Then the according computation threads get a request to work on a request on the according partition put in their internal request queue as shown in algorithm 2, by looking the threads up in the map created earlier. The behaviour of the computation threads is defined in algorithm 5

If the system is running in the lock-based dynamic mode, there is a lock created for every partition and an additional central request queue for all threads to see on which partition there are requests to process. When a request arrives, the dz-expression of its desired subspace is matched to the partitions existing in the system, and the request is put in the request queues of the partition covered or partially covered by the request. In addition, for each of these partitions there is a request to process one request on this partition put in the central request queue. The threads look in this central queue for the next request of which the partition lock is available, get the lock and compute a request of the corresponding partition's request queue, as shown in 5.

The algorithms used for request dispatching are as shown in 2 and 4.

Algorithm 1 Algorithm for finding partitions affected by a request.

```
1: function getAffectedPartitions(request)
2:     affectedPartitionss[]
3:     for partition in partitionList do
4:         if partition in request.getRequestspace then
5:             affectedPartitions.add(partition)
6:         else
7:             if request.getRequestspace in partition then
8:                 returnpartition                ▷ Request is only inside this partition
9:             end if
10:        end if
11:    end for
12:    returnaffectedPartitions
13: end function
```

Algorithm 2 Random assignment of event space partitions to computation instances

```
1: responsibleInstances < dz − expr, calcInstance >  ▷ Mapping partition to computation instance
2: partitions[]                                      ▷ Dz-expressions representing all partitions
3: requestQueues < dz − expr, queue >                ▷ Mapping partition to queue for requests
4:
5: function InitializeStaticAssignment(calcInstances[], newPartitions[])
6:     partitionList ← newPartitions.getCopy()
7:     while not newPartitions.isEmpty do
8:         for calcInstance in calcInstances do      ▷ assign one partition to every instance
9:             if not newPartitions.isEmpty then
10:                partition ← newPartition.getRandom()
11:                responsibleInstances.put(partition, calcInstance)
12:                newPartition.remove(partition)
13:            end if
14:        end for
15:    end while
16: end function
17:
18: function HandleRequest(partitionDz, request)
19:    affectedPartitions ← getAffectedPartitions(request)
20:    for partition in affectedPartitions do
21:        calcInstance ← responsibleInstances.get(partitionDz)
22:        requestQueues.get(partition).put(request)
23:        calcInstance.notifyNewRequest(partition)
24:    end for
25: end function
```

Algorithm 3 Computation thread code for partitions being assigned to computation instances

```
1: requestQueue
2:
3: function MainRoutine
4:     while true do
5:         ComputeRequestOnSubspace(thisThreadRequestQueue.getNextPartition())
6:     end while
7: end function
8:
9: function notifyNewRequest(partition)
10:    thisThreadRequestQueue.put(partition)
11: end function
12:
13: function ComputeRequestOnSubspace(partition)
14:    request ← thisThreadRequestQueues.get(partition).getNext()
15:    request.compute()
16: end function
```

Algorithm 4 Random assignment of requests to computation instances

```
1: instances[]                                              ▷ The computation threads
2: requestQueues < dz − expr, queue >                       ▷ The queue for requests
3: partitionsPending   ▷ The central queue for threads to look up partitions with pending
   requests
4: locks < dz − expression, lock >          ▷ Mapping partitions to locks for synchronization
5:
6: function HandleRequest(partition, request)
7:     affectedPartitions ← getAffectedPartitions(request)
8:     for partition in affectedPartitions do
9:         requestQueues.get(partition).put(request)
10:        partitionsPending.put(partition)
11:    end for
12: end function
```

Algorithm 5 Computation thread code for random assignment of requests to computation instances

```
1:  function MainRoutine
2:      while true do
3:          gotlock ← noLock
4:          while gotlock is noLock do
5:              gotlock ← centralQueue.next.tryLock()          ▷ non-blocking tryLock()
6:          end while
7:          centralQueue.removeRequest(gotlock.request)
8:          ComputeRequestOnPartition(gotlock.getPartition)
9:          gotlock.unlock()
10:     end while
11: end function
12:
13: function ComputeRequestOnPartition(partition)
14:     request ← requestqueue(partition).getNext()
15:     request.compute()
16: end function
```

Chapter 4

Evaluation

The performance gain aimed for in this thesis is a higher throughput of (un)advertisement and (un)subscription requests at computation stage. In this case especially the time a request has to wait until it is computed, and the overall time from incoming to process completion is of interest. Therefore the measure times are measured under different conditions. With these tests the behaviour of the implementation of a controller with multiple computation instances is evaluated.

Three metrics were measured when conducting tests:

Average Waiting Time The average time it takes for a request from being put in a waiting queue until the computation for it is started.

Average Computation Time The average time it takes for computing new flow rules for a request and possibly trees, too.

Average Processing Time The overall time from incoming until the computation finished and the flow rules are to be deployed on the network's switches.

Each test is run once each with the use of one, two, four and eight threads as computation instances. In the controller the event space is divided to 32 partitions, which are assigned randomly to the threads processing the requests, ensuring that each thread is responsible for the same amount of partitions.

With increasing amount of threads working in parallel on incoming requests, the average time requests have to wait for them to be processed is expected to decrease until there are more computation threads than cpu-cores on that machine. Then the calculation is expected to slow down due to cpu-scheduling of the computation threads, which also would lead to longer waiting times. The overall time it takes in average from incoming of an event until it has been processed is expected to show these characteristics, too.

4.1 Test Environment

The tests were conducted on a machine with 4 cores clocked at 2,66GHz and 4GB RAM, running a lightweight linux 64bit (Lubuntu) on which network components were simulated using mininet [12]. Java client programs were run On the virtual hosts for subscribing and advertising, depending on the test. For the tests, a network with a fattree topology consisting of 20 SDN-switches and 16 virtual host machines as showed in figure 4.1 was simulated.

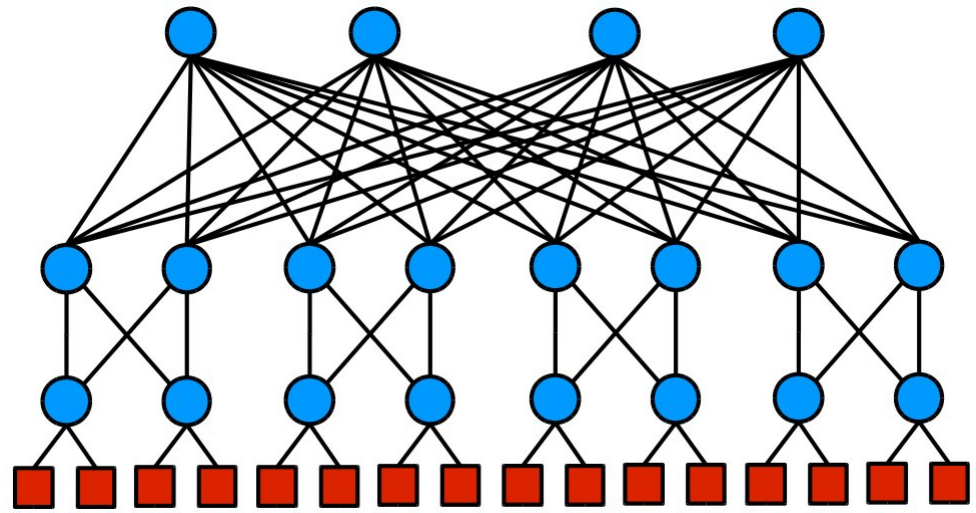

Figure 4.1: The fattree network topology as used for evaluations. Switches are represented by circles, client machines are represented by squares. The lines connecting them

To ensure computational load by a high chance of subscriptions and advertisements matching, all subscriptions and advertisements are in a one dimensional event space of natural numbers with the range of 0 to 100.

Each subscription, unsubscription, advertisement and unadvertisement request sent by a client to the controller gets internally divided to multiple internal requests, each according to one affected partition of the event space, as it is divided by spatial indexing. The partitions are initially assigned to the available worker threads, which are processing the internal requests affecting the partitions they are assigned. From now on, we only look at the internal requests, each affecting exactly one partition and resulting in one computation action. The data shown later is taken per test run from 1500 such internal requests conducted.

4.2 Tests Conducted

Advertising and Unadvertising For evaluating the system's throughput of advertisements and unadvertisements, on each of the 16 hosts a client is started, subscribing five times in a random part of the event space on every host, resulting in 80 subscriptions total. After that five client programs on every host are started to advertise and unadvertise as fast as possible

in a random part of the event space. This results in 80 processes running at the same time, alternatively advertising and unadvertising.

Advertising For evaluating the system's throughput of advertisements, with live advertisements accumulating, the test for advertisements and unadvertisements is slightly modified. Unlike in that test, now the client programs are made not to unadvertise and send out advertisement requests as fast as possible. So in this test, there are 80 random subscriptions and 80 client programs permanently sending advertisement requests.

Subscribing and Unsubscribing For evaluating the system's throughput of subcriptions and unsubscriptions, this test is a modified version of the test for Advertising and unadvertising. The controller is startet with the same configuration, but this test initially starts on each of the 16 hosts a client to advertise five times in a random part of the event space, resulting in 80 advertisements total. Then on every host five clients are started, each subscribing and unsubscribing as fast as possible. This results in 80 processes running at the same time, alternatively advertising and unadvertising.

Subscribing For evaluating the system's throughput of subscriptions, with live subscriptions accumulating, the test for subscriptions and unsubscriptions was slightly modified by making the client programs only subscribe, and not unsubscribe again. This results in 80 client programs sending out subscription requests as fast as possible and live subscriptions accumulating in the system.

Advertising and Unadvertising with lock based synchronisation The previous tests were based on partitions being assigned to computation threads in a static way. This test is run for evaluating the system's throughput of advertisements and unadvertisements with computation threads waiting for new requests in a fifo-like request queue, as described in chapter 4. So this test evaluates the throughput of advertisements and unadvertisements in a lock based implementation.

Subscribing and Unsubscribing with lock based synchronisation This test is the subscription and unsubscription counterpart to the previous test. It is run for evaluating the system's throughput of subscription- and unsubscriptions requests in a lock based implementation.

4.3 Computation Time

The time it takes for routing and flow rule creation is conceptually independent to the degree of parallelisation presented in this thesis, though there are not all flow rules stored together like it is done the original way, but only the relevant to the particular partition being worked on. This Nevertheless there are changes due to different access frequencies to common data and

the behaviour of the system's thread scheduler. The impact of the former can be seen in most graphs, the computation time decreases slightly with the increasing amount of threads working in parallel, resulting in more frequent access to common data. This effect is only visible, until the effects of scheduling overhead show up, especially when there are more computation threads than cores available for computing. At this stage due to the context switching overhead and the connected data movement, computation takes more time than with the use of less threads. Only for the subscribing and unsubscribing tests there seems to be an overhead showing due to the very short computation time.
Computing subscriptions as seen in figures 4.4, 4.5 and 4.7 is always much faster than computing advertisements as seen in figures 4.2, 4.3 and 4.6. But both are relatively fast and do not alter significantly with the amount of threads used for computing, compared to the change in average waiting times seen in the next section.

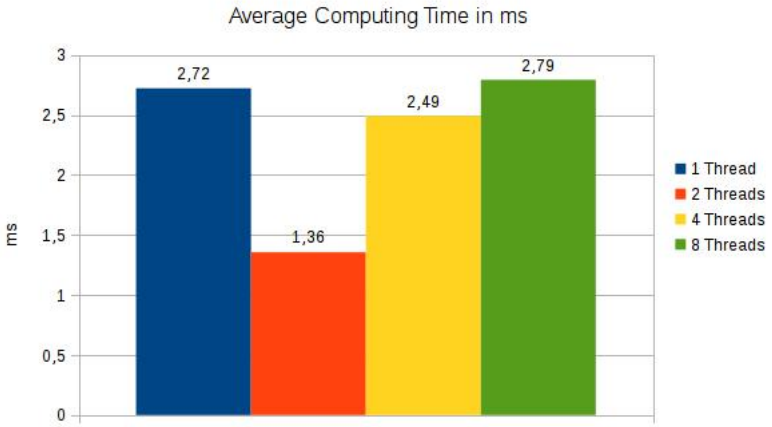

Figure 4.2: Average computing time for requests in the *Advertising and Unadvertising* test

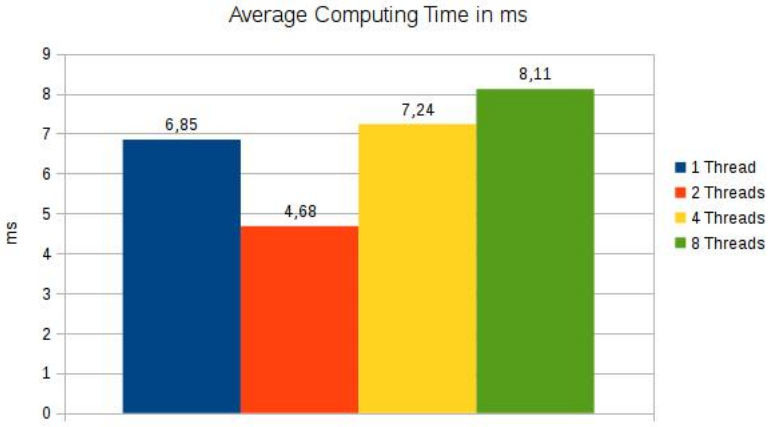

Figure 4.3: Average computing time for requests in the *Advertising* test

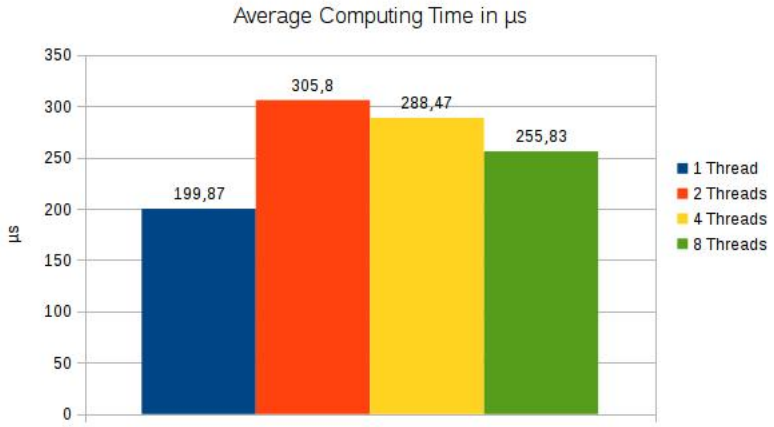

Figure 4.4: Average computing time for requests in the *Subscribing and Unsubscribing* test

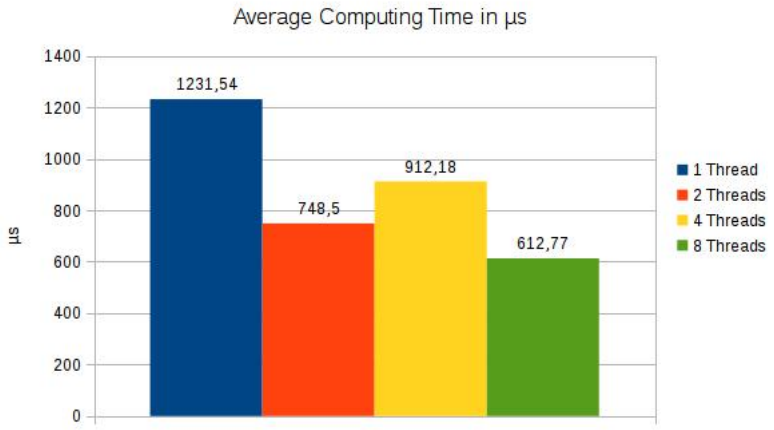

Figure 4.5: Average computing time for requests in the *Subscribing* test

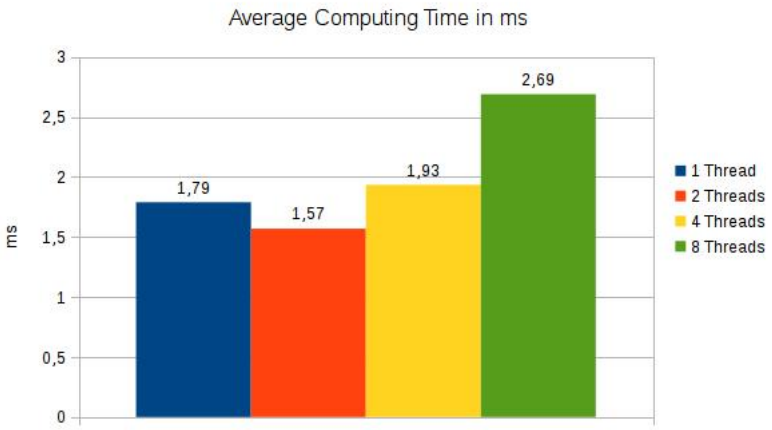

Figure 4.6: Average computing time for requests in the *Advertising and Unadvertising with lock based synchronisation* test

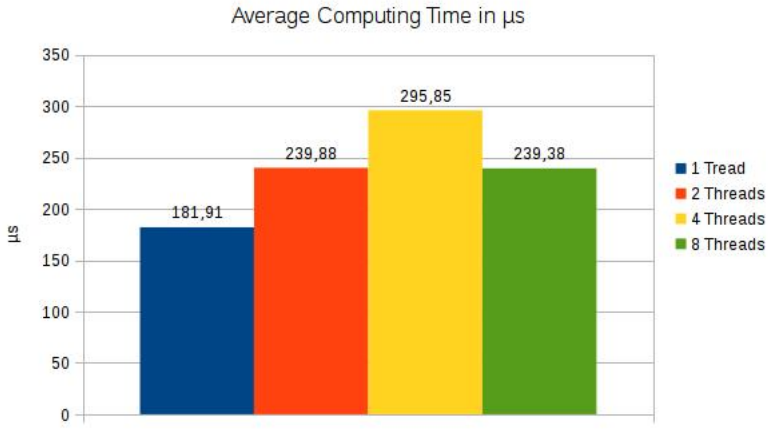

Figure 4.7: Average computing time for requests in the *Subscribing and Unsubscribing with lock based synchronisation* test

4.4 Waiting Time

The effects of parallel processing of requests show in the time, a request has to wait in average until it gets processed. Figure 4.8 shows how the average waiting time changes with the degree of parallelisation with a static distribution of partitions to threads and clients sending advertisement and unadvertisement requests. Figures 4.8 and 4.9 show that the CPU- and memory-intensive processing of (un)advertisement requests benefits most from parallelised computation. Figure 4.9 shows that by using four threads in parallel, the waiting time is reduced to even less than one fourth than when using only one thread with sequential computing, probably due to preventing of the request queue filling up.
The tests running (un)subscribing clients do show the same effect very limited in microsecond range, see figures 4.10 and 4.11.

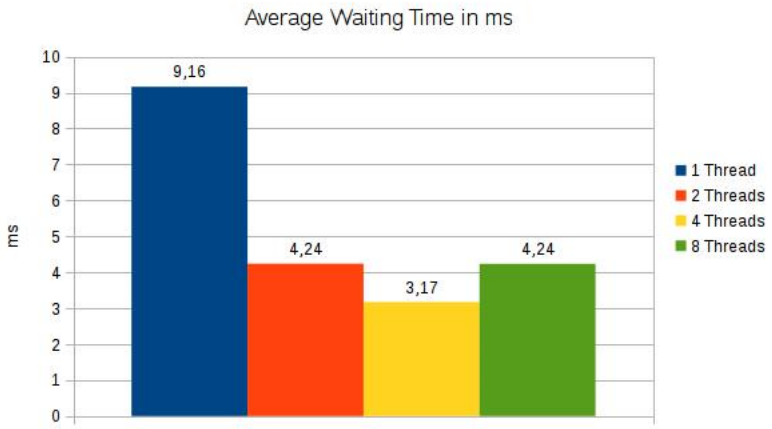

Figure 4.8: Average waiting time for requests until computation in the *Advertising and Unadvertising* test

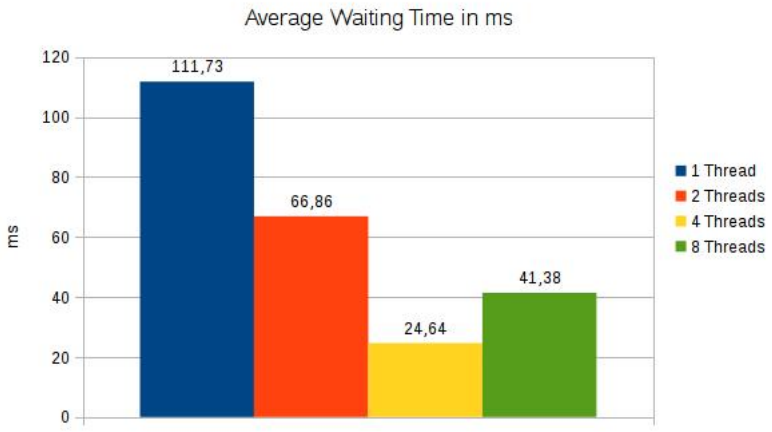

Figure 4.9: Average waiting time for requests until computation in the *Advertising* test

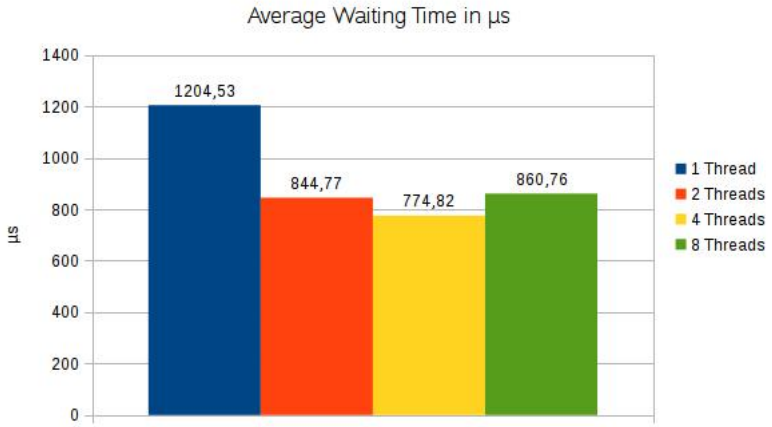

Figure 4.10: Average waiting time for requests until computation in the *Subscribing and Unsubscribing* test

4.5 Processing Time

The time determining the throughput of requests is the average time from submitting a request until all according changes in flow rules are computed. It is the combined average waiting time and average computing time. For advertisement handling the performance gain of the random partition distribution is large, as figures 4.14 and 4.15 show. The performance gain is not that big for handling (un)subscription requests, but is improved in a reasonable way, as seen in figures 4.16 and 4.17. The lock based approach does show the effects of parallelising when handling advertisements in figure 4.18, but its overhead eliminates any gains when handling subscriptions, as seen in figure 4.19.

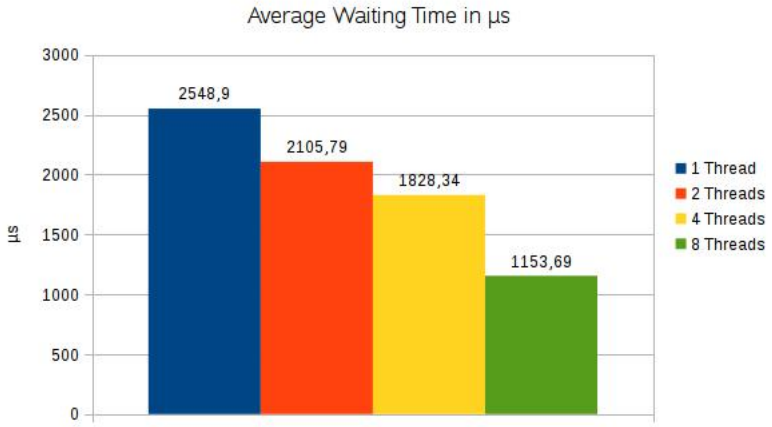

Figure 4.11: Average waiting time for requests until computation in the *Subscribing* test

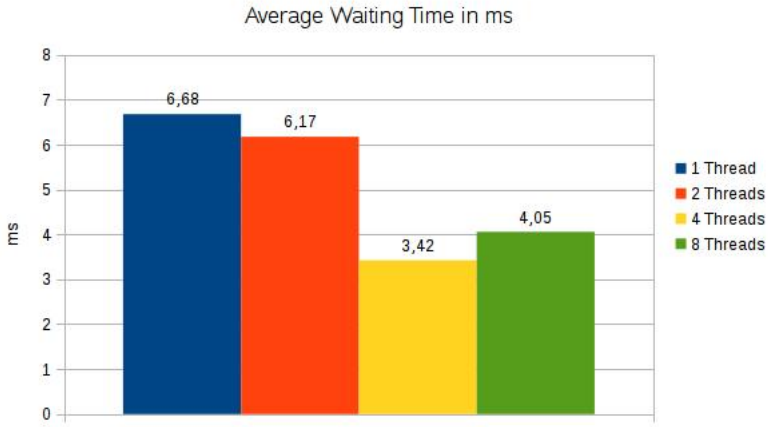

Figure 4.12: Average waiting time for requests until computation in the *Advertising and Unadvertising with lock based synchronisation* test

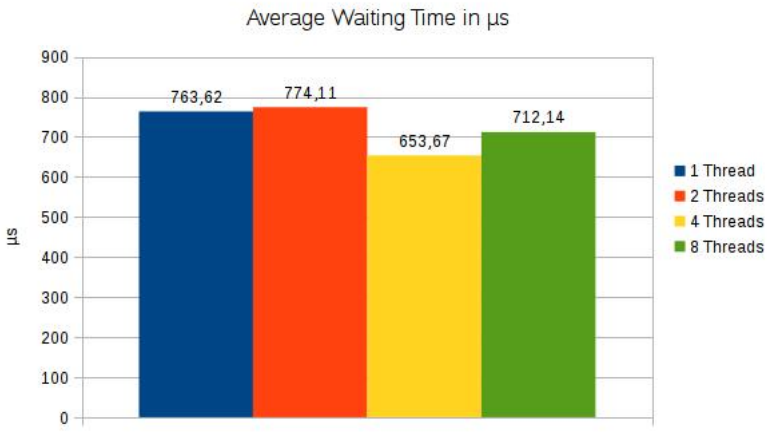

Figure 4.13: Average waiting time for requests until computation in the *Subscribing and Unsubscribing with lock based synchronisation* test

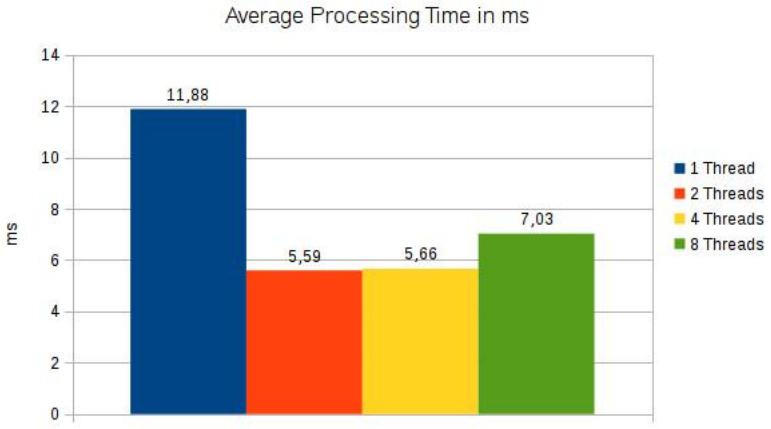

Figure 4.14: Average overall processing time for requests in the *Advertising and Unadvertising* test

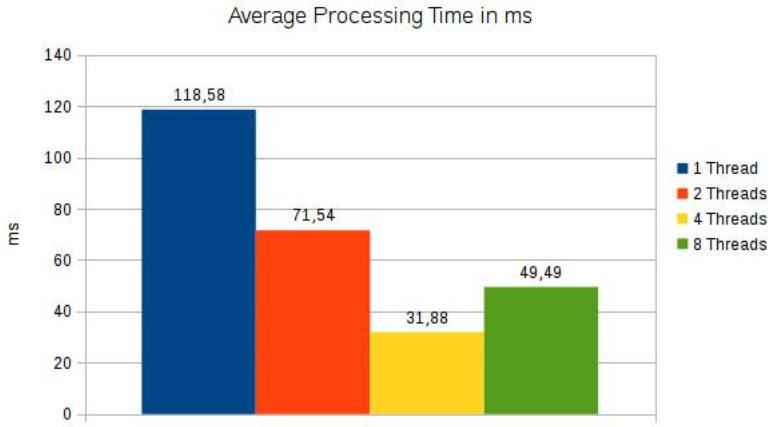

Figure 4.15: Average overall processing time for requests in the *Advertising* test

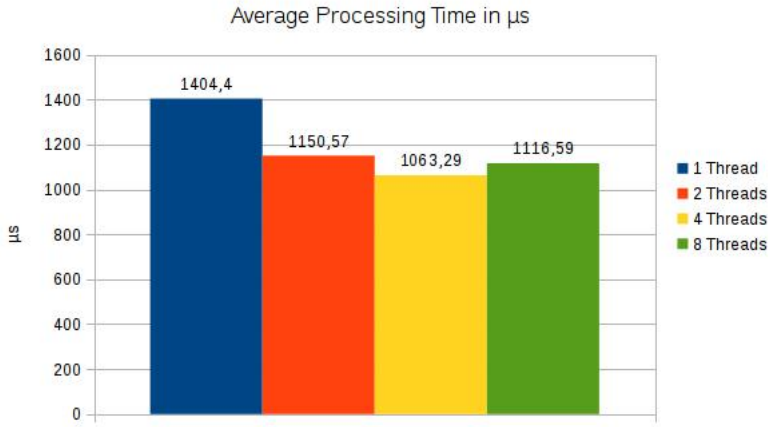

Figure 4.16: Average overall processing time for requests in the *Subscribing and Unsubscribing* test

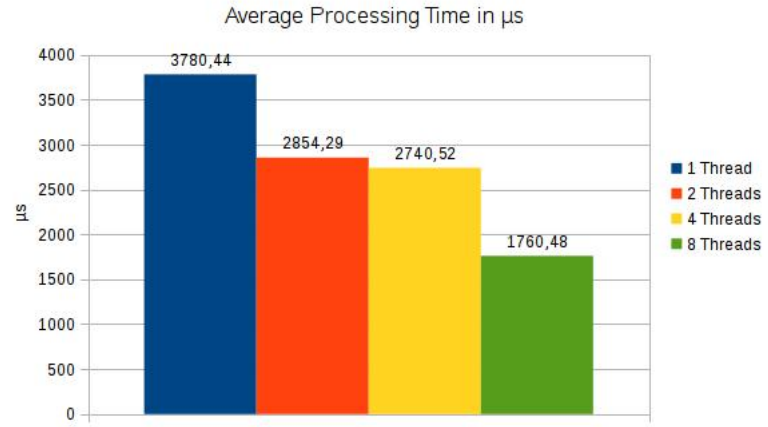

Figure 4.17: Average overall processing time for requests in the *Subscribing* test

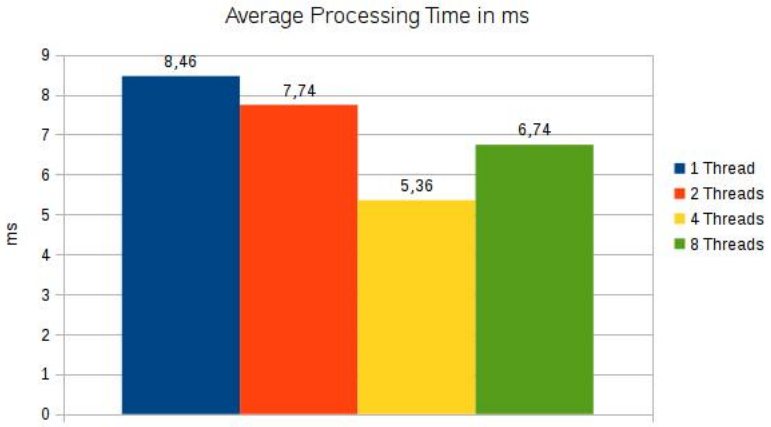

Figure 4.18: Average overall processing time for requests in the *Advertising and Unadvertising with lock based synchronisation* test

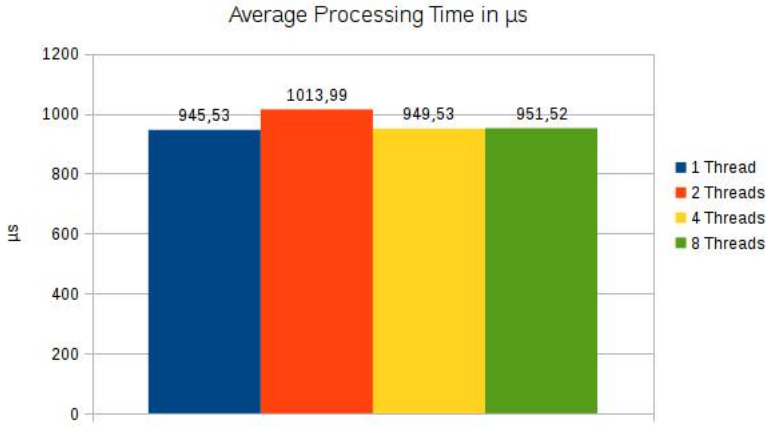

Figure 4.19: Average overall processing time for requests in the *Subscribing and Unsubscribing with lock based synchronisation* test

Chapter 5

Conclusion and Future Work

Parallel processing of requests as described in this thesis has the ability to decrease the processing time of requests heavily while producing little overhead, especially when computation of requests takes long. Though the lock based approach does not show that much performance gain, the basic approach shows the effectiveness of the proposed way to process requests in a parallel manner. Furthermore it shows possible performance gain. However, further research needs to be done regarding the behaviour when requests cluster at a few areas in the event space. In addition, the optimal performance depending on the number of partitions and computation instances needs to be evaluated. Scaling of parallel processing as described in this thesis with the network size may be optional to look at. Optimising initial event space partitioning and partition distributing strategies not evaluated in this thesis are to be evaluated, too.

Not only the computation of flow rules on different partitions can be done independently due to non-overlapping partitions and no interference of data. As described in chapter 2, flow rules of each partition are independent from the ones of other partitions and can therefore be deployed independently to those of other partitions. So the deployment of those flow rules onto the switches in the network can be parallelised without further synchronisation, too. There can be separate controllers for deploying flow rules, up to one for every single partition. This leads to the possibility of a much higher throughput of requests and a better vertical scalability of the system.
The same goes for receiving requests, there can be multiple controllers in the network for receiving requests from clients and dispatching them to computation instances, this forwarding of requests might even be done by in-network filtering. So the scalability of all three stages *request dispatching*, *flow rule computation* and *flow rule deployment* is to be evaluated.

Bibliography

[1] B. Koldehofe, F. Dürr, M. A. Tariq, and K. Rothermel, "The power of software-defined networking: line-rate content-based routing using Openflow," in *Proceedings of the 7th Workshop on Middleware for Next Generation Internet Computing*, MW4NG '12, (New York, NY, USA), pp. 3:1–3:6, ACM, 2012.

[2] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf, "Design and evaluation of a wide-area event notification service," *ACM Trans. Comput. Syst.*, vol. 19, pp. 332–383, Aug. 2001.

[3] P. R. Pietzuch, *A scalable event-based middleware*. PhD thesis, University of Cambridge, June 2004.

[4] M. A. Tariq, B. Koldehofe, and K. Rothermel, "Efficient content-based routing with network topology inference," in *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*, DEBS '13, (New York, NY, USA), pp. 51–62, ACM, 2013.

[5] G. B. Mishra, "Providing in-network content-based routing using OpenFlow," Master's thesis, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, June 2013.

[6] B. Koldehofe, F. Dürr, M. A. Tariq, and K. Rothermel, "The power of software-defined networking: Line-rate content-based routing using OpenFlow," in *Proceedings of the 7th international ACM middleware for next generation Internet computing (MW4NG) workshop of the 13th international middleware conference*, 2012.

[7] B. Koldehofe, F. Dürr, and M. A. Tariq, "Event-based systems meet software-defined networking," in *Proceedings of the 7th ACM International Conference on Distributed Event-Based Systems (DEBS)*, 2013.

[8] M. Li, F. Ye, M. Kim, H. Chen, and H. Lei, "A scalable and elastic publish/subscribe service," in *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pp. 1254–1265, May 2011.

[9] R. Barazzutti, P. Felber, C. Fetzer, E. Onica, J.-F. Pineau, M. Pasin, E. Rivière, and S. Weigert, "Streamhub: A massively parallel architecture for high-performance content-based publish/subscribe," in *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*, DEBS '13, (New York, NY, USA), pp. 63–74, ACM, 2013.

[10] S. Bhowmik, "Distributed control algorithms for adapting publish/subscribe in software defined networks," Master's thesis, Universität Stuttgart, Holzgartenstr. 16, 70174 Stuttgart, 2013.

[11] "Opendaylight sdn controller. project website." `http://www.opendaylight.org/`, 2013.

[12] "Mininet network emulator. project website." `http://www.mininet.org`, 2013.