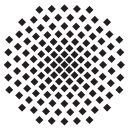

Universität Stuttgart


Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelor's Thesis Nr. 144

# Integrating Storage Components to Control Event-Rates in SDN-based Content Routing Networks

André Kutzleb

| | |
|---|---|
| **Course of Study:** | Softwaretechnik |
| **Examiner:** | Prof. Dr. Kurt Rothermel |
| **Supervisor:** | Dr. Boris Koldehofe |
| | M. Sc. Sukanya Bhowmik |
| **Commenced:** | 19.05.2014 |
| **Completed:** | 18.11.2014 |
| **CR-Classification:** | C.2.1,C.2.4,D.4.4 |

# Abstract

Software-defined Networking(SDN) makes it possible to build content-based routing systems which achieve line-rate performance. As recent work has shown, a content-based publish/-subscribe system can be built using SDN-switches instead of software-based brokers found in traditional publish/subscribe systems to forward events. These switches provide content-based routing just like the brokers in traditional publish/subscribe systems, however, they are operating on hardware and simply forward events according to their forwarding tables. While this eliminates the need for a network of software-based brokers and results in a significant performance boost, it also requires solutions for problems not existing or already solved in broker-based publish/subscribe systems.

One of these problems is the control of event rates. While content producers (publishers) in traditional publish/subscribe systems depend on brokers to forward their events, in a SDN-based system a publisher can simply publish at line-rate with no broker as intermediary to control the event rate - the switches will simply forward any event to the content consumers (subscribers) at a rate limited only by the forwarding capability of their hardware. Because of this lack of any kind of rate control, an additional mechanism is necessary in order to avoid the overload of subscribers.

This thesis provides a solution to this problem by placing event caches in the network which temporarily buffer events whenever a subscriber is overloaded and retransmit them at a controlled rate once it is possible to do so without overloading the subscriber. Additionally, an algorithm for finding good positions in the network to place caches at is proposed.

Evaluations of this caching mechanism show the feasibility of this approach for controlling event rates. These evaluations show that in some situations a reduction in event loss of more than 99.7% has been achieved with the introduction of caches.

# Kurzfassung

Software-defined Networking (SDN) macht es möglich hochperformante, inhaltsbasierte Routingsysteme zu realisieren. Kürzlich erschienene Publikationen zeigen dass ein inhaltsbasiertes Publish/Subscribe-System implementiert werden kann, dass SDN-Switches (statt wie in traditionellen Systemen dedizierte Server, so genannte *Broker*) verwendet, um Nachrichten zuzustellen. Diese Switches bewerkstelligen die inhaltsbasierte Zustellung von Nachrichten genau wie Broker, jedoch stellen sie diese Nachrichten einfach anhand ihrer Routing-Tabelleneinträge zu. Dies macht Broker überflüssig und hat eine Performanzsteigerung zur Folge, sorgt jedoch auch dafür, dass Lösungen für Probleme gefunden werden müssen, die in traditionellen Publish/Subscribe-Systemen entweder nicht vorhanden, oder auf eine Art und Weise gelöst worden sind, die sich nicht auf ein SDN-basiertes System übertragen lässt.

Eines dieser Probleme ist die Flusskontrolle. Während Publisher (Nachrichtenerzeuger) in traditionellen Publish/Subscribe-Systemen ihre Nachrichten über die Broker zustellen lassen (welche den Nachrichtenfluss kontrollieren können), so können in einem SDN-basiertem System die Publisher mit beliebiger Rate Nachrichten senden - die Switches leiten die Nachrichten einfach an alle Subscriber (Nachrichteninteressenten) weiter, wobei sie nur durch die Performanz ihrer Hardware eingeschränkt werden. Mangels jedweder Flusskontrolle ist ein Mechanismus nötig, der eine Überlastung der Subscriber verhindert.

Diese Arbeit stellt eine Lösung für dieses Problem vor, die die Platzierung von Caches (Speicher im Netzwerk) vorsieht, welche für den Fall, dass ein Subscriber überlastet ist, zeitlich beschränkt Nachrichten zwischenspeichern kann und diese mit einer kontrollierten Rate an die Subscriber nachsendet. Des weiteren wird ein Algorithmus vorgeschlagen, der geeignete Stellen für die Platzierung von Caches im Netzwerk ermittelt.

Im Rahmen dieser Arbeit durchgeführte Evaluationen zeigen die Machbarkeit des vorgeschlagenen Caching-Mechanismus zur Flusskontrolle. In bestimmten Situationen konnte eine Reduzierung der Verlustrate um mehr als 99,7% erreicht werden.

# Contents

# List of Figures

# List of Algorithms

# List of Tables

# Chapter 1

# Introduction

The ever increasing amount of interconnected and mobile devices proves to be a huge challenge in a world where networks were originally conceptualized as a rather static and hierarchical connection between mostly immobile computers. While the requirements to networks have changed significantly since the early days of networking, the protocols used for communication stayed largely the same. The success of the Internet caused many of these old protocols to be basically set in stone - changing them would require an update or replacement of every single network device.

One approach to adapt existing networks to these new requirements while still honoring well established protocols is software-defined networking (SDN)[1]. Using SDN, network devices (in the context of this thesis, multilayer SDN-switches) can be programmed to change their behavior to adapt to a changing environment. While existing network devices would still need to be upgraded or replaced to be SDN-capable, the prospect is that further changes can simply be applied by programming them into the switches using SDN.

A SDN-capable network can be used to improve on a widely used concept in traditional networks - content-based publish/subscribe. In contrast to synchronous communication between hosts in a network, a publish/subscribe system consists of publishers (producers of information), and subscribers (consumers of information). Information is exchanged in the form of events, which are published (produced) by these publishers and received (consumed) by subscribers. In *content-based* publish/subscribe systems, publishers publish certain content, and if a subscriber is interested in that content, relevant events are delivered to it asynchronously by the publish/subscribe system. Traditionally, a publish/subscribe system consists of dedicated servers (so called brokers) which are distributed in the network and directly communicate with publishers and subscribers. They receive events from publishers and forward them to the other brokers and eventually to the interested subscribers. These brokers have to scale with the amount of events that have to be delivered in the network, but not only that, they need to be as fast as possible to reduce the delay in which events are delivered. The brokers also have to constantly exchange state information with each other to correctly deliver events to relevant recipients. Performance issues resulting from this structure can be alleviated [2] [3], with recent work [4][5] demonstrating that a publish/subscribe system which is aware of the underlying network topology can significantly perform better than traditional systems. This approach of underlay-aware publish/subscribe eventually led to SDN.

Given the flexibility SDN provides, a SDN-capable network can be programmed to function as a content-based publish/subscribe system [6] [7]. The brokers previously required essentially become obsolete. One part of the broker's functionality is now done by the SDN-switches - they can provide content-based routing on hardware and therefore do not share the scalability problems of software-based brokers, being limited only by the maximum throughput of their hardware. However, these SDN-switches still need to be configured to forward events according to their content - this part is now done by one or more controllers (control servers that the SDN-switches connect to) which program the SDN-switches accordingly. Note that these controllers *do not* have to actively filter or distribute events, they only program the SDN-switches. The concepts of such a SDN-based publish/subscribe system are described in [8], while recent work such as [9] show its feasibility by providing an implementation and evaluating it. Further research went into distributing the control logic among multiple controllers to avoid bottlenecks resulting from having only a single controller to configure the SDN-switches, showing that the control logic can be scaled horizontally[10].

One significant functionality present in traditional publish/subscribe systems is still not covered by the controllers or the SDN-switches: the brokers can usually cache events, delivering them once the subscriber is able to receive them. This can become necessary in case a subscriber is currently off-line or overloaded. A SDN-based system with its ability to distribute events at line-rate speed can mercilessly overwhelm an unprepared or underpowered subscriber. The only option currently possible is to limit the rate at which the publishers may publish - however, this is a major drawback and unnecessarily restricts the possibility of line-rate transmission that such a publish/subscribe system can archive.

While the concepts of caching have been discussed for traditional content-based routing systems [11][**?**], no system exist which provides a solution for this problem in a SDN-based publish/subscribe system. This thesis focuses on proposing a design and the implementation of a system which uses caches in the network as buffers to mitigate the effects that event-rates too high to be handled by subscribers would otherwise have. The design should be completely transparent to users of the publish/subscribe system and should use the inherent advantages that SDN-based contend-routing provides to avoid additional strain on the network as much as possible. The implementation is evaluated using different usage scenarios and the results for latency, loss rate and other factors are compared to the system without any caching. Furthermore, different mechanisms for detecting overload on subscribers are described. Additionally, an algorithm that suggests these locations is proposed.

## 1.1 Thesis Organisation

The remaing parts of the thesis are organized as follows:

Chapter 2 provides necessary background information about:

- 2.1 Content-based publish/subscribe and its notation.
- 2.2 Software-defined Networking and an actual implementation of its concepts, the Openflow protocol.

- 2.3 The content-based publish/subscribe system that this thesis is based on. This system is closely related to [12] and [9].

Chapter 3 contains the problem statement and the system model. It also explains why the caching problem for SDN-based systems is unique by differentiating it from comparable content-based networking solutions.

Chapter 4 explains how the caching mechanism works and how it is implemented into the system explained in Section 2.3.

Chapter 5 is centered around finding good locations to place caches at. Different factors required for choosing a location are evaluated and an algorithm for finding such locations is proposed that takes these factors into account.

Chapter 6 evaluates the caching implementation proposed in Chapter 4 using different scenarios. The results of this evaluation are compared to the same scenarios without caching. It is also evaluated how many subscribers a cache can support.

Chapter 7 concludes this thesis and gives a summary of the work done and points out what future work can be based on this thesis.

# Chapter 2

# Background

This chapter is aimed at providing information necessary to understand the concepts in this thesis. In particular, it provides general information about content-based publish/subscribe and the application of this concept based on SDN.

## 2.1 Content-based Publish/Subscribe

Publish/subscribe is a concept for asynchronous communication. There are 4 elements in a publish/subscribe system: Publishers, subscribers, events and the event delivery service. Publishers and subscribers are both also known as *participants*.

The notation used to describe publish/subscribe here originates from [10] and is reused with slight changes. *Note that this is the notation for general content-based publish/subscribe. The actual and more restricted notation used for the actual implementation is introduced in 2.3.*

#### Event

*Events* are pieces of information that are published by publishers, then forwarded by the event delivery service, and finally received by subscribers.

A concrete event $e$ contains attribute-value pairs so that:

$event\ e = < attr_0, value_0; attr_1, value_1; ...; attr_n, value_n >$

Attributes are identifiers for certain content, while the value is the actual data of this content type.

#### Publisher

A publisher is an element in the network that *publishes* events. In order to be allowed to publish events, this publisher has to *advertise* its intent to publish at the event delivery service. This action is known as *advertising* and lets the event delivery service know that this publisher can now publish events. The message used to advertise is called an *advertisement*. If a publisher does not want to publish events anymore it can *unadvertise* its advertisement from the event delivery service. This message is known as an *unadvertisement*.

An advertisement is expressed like this:

*advertisement* $adv = < f, pub >$

where $f$ is a filter on attribute-values pairs that any published event $e$ from this Publisher has to satisfy ($e \sqsubset f$). This lets the event delivery service know what kind of events it can expect from this publisher.

#### Subscriber

A *subscriber* interacts in a very similar way with the event delivery service. A subscriber is a consumer of events. in order to receive events from the event delivery service, the subscriber has to *subscribe*. The associated message is a *subscription*. After subscribing was successful, the subscriber now receives events from the event delivery service. If a subscriber is no longer interested in events, it can *unsubscribe* from the event delivery service. This message is called an *unsubscription*.

A subscription is expressed like this:

*subscription* $s = < f, sub >$

where $f$ is a filter that expresses the kind of events that the subscriber is interested in receiving. Every event $e$ published has to satisfy $f$ in order to be delivered to it ($e \sqsubset f$).

#### Event Delivery Service

The *event delivery service* accepts subscriptions, unsubscriptions, advertisements and unadvertisements. It is the central part of the publish/subscribe system. Its goal is to deliver published events from publishers to interested subscribers. It acts as an abstraction for the underlying systems used to actually deliver the events. The service can handle multiple subscribers and publishers within the technical limitations of the concrete publish/subscribe system used.

## 2.2 Software-defined Networking (SDN)

Software-defined Networking (SDN) is an approach to enable network administrators to easily influence the behavior of their network devices without sacrificing the inherent performance advantages such hardware-based devices provide. The general idea is split the network into two parts: The *control plane* decides where and how data is sent in the underlying network, which is called the *data plane*.

Using SDN, a network can be easily reconfigured without the need of firmware updates or hardware upgrades. Also, the controller provides a global view of all components in the network, which allows, among other things, for easy detection of link or switch failures and simplification of routing. In summary, SDN allows a network to be easily adapted for a specific role instead of having traditional hardware that dictates a certain way of how it can be used, while not sacrificing the performance advantage of forwarding packets on hardware.

### Openflow

This thesis is based on a publish/subscribe system which is in turn based upon Openflow [13]. Openflow is a concrete implementation of the SDN concept. The control plane in Openflow consists of a dedicated network that connects each Openflow-capable Switch to a control server, a so called *SDN-controller*. Through this dedicated network, the switches supply the controller with information about the network, while the controller can modify the forwarding tables of the switches. The *data plane* in Openflow is the network where the actual data is transported.

There are three important features offered by Openflow:

### 1. Flow Rules

The controller can manipulate the forwarding tables of switches by installing *flow rules*. These rules decide what should happen with packets received by the switch. A single flow rule contains two parts: *Match fields* and *actions*.

### Match Fields

The match fields are filters for packets. These filters express what values must be present in certain fields of an incoming packet in order to satisfy this filter. Fields that as of Version 1.0 of Openflow can be matched are shown in Figure 2.1. Some fields allow matching against absolute values only, while others allow the use of wildcard-bitmasks. These fields are used to provide content-based routing which is explained in section 2.3 by mapping the content to these packet fields.

| Ingress Port | Ether source | Ether dst | Ether type | VLAN id | VLAN priority | IP src | IP dst | IP proto | IP ToS bits | TCP/UDP src port | TCP/UDP dst port |
|---|---|---|---|---|---|---|---|---|---|---|---|

Figure 2.1: Fields from packets used to match against flow entries, as of Openflow v. 1.0.

All flow rules on a switch do either need non-intersecting sets of match fields or in case of an intersection need to provide a unique numerical priority. The reason for this is that each incoming Packet is matched against all existing flow rules and only the following scenarios are allowed to occur:

1. No flow rule matches → depending on setup the packet is *dropped* or *sent to the controller*.
2. A single flow rule matches → The *Actions* for this flow rule are executed (see next paragraph).
3. Multiple flow rule match → The Actions for the flow rule that matches the most (if wildcard fields present) are executed, or, if the match is equal, use the flow rule that has the highest priority.

Note: This matching is usually done on Ternary Content-addressable memory (TCAM). This memory allows for near instant matching of packet fields against all flows and is therefore very fast.

#### Actions

When for a given packet a flow rule is matching, the actions of this flow-rule are executed. These actions can be chained and are executed in sequential order. All actions possible as of Openflow Version 1.0 can be seen in Figure 2.2. Usually packets are simply forwarded to a specific port, however much more detailed action chains are possible.

#### 2. Exchanging Packets between Control Plane and Data Plane

Packets can be explicitly (if stated in a flow rule) or implicitly (default behavior if no matching flow present) be forwarded by the switches from the data plane to the control plane and ultimately be received by the controller. There, the packet can be analyzed on the application level. This therefore allows for hosts to send any messages to the controller.

The controller can also construct packets on the application layer and then inject these packets into the data plane. This can be used to send messages from the controller to the hosts in the network.

Both capabilities combined means that a bidirectional communication between the controller and hosts in the data plane can be accomplished.

#### 3. Acquiring Traffic Statistics from Switches

Openflow switches allow polling of traffic statistics as detailed as traffic information for each single flow rule installed. In Table 2.3 it is shown what counters are accessible through the Openflow protocol as of Version 1.0. This data is easily accessible from the controller and can be used to monitor the traffic in the network.

The flow rules and the communication between control and data plane are used to build the publish/subscribe system described in the next Section 2.3, while the statistics can be used to detect overload in the network.

## 2.3 Using SDN for content-based Publish/Subscribe

This section briefly explains how the concept of publish/subscribe explained in Section 2.1 is implemented using the basics of SDN explained in 2.2. The described system is called HPMOM (High Performance Message Oriented Middleware), and is the concrete implementation of a content-based publish/subscribe system used in this thesis. HPMOM is very similar to the PLEROMA system [9] - however, HPMOM uses *categories* to distinguish content, which is a useful high level abstraction for explaining the caching mechanism.

| Action | Associated Data | Description |
|---|---|---|
| Set VLAN ID | 12 bits | If no VLAN is present, a new header is added with the specified VLAN ID and priority of zero.<br>If a VLAN header already exists, the VLAN ID is replaced with the specified value. |
| Set VLAN priority | 3 bits | If no VLAN is present, a new header is added with the specified priority and a VLAN ID of zero.<br>If a VLAN header already exists, the priority field is replaced with the specified value. |
| Strip VLAN header | - | Strip VLAN header if present. |
| Modify Ethernet source MAC address | 48 bits: Value with which to replace existing source MAC address | Replace the existing Ethernet source MAC address with the new value |
| Modify Ethernet destination MAC address | 48 bits: Value with which to replace existing destination MAC address | Replace the existing Ethernet destination MAC address with the new value. |
| Modify IPv4 source address | 32 bits: Value with which to replace existing IPv4 source address | Replace the existing IP source address with new value and update the IP checksum (and TCP/UDP checksum if applicable).<br>This action is only applicable to IPv4 packets. |
| Modify IPv4 destination address | 32 bits: Value with which to replace existing IPv4 destination address | Replace the existing IP destination address with new value and update the IP checksum (and TCP/UDP checksum if applicable).<br>This action is only applied to IPv4 packets. |
| Modify IPv4 ToS bits | 6 bits: Value with which to replace existing IPv4 ToS field | Replace the existing IP ToS field. This action is only applied to IPv4 packets. |
| Modify transport source port | 16 bits: Value with which to replace existing TCP or UDP source port | Replace the existing TCP/UDP source port with new value and update the TCP/UDP checksum.<br>This action is only applicable to TCP and UDP packets. |
| Modify transport destination port | 16 bits: Value with which to replace existing TCP or UDP destination port | Replace the existing TCP/UDP destination port with new value and update the TCP/UDP checksum<br>This action is only applied to TCP and UDP packets. |

Figure 2.2: Field-modify actions, as of Openflow v. 1.0

| Counter | Bits |
|---|---|
| Per Table | |
| Active Entries | 32 |
| Packet Lookups | 64 |
| Packet Matches | 64 |
| Per Flow | |
| Received Packets | 64 |
| Received Bytes | 64 |
| Duration (seconds) | 32 |
| Duration (nanoseconds) | 32 |
| Per Port | |
| Received Packets | 64 |
| Transmitted Packets | 64 |
| Received Bytes | 64 |
| Transmitted Bytes | 64 |
| Receive Drops | 64 |
| Transmit Drops | 64 |
| Receive Errors | 64 |
| Transmit Errors | 64 |
| Receive Frame Alignment Errors | 64 |
| Receive Overrun Errors | 64 |
| Receive CRC Errors | 64 |
| Collisions | 64 |
| Per Queue | |
| Transmit Packets | 64 |
| Transmit Bytes | 64 |
| Transmit Overrun Errors | 64 |

Figure 2.3: Required list of counters for use in statistics messages, as of Openflow v. 1.0

### Categories, Attributes and Events

HPMOM is a hybrid between a topic-based and a content-based publish/subscribe system. This means that an event cannot contain a freely chosen set of attributes - every event has to belong to a certain *category*. A category has a unique name (like a topic in topic-based publish/subscribe) and a fixed $n-tuple$ of attributes - in HPMOM, there are 4 possible attribute types available for events: *Long*, *Double*, *String* and *Boolean*. A *range* of possible values has to be present for each of the attributes as well.

The description below shows what ranges are possible for each attribute type:

**Long**
: Attributes of type Long allow values ranging from $Long_{min} = -2^{63}$ to $Long_{max} = 2^{63}$. A Long attribute of a category can restrict the values it is allowed to have, as long as this restriction is included in $[Long_{min}...Long_{max}]$, e.g. $[-100 \dots 100]$).

**Double**
: Attributes of type Double allow values ranging from $Double_{min} = 4.94065645841246544 \cdot e^{-324}$ to $Double_{max} = 1.79769313486231570 \cdot e^{308}$. A Double attribute of a category can restrict the values it is allowed to have, as long as this restriction is included in $[Double_{min}...Double_{max}]$, e.g. $[-1.0 \dots 1.0]$).

**String**
: Attributes of type String allow for any String literal as value. A String attribute of a category can restrict this to one specific string literal, e.g. exactly "Hello".

**Boolean**
: Attributes of type Boolean allow the boolean values "true" and "false". A Boolean attribute of a category can restrict this to either exactly "true" or exactly "false".

A Category in HPMOM can be expressed as:

$Category\ cat =$

$< categoryName; attr_0name, attr_0type, attr_0range; ...; attr_nname, attr_ntype, attr_nrange >$

As a result of the category concept, events also look a bit different than described in 2.1, as can be seen here:

$Event\ e =$

$< categoryOfEvent; attr_0, value_0; attr_1name, att_1value; ...; attr_nname, attr_nvalue >$

### Publisher and Subscriber

In HPMOM, an advertisement can be defined as

$advertisement\ adv = < f < cat >, pub >$

where $f < cat >$ is a filter on one of the categories currently existing in the system. Any published events $e$ from this Publisher have to satisfy this filter ($e \sqsubset f$). The filter is a restriction on the ranges of the category - just like attributes of categories may be restricted to a range within the allowed ranges of the attribute types themselves.

For instance, let's assume we have the following category in the system:

$Category\ L =< categoryName = "L"; attr_0name = "l", attr_0type = Long, attr_0range = [-100|100] >$

This category L has one attribute, which is of type Long and has the name "l". Valid values for "l" are in the range of $[-100|100]$.

Now, a publisher wants to publish events for category L, so he issues an advertisement:

$advertisement\ adv_L = < f < L >, pub >$

The filter $f < L >$ can restrict the ranges of values for which the publisher can publish events, for each of the attributes for category L.

We suppose that the publisher wants to publish events of Category L with values for "l" inside $[50|100]$. The filter therefore has to contain a restriction for "l" so that only these values are now allowed. The filter now looks like this:

$f < L >=< l = [50|100] >$

The events that the publisher can publish are always a subset of what is permitted by a category and its restrictions. This can range from very restricted ranges (a single value, e.g. $f < L >=< l = [50|50] >$ only allows events where $"l" = 50$) to no restriction at all (all values allowed by the Category, $f < L >=< l = [-100|100] >$ ).

The subscriber works in a very similar way. The subscription is defined as

$subscription\ s = < f < L >, sub >$

where $f < L >$ is a filter just like the filter explained for the advertisements. In this context, the filter can be used to express more detailed what events the subscriber is interested in.

For instance, if a subscriber is only interested in category L with values for $l = [50|100]$, it can express this with the filter $f < L >=< l = [50|100] >$.

### Architecture

HPMOM consists of 2 Parts. The Controller (a SDN-Controller with added publish/subscribe functionality), and Hosts in the SDN-Network that use the publish/subscribe API of HPMOM. Figure 2.4 shows the architecture of the whole system.

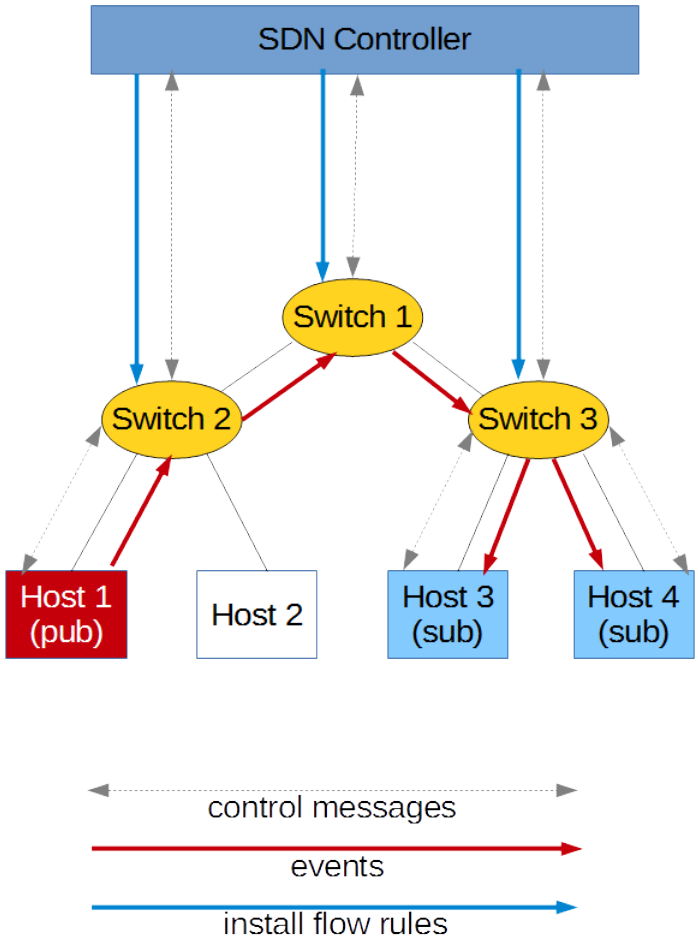


Figure 2.4: HPMOM system showing event flow from Host 1 to Host 3 and 4.

### Clients

Users of HPMOM use the ClientAPI to access the publish/subscribe functionality. The ClientAPI delivers control messages to the controller which handles the publish/subscribe logic by sending these messages with a certain IP to the network. The switches forward these messages to the controller because having been programmed to do so. When the controller needs to deliver a message to the ClientAPI, it injects that message to the switch closest to that ClientAPI which then forwards the message to that ClientAPI. This bidirectional communication is used for any messages relevant to HPMOM except for the events itself.

The ClientAPI itself offers the ability to:

- Subscribe
- Unsubscribe
- Advertise
- Unadvertise
- Receive events (given an active subscription)
- Publish events (given an active advertisement)

While the first four points are sent to and handled by the controller, publishing and receiving events works differently.

Events are published to a multicast IP address which has information about the content of the event encoded in it. The switches then forward the events to all relevant subscribers. This is described in more detail in the next paragraph.

### SDN-Controller

The SDN-Controller contains a module for publish/subscribe functionality. The main task of this module is to create distribution trees in the network for event delivery. This is done by installing forwarding rules on the switches which use longest prefix match on multicast IP-addresses to determine where events have to be sent to. Since each published event has information about its content encoded in its destination address, the switches can forward these events to the correct destinations. The details of how events are encoded and how the flow rules on the switches have to be implemented is thoroughly described in [12]. For the work done in this thesis, The abstraction of content-based on numeric values (Long-attributes) suffices to explain the caching mechanism.

# Chapter 3

# Caching Events in SDN-based Publish/Subscribe

This chapter is dedicated to the challenges that must be overcome to cache events in a SDN-based publish/subscribe system and to distinguish these challenges from those present in comparable, non-SDN-based systems.

Any traditional content-based routing system involving brokers (e.g. PADRES[14], SIENA[15], JEDI [16]), or content stores in the case of Content Centric Networking [17], store events at some point. This is because brokers/content stores are are software-based applications which communicate with participants in the network, meaning they can easily store events in volatile or non-volatile storage. A SDN-based publish/subscribe system differs the most from these traditional content-based routing approach in that the content delivery is done exclusively by hardware [6]. SDN-switches can do this delivery very fast, however, the switches do not have any knowledge that they are actually the brokers of the publish/subscribe system - they simply move packets from publishers to subscribers according to the entries in their forwarding tables. The reason why this is a very important difference is that the software-based brokers can relatively easy notice that a particular subscriber cannot keep up with a certain event rate - since this usually involves a queue that fills up at the broker. If such a scenario occurs, the broker can actively react to it.

A possible action by the broker is to queue the events as long as space for it is available, and then transmit the events once the subscriber can handle it again. If this kind of handling proves insufficient (like in a scenario where the event rate inevitably overloads the subscriber), the broker itself can simply stop to accept events from its source (preceding brokers and/or publishers). This will then cause their respective queues to fill up, causing their predecessors to slow the transmit rate, and so on. This kind of closed-loop flow control regulates the event transmission in a similar way than the TCP-protocol handles overload. This has both positive as well as negative side effects. If a subscriber simply stopped accepting events alltogether the network can be slowed to a halt if no other mechanisms are implemented to escape this kind of deadlock (e.g. individual queues for each subscriber with an absolute maximum of allowed events before that subscriber is terminated). This kind of closed-loop flow control is simply not possible using SDN-switches as they can only forward events without queuing them. Also, brokers in traditional systems can simply reduce the rate at which events are accepted by the first broker connected to a publisher, which allows to prevent overload situations in the first

place. In general, software-based brokers have the ability to make informed decisions, while switches *do not*.

While the focus in this work lies with the short term caching of events during overload, caching in related work [**?**] [11] mainly addresses the aspect of historic access of events. This means that subscribers can access events published in the past, which allows time decoupled event delivery. The solution proposed in this thesis can be adapted to provide this form of caching, but the focus and implementation clearly addresses overload situations.

## 3.1 System Model

The system model required by the proposed caching mechanism is a SDN and content-based publish/subscribe system with the following properties:

1. The communication between the controller and the subscribers as well as the communication between the controller and the caches is reliable (*exactly once* delivery of messages).
2. Subscribers are eventually able to receive all events intended to be delivered to them (no permanent overload situation unless the communication is intended to fail).
3. The underlying publish/subscribe system works reliably.
4. An additional category (see 2.3) can be introduced for each existing category that has to be cached.
5. Events are unique and can be differentiated from any other event ever published.
6. Event rates do not exceed the capabilities of the caches.

Given this System model, caching can be provided as described in the next section.

## 3.2 Problem Statement

Content-based publish/subscribe systems based on SDN currently lack the ability to control the rate of events inside the system. This however is required to avoid event loss caused by publishers publishing at rates exceeding the receive capability of subscribers.

This thesis aims to:

**Control the event rates to subscribers**
: In case a Subscriber is overloaded, caches present in the network will start to cache events relevant to this subscriber. The caches will retransmit the events to the subscriber at a controlled rate until caching is no longer required. More formally, the goal is that $\forall\ s \in subscriber \mid receiveRate(s) \leq maxSubRate(s)$ (No subscriber should receive more events/s than it is able to receive). Each event published and relevant to subscriber $s$

will be delivered *at most once*, meaning that the caching will not cause duplicates, but will not guarantee lossless delivery of events either. [1]

The mechanism does *not* guarantee:

1. *Exactly once* delivery of every event relevant to the subscriber.
2. Global chronological event delivery order.
3. Per-publisher chronological event delivery order.

**Find good Cache Positions**

To minimize the impact on the performance of the system and maximize the efficiency of the cache, an algorithm is proposed which presents favorable positions for caches in the network. The concrete goal of the algorithm is to order possible cache locations according to a cost function. This cost function provides a score [0.0 ... 1.0] for each cache location, expressing its quality as a position for a cache according to different metrics.

[1] The caching alone cannot avoid event loss altogether if the underlying system does not guarantee *exactly once* delivery of events, as is the case with the used publish/subscribe system, HPMOM. The proposed mechanism will still significantly reduce event loss, as shown in Chapter 6.

# Chapter 4

# Cache Mechanics

This chapter describes how the caching mechanism to control event rates can be implemented in the HPMOM system. A cache is a host in the network that will receive events from publishers and temporarily caches them for subscribers that are receiving too many events for them to handle. Once the subscriber can receive events without being overloaded by the rate of events they receive, the cache will retransmit the events to the subscriber at a controlled rate.

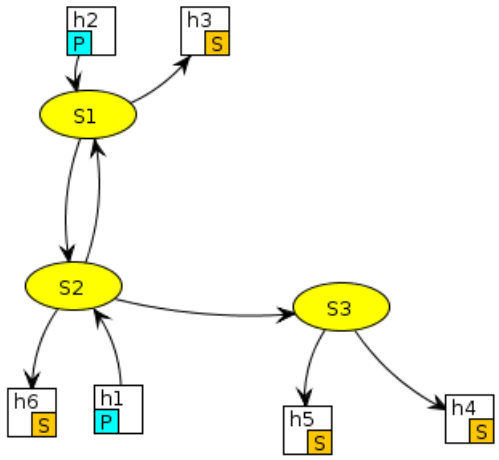


Figure 4.1: A publish/subscribe system with 4 subscribers, 2 publishers and 3 switches.

### Overview

A cache can take advantage of the fact that it is used in a content-based publish/subscribe network. When a cache is placed in the network, it will behave like a normal participant - it will subscribe and publish certain events.

In order to provide the caching service, the cache can first simply subscribe to every content that particular cache is responsible for. The publish/subscribe system will then send every events to every interested subscriber as well as to the cache, which - provided that the cache is not the only subscriber in the network - requires negligible overhead to do. As long as no one needs the services of the cache, it can just drop any events it receives. If a subscriber

is overloaded, its subscription will be paused to stop the overload. However, the cache still receives all events from the publishers. The subscriber can now tell the publish/subscribe system that it paused its subscription, since it was overloaded.

The responsible cache now comes into play - the cache is told that a subscriber is overloaded, in which case the cache will stop to drop all events that it receives - instead, it will enqueue any event it receives that matches the content the subscriber was interested in. The cache can now start to retransmit events from this queue to the subscriber at a controlled rate - again, using the pub-sub-system. Once the queue for the subscriber is empty, queuing of events for the subscriber and retransmission to the subscriber stops and the subscriber resumes its actual subscription, which restores the whole system to its initial state. Without any caches present, the publish/subscribe scenario used to explain the mechanics in this section looks like depicted in Figure 4.1.

In the following sections, the overload detection mechanism as well as details of the cache mechanism itself are described.

#### Restriction

As already mentioned in the problem statement, it is assumed that the cache is able to handle its own subscription without beeing overloaded itself. An overloaded cache could act in different ways in such a scenario. The controller could, for example, limit the publish-rates of publishers until the cache is no longer overloaded, or limit the publish rate pro-actively so that such a scenario may not even occur in the first place.

## 4.1 Overload Detection

This section explains the mechanic used to detect overload of subscribers. Detection of overload situations is important and needs to be done as quickly as possible - the faster caching starts, the fewer events are lost due to overload.

In order to know when a subscriber is overloaded, a threshold is needed. This threshold can be expressed at a maximum rate of events a subscriber may receive per second (*maxSubRate*). Any rate of events exceeding this *maxSubRate* will result in event loss. In order to detect overload, a very simple approach is to count the number of events that arrived during the last second. While this method is very simple to implement, it has two major drawbacks:

1. Assuming that the subscribers are passively monitored, overload situations may only be detected after a second has passed - since this is the moment when the arrived events are counted.

2. Events arriving during this second may be distributed in any way. If, for example, a subscriber has a *maxSubRate* of 1,000 events/s, These 1,000 events may exclusively arrive during the first 10ms of the measurement. While this still means that only 1,000 events/s have been received by the subscriber, the subscriber received events at a rate of 100 events/ms for 10ms. This rate of events may overload the subscriber temporarily.

To mitigate these problems, the implementation of the overload detection uses the same technique, but monitors the rate every 100ms instead of once per second. If the rate during these 100ms exceeds the maxSubRate for 100ms, the controller is immediately notified, which will then decide on how the overload should be handled. If the rate does not currently exceed the maxSubRate, the subscriber will only periodically notify the controller of its current load level.

## 4.2 Idle Cache

After having placed a cache in the network, the cache will register itself with the controller. The controller will then assign a responsibility for certain content to the cache. Now, the cache can subscribe to the content specified in the responsibility, for example, it will subscribe to $f < L >=< l = [-100|100] >$, meaning the cache is responsible for the whole category L. From the perspective of the publish/subscribe network, the cache is now simply another subscriber. This will now look like depicted in Figure 4.2. The black arrows show the normal paths events take from publishers to subscribers, while the green arrow shows the additional hop required for the events to be delivered to the cache. Any published events that the cache receives are dropped for now, since it currently has no reason to cache them. The cache will remain in this state as long as no subscriber is overloaded.

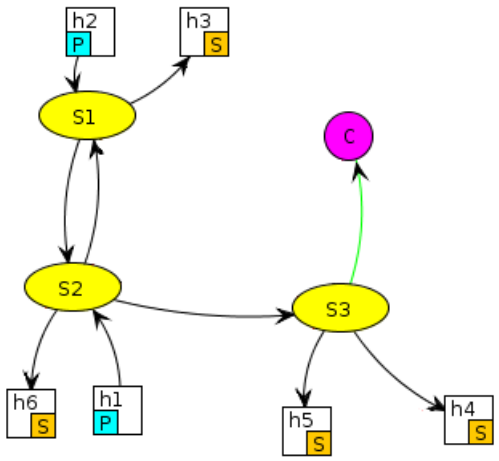


Figure 4.2: A publish/subscribe system with a cache.

## 4.3 Active Cache

As soon as one subscriber (e.g. having a subscription for L with $f < L >=< l = [0|25] >$) is overloaded by receiving too many events, it informs the controller about this. The subscription

of the overloaded subscriber will be paused for the time being - the subscriber will therefore no longer receive events. Having a global view about the participants of the network, the controller will now signal the responsible cache(s) which subscription needs caching. The cache(s) will now install a filter that matches the subscription of the overloaded subscriber($f < L >=< l = [0|25] >$), queuing all events they receive and that are matching this filter. The cache(s) will then begin to retransmit events to the subscriber from this queue at a controlled rate, by using another category, namely $L_{cached}$. If the cache would simply retransmit using an advertisement for Category L with $f < L >=< l = [-100|100] >$, not only a loop would result where the cache feeds itself new events, but the retransmission would also cause duplicates in the network - any other subscriber interested in the same content as the overloaded subscriber would receive the same event at least twice - at first from the original publisher, and the second time or more often from caches retransmitting them. This is why for each category in the system, a cached category variant has to exist. This is the original category with an added Long attribute "subId" that is used for addressing specific subscribers.

For the

$Category\ L =< categoryName = "L"; attr_0name = "l", attr_0type = Long, attr_0range = [-100|100] >$

the cached variant is

$Category\ L_{cached} =< categoryName = "L_{cached}"; attr_0name = "l", attr_0type = Long, attr_0range = [-100|100], attr_1name = "subId", attr_1type = Long, attr_1range = [0|1000] >$

All Subscriptions are required to subscribe to the relevant cached category as well - with the same filter used for their respective original subscription, and using a unique number for the subId which is provided by the controller.

In our example, the subscriber having a subscription for category L with $f < L >=< l = [0|25] >$, also has a subscription for $L_{cached}$ with $f < L_{cached} >=< l = [0|25], subId = 1 >$, provided that the subscriber has been assigned the id 1.

The cache also has an advertisement for that category - when it got its responsibilities assigned, beside subscribing to category L with $f < L >=< l = [-100|100] >$, it also advertised for $L_{cached}$ with $f < L >=< l = [-100|100], subId = [0|1000] >$.

Based on this, the cache can now take events from the queue for the subscriber of category L, convert them to events of category $L_{cached}$ (using the subId of the subscriber for the attribute "subId"), and then publish them to its advertisement for $L_{cached}$ with $f < L >=< l = [-100|100], subId = [0|1000] >$. Since the subIds are unique, the publish/subscribe system will take care that this event will only be delivered to the subscriber with that subId. In Figure 4.3 the red arrows display this additional distribution network required for this retransmission. It spans from the cache to every subscriber.

During retransmission, the controller will keep monitoring the current load on the subscriber and adjust the transmission rates if necessary with the goal to have the caches transmit as many events as possible without causing overload on the subscriber.

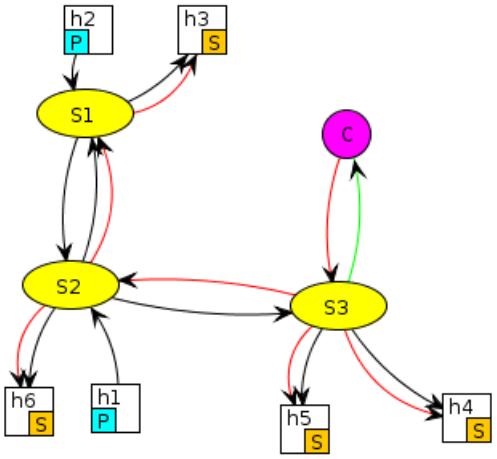


Figure 4.3: A publish/subscribe system with a cache showing event retransmission paths.

For as long as the the cache has events to transmit, it will remain in this state and keep retransmitting events to the subscriber(s) - unless too many events pile up, in which case a failure state triggers. this is explained in the next paragraph.

#### Failure State

The retransmission is supposed to happen at the fastest rate possible without overloading the subscriber - and this is for a very good reason. Suppose that the constant rate of event relevant to a subscriber causes 80% load on the subscriber. A peak in events then causes overload and therefore the subscription to be paused and caching to be activated. The cache(s) must now retransmit with a rate that is as close as possible to 100% (but at least 80%) of the subscribers allowed load - else the cached events at the cache will keep piling up until the queue for this subscriber is full. There may also exist scenarios where the subscriber will never be able to keep up, at which point the caching just postpones the inevitable: the failure of the subscriber. In any case, there will be scenarios where the cache must make a decision - it cannot enqueue new events forever, there must be an upper limit of how many events can be enqueued in total for a cached subscription, as well as a well defined reaction to breaches of this upper limit.

Once the queue for retransmission is full (essentially stating that this subscriber will - under the current circumstances - never be able to receive all these events), the cache signals the controller that the subscription in question is beyond saving, and verbosely terminates the subscription so that the subscriber is made aware that it cannot cope with the current event rate. This behavior does not introduce problems on its own - consider the same network without a cache. In that case, the subscriber would have failed more rapidly since it had no chance to ever keep up with the event rate in the first place. The cache is just delaying the inevitable - however, if the event transmit rate is lowered, the subscriber has a chance to recover thanks to the cache, whereas it would definitely have lost events without it.

## 4.4 Synchronizing Cache

A very important decision to make is when the subscription can keep up with the uncached event rate again. One use-case could actually be to never stop the caching - in theory, a subscription could just indefinitely be served by the cache, which has its advantages - it then acts as a constantly active cache protecting the subscription from overload. However, since the caching costs the cache effort and causes millisecond delay in event delivery as shown in Chapter 6 - its in general desirable to go back to the initial state in which the cache is idle and the subscription receives its events uncached. A good point to stop the caching is when the event queue at the cache runs empty once or multiple times. The actual implementation stops the caching when the queue was empty more than 50% of the time when attempting to take an event for retransmission in a period of 100ms.

There are some problems that can occur while transitioning to the original state. When the cache stops queuing events for retransmission before the original subscription restarts, events may be lost and vice-versa. Even if both could act at exactly at the same time, the network may currently be transmitting events that then may be lost nevertheless. One way to circumvent that is to have a small period of time where both the cache and the subscription are active. This means that the subscription restarts and then waits some time $t$ that ensures that by the end of $t$, any event in the network must have arrived. it is then save to stop the caching - however, during this period $t$, duplicates must be identified and sorted out, since both the cache and the pub-sub-network may be delivering the same events to the subscription. Given a scenario where the subscriber is at its limit load-wise, this period where duplicated events arrive may in turn cause the subscriber to overload again, which would then cause the caching to restart - which would cause the system to be trapped in this loop of trying to transition to the original state and trying to provide caching. To avoid this problem, the simple transition method was deemed acceptable for this thesis, even if it causes minor event loss during transitions.

# Chapter 5

# Cache Placement Strategy

While Chapter 4 described the mechanics of how caching works, this chapter is dedicated to finding good positions for the caches in the network. Finding a good position for a cache can have a significant impact on the network performance - a randomly chosen position, for example, may cause events to be sent across the network and back, depending on where the subscribers and publishers reside and what content they publish.

A publish/subscribe system may constantly change during runtime (publishers/subscribers can come and go, new content is introduced while old content is not being used anymore, event rates may change at any time, as well as the content of events). This means that in theory, caches would have to be constantly moved from one point to another to always perform optimally. A dynamic algorithm taking these runtime changes into account may justify a thesis on its own - the algorithm proposed in this section aims at finding good cache positions independently of the runtime behavior of the system. While these positions may prove suboptimal for some scenarios, the algorithm is expected to provide above average quality positions for caches in most scenarios.

## 5.1 Calculating Position

In order for the algorithm to produce meaningful positions, the following information must be available:

- The current topology of the network (hosts, switches and links between them)
- The maximum rate at which a host can publish events (if it would act as a publisher)
- The maximum rate at which a host can receive events (if it would act as a subscriber)
- The maximum capacity of the links in the network in events/s

Algorithm 1 considers each switch to be a viable position to connect a cache to. In order to generate a score for each of these positions, a number of publish/subscribe scenarios are to be simulated. In this example, a total of 1000 *hoststates* are randomly generated in Algorithm 2. Each scenario assigns to each host the role of a publisher and/or a subscriber (or neither). Each publisher publishes all event to all subscribers. These hoststates can be interpreted as concurrent usage scenarios for different types content at the same time, or as generalized

usage scenarios during different points in time - all in all it is supposed to be an abstraction of how the system may deliver events during runtime.

**Algorithm 1** calculateCachePositions

This algorithm calculates favorable positions to place a cache in the network.

```
1: var hoststates array [1...1000]:= generateRandomStates()
2: var scores
3: for ∀ state hoststate ∈ hoststates do
4:
5:    for ∀ switch s ∈ network do
6:       place cache c at s
7:       var float score = calculateScore(hoststate)
8:       scores.addScoreFor(s,score)
9:       remove cache c from s
10:   end for
11: end for
12: return scores
```

**Algorithm 2** generateRandomStates

This algorithm generates an array of host states. Each state makes each host in the network randomly either a publisher, a subscriber, both publisher and subscriber, or neither publisher nor subscriber.

```
1: var scenarios := 1000
2: var hoststate array [1..1000][] states
3: for i := 1 to scenarios do
4:
5:    for ∀ host h ∈ network do
6:       states[i][h] = chooseOneOf(publisher,subscriber,both,neither))
7:    end for
8: end for
9: return states
```

Using this information, for each possible cache position and each hoststate simulations are run to find out how the effects on various metrics are. For each hoststate, a score between 0.0 (terrible position) and 1.0 (perfect position) is calculated for the given position. Finally, after all simulations ran, the caches can be ordered by the average score they achieved in these simulations. The result is a list of cache positions (switches) in descending order representing the predicted quality of the cache position.

## 5.2 Calculating the Score

Algorithm 3 does calculate this score. A total of 5 different simulations(Algorithm 5, 6, 7, 8 and 9) are run inside this algorithm, each returning an individual sub-score. Each of these

Table 5.1: Variables used for calculating the different sub-scores.

| property | meaning |
|---|---|
| maxPubRate | maximum events/s published by a publisher |
| globalMaxPubRate | sum of maxPubRate of all publishers |
| linkUsage | current usage of a network link (depending on current simulation) |
| maxLinkRate | maximum capacity of a network link |

sub-algorithms provide different metrics for how the cache influences the publish/subscribe system depending on its position. Table 5.1 shows the variables used in these 5 simulations that require further explanation.

### 5.2.1 Algorithm: calcPubCacheFlowScore

The Algorithm 5 calculates the additional flow rules necessary to forward events to the cache and converts this number into a score. Since the cache is a subscriber for the content that it provides caching for, the publish/subscribe system must make sure that the cache receives these events by installing additional flow rules on the switches in the network. If, for example, the cache would be placed far away from the current distribution tree for events, a large amount of additional flow rules may be required to reach it.

The score calculated by this algorithm therefore represents one part of the impact (the other part being is represented by the Algorithm 6) placing a cache at a certain switch has on the amount of flow rules required.

### 5.2.2 Algorithm: calcCacheSubFlowScore

Since the cache has to be able to retransmit events to any relevant subscriber, it advertises for the content it provides caching for. All subscribers which may require caching for this content subscribe to this content. This means that a secondary distribution tree for events has to be installed on the switches in the network, spanning from the cache to every relevant subscriber. This may require a substantial amount of additional flow rules to be installed on the switches in the network. The amount required may highly depend on the location of the cache. The Algorithm 6 calculates the amount of flow rules required for this additional distribution tree, and turns the result into a score. Together with the Algorithm 5, this algorithm represents the impact on the total flow rules required depending on the cache location.

### 5.2.3 Algorithm: calcMaxPubMsgPerSecScore

Each event published has to be delivered to every interested subscriber. Assuming a network consisting of a publisher, a switch, and a subscriber, each event would appear twice in the network - once on the link from the publisher to the switch, and once on the link from the switch to the subscriber. If the distance between publisher and subscriber increases (by having

more switches in between), the event will appear once more for each additional hop required. Now, given multiple subscribers, At some point in the network the event has to be duplicated in order to reach all subscribers.

This means that a single event may appear multiple times in the network. Now, assuming that all publishers publish at their *maxPubRate*, the total amount all of these events will appear in the network can be calculated. Placing a cache in the network is bound to increase this value, since at least one additional hop for all events is required in order to reach the cache (since it acts as a subscriber). The Algorithm 7 calculates the total amount all events will appear in the network assuming that all publishers publish with their *maxPubRate*, and converts this into a score. Thus, this score represents the additional load put on the network by placing the cache at a certain position.

### 5.2.4 Algorithm: calcMaxCacheMsgPerSecScore

A cache placed in the network could, in the worst case, be responsible for retransmitting all events it receives to all subscribers. This is the case when all subscribers currently require caching. If this scenario is combined with all publisher publishing with their associated *maxPubRate*, then the cache will receive events at the *globalMaxPubRate*. The amount of times each of these retransmitted events will uniquely appear in the network is calculated by the Algorithm 8, resulting in a score. This score represents how big the impact on the network is assuming the cache would be active for all subscribers.

### 5.2.5 Algorithm: calcMaxLinkOverload

Assuming that all publishers in the network publish with their respective *maxPubRate*, each link in the network will have to transmit events at a certain rate. This rate depends on the location of the link. Depending on the location of publishers and subscribers in the network, certain links may be used heavily while others may not be used at all. The resulting *linkUsage* for each link is only theoretical since the *linkUsage* is ultimately bounded by the *maxLinkRate* of that link. Any *linkUsage* exceeding the *maxLinkRate* of a link can be considered as *overload*. Algorithm 9 calculates the sum of the overload for all links, and outputs a score for this value. This score therefore represents the amount of overload the cache causes if placed at a certain position, since the cache has to receive events from all publishers which may increase or cause overload.

## 5.3 Results

The implementation of the cache placement algorithms in Java can be seen in Figure 5.1. This software allows for the generation of graphs, for which the cache placement algorithm can then be run. While this cache placement strategy was not extensively evaluated, the results for a few manual tests seem to provide solid suggestions for cache positions. This can

**Algorithm 3** calculateScore

This algorithm calculates a score between 0.0 and 1.0 representing how good this position is for a cache. Various metrics are taken into account for this decision. Each metric can have a different weight associated with it, in order to prioritize certain metrics, or, for example, to exclude one from the score entirely.

1: var float array [1..5] scores
2: var float array [1..5] weights {given weight for the different metrics, [0.0|1.0]}
3: scores[1] := weights[1] * calcPubCacheFlowScore()
4: scores[2] := weights[2] * calcCacheSubFlowScore()
5: scores[3] := weights[3] * calcMaxPubMsgPerSecScore()
6: scores[4] := weights[4] * calcMaxCacheMsgPerSecScore()
7: scores[5] := weights[5] * calcMaxLinkOverloadScore()
8: var float finalScore := sum(scores) / sum(weights)
9: **return** finalScore

**Algorithm 4** calcPubSubFlow

This algorithm is used to calculate the number of flow rules necessary to reach each subscriber from all publishers. This value does not change with the cache position and is not used as a metric (since it is constant for all cache positions), however, it is required for the following Algorithm 5 to count the number of additional flows necessary because of the presence of the cache.

1: Create tree of network with undirected edges (network links)
2: Lay paths along tree from each publisher to each subscriber (directed edge paths)
3: Merge all duplicate directed edges (same source, same destination)
4: var integer flowcount := sum of the in-degree of all switches of the directed edges present after merge
5: **return** flowcount

**Algorithm 5** calcPubCacheFlowScore

This algorithm is used to calculate the additional number of flow rules necessary to reach the cache from all publishers.

1: Create tree of network with undirected edges (network links)
2: Lay paths along tree from each publisher to each subscriber and to the cache (directed edge paths)
3: Merge all duplicate edges (same source, same destination)
4: var integer flowcount := sum of the in-degree of all switches of the directed edges present after merge minus calcPubSubFlow()
5: **return** scoreOf(flowcount){converts the absolute value of flowcount to a score between [0.0|1.0]}

**Algorithm 6** calcCacheSubFlowScore

This algorithm is used to calculate the number of flow rules necessary to reach all subscribers from the cache.

1: Create tree of network with undirected edges (network links)
2: Lay paths along tree from the cache to each subscriber (directed edge paths)
3: Merge all duplicate edges (same source, same destination)
4: var integer flowcount := sum of the in-degree of all switches of the directed edges present after merge
5: **return** scoreOf(flowcount){converts the absolute value of flowcount to a score between [0.0|1.0]}

**Algorithm 7** calcMaxPubMsgPerSecScore

This algorithm calculates the total unique occurences of all published events in the network, if all publishers publish with their *maxPubRate*.

1: Create tree of network with undirected edges (network links)
2: Lay paths along tree from each publisher to each subscriber and the cache (directed edge paths).
3: Merge all duplicate directed edges (same source, same destination) sharing the same source (publisher)
4: For each directed edge, add the *maxPubRate* of its source publisher to the network link (*linkUsage*
5: var integer eventSum = Sum of *linkUsage* for all network links
6: **return** scoreOf(eventSum){converts the absolute value of eventSum to a score between [0.0|1.0]}

**Algorithm 8** calcMaxCacheMsgPerSecScore

This algorithm calculates the total unique occurences of all events cached in the network, if the cache receives messages with the *maxPubRate* and retransmits them to each subscriber with that rate.

1: Create tree of network with undirected edges (network links)
2: Lay paths along tree from cache to each subscriber (directed edge paths)
3: var integer eventSum = for each network link, multiply the *globalMaxPubRate* with the number of paths passing this network link (*linkUsage*
4: **return** scoreOf(eventSum){converts the absolute value of eventSum to a score between [0.0|1.0]}

**Algorithm 9** calcMaxLinkOverload

This algorithm calculates the sum of uniquely appearing messages that overload their respective link, if all publishers publish with their *maxPubRate*.

1: Create tree of network with undirected edges (network links)
2: Lay paths along tree from each publisher to each subscriber and the cache (directed edge paths)
3: Merge all duplicate directed edges (same source, same destination) sharing the same source (publisher)
4: For each directed edge, add the *maxPubRate* of its source publisher to the network link (*linkUsage*)
5: var integer maxLinkOverload := For all network links sum up (abs(min(0, (*maxLinkRate* - *linkUsage*)))).
6: **return** scoreOf(maxLinkOverload){converts the absolute value of maxLinkOverload to a score between [0.0|1.0]}

also be observed in Figure 5.1: for the displayed network, the Algorithm suggested switch s9 as position for a cache, which - given its central position in the network - could intuitively be considered a good pick as well.

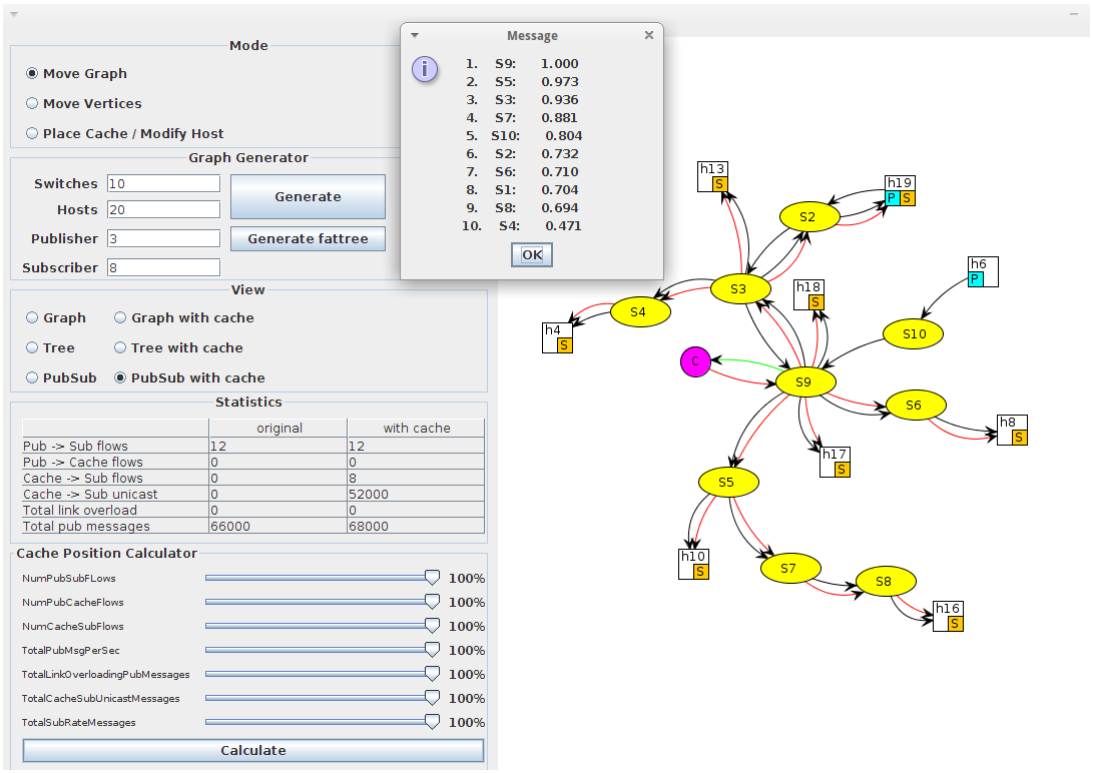


Figure 5.1: Cache placement suggestions for a random topology.

# Chapter 6

# Evaluation

Introducing caches to the publish/subscribe system may drastically change its runtime behavior. While less event loss may occur with caches present, it is assumed that the latency of event delivery may rise significantly.

In order to analyze the effects caching has on the publish/subscribe system, this section focuses on answering the following questions:

- How does caching influence the average latency of event delivery?
- What is the effect of caching on event loss?
- How does the caching influence the rates at which event are transferred?
- Which situations do profit from caching, and which do not?

This chapter is split into four sections. Section 6.1 explains the test environment and general parameters of the tests conducted. Section 6.2 explains the different traffic generation models used for simulating event rates in this evaluation. Section 6.3 tries to isolate general effects of caching in simple scenarios using different event rates. Finally, section 6.4 aims to provide insight into how caches scale with large large amounts of subscribers in need of caching.

## 6.1 Test Environment and Network Topology

All tests have been executed on a virtual machine running Ubuntu Linux version 14.04. The virtual machine uses a processor with 4 cores at a clock speed of 2.4GHz each. The virtual machine has 8GB of RAM available and is located on an SSD. Inside this virtual vachine, a virtual network is started, consisting of open vSwitches and virtual hosts. Open vSwitches are virtual openflow-capable switches. The virtual hosts are realized by linux lightweight container virtualization, with each host essentially being a small virtual machine on its own. The switches and hosts then have their network interfaces connected. This is done by a tool by the name of mininet. Mininet orchestrates the setup of such a network according to a certain topology - it starts all required switches and hosts and connects them to each other.

The topology used for all tests is called a *fattree*.[1] A depiction of this topology can be seen in figure 6.1. It consists of 10 switches and 8 hosts. One interesting feature of this topology is that it allows for a total of 8 unique shortest paths for any path connecting one host from $\{h1, h2, h3, h4\}$ with any host from $\{h5, h6, h7, h8\}$ which makes this topology very tolerant to link or switch failures.

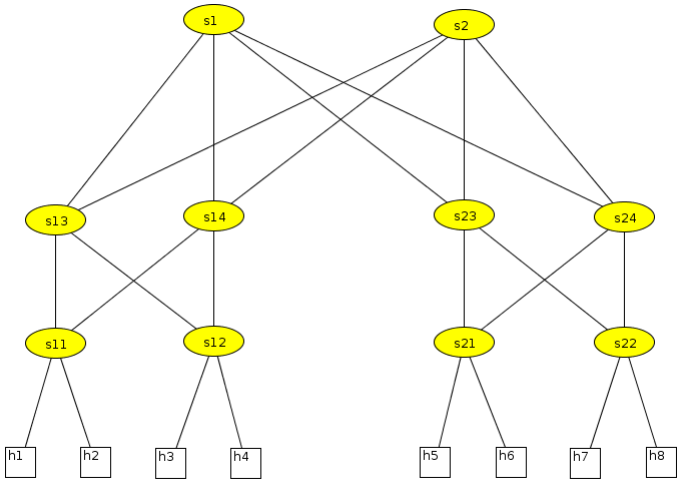


Figure 6.1: A fattree.

## 6.2 Traffic Generation Models

### 6.2.1 Greedy Source

A greedy source is a very simple traffic generation model - it simply means that a publisher is publishing at its maximum rate for as long as it has events to publish. This is a typical scenario where a large amount of data has to be transmitted as fast as possible, e.g. a file transfer.

### 6.2.2 Poisson Distribution

This is a traffic generation model where a publisher publishes according to a poisson distribution for the parameter $\lambda = events/s$. This means that on average $\lambda$ events will be sent, however, the actual rate per second may deviate from this value by a semi-random poisson distribution.

[1] There are multiple defintions for what a fattree is - during this thesis, a fattree is considered to be a topology as displayed in Figure 6.1

### 6.2.3 Poisson Distributed Event Rate combined with Bursts

This is the combination of a traffic generation model that distributes bursts of events semi-randomly over a period of time to generate situations where caching must be activated, and the poisson distribution traffic generation model from subsection 6.2.2 with a small value for lamdba to have a relatively constant workload that does not require caching on its own.

## 6.3 General Caching Properties

This section is aimed at identifying the key runtime behavioral changes with and without a cache present in the network while one publisher publishes events received by one subscriber using different event rates and distributions.

For the tests in this section, the actual content of the events does not play a role - anything the publisher publishes will be of interest to the subscriber. While this is a rather trivial scenario, it helps to isolate distinct runtime behaviors that may otherwise be hard to nail down when taking variable content into account.

Also for tests in this section, host $h1$ will always have the role of a publisher, while $h8$ will always be a subscriber. The cache $c1$ is also always connected to $s1$. Figure 6.2 shows how the events may be routed from $h1$ to $h8$ while the cache is inactive. The presence of the cache makes all routes not traversing $s1$ sub-optimal, since the cache will also be subscribing for the same content as $h8$. If in this state the subscriber is overloaded and the cache becomes active, the situation may look like depicted in 6.3. This again only shows one of four optimal states the network can be in in this scenario - two shortest paths exist from $h1$ to $s1$ while another 2 shortest paths exist from $s1$ to $h8$ which leaves 4 possible optimal combinations.

A total of 100,000 events are published by $h1$. Depending on the scenario, these events are published with varying rates, while the subscriber usually has an arbitrary receive limit of 1,000 events per second.

This section is split up into five subsections. Subsection 6.3.1 isolates the minimum overhead in latency that caching introduces. The following three subsections analyze the runtime behavior for event rates, latency and packet loss for different types of event distribution patterns for the publisher. In particular, publisher $h1$ will publish at event rates according to:

- a greedy source in subsection 6.3.2.
- a poisson distribution combined with occasional bursts in subsection 6.3.3.
- a poisson distribution in subsection 6.3.4.

Finally, subsection 6.3.5 summarizes the results of the previous three test scenarios and concludes this section.

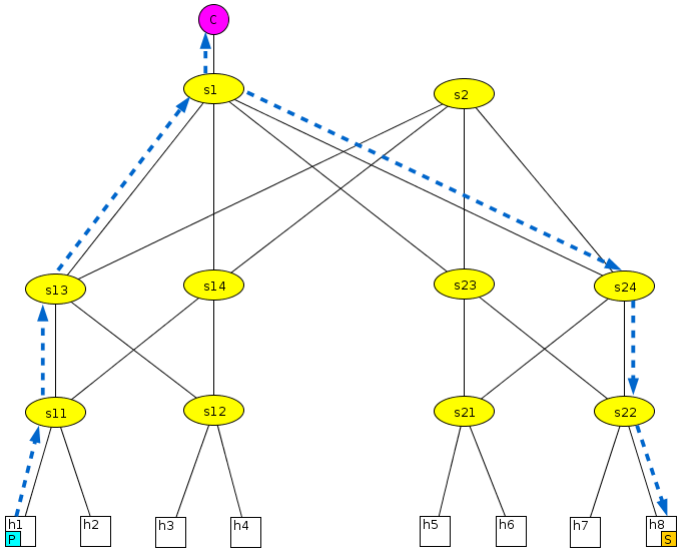


Figure 6.2: Publisher $h1$ publishes events along the blue dotted arrows to subscriber $h8$ as well as to the cache. Events need to be duplicated once at $s1$.

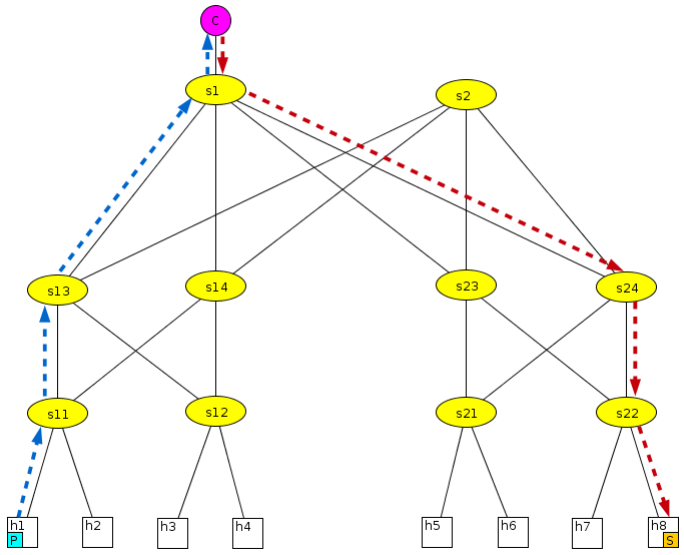


Figure 6.3: Publisher $h1$ publishes events along the blue arrows exclusively to the cache. The cache retransmits events at a controlled rate along the red arrows to $h8$.

### 6.3.1 Latency Overhead

The end-to-end latency between a publisher and subscriber in HPMOM is in the order of microseconds (which in the scope of this thesis is simply rounded to one millisecond for easier comparison). Even though a cache is essentially nothing more than an intermediary in the form of another subscriber and publisher, it is of interest just by how much this cache delays events. While it could in theory be true that the additional delay is in the order of microseconds as well, the cache has to do more complex processing of the events, as described in Chapter 3. Also, the events have to be propagated to the application layer in order to be processed in the first place, and finally be sent again.

So how high is the average delay caused by caching if the cache would forward any event instantly to the subscriber?

To find out, the test setup described in 6.3 was used to transfer the 100,000 events from publisher $h1$ to the cache which then transferred them to subscriber $h8$, at a constant rate of 100 events/s and then for another test with a constant rate of 1,000 events/s.

For a rate of 100 events/s, the average event latency from publisher until reception by the subscriber is 6ms, whereas for the test with 1,000 events/s the latency was 8ms.

These numbers certainly leave room for optimizations, however, these latencies are very negligible compared to the measured average latency in the following subsections - latencies there are measured in the order seconds rather than milliseconds due to the time cached events may spent piled up in a queue.

### 6.3.2 A Greedy Source

An interesting yet simple scenario for caching is a greedy source as a publisher. A greedy source in the publish/subscribe system is a publisher that publishes all events it can at the maximum rate it is allowed to or technically able to. This overload situation has to be handled as quickly as possible by the caching mechanism to avoid massive loss of events.

#### Event Rates

For this first test, a publisher starts to publish with a maximum allowed rate of about 2,000 events/s shortly after the start of the test. Figure 6.4 shows this happening at about 2 seconds in and goes on until all 100,000 events have been published.

Also at 2 seconds a slight peak can be observed for the normal event receive rate of $h8$ - these are the events that arrived before the overload detection triggered the caching. While the publisher $h1$ finishes its activity halfway through the test, the cache keeps publishing events close to the 1,000 event/s limit of the subscriber. This activity extends well beyond the 2 minute mark, until all cached events have been sent at a controlled rate. Between second 2 and second 8 the speed at which the cache retransmits can be seen to increase to about 850 events/s, which is the upper limit deemed to be safe by the overload detection. Occasionally,

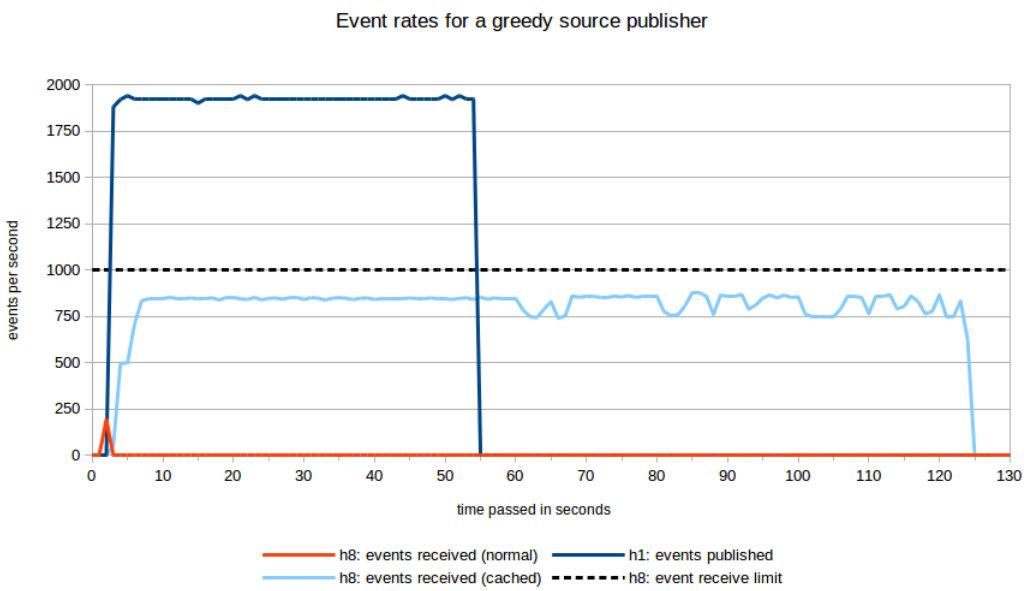


Figure 6.4: Event rates for a greedy source publisher $h1$ which publishes events to $h8$, with caching active.

this rate drops to about 750 events/s, possibly triggered by short violations of the safety threshold that forces a rate reduce if the threshold of 900 events/s is violated.

All in all, the cache acts as an effective buffer, spreading the events out to prevent subscriber overload.

### Latency

Beginning from the moment where the caching starts (roughly at the 2 second mark), the average end-to-end latency is increasing rapidly. Since the subscriber could at most receive 1,000 events/s while the publisher publishes at twice that rate, Events keep piling up with at least a speed of 1,000 events/s at the cache. Essentially, for any event sent by the cache, two more are enqueued for retransmission. This has a very high impact on latency, which can be seen in Figure 6.5. While the queue length of the cache is constantly increasing up to about the 1 minute mark (see Figure 6.6), it starts to fall at the same rate once the publisher ceases to send more events. however, the latency continues to grow until all events have been delivered by the cache. The average end-to-end latency of events is roughly 35 seconds. This points out an unavoidable problem - if the rate of events exceeds the rate at which they can be received, average per-event latency skyrockets.

### Packet Loss

The instant increase to a publish rate of 2,000 events/s at about 1800ms in causes event loss to occur at the subscriber between 1807ms and 1882ms. During these 75ms two phases contribute to the event loss, with both phases being equally long.

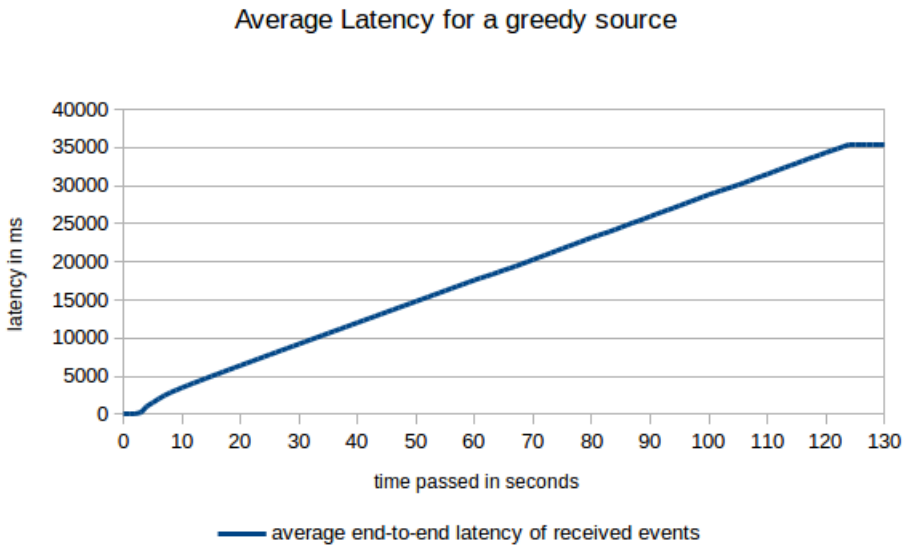


Figure 6.5: Average latency change for a greedy source publisher $h1$ which publishes events to $h8$.

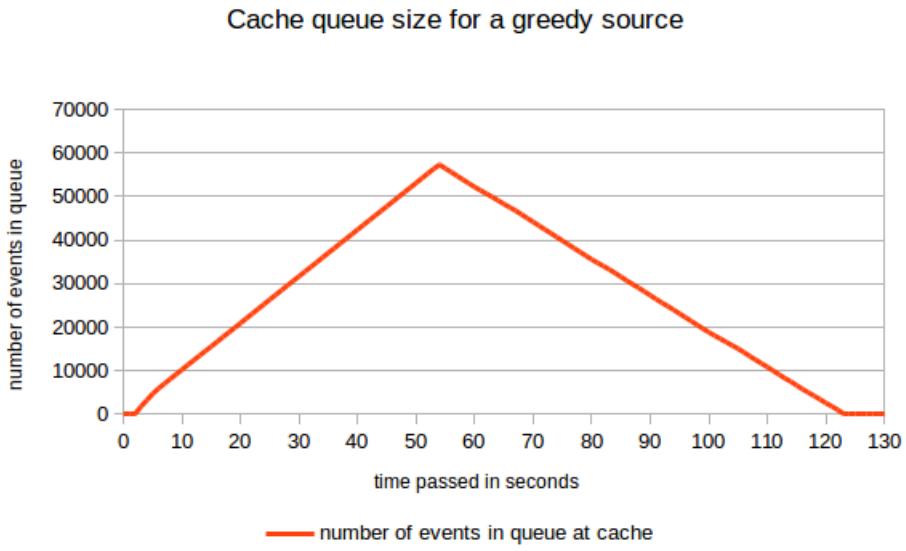


Figure 6.6: Queue length at the cache for a greedy source publisher $h1$ which publishes events to $h8$.

First, roughly every second event is dropped by the overloaded subscriber (due to the fact that the subscriber can only receive at 1,000 events/s while the publisher publishes at 2,000 events/s).

With the controller telling the responsible cache to start caching and with the beginning reconfiguration of the switches in the network, the second phase begins. For as long as it takes to reconfigure the switches and the cache to start queuing events for the subscriber, loss of events currently in the network may occur. Loss during this period is very likely to be absolute, since reconfiguring paths without taking precautions for the current events in the network essentially creates *black holes* where those events disappear.

All in all, during these 75sm, 112 events were lost. Since in this scenario the cache gets active only once, no further loss related to caching occurs.

Total loss therefore accounts for 112, which is 0,122% of the total 100,000 messages sent.

#### Effect compared to not Caching

A greedy sender scenario is very well suited for caching. As can easily be seen in Figure 6.7, a subscriber without cache would be very negatively effected by the high event rate. In this case, the subscriber lost 47% of all events as compared to only 0.122% when using a cache. The still fixed maximum receive rate of the subscriber however forced the cache to store large amounts of events (up to 56,000), which massively influences the average event latency. In particular, the average latency without a cache was 1ms and potentially even lower - whereas the average latency with active caching peaked at 35 *seconds*. Clearly, where timeliness as opposed to low loss rate is of the essence, caching should not be taken into consideration. If however an increase in latency can be agonized, caching provides significant protection against event loss and can very well be considered.

### 6.3.3 Poisson Distributed Event Rate combined with Bursts

The basis for this scenario is a certain level of background event rate with a poisson distribution for $\lambda = 100$ events per second combined with randomly distributed peaks reaching rates of up to 2,000 events/s. Just like with a greedy sender, the peaks require caching to become active as fast as possible to avoid event loss. As opposed to the greedy sender, the cache has the ability to become active and inactive multiple times during the test. Meanwhile, the poisson distributed event rate is supposed to resemble a constant workload. This workload prevents trivial situations where no events are currently transmitted and switching from caching to not caching would have no effect on event loss.

Since the event rate for this distribution pattern is on average much lower than the greedy sender, the figures in this section only show a representative part of the whole test.

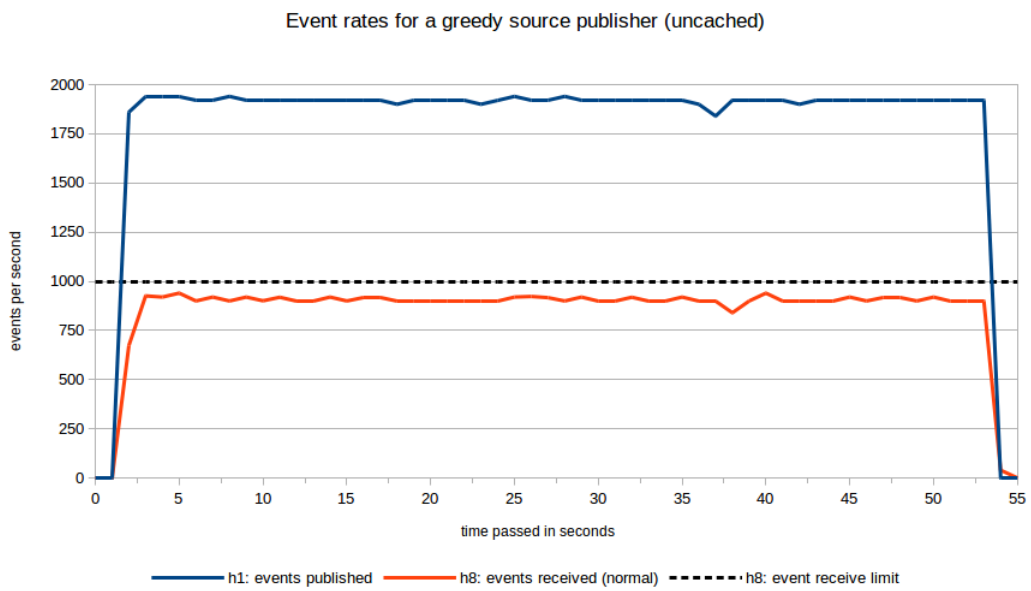


Figure 6.7: Event rates for a greedy source publisher $h1$ which publishes events to $h8$, without caching.

### Event Rates

The graph showing the behavior of the event rates for this scenario is a lot more complex than for the greedy sender, as can be seen in Figure 6.8.

While no peaks occur, the cache remains inactive, and the receive rate for normal events roughly matches the publish rate. In case of a peak of events such as at roughly 8 seconds in, the publish rate can be seen to diverge from the receive rate of normal events, which itself moves to zero. At the same time, the cached event receive rate starts to increase. The reason for that is that at this 8 second mark the caching starts and goes on to the 14 second mark, essentially transforming the short, intense peak of published events into a longer, more moderate peak.

With the increase of normally received events at the 14 second mark, it becomes clear that the caching was stopped. At the same moment however the publish rate peaks again, causing the caching to restart shortly after, essentially repeating the pattern from the first peak.

During second 22 and 36 no caching is required. After second 36 an interesting period starts. With 3 consecutive peaks occurring close together, the caching does not stop until all these peaks are over, essentially smoothing them into one long peak of moderate intensity.

This all goes to show that for this scenario the cache acts as an efficient buffer for peaks while not affecting the network during times where the rates are low enough for the subscriber to handle.

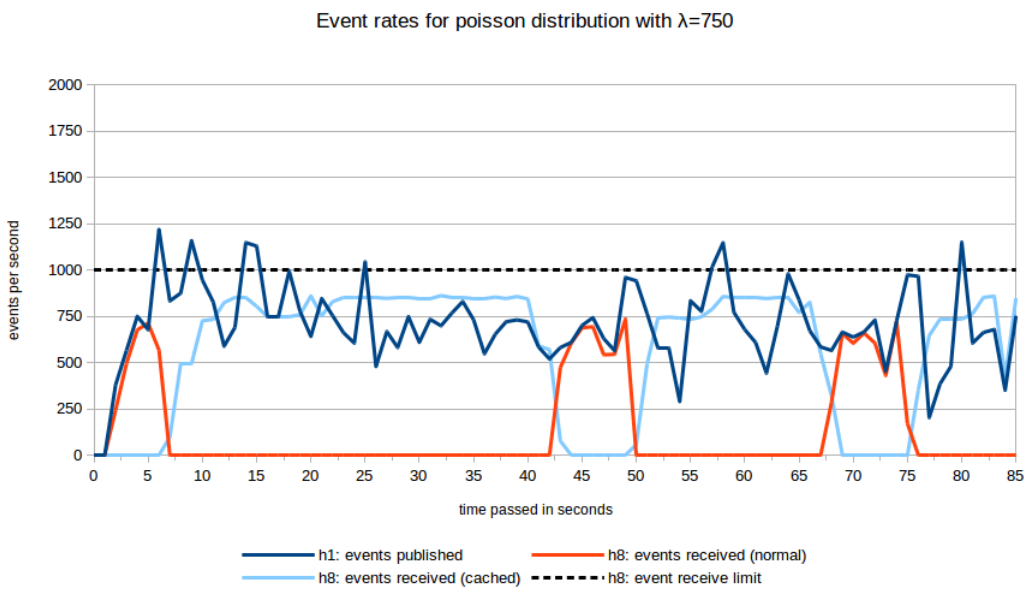


Figure 6.8: Event rates for events published with a poisson distribution combined with bursts from publisher $h1$ to subscriber $h8$, with caching active.

### Latency

The effects on latency are not as severe as with the greedy sender, however, they are still noticeable. Figure 6.9 shows how during periods where caching is active (high rates of cached events which can be seen in Figure 6.8) the average latency rises as well. When no caching takes place, the average latency starts to lower, since the events are then forwarded without any noteworthy delay again.

At the end of the test, the average latency ended up being 1,700ms.

### Packet Loss

Over the course of the test, multiple situations occur in which events are lost. These show the same two phases as described in Subsection 6.3.2, however, they are not expressed as severe. Over the course of the test, 176 events are lost. Given the fact that all in all 30 peaks occur that surpass the receive limit of 1,000 events/s, this number is relatively small compared to the packet loss of a greedy sender which lost 112 events in a single peak alone. The relatively low average loss of 5.9 events per peak has two main reasons.

Firstly, the peaks build up relatively slowly compared to the greedy sender - they reach their peak only after 500ms. While not even a remarkable timespan in Figure 6.8, this gradual buildup results in better detection of the overload situation and faster handling by the controller, meaning that on average no more than these 5.9 events are lost, even though the event rate at the top of the peaks can be as high as the rate for the greedy sender.

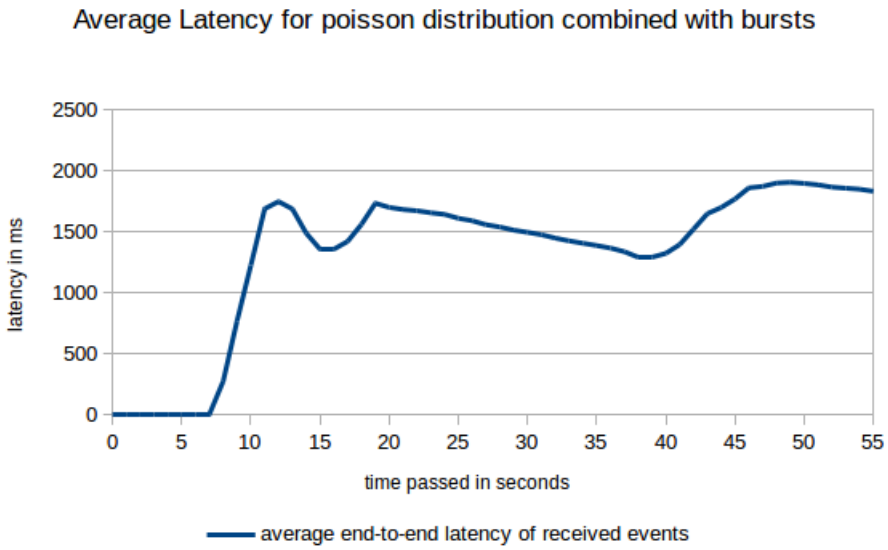


Figure 6.9: Average latency for events published with a poisson distribution combined with bursts from publisher $h1$ to subscriber $h8$.

Secondly, if peaks are close enough together so that caching stays active during multiple peaks, these multiple peaks share only one instance of loss due to spontaneous overload.

With 176 events lost out of a total 100,000 events, the loss rate is 0.176%.

#### Effect compared to not Caching

The effects of caching in this scenario are more subtle than observed for the greedy source in the previous subsection. Figure 6.10 shows that the same scenario without a cache can receive the basic workload just as well, however the bursts of events are cut off before reaching the limit of 1,000 events/s - causing event loss due to overload for each of those bursts. The combined total event loss for the test without a cache is 16695 events, or 16.695% of the total of 100,000 events sent. Compared to the loss with a cache present (0.176%, or 176 events in total), this is quite significantly worse.

This again comes at the cost of average event latency - without caching, the latency is 1ms or less, whereas caching causes an delay of 1700ms on average.

All in all however, a scenario like this where the workload is generally below the event receive limit of the subscriber and only moderately frequent bursts violate this limit seem plausible for caching.

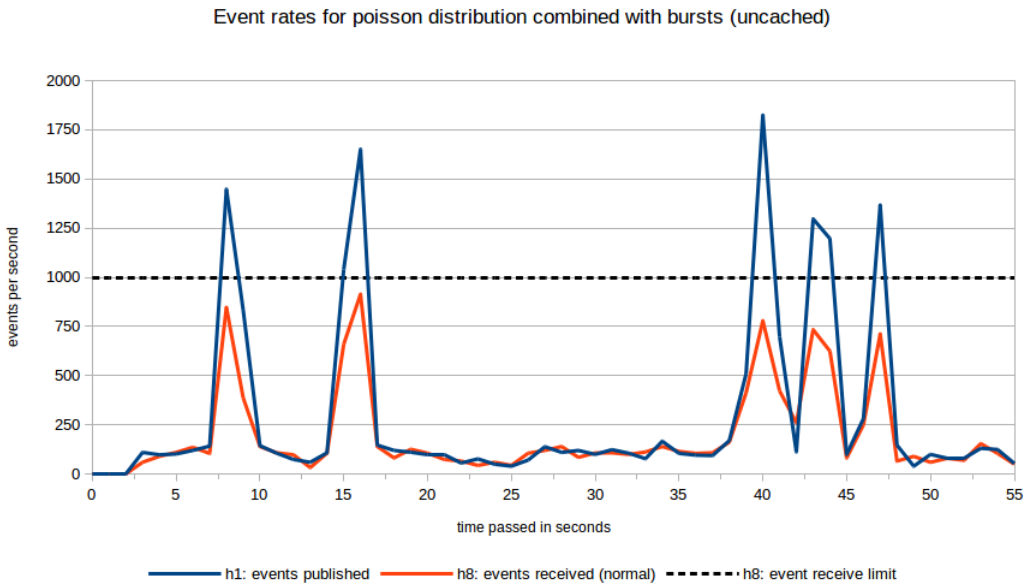


Figure 6.10: Event rates for events published with a poisson distribution combined with bursts from publisher $h1$ to subscriber $h8$, without caching.

### 6.3.4 Poisson Distribution

One final scenario for the same simple setup has the publisher send at random rates changing once a second according to a poisson distribution with $\lambda = 750$, causing an average of 750 events per second to be sent. This distribution is alot harder for the cache to handle. On average, the subscriber could handle this rate, but since event rates can peak at well above 1,000 events/s at times, caching has to take place to prevent event loss. however, since the rate is relatively close to the limit regularly, it is not easy to decide when caching should stop.

This subsection therefore shows a distribution where caching is of questionable efficiency when the caching mechanic is supposed to constantly make the right decisions: Caching or not caching.

#### Event Rates

Figure 6.11 shows the event rates for a representative section of the test conducted. The average of 750 events/s is close to the rate of 800 - 900 events/s at which the cache will retransmit the events (leaving at least 100 events/s as a threshold to not cause more than 90% load on the subscriber). This enables the cache to retransmit events at a faster rate than the publisher publishes on average, therefore being able to catch up in case a peak high enough for caching to start occurs. In practice however this causes the cache in its current

implementation to be active for extended periods of time, as shown in Figure 6.11 by the cached receive rate.

For scenarios like these, the point at which caching starts and stops as well as the exact transmission rate may be fine tuned to perform better overall, since for large periods of time, the caching was not actually needed.

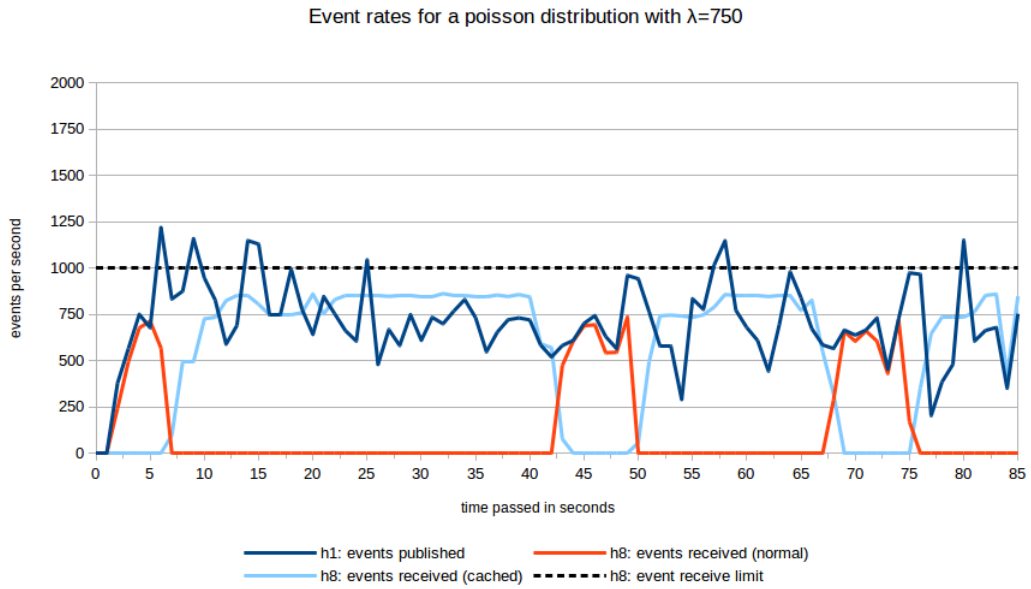


Figure 6.11: Event rates for events published with a poisson distribution with $\lambda = 750$ from publisher $h1$ to subscriber $h8$.

### Latency

Even though the average rate of published events (750/s) is well below the limit of 1,000 events/s, the average latency of events still ends up as 1,169ms. While the cache can retransmit events with a delay below 10ms (As shown in Subsection 6.3.1), the time it takes to empty the currently queued events at a rate only slightly above the average publish rate can take several seconds (given preceding peaks filling up that queue) and thereby cause noticeable delay (see Figure 6.12 and 6.13). Also, the speed of the retransmission by the cache increases incrementally to a maximum, contributing to the delay due to this ramp-up time.

However, some of the delay is justified by all peaks actually violating the 1,000 events/s border, since delivery of the events during these peaks could, without caching, under no circumstances be possible with small latency *and* without loss.

### Packet Loss

While not as prominent as in the other scenarios, the positive effect on event loss is still measurable with this distribution pattern. The total amount of events lost accumulates to

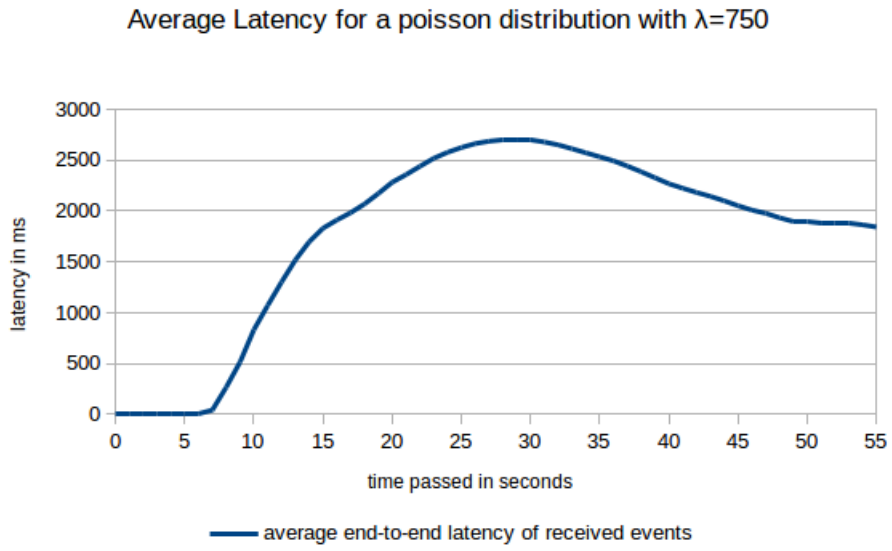


Figure 6.12: Average Latency for events published with a poisson distribution with $\lambda = 750$ from publisher $h1$ to subscriber $h8$.

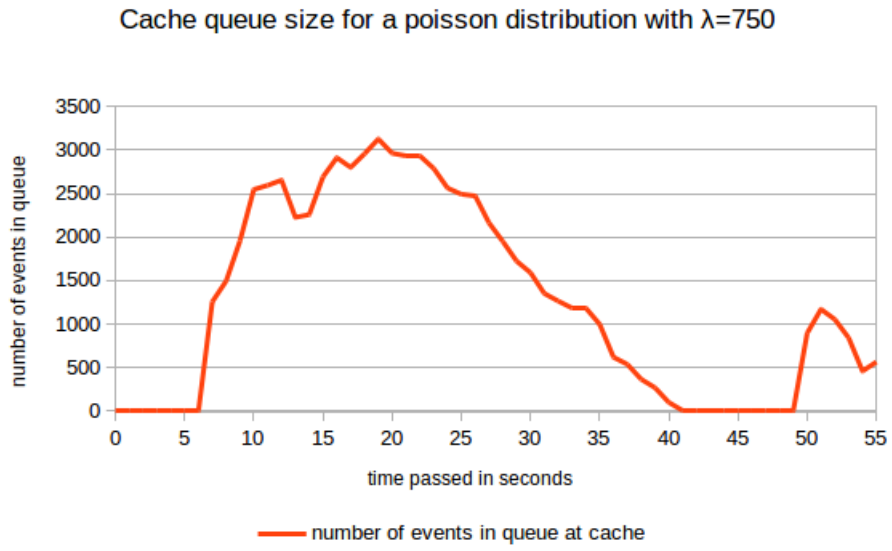


Figure 6.13: Queue length at the cache for events published with a poisson distribution with $\lambda = 750$ from publisher $h1$ to subscriber $h8$.

146, which is is 0,146% of the total 100,000 events sent. This marginal loss can mostly be attributed to losses due to peaks above the 1,000 events/s mark, and no caching was active at that time.

#### Effect compared to not Caching

The loss rate of this test using caching (0,146% loss) is still lower than without caching (1,413%), however the difference is quite small as compared to the previous tests. Depending on the requirements, the difference may not be large enough to justify the very high average latency (1169ms). This shows that for situations where latency is more important than a relatively small amount of loss caching should maybe not be used or has to be fine tuned for the situation in order to perform better.

### 6.3.5 Summary of Simple Scenarios

Overall, the results of these three simple scenarios from the previous section can be summarized as:

**Caching $\longrightarrow$ less event loss**
With all presented scenarios caching decreases the loss of events significantly.

**Caching $\longrightarrow$ higher average latency**
While the theoretical latency overhead is below 10ms, the latency during the tests rose to magnitudes higher than that.

**Caching $\longrightarrow$ not necessarily suitable for all situations**
As an example, the poisson distribution shown in Subsection 6.3.4 shows that slightly lower event loss comes at the cost of significantly higher latency.

## 6.4 Benchmarking the Cache

Since a single cache proved to be capable of caching and retransmitting at rates of over 180,000 events/s, a reasonably large network using many publishers, subscribers and caches was ruled out for this evaluation and since the test environment (see Section 6.1) does simulate the network on a single machine, and therefore any additional component of the simulation negatively influences the performance of the cache directly.

This means that the evaluation in this section focuses on the performance of the caching mechanism itself. Just like in the previous evaluation, a cache is installed at switch s1 (see Figure 6.2). Instead of real publishers actually sending events to the cache, the cache will receive one event every time it attempts to receive an event from its subscription. This simulates any amount of publishers publishing at a rate that is constantly at 100% of the caches limit. The cache is not aware of this deception and cannot tell that these events are artificially created. The cache will further be told to actively cache for a varying amount of subscribers. These subscribers do not exist as well, and events that the cache will forward to

them will be dropped by switch s1 in the network immediately. This means that the cache itself is not aware that it is part of an artificial simulation and will behave exactly the same as it would in an environment with actual subscribers and publishers. This trick spares the majority of the resources of the testing environment for the cache itself - instead of distributing the processing power for no real gain among hundreds of participants in the network for the sake of evaluating the cache.

**Setup** The cache is responsible for category L with $f < L >=< l = [-100|100] >$, and it will receive random events in this range. Each event is offered to retransmit queues for a varying amount of subscribers. These queues are emptied at the highest rate possible. This was done for 1, 100, 250, 500, 1000, 2000, 3000, 4000 and 5000 simulated subscribers. These subscribers have subscriptions for Category L with $f < L >=< l = [-100|100] >$, meaning that each single event arriving at the cache is of interest to every single subscriber.[2] For any amount of subscribers, the following metrics were determined:

**Average transmit rate (total)**
How many events/s does the cache transmit to all subscribers on average?

**Average transmit rate (per subscriber)**
How many events/s does the cache transmit to each single subscriber on average?

**Average offer time**
How long does it take for each event to be offered/range-checked to all retransmission queues?

**Average processing time**
How long is each event processed on average?

The following subsections cover the results for this simulation for each number of subscribers. Each simulation collected the average results for the metrics from a simulation period of one minute.

### 6.4.1 Average Transmit Rate (total)

Figure 6.14 shows how, for different number of subscribers, the amount of total transmitted events by the cache changes. With only one subscriber requiring caching, the cache is able to transmit at about 180,000 events/s. With an increase in subscribers, the total number of transmitted events slowly and linearly decreases. This can be explained by organizational overhead: each subscriber requires its own queue since the subscribers may receive events at different rates and cannot share a global queue. Also, each event has to be range-checked against the ranges of the subscription. All in all, the global transmit rate scales quite well with the amount of subscribers. As the Figure 6.14 shows, a single cache can, for example,

[2] while the subscribers could simulate different ranges for $f < L >$, this would not add anything meaningful to this evaluation - each singe event is range-checked against the range of the subscriber, so no performances is gained from using the same ranges for the cache and the subscribers.

support 100 subscribers while still operating at 91% of the peak performance possible for a single subscriber.

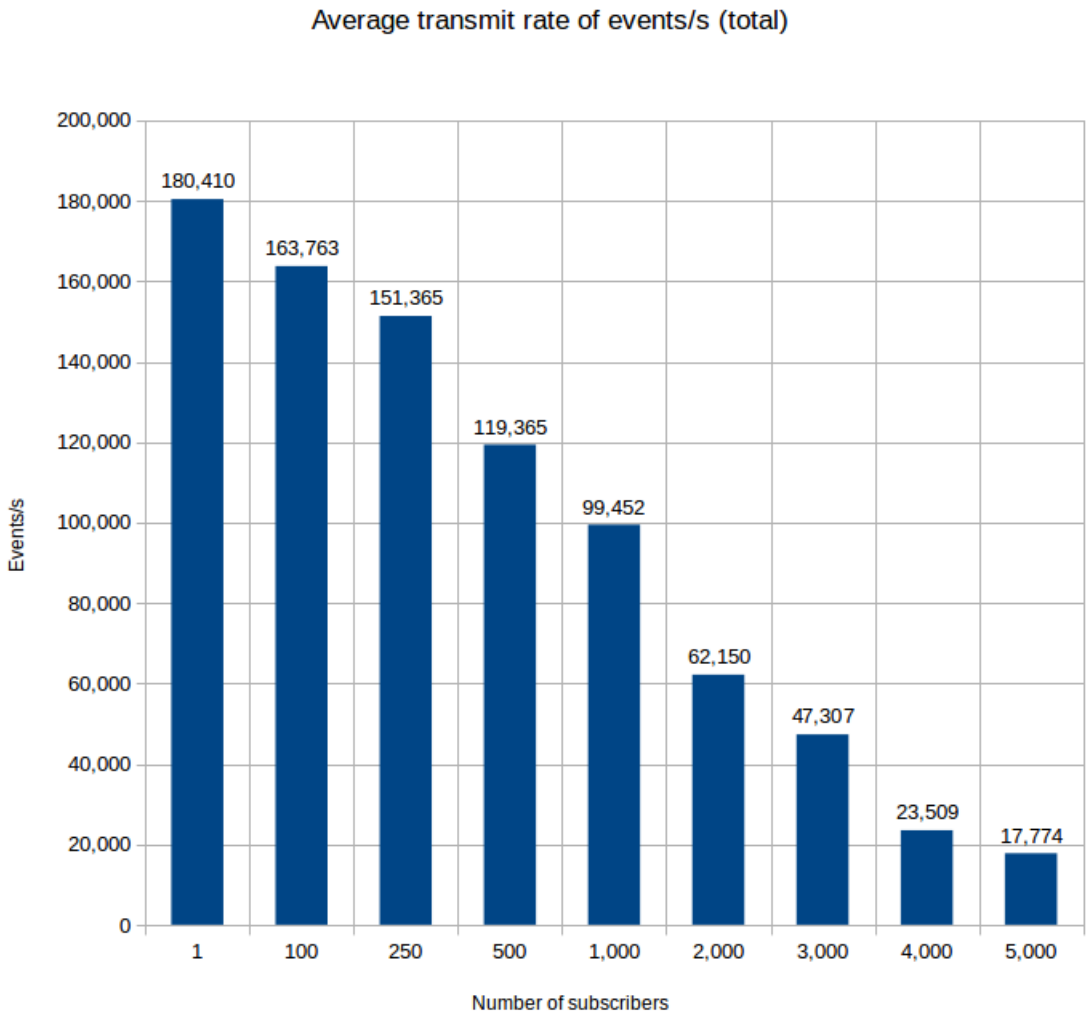


Figure 6.14: Number of events/s the cache transmits to all subscribers on average.

### 6.4.2 Average Transmit Rate (per subscriber)

As explained in Chapter 4, the retransmission of the cache is done via unicast directly to single subscribers. This means that each additional subscriber served by the cache decreases the rate at which each subscriber can be served individually. This effect can be seen in Figure 6.15. The total transmit rate shown in 6.14 can be considered a shared medium for all subscribers. While 100 subscribers can still be served with 1,637 events/s on average, this rate quickly degrades with more subscribers. With 1,000 subscribers being served by the cache,

the transmission rate dropped to 99 events/s per subscriber. At 5,000 subscribers, the average rate dropped to 3 events/s per subscriber.

While this is not suitable for high event rates, it can be useful in case a burst of events forces a large amount of subscribers to start using the cache. The cache can still act as buffer for this burst, and then retransmit the events. With every subscriber that resumes normal operation, higher rates can be achieved by the remaining transmissions, until eventually all subscribers have resumed normal event receiving. In a scenario like this, it would be preferable to have the cache focus its available capacity on only a few subscribers at once in order to quickly remove the total number of subscribers currently served (Figure 6.14 shows the positive effect on total event rate with fewer subscribers active).

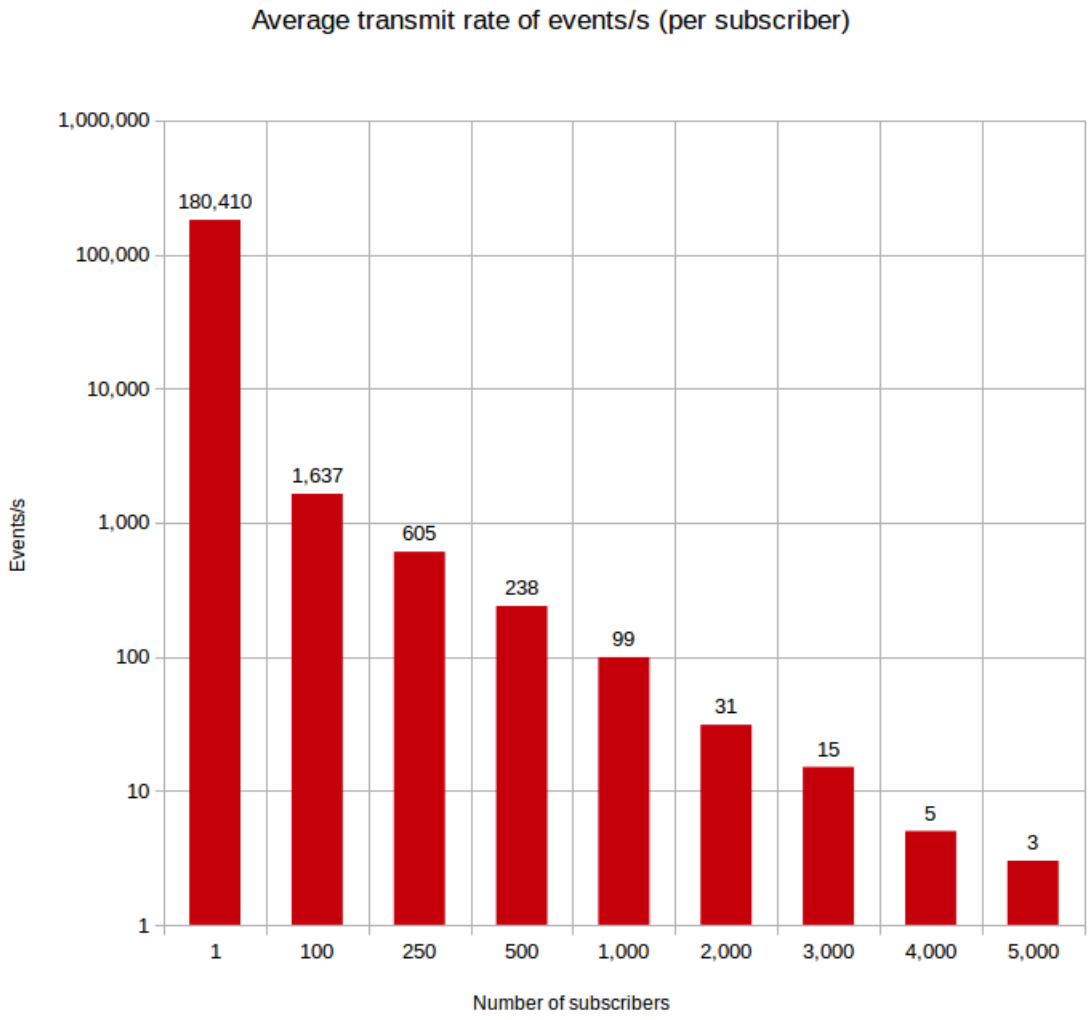


Figure 6.15: Number of events/s the cache transmits to each single subscriber on average.

### 6.4.3 Average Offer Time

This subsection evaluates how an increasing amount of subscribers affects the time the cache requires to offer a event to all subscribers. This offering consists of doing a check whether the event is relevant to the subscriber, and if it is, to add the event to the queue of that subscriber. For $CategoryL =< categoryName = "L"; attr_0name = "l", attr_0type = Long, attr_0range = [-100|100] >$, and every incoming $event\ e = < categoryOfEvent = "L"; attr_0 = "l", value_0 = X >$, it has to be checked that $-100 \leq X \leq 100$. Such a range check has to be done for each Attribute in the category. Overall however, this check is very fast. Figure 6.16 shows the average time this offering to all subscribers takes.

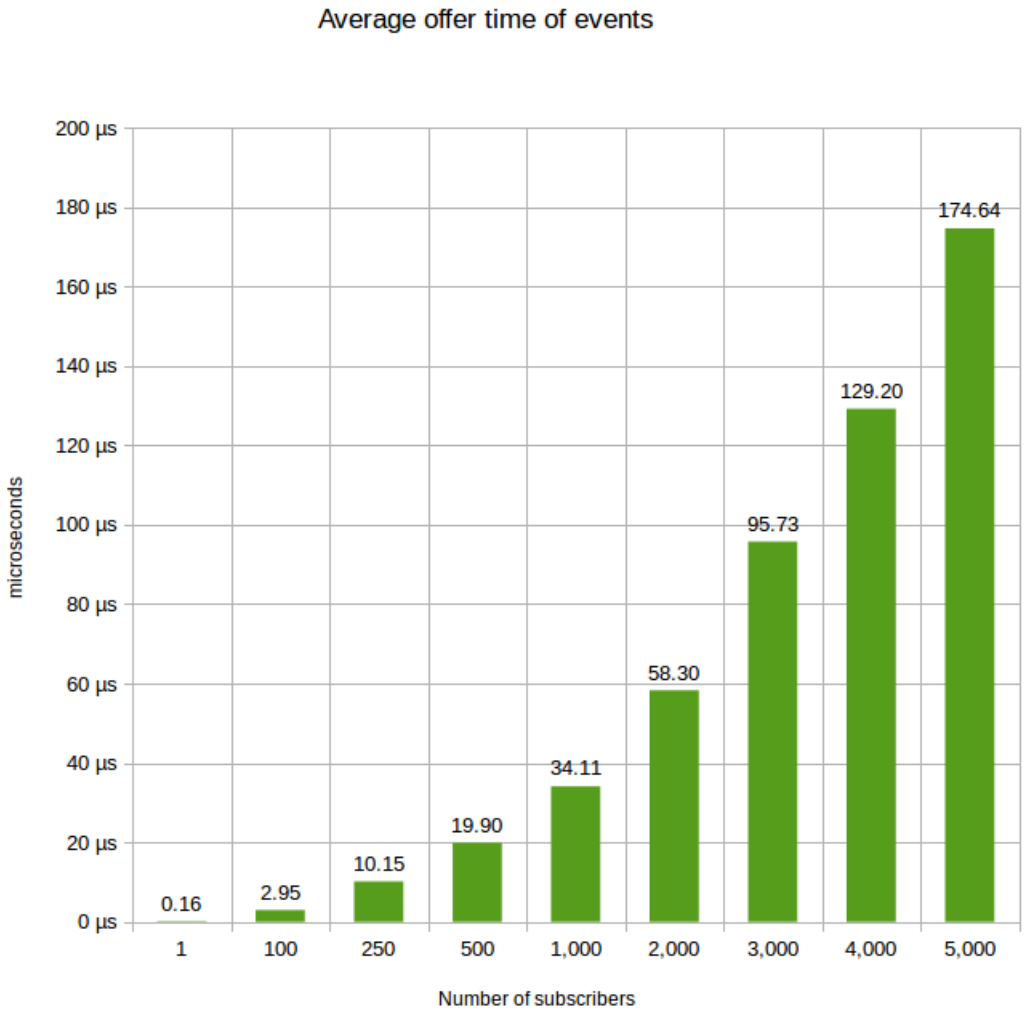


Figure 6.16: Average event offer time inside the cache.

### 6.4.4 Average Processing Time

Figure 6.17 shows how long it takes the cache to handle an event on average. This is the total amount of time it takes to take an event from the receive queue, convert it to a cached event, and finally offering it to every transmission queue[3].

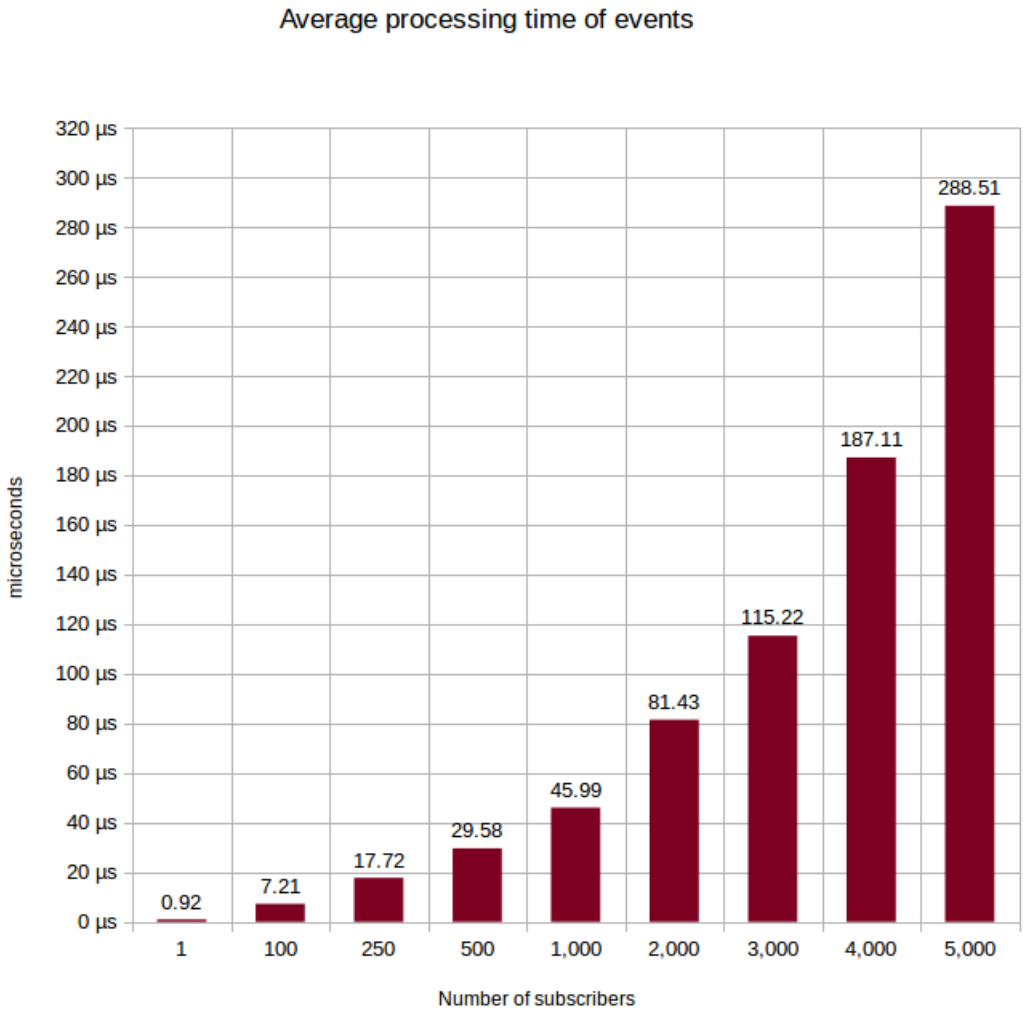


Figure 6.17: Average event processing time inside the cache.

[3]The processing time therefore includes the average offer time measured in the previous subsection

# Chapter 7

# Conclusion and Future Work

SDN-based content-routing networks require new solutions for problems already solved or not even existing in traditional, broker-based systems. This thesis presented a solution for controlling event-rates in a SDN- and contend-based publish/subscribe systems. For this, a caching mechanism was implemented - caches are hosts which use the semantic of the underlying content-based network to subscribe to content that they aim to provide caching for. In case of an overload of a subscriber in the network, the cache will temporarily store the events intended for the subscriber. Said subscriber will suspend its original subscription until further notice. Now, the cache will transmit the events at a controlled rate to said subscriber. Finally, the subscriber will resume its original subscription. The infrastructure necessary for controlling the behavior of the caches and subscribers were implemented. This infrastructure was used to implement a centralized control module working at the SDN-controller which orchestrates the actions of all caches and subscribers and monitors their current status. In order to decide on when a subscriber should be cached, three mechanisms were discussed to detect overload situations, of which one was used in the implementation of the caching mechanism. Furthermore, an algorithm was proposed and implemented which is aimed at finding good positions to place caches in the network according to various properties known about this network.

Two sets of evaluations were conducted. The first set of evaluations intends to provide a general understanding of how a cache will affect the runtime dynamics of a publish/subscribe system. This primarily isolated effects caching has on latency and event loss. The goal of the second set of evaluations was to find out how many subscribers a cache can handle and how the amount of subscribers affect the performance of said cache.

Since the implemented caching mechanism is the first of its kind for SDN- and content-based publish/subscribe, a critical look at the concept of the caching mechanism itself and the results of the evaluation are necessary.

In general, controlling event rates in SDN-based content-routing networks is definitely possible. There is a lot of room for improvements, which may be addressed in future work. For once, event loss could be further minimized by using a more sophisticated synchronization mechanism for subscribers to switch to and from caching. Also, the cache could keep a fixed amount of events so that every lost event can be retransmitted even if the caching started after a subscriber experienced event loss.

Also, one of the biggest restrictions currently is the fact that retransmission of events is done via unicast. If subscribers interested in the same cached events at the same time could be grouped, cached events would only have to be sent once since the publish/subscribe system delivers the events to all members of that group. This way, the number of subscribers a cache can handle would increase manifold.

In order to react to changes in the system, it would also be necessary to implement a mechanism which enables caches to be moved around in the network. Caches should be able to split up their responsibilities so they can scale horizontally at any time when they are overloaded, and then be able to merge with each other again in case multiple caches are not needed anymore.

Optimization can be done for the overload detection mechanism - this can decrease the time it takes for caching to become active, which results in fewer lost events and allows for the caching of shorter peaks. Also, the prediction algorithm for constantly adapting the transmission rate according to the subscribers load level could be improved - this would lower the average event latency by starting to transmit events faster and by keeping the rate closer to the limit of the subscriber.

Finally, the evaluations conducted were rather basic and ran on a simulated network on a single machine - in order to fully understand the impact of caching, a scenario which involves changing numbers of publishers, subscribers and caches, varying content and event rates as well as changes in the topology, with everything *running in a real network* (or at least in a simulated, but distributed network) should be considered.

# Bibliography


[1] Wikipedia, "Software-defined networking — wikipedia, the free encyclopedia," 2014. [Online; accessed 17-November-2014].

[2] M. A. Tariq, B. Koldehofe, G. Koch, and K. Rothermel, "Providing probabilistic latency bounds for dynamic publish/subscribe systems," in *Proceedings of the 16th ITG/GI conference on kommunikation in verteilten systemen (KiVS)*, Springer, 2009.

[3] M. A. Tariq, B. Koldehofe, G. G. Koch, and K. Rothermel, "Distributed spectral cluster management: A method for building dynamic publish/subscribe systems," in *Proceedings of the 6th ACM international conference on distributed event-based systems (DEBS)*, 2012.

[4] M. A. Tariq, B. Koldehofe, G. G. Koch, I. Khan, and K. Rothermel, "Meeting subscriber-defined qos constraints in publish/subscribe systems," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 17, pp. 2140–2153, 2011.

[5] 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.

[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] 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.

[9] M. A. Tariq, B. Koldehofe, S. Bhowmik, and K. Rothermel, "PLEROMA: A SDN-based High Performance Publish/Subscribe Middleware," in *To appear in Proceedings of the ACM/IFIP/USENIX Middleware Conference*, ACM press., Dezember 2014.

[10] S. Bhowmik, "Distributed Control Algorithms for Adapting Publish/Subscribe in Software Defined Networks," master thesis, University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, Germany, November 2013.

[11] V. Sourlas, G. Paschos, P. Flegkas, and L. Tassiulas, "Caching in content-based publish/subscribe systems," in *Global Telecommunications Conference, 2009. GLOBECOM 2009. IEEE*, pp. 1–6, Nov 2009.

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

[13] O. N. Foundation, "Openflow switch specification."

[14] E. Fidler, H.-A. Jacobsen, G. Li, and S. Mankovskii, "The PADRES Distributed Publish/Subscribe System," *International Conference on Feature Interactions in Telecommunications and Software Systems (ICFI'05)*, pp. 12–30, July 2005.

[15] 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.

[16] G. Cugola, E. Di Nitto, and A. Fuggetta, "The jedi event-based infrastructure and its application to the development of the opss wfms," *IEEE Trans. Softw. Eng.*, vol. 27, pp. 827–850, Sept. 2001.

[17] a. X. c. PARC, "Content centric networking." `http://www.ccnx.org/what-is-ccn/`, 2014.