Université de Neuchâtel
Faculté des Sciences
Institut d'Informatique
Chaire de Systèmes Complexes
Programme Doctoral

Scalable Content-Based Publish/Subscribe and Application to Online Trading

par
Raphaël P. Barazzutti

Thèse soutenue à la Faculté des Sciences le 2 mai 2019
pour l'obtention du grade de Docteur ès Sciences

Acceptée sur proposition du jury :

Prof. Pascal Felber, directeur de thèse
Université de Neuchâtel

Prof. Peter Kropf, président du jury
Université de Neuchâtel

Dr Valerio Schiavoni, rapporteur
Université de Neuchâtel

Prof. Rachid Guerraoui, rapporteur
École Polytechnique Fédérale de Lausanne

Prof. Benoît Garbinato, rapporteur
HEC Lausanne, Université de Lausanne

git: 79d5567 (2019-07-21)

UNIVERSITÉ DE NEUCHÂTEL

Faculté des Sciences
Secrétariat-décanat de Faculté
Rue Emile-Argand 11
2000 Neuchâtel – Suisse
Tél : + 41 (0)32 718 21 00
E-mail : secretariat.sciences@unine.ch

IMPRIMATUR POUR THESE DE DOCTORAT

La Faculté des sciences de l'Université de Neuchâtel autorise l'impression de la présente thèse soutenue par

Monsieur Raphaël P. BARAZZUTTI

Titre:

"Scalable Content-based Publish/Subscribe and Application to Online Trading"

sur le rapport des membres du jury composé comme suit:

- Prof. Pascal Felber, directeur de thèse, Université de Neuchâtel, Suisse
- Prof. Peter Kropf, Université de Neuchâtel, Suisse
- Dr Valerio Schiavoni, Université de Neuchâtel, Suisse
- Prof. Rachid Guerraoui, EPF Lausanne, Suisse
- Prof. Benoît Garbinato, HEC Lausanne, Université de Lausanne, Suisse

Neuchâtel, le 9 mai 2019

Le Doyen, Prof. P. Felber

À ma famille

Contents

List of Figures **xiii**

List of Tables **xv**

List of Acronyms **xvii**

Acknowledgments **xxi**

Abstract **xxiii**

Résumé **xxv**

1 Introduction **1**

1.1 Context 1

1.1.1 Publish/Subscribe 1

1.1.2 Scalable Cloud Architectures 3

1.1.3 Online Trading 8

1.2 Motivations and Contributions 9

1.2.1 Scalable Publish/Subscribe Engine 9

1.2.2 Concurrent Order Book 10

1.3 Organization of the Thesis 10

2 Related work **13**

2.1 Publish/Subscribe 13

2.1.1 Infrastructure Provisioning 13

2.1.2 Elasticity 18
2.2 Online Trading 21
3 Infrastructure Provisioning **23**
3.1 Introduction 23
3.2 Infrastructure Model and Assumptions 25
3.2.1 Communication Assumptions 25
3.2.2 Workload Assumptions 25
3.2.3 Interconnection Points 26
3.2.4 Matching Operators 27
3.2.5 Dispatchers 27
3.3 Scaling the Infrastructure 28
3.3.1 Scaling with Publishers and Subscribers 28
3.3.2 Scaling with Publications (Matching Constraints) 29
3.3.3 Scaling with Subscriptions 30
3.3.4 Dispatching Filtered Messages 31
3.3.5 Scaling with Publications (Communication Constraints) 32
3.3.6 Putting Everything Together 32
3.4 A Stage-Driven Architecture 33
3.4.1 Interconnection Points Provisioning and Scalability 33
3.4.2 Matchers, Dispatchers Provisioning and Scalability 34
3.5 Experimental Results 35
3.6 Summary 38

4 Scalability and Performance **39**

4.1 Introduction 39

4.2 Architecture 43

4.2.1 Connection To and From Clients 44

4.2.2 Content-Based Routing Operators 46

4.3 Implementation 50

4.3.1 StreamMine Stream Processing Engine 50

4.3.2 Filtering and Clustering Libraries 52

4.4 Evaluation 54

4.4.1 Experimental Workload 55

4.4.2 Baseline Filtering Performance 55

4.4.3 Performance of Operators 56

4.4.4 Impact of Clustering 59

4.4.5 Overall Performance 61

4.4.6 Comparison with PADRES 61

4.5 Summary 63

5 Elasticity **65**

5.1 Introduction 65

5.2 The StreamHub Scalable Pub/Sub Engine 67

5.3 Elasticity Mechanisms 70

5.3.1 Slice Migration with Low Impact on Delays 70

5.3.2 The e-StreamHub Manager 72

5.4 Elasticity Enforcer and Policy 73
5.5 Evaluation 75
5.5.1 Experimental Setup 75
5.5.2 Workload 76
5.5.3 Baseline StreamHub Performance 77
5.5.4 Operator Slice Migration Performance 78
5.5.5 Elastic Scaling under Varying Workloads 79
5.6 Summary 83
6 Application to Online Trading **85**
6.1 Introduction 85
6.2 Online Trading and the Order Book 87
6.3 A Concurrent Order Book 90
6.3.1 Coarse-Grained Locking 92
6.3.2 Two-Level Fine-Grained Locking 92
6.3.3 Toward Lock-Free Algorithms 94
6.4 Generating Workloads 98
6.5 Evaluation 100
6.6 Summary 101
7 Conclusion **103**
7.1 Summary of Contributions 103
7.2 Future Directions 105
7.2.1 Scalable Content-Based Publish/Subscribe Engine 105

7.2.2 Elasticity 105
7.2.3 Application to Online Trading 106
A Publications **107**
B Bibliography **109**
C Curriculum Vitae **121**

List of Figures

1.1 Publish/subscribe engine. 1

2.1 Broker overlay network. 16

3.1 Publisher Interconnection Point operator. 26

3.2 Subscriber Interconnection Point operator. 26

3.3 Matching operator. 27

3.4 Generic architecture, its operators, and its constraints when routing publications. 28

3.5 Modular stage-driven Pub/Sub architecture 29

3.6 Partitioning publications. 29

3.7 Partitioning subscriptions. 31

3.8 Publication filtering time of the SIENA algorithm 36

3.9 Number of matching operators regarding the number of number of subscriptions (part 1). 37

3.10 Number of matching operators regarding the number of number of subscriptions (part 2). 38

4.1 Architectural principles. 43

4.2 User view of StreamHub. 45

4.3 StreamHub processing operators. 47

4.4 Path taken by subscriptions and publications in StreamHub. 48

4.5 Details of an operator slice from Figure 4.1. 51

4.6 Workload characteristics: cumulative distribution of matching set sizes for publications and matching ratios for subscriptions. 54

4.7 Performance of the counting `libfilter` for filtering incoming publications with respect to the size of the stored subscriptions set. 55

4.8 Scaling of StreamHub operators. 57

4.9 Scaling of the matcher operator 58

4.10 Overhead of clustering for storing subscriptions and impact on the filtering 59

4.11 Throughput of StreamHub with 100 K subscriptions 60

4.12 Throughput of PADRES with 100 K subscriptions. 62

5.1 Typical volume of ticks (Frankfurt Stock Exchange, 2011/11/18). 66

5.2 StreamHub engine deployed on a public cloud. 68

5.3 The five main steps of a slice migration between two hosts. 71

5.4 Interaction of the e-StreamHub components. 72

5.5 Example of a slice placement decision. 74

5.6 Performance of static e-StreamHub (100 K subscriptions). 77

5.7 Impact of migrations on delay. 79

5.8 Elastic scaling under a steadily increasing and decreasing synthetic workload and 20 K encrypted subscriptions. 80

5.9 Elastic scaling under load from the Frankfurt stock exchange. 81

6.1 Order book, as seen on real trading platform. 86

6.2 Internal structure of the order book. 88

6.3 Order processing time and speedup for different order book implementations on Intel Haswell and IBM POWER8. 101

List of Tables

4.1 Operators supporting scalable CBR. 46

4.2 End-to-end delays (settings as Figure 4.11). 60

5.1 Migration times of operator slices 78

List of Acronyms

AP*	Access Point
API	Application Programming Interface
ASPE	Asymmetric Scalar-product Preserving Encryption

CAS	Compare-And-Set
CBR	Content-Based Routing
CEP	Complex Event Processing
CPU	Central Processing Unit

D*	Dispatcher
DAG	Directed Acyclic Graph
DCCP*	Data Converter and Connection Point

EP*	Exit Point
ESP	Event Space Partitioning

FIFO	First-In First-Out
FPGA	Field-Programmable Gate Array

IaaS	Infrastructure as a Service
IP	Internet Protocol

KPI	Key Performance Indicator

LF	Lock-Free

M* Matcher

NAT Network Address Translation

OS Operating System

PaaS Platform as a Service
PIP* Publisher Interconnection Point

QoS Quality of Service

RAM Random Access Memory

SGL* Single Global Lock
SIP* Subscriber Interconnection Point
SLA Service Level Agreement
SPE Stream Processing Engine

TCP Transmission Control Protocol

VE Virtual Environment
VM Virtual Machine

XML Extensible Markup Language

Units

Digital information

b	bit		
B	byte	1 B	= 8 bit
kB	kilobyte	1 KB	= 10^3 byte
MB	megabyte	1 MB	= 10^6 byte
GB	gigabyte	1 GB	= 10^9 byte

Time

s	second		
ms	millisecond	1 ms	= 10^{-3} second
μs	microsecond	1 μs	= 10^{-6} second
ns	nanosecond	1 ns	= 10^{-9} second

Frequencies

Hz	hertz	1 Hz	= second^{-1}
kHz	kilohertz	1 kHz	= 10^3 hertz
MHz	kilohertz	1 MHz	= 10^6 hertz
GHz	kilohertz	1 GHz	= 10^9 hertz

* These acronyms are specific to this thesis, they were introduced to simplify the figures as well as the text.

Acknowledgments

First, I would like to express my gratitude to my advisor, Prof. Pascal Felber, for all his help and guidance he has given me as well as for the many shared sporting activities during my Ph.D. studies. Pascal has been a continuous source of ideas and motivation in this thesis.

I would like to express my thanks to Prof. Peter Kropf, Prof. Rachid Gerraoui, Prof. Benoît Garbinato, and Dr. Valerio Schiavoni for being part of the jury and for the time and energy that they dedicated to evaluate this PhD dissertation.

I sincerely thank my friends who extensively read my thesis during the various steps of the writing process: Nada Naji, Valerio Schiavoni, and Yaroslav Hayduk.

I would also like to thank Marcelo Pasin, Hugues Mercier, Etienne Rivière, and Emanuel Onica for our fruitful collaboration and all the nice moments spent together.

Special thanks to my friends at the Institute, Dorian, Maria, Mascha, Heverson, Lorenzo, Lucas, Mirco, Patrick, and Raziel for their humor and great discussions.

Other special thanks go to Lawyer Georges Schaller, former President of the Bar Association of the State of Neuchâtel, for ~~putting tremendous pressure~~ *his encouragement* in the writing process of this thesis.

This thesis is dedicated to my family who has given me unconditional love and support in the process of my studies. The work I have accomplished would not have been possible without their love.

Abstract

Publish/subscribe is a popular messaging pattern that provides efficient and decoupled information dissemination in distributed environments. Publishers generate a flow of information as publications, which are routed to subscribers based on their interests expressed as subscriptions.

Matching events against filters has a non-negligible processing cost. Our first contribution within this thesis is the design and the analysis of a generic, modular and scalable infrastructure for supporting high-performance content-based publish/subscribe.

Pub/sub systems must provide high throughput, filtering thousands of publications per second matched against hundreds of thousands of registered subscriptions with low and predictable delays, and must scale horizontally and vertically. As large-scale application composition may require complex publications and subscriptions representations, pub/sub system designs should not rely on the specific characteristics of a particular filtering scheme for implementing scalability. The second contribution of this thesis is the design and the implementation of a novel and pragmatic tiered approach, StreamHub, that offers high-throughput and scalability. We divide the whole process in the three operations involved in pub/sub and leverage their natural potential for parallelization.

In many real-world scenarii, the amount of stored subscriptions and the incoming publications rates varies over time, and similarly their linked computational cost. We propose e-StreamHub, an elastic content-based pub/sub system. The third contribution of this thesis includes: (1) a mechanism for scaling both out and in, of stateful and stateless pub/sub operators, (2) a local and global elasticity policy enforcer maintains high system utilization and stable end-to-end latencies and (3) an evaluation using real-world workload from the Frankfurt Stock Exchange.

Lastly, we focus on a domain-specific application from the financial field, where a data structure, named as order book, is used to store and match orders from buyers and

sellers arriving at a high pace. This application has interesting characteristics as it exhibits some clear potential for parallelism, but at the same time it is relatively complex and must meet some strict guarantees (notably w.r.t the ordering of operations). In this last contribution, we propose several approaches for introducing concurrency in the shared data structure, in increasing the order of sophistication starting from lock-based technique to partially lock-free designs. Corollary we propose a workload generator for constructing histories according to realistic models from the financial domain.

Keywords: publish/subscribe, cloud, scalability, elasticity, order book, concurrency

Résumé

Le système pub/sub basé sur le contenu est un candidat idéal pour concrétiser la communication d'applications à grande-échelle. Il permet de découpler les producteurs de messages (publishers) des consommateurs (subscribers), qui communiquent alors de manière indirecte. Les producteurs génèrent un flux d'informations (publications) qui sont acheminées vers les abonnés en fonction des leurs intérêts (exprimés au travers d'abonnements).

Le filtrage des messages a un coût de traitement non-négligible. La première contribution dans cette thèse est la conception et l'analyse d'une infrastructure générique, modulaire et supportant le passage à l'échelle permettant d'avoir un système pub/sub basé sur le contenu à haute performance.

De tels systèmes pub/sub doivent fournir un débit très élevé, en filtrant des milliers de publications face à des centaines de milliers d'abonnements tout en garantissant une faible latence ainsi qu'un passage à l'échelle horizontal et vertical. La composition d'applications à grande-échelle peut nécessiter des formes complexes de publications et d'abonnements, la conception d'un système pub/sub ne doit pas dépendre des caractéristiques de filtrage particulières pour mettre en œuvre le passage à l'échelle.

La seconde contribution de cette thèse est la conception et la mise en œuvre de StreamHub, une approche à plusieurs niveaux innovante et pragmatique offrant une haute performance ainsi qu'un passage à l'échelle. Nous séparons l'ensemble du processus en trois opérations et tirons avantage de leur potentiel naturel de parallélisation.

Dans les scénarii du monde réel, la quantité d'abonnements ainsi que le débit de publications varie au cours du temps et par conséquent les coûts de traitement y étant liés. La troisième contribution de cette thèse est e-StreamHub, un système pub/sub élastique. La troisième contribution de cette thèse contient : (1) un mécanisme permettant la réduction/augmentation des ressources utilisées, (2) un système global

et local d'application de polices d'utilisation maintenant un usage élevé du système ainsi que des latences stables et (3) une évaluation faite avec des données réelles provenant de la bourse de Francfort.

La quatrième contribution se concentre sur une application spécifique du monde de la finance, dans laquelle, une structure de donnée nommée carnet d'ordres, est utilisée pour contenir et mettre en correspondance les ordres d'achats et de ventes arrivant à une rythme soutenu. Cette dernière a des propriétés intéressantes mettant en évidence un grand potentiel de parallélisation mais est aussi relativement complexe et requiert le respect de certaines garanties (notamment en rapport à l'ordre des opérations). Nous proposons de nombreuses approches pour tirer profit de la concurrence dans une structure de donnée partagée, en augmentant le niveau de sophistication en partant de solutions basées sur des verrous jusqu'à des conceptions partiellement sans verrouillage. Comme corollaire, nous proposons un générateur de données historiques synthétiques suivant des modèles réalistes venant de l'éconophysique.

Mot-clés : publish/subscribe, informatique en nuage, passage à l'échelle, élasticité, carnet d'ordres, programmation concurrente

"La capacité d'imaginer est

plus apaisante que la vie elle-même..."

Tahar Ben Jelloun

1

Introduction

1.1 Context

1.1.1 Publish/Subscribe

Publish/subscribe, often named as pub/sub, is a messaging pattern where senders/publishers do not program explicitly neither specify to which receivers/subscribers the messages are supposed to be delivered. On the other end, subscribers only express interest using subscriptions and only receive the messages that are of interest [41] (illlustrated in Figure 1.1).

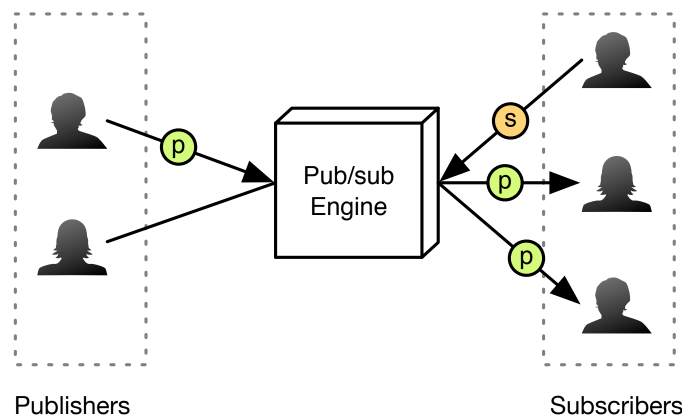


Figure 1.1 – A publish/subscribe engine.

Topic-Based

In topic-based publish/subscribe [19], subscribers register to a set of topics (also referred to as channels or subjects). Each publication sent to the system is tagged with a specific topic and then delivered to all subscribers who registered to that topic. The topic-based publish/subscribe pattern is widely used, and there are plenty of fast implementations that are basically looking up, for each incoming message that has a given topic, which are the interested subscribers and then delivering that message to all of them. The main downside of topic-based publish/subscribe is its lack of expressiveness and flexibility.
Examples of topic-based publish/subscribe systems include iBus [3], SCRIBE [28] and Bayeux [120].
Several implementations of topic-based publish/subscribe are used within the industry:
PaaS [1]/Cloud-Based: Major cloud computing services provide scalable topic-based publish/subscribe: Google Cloud Pub/Sub [2], Amazon Simple Notification Service, [3] and Microsoft Azure Service Bus. [4]
Libraries: øMQ provides among others an implementation of topic-based publish/subscribe. [5]
Brokers: The Java Message Service specification (JMS) [38] includes topic-based publish/subscribe. Broker are standalone applications, and they require the use of dedicated connectors that implements the JMS specifications. This adds another level of decoupling since applications using JMS are not supposed to have any dependency against a specific implementation. The connector can be selected at runtime. Several brokers implement the JMS specifications, Apache ActiveMQ, [6] and RabbitMQ, [7] just to name a few. Google provides JMS bindings for the Google Cloud Pub/Sub [8] as well.

[1] PaaS, Plateform as a Service is defined in Section 1.1.2
[2] Google Cloud Pub/Sub: `https://cloud.google.com/pubsub/`
[3] Amazon Simple Notification Service: `https://aws.amazon.com/sns/`
[4] Microsoft Azure Service Bus: `https://azure.microsoft.com/en-us/services/service-bus/`
[5] øMQ pub-sub pattern: `http://zguide.zeromq.org/php:chapter5`
[6] ActiveMQ: `http://activemq.apache.org`
[7] RabbitMQ: `https://www.rabbitmq.com`
[8] Google Cloud Pub/Sub: `https://github.com/GoogleCloudPlatform/pubsub`

Content-Based

In content-based publish/subscribe, publications are not attached to a specific topic. The publications are analyzed at runtime by the engine to determine which subscribers should receive a given publication. In the usual model, publications are a set of attribute-value pairs (or a map), attributes are usually strings and values are commonly strings, integers, boolean, floating-point numbers but can virtually be anything. The subscriptions can select publications of interest via a set of predicates, that are constraints over attributes. This variant of publish/subscribe is way more expressive than the topic-based one. Subscribers are not limited anymore to the selection of a topic among a set of predefined ones, and conversely the consumers can build any combinations of constraints over the attributes. Examples of content-based publish/subscribe systems includes Siena [26], XSiena [66] and PADRES [63]
Content-based publish/subscribe might be modeled as a superset of topic-based publish/subscribe. Conceptually a content-based publish/subscribe engine can be easily adapted to handle a topic-based workload.

Popular Message Pattern

Both variants of publish/subscribe, topic-based and content-based, are widely used in nowadays applications. Classical examples of publish/subscribe include event-driven applications such as the delivery of financial quotations disseminated to several interested parties, sports reporting, traffic condition monitoring, modern multimedia applications, and monitoring of the Internet of things devices [41, 51, 67, 103].

1.1.2 Scalable Cloud Architectures

The Shift Towards Multi-Core

Computer scientists are familiar with Moore's law that states that the number of transistors on an integrated circuit doubles approximately every eighteen months. A corollary to this observation was an awesome side effect, an extremely convenient fact that the clock speeds were following the same rules up to early 2000's. This situation was somehow comfortable for the programmer since every new CPU generation was running any piece of software faster.
While being able to still follow Moore's law in term of the number of transistors, CPU manufacturers were not able anymore to increase clock speeds since 2004. This

stagnation is mainly due to two reasons: higher-frequencies started to be way too greedy in term of electric power and that the transistors switching frequencies were not evolving anymore. At this time began a new era, the multi-core era, instead of increasing the CPU frequency of a single processor, they started to build chips containing multiple processors and cores.
Thanks to that, on multiple cores processors, the aggregated number of computations done by a unit of time on a single chip is still following Moore's law. Taking advantage of these architectures is obvious when running several different programs in parallel but can be much more complex when a single program needs to benefit of this parallelism. This kind of parallelism is referred to as *vertical scaling* within this thesis.

The Shift Towards Clusters

There are plenty of tasks that are heavily greedy in term of computing resources, to name a few, let us cite MRI scanners image rendering, weather forecast, indexing a massive corpus of documents, and 3D raytracing.
For these heavy tasks, computer manufacturers built specific hardware to sustain the load. These extraordinary architectures ended to have sometimes a higher price per computation unit (in term of hardware and electricity consumption) than standard personal computers available on the market.
Some nice examples of this shift might be the rendering of images of the blockbuster movie "Titanic", that was done on a cluster of affordable PCs running on Linux [108]. Prior to this movie most of the movies were rendered on expensive dedicated super-computers.
Another meaningful example is the indexing of web pages done by Google, at a time where most of the competitors were using mainframes (such that Altavista). Google decided to use a large number of affordable PCs in order to guarantee a lower cost of this process. Google built a programming model named *MapReduce* [39] that allows the programmer to easily split of a set of heavy computations within a set of nodes. This kind of parallelism is referred to as *horizontal scaling* within this thesis.

Decoupling the Software and the Infrastructure

We define *computer virtualization* as the act of creating a *virtual computer* that runs within an actual computer.

It started in the 60's and was quite popular in both the computing industry and academic research, then virtualization lost popularity until the end of the century, mainly due to the falling cost of hardware as well as the sophistication of multi-user operating systems [25].
At the end of the 90's the popular computer architectures did not provide any support for virtualization. In 1997, Ed Bugnion and al., introduced in Disco [24], an efficient implementation of full virtualization on the MIPS architecture with an elegant trapping mechanism. Disco was emulating at this time only the Silicon Graphics's MIPS R10000, but the paper was already describing how that technology could be applied to the x86-32 [9] architecture. In 1998, Ed Bugnion and al. [25] started a company named VMWare and brought virtualization to the x86-32 architecture. Initially, this technology aimed at addressing interoperability, operating system migration, reliability, and security. Virtualization of the x86-32 architecture was a challenging problem, due to the lack of hardware supported virtualization and the complexity of the x86-32 architecture.
Operating-system level virtualization or *containerization* is another mechanism that refers to an operating system feature that allows the existence of multiple isolated user-space *instances*. This approach, that might be considered as an extended *chroot*, offers many of the advantages of virtualization with a negligible overhead and no need of specific hardware feature to run. Many implementations came up like BSD Jails on FreeBSD, OpenVZ on Linux and more recently the Linux kernel introduced a functionality named *cgroups* that helped the creation of Linux Containers (LXC). Thanks to the negligible overhead [57] of this kind of virtualization, it became convenient to dedicate containers to single applications. More recently, operating-system level virtualization is being used along with union-capable file system such as OverlayFS and others. Especially with a software technology named Docker [89].
The research done within this thesis that was relying multiple computing instances (except in *Application to Online Trading* (Chapter 6)) was done using OpenNebula, [10] a cloud computing virtualization platform and using KVM [11] as hypervisor.

[9] In this thesis, x86-32 refers to Intel's 8086 32-bits architecture
[10] OpenNebula, `https://www.opennebula.org`
[11] KVM (Kernel-based Virtual Machine), `https://www.linux-kvm.org`

The Rise of Cloud Computing

Cloud computing came out during the 2000s, even though its roots are older as well as its metaphoric name[54]. Two decades before, the cloud symbol was already used to represent computing and networking infrastructure in the documentation of Internet ancestors, in 1977 for ARPANET, and in 1981 [12] for CSNET [13].

On-Premise refers to a scenario in which the user owns or rents physical machines and is responsible of everything starting from the hardware up to the management of the software. On-premise is the term used to define the approach prior the cloud computing era.

Infrastructure as a Service (IaaS) refers to services that free the user of taking care of the underlying infrastructure, such as data storage, network interfaces, backups. Commonly, the IaaS is built using virtualization, thanks to this, the users has an uniformized view of the actual hardware.

Provisioning and monitoring compute, networking, storage, and other services is done in a self-service manner (with the help of a web interface and/or directly with publicly available API).

IaaS examples include but are not limited to Amazon Web Services (AWS), Google Compute Engine (GCE), Microsoft Azure, Digital Ocean.

Platform as a Service (PaaS) refers to services that provide cloud components to other softwares. These services might be databases, mail transfer agent, load balancer, web applications containers, or others.

Commonly, PaaS offers many advantages such that: reducing the amount of coding, offering scalability and high availability.

PaaS examples include, but are not limited to AWS Elastic Beanstalk, Windows Azure, Heroku, Google App Engine, Amazon Relational Database Service Google Cloud Pub/Sub, Amazon Simple Notification Service, and Microsoft Azure Service Bus.

Software as a Service (SaaS) refers to services that provide the full application to final users. It defines software that are actually not running on the computer of the user.

[12] ARPANET: `https://www.computerhistory.org/internethistory/1970s/`

[13] CSNET: `http://gu.friends-partners.org/Bookwriting/PART_I/Chapter_I/Total/Insertions/NSF/CSNET/CSNET_2P.html`

For the users, this approach comes with some advantages like not having to manage the software, to install it neither to update it, no need to care about backups. The main drawbacks are that the application requires an Internet connection to work properly.
Usually, this approach comes with another kind of billing, instead of buying the software, the user rent it.
SaaS examples include, but are not limited to Google Apps, Microsoft Office Online, SalesForce, Dropbox, and Github.
Bare-Metal Cloud refers to service that provides physical machines with an ease comparable to IaaS. As seen previously IaaS is built on top of virtualization, it comes with many advantages but has still some drawbacks such that the perturbation on performances done by "noisy neighbors" on a shared instance, the inability to access a specific hardware through the virtualization layer, or more importantly some concerns about security. Virtualization security relies much on the isolation mechanisms provided by the hardware, sadly these ones are still not fully bulletproof; exploits include gain of kernel privileges [102] and reading host memory [72, 80]. To provide a similar comfort and flexibility with physical hardware, another model emerged, *bare-metal servers*, a service in which physical machines can be provisioned, un-provisioned with the same ease than with virtualization [91].
The cloud computing era started two decades ago and major industry leader entered the race. In August 2006, Amazon created a sub-organization named Amazon Web Services (AWS) and announced the public availability of Elastic Compute Cloud (EC2), an IaaS that allows to run virtual machines instances on their infrastructure. In April 2008, Google introduced Google App Engine, a PaaS that allows programmers to write computer programs, that run on their infrastructure within a specific framework (a handful of computer languages are supported). Later, in February 2010, Microsoft joined the race with Microsoft Azure.

Tribute to Open-Source Software

Cloud computing is somehow a big stack of various technologies that contains operating systems, multiple types of virtualization, orchestration mechanisms, distributed datastores, large-scale data processing software, etc. Development of each brick of that set of technologies is complex and requires much work. We can easily argue that

without open-source software contributions, technology improvements would not have gone that far.
In the last two decades, contributions of open source to the various aspects of computing are tremendous.
Open-source not only powers crucial parts of our cellphones, tablets, browser, or networking gears, but it is also behind many of the technologies used in the cloud.

1.1.3 Online Trading

High-Frequency Trading

A given asset can be traded on different markets, and under certain circumstances, it might be priced differently on two different ones. Knowing that information before all other investors is a great advantage, since it allows an investor to make a safe and profitable pair of transactions; buy the products where it is cheaper and sell it where its price is higher. We denote the technique described above as *market arbitrage*, it relies mainly on having the financial informations quickly and taking the decision before adversaries.
According to this, it became soon a technical challenge to transmit extremely fast this data from one stock exchange to the other. We note that in the early 1930's telegram communications were established between North-America East-Coast and Europe, with bandwidth reaching already 3'200 characters per minute for this specific purpose [90].
In 1971, the National Association of Securities Dealers founded the Nasdaq [14] Stock Market, the world first electronic stock-market [109]. At the time of its launch Nasdaq was mainly operated as a bulletin board, fully electronically operated markets came a bit later. GLOBEX trading system introduced in 1992, provided an full-fledged electronic stock-market that can handle trades electronically. Since then, most stock-exchanges moved quickly to systems that are fully electronically operated, allowing traders to do various kind of arbitrage or apply other algorithms at a very high pace.
Arbitrage techniques rely on having the best selling and buying price but they also benefit of extra information. Having, in a fast manner, a more in-depth knowledge such as all potential buyers and sellers, with the quantities (referred to volume) and

[14] Nasdaq was initially an acronym for the National Association of Securities Dealers Automated Quotations

the price at which they are willing to trade, gives a great advantage. The structure that holds this information is named *order book*. In Chapter 6, I will present various algorithms that leverage concurrency to speed up its management.

1.2 Motivations and Contributions

1.2.1 Scalable Publish/Subscribe Engine

In Section 1.1.1, I argued that the publish/subscribe pattern is heavily used, and that nowadays many applications are relying on it. Many of these applications are already running on the cloud, and they need an engine that is scalable.

Our first contribution within this thesis is the design and the analysis of a generic, modular and scalable infrastructure for supporting high-performance content-based publish/subscribe.

An implementation should scale vertically, and horizontally, and compete in the same league as existing implementations. The evaluation of the performance is done with various number of computing nodes, subscriptions, and with different types of messages (i.e., various distribution of messages according to their similarities). The metrics that will allow the comparison of the engine and its variants of our algorithms against other well-established ones are as follows:

— **latency:** measure of the time delay required for message to travel across a engine (between submission and delivery)

— **throughput:** maximum number of events that can be handled per unit of time.

The second contribution of this thesis is the design and the implementation of a novel and pragmatic tiered approach, StreamHub, that offers high throughput and scalability.

Furthermore, the solution should provide elasticity in order to sustain a load that is varying. The impact of this feature has to be evaluated and should be kept as small as possible.

We propose e-StreamHub, an elastic content-based pub/sub system. The third contribution of this thesis includes: (1) a mechanism for scaling both out and in, stateful and stateless pub/sub operators, (2) a local and global elasticity policy enforcer maintaining high system utilization and stable end-to-end latencies and (3) an evaluation using real-world workloads from the Frankfurt Stock Exchange.

The architecture of our system is based on a Complex Event Processing named StreamMine3G developed in collaboration with colleagues at TU Dresden, has permitted to decompose the process in stages of operators.

1.2.2 Concurrent Order Book

In section 1.1.3, I argued that time that trading market financial applications like trading require a fast representation of the order book (and obviously an excellent connection to the stock exchanges). Current matching engines work sequentially [93] on dedicated field-programmable gate arrays (FPGA) [77] and require specific connections [1].

Chapter 6 evaluates the possibility of leveraging parallelism into this problem. The potential for performance improvements by taking advantage of the parallel processing capabilities of modern multi-core CPU gives us a great incentive.

In this last contribution, we explore various matching algorithms and data structures in order to reduce latency. We compare a sequencial algorithm, a lock-based algorithm, and some more sophisticated partially lock-free algorithms.

In order to do proper evaluation of the aforementioned algorithms, we need to build a workload that has similar properties than the real market. Considering that real market data are hardly available (either there are extremely expensive or are incomplete), we de decide to build a generator that follows state-of-the-art models used in econophysics [10, 15, 86].

1.3 Organization of the Thesis

The rest of this thesis is organized as follows: In Chapter 1, I present the context on content based publish/subscribe, scalable cloud architectures and online trading mechanisms, as well as the motivation of this thesis. In Chapter 2, I present related work. In Chapter 3, I show how a content-based publish/subscribe system can be properly provisioned in order to run in the most efficient way on a IaaS environment. In Chapter 4, I show how a content-based publish/subscribe system can be implemented to offer scalability properties. Later in Chapter 5, I present a refinement of the system which provides elasticity to the system, that can dynamically scale according to the load. In Chapter 6, I propose various approaches to maintain an order book and do

the actual matching of stock exchange orders in a parallel manner. Conclusion and future directions are presented in Chapter 7.

"Wie komm ich am besten den Berg hinan?" —
— *"Steig nur hinauf und denk nicht dran!"*
Friedrich W. Nietzsche, *Die fröhliche Wissenschaft*

2
Related work

2.1 Publish/Subscribe

Content-based Pub/Sub systems have been widely studied by the research community in the past years. By design, they are conceived to support a large number of users. In this section, we focus on related work close to the objectives of our research, i.e., Pub/Sub infrastructures designed to dynamically scale with various parameters, and papers that study the behavior of such systems analytically.

2.1.1 Infrastructure Provisioning

Analytical Frameworks for Pub/Sub

Raiciu *et al.* [95] studied the dimensioning and scaling of content-based Pub/Sub systems. They first analyzed a few applications suitable for content-based Pub/Sub and extracted their characteristics. They discovered that these applications exhibit significant diversity, from input loads (publishers bandwidth as well as publication frequency) to requirements (tolerable latency). They focused their study on throughput and latency, and analyzed their effects on the architecture parameters. However, they do make several unrealistic assumptions, such as using communication links with infinite bandwidth.

Wang *et al.* [113] studied the impact of subscriptions partitioning and existing routing algorithms on the throughput of Pub/Sub platforms. They mainly focused on the

impact of subscription summaries on the routing algorithm rather than the impact of subscription partitioning.

In [99], Rose *et al.* proposed Cobra, a provisioning technique for aggregating blogs and RSS feeds. In a sense, it operates like a search engine, as one of its operators is a *crawler* which retrieves information. Its similarity with our work is its decoupled architecture between crawlers, filters, and reflectors as well as its off-line service provisioning technique to determine the minimal amount of physical resources required. However, while the general ideas are similar, their implementations are significantly different. The purpose of each stage is different, therefore leading to different analytics. Cobra's filtering stage considers text matching algorithms, while we allow more general filtering algorithms and adapt our architecture accordingly.

Yoon *et al.* [118] also investigated provisioning techniques when SLAs need to be enforced for a Pub/Sub service. The provisioning is accompanied by broker topology transformations and only applies to reverse-path forwarding-based systems where each broker endorses all roles (contact point, matcher, dispatcher for other brokers).

Content-Based Pub/Sub

SIENA [27] is one of the first content-based Pub/Sub frameworks. It uses a distributed network of dedicated event brokers and reverse path forwarding. Brokers assume all roles, serving as contact points for publishers and subscribers, matching events, and forwarding events to other brokers who may have at least one matching subscription. The matching operation is performed several times in order to determine the propagation of publications to brokers. The SIENA algorithms have become reference solutions for the problem of routing content-based events and subscriptions in an application-level network.

PADRES [63] is a Pub/Sub system with the capability to produce complex correlations among both past and future events. The system, using advertisement-based routing, includes algorithms to optimize the retrieval and correlation of heterogeneous data streams. It also exploits the relationships between subscriptions (equivalence, containment), supports multiple data partitioning schemes (each with its own trade-offs), and handles network failures. Recent work on PADRES, such as that by Cheung and Jacobsen [33], adds load-balancing to increase its scalability.

Scalable Pub/Sub Architectures

The need for designing Pub/Sub infrastructures that can scale with large user and subscription populations has long been recognized. Classical broker-based systems such as SIENA [27] or PADRES [63] exploit similarities between subscriptions (in particular containment relationships) to improve the matching speed. More specifically, when subscriptions are aggregated along the way from consumers to producers of information, taking advantage of containment relationships between subscriptions: a single aggregated subscription may represent the interests of many downstream subscriptions, thus reducing the number of subscriptions managed by the broker and improving filtering performance (see for instance [64]). Generally, such mechanisms produce higher gains when the considered subscriptions are similar or describe overlapping interests. By clustering the subscribers in a smart way, one can scale to large user populations with a reasonable number of brokers. These systems are designed to be deployed in large-area networks and do not explicitly consider deployments in clusters with a given scalability objective. In broker overlays (illustrated in Figure 2.1), under some workloads, publications may have to traverse a number of brokers that have no local interested subscriber but still have to filter the publication against stored subscriptions. These are called *forwarder-only* brokers. While techniques for rewiring the broker overlay have been proposed to tackle this problem [69], the presence of such forwarder-only brokers is intrinsic to a design where communication flows depend on the filtering scheme and on the current workload of stored subscriptions. Since all brokers play all roles in the pub/sub operation, the allocation of publishers and subscribers to brokers has a strong impact on the balance of load and on the overall filtering efficiency. A bad placement may result in a high number of messages being propagated between brokers. Some optimizations were proposed to address this problem by connecting subscribers with similar subscriptions to the same brokers [31], or by linking publishers and their expected subscribers to the same brokers [32, 79]. Cheung *et al.* [33] proposed to use similar techniques to rewire the PADRES overlay in order to reduce the environmental footprint of the pub/sub system. Again, these mechanisms are dependent on the filtering scheme and require the ability to determine proximity relations between subscribers and publishers. This would not be possible, for instance, with encrypted filtering approaches.

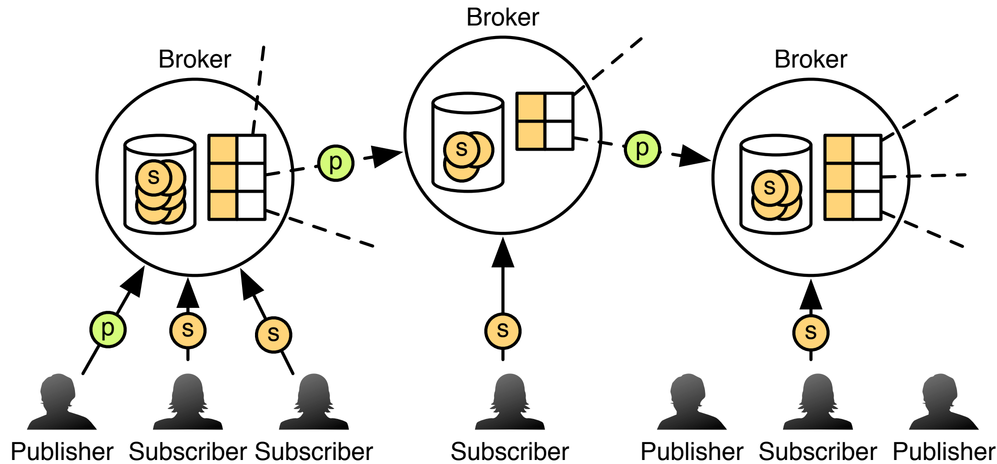


Figure 2.1 – A broker overlay network for publish/subscribe.

In contrast to systems based on overlay of brokers, the logical connections between the elements in our proposed architecture are independent of the nature of the subscription and publication workloads and of the nature of the filtering scheme. We support scaling each of the pub/sub operations independently by simply adding more processors to the set of nodes that support this operation. We do not require any specific optimization support from the filtering scheme, though we can leverage their existence for improving single-node performance inside filtering libraries. Note that our approach is readily applicable to architectures like Google's GooPS [97], where pub/sub is implemented by regional data centers consisting of clusters of brokers and interconnected by dedicated network links. In the context of content-based routing of XML documents, Felber *et al.* [45] proposed several scalable strategies for parallel filtering that offer various trade-offs in terms of throughput, latency, resource consumption, and complexity. They can be combined together into a hierarchical routing architecture that can adapt to the specifics of individual services. These strategies explore similar trade-offs to those discussed in Section 3.3, but at a much coarser level. There is no generalization of the model nor any analytical study.

Baldoni *et al.* [11] underlined the lack of self-organization capabilities of current content-based Pub/Sub systems, thereby requiring human intervention for set-up and management of the application-level network. Their solution describes a fully decentralized Pub/Sub system, where all nodes act as both publishers/subscribers and brokers. Self-configuration is achieved through the use of configuration middleware between the overlay supporting event propagation and the subscription layer that

deals with subscription installation. The configuration middleware system decides on subscription-to-node and event-to-node mappings for better performance in this peer-to-peer context.

Baldoni *et al.* [12] also tackled the problem of event dissemination and proposed a novel approach by reorganizing the overlay topology. They presented a self-organizing overlay network based on the principle that brokers matching similar events should be located as close from each other as possible.

Farroukh *et al.* [44] also used event partitioning. They introduced a parallel matching engine on current chip multi-processors to increase the throughput and reduce the matching time. Their engine exploits the characteristics of multi-core computers to partition the events and perform the processing.

Filtering Mechanisms

Our architecture supports *pluggable* filtering mechanisms. As the design of new such mechanisms is not the focus of this paper, we only briefly discuss below a few well-known algorithms that can be readily used in our StreamHub implementation (see Section 4.3.2). Note that this list is far from being exhaustive.

SIENA uses a counting algorithm [26] for efficiently matching publications against subscriptions. Individual predicates are stored in a forwarding table and a subscription is detected as matching when all its predicates have been encountered. We use an implementation of this counting algorithm as the default filter in StreamHub for non-encrypted matching. Additional details on its operation are given in Section 4.3.2. Encrypted filtering can be supported for instance by *asymmetric scalar-product preserving encryption* [34], combined with pre-filtering [13] for efficiency. Gryphon [2] inserts the set of subscriptions into a matching tree: leaves contain subscriptions, non-leaf nodes contain tests, and outgoing edges represent the results of the tests. A publication traverses down the tree by following all matching paths and reports a matching subscription for each leaf node reached. PADRES uses a scalable filtering engine [44] that can leverage multiple cores on a shared memory architecture. By splitting the state of subscriptions and using multiple threads synchronized using either locks or transactional memory, the filtering throughput is significantly improved. We also exploit all the cores available on a machine and provide synchronization mechanisms for concurrent accesses to a shared state.

Other examples of filtering mechanisms that can be leveraged in the context of StreamHub include, but are not limited to, the following. RAPIDMatch [68] is a tree-based filtering mechanism that takes into account the sparseness of criteria definitions over the whole attribute set in some pub/sub workloads for greater efficiency. TAMA [119] trades accuracy and space complexity for efficiency by clustering range-based subscriptions in predefined sets based on discrete cuts of the definition range of each attribute. The use of discrete cuts leads to the presence of false positives, while the presence of subscriptions in multiple buckets leads to higher memory consumption, in return for faster filtering. Fabret *et al.* [42] use schema-clustering to minimize the number of filtering operations performed, along with techniques to improve cache performance of the algorithm. Filtering mechanisms also exist for boolean expressions [20, 47], XML documents and XPath expressions [4, 29], and compact data representation using Bloom filters [65].

Clustering Subscriptions

Similarly to filtering mechanisms, we support subscription clustering by the means of pluggable libraries. Subscription clustering, as described in Section 4.3.2, splits the whole set of subscriptions maintained by the pub/sub system into clusters according to similarity (if a proximity metric is available on the subscription). This typically increases the level of containment and the potential for aggregation, which in turn improves the performance of the filtering operation. Classical clustering algorithms include K-means [114], Event Space Partitioning (ESP) [112], or R-trees [16].

2.1.2 Elasticity

In this section we present an overview of related work with specific focus on elastic platforms, event processing systems, and pub/sub middleware.

Elastic Platforms and Infrastructures

Elastic infrastructures and platforms as a service (IaaS and PaaS) support ad hoc addition and removal of virtual machines (VMs) to/from a virtual environment (VE). A typical example is the Amazon EC2 Auto Scaling service [6]. It relies on basic elasticity policies by setting simple thresholds on resource utilization. The *elastic site* mechanism for Nimbus and EC2 clouds proposes more elaborate policies, for

workloads composed of independent tasks under batch scheduling [82]. AutoScale [49] is an example of an adaptive IaaS elasticity policy, where the goal is to maintain the minimum amount of spare capacity to handle sudden load increases while keeping the total number of servers as low as possible.

All elastic scaling solutions at the IaaS level require that the VMs composing the VE, as well as the jobs that run on these VMs, are stateless or independent. These requirements are not fulfilled by the nodes of a pub/sub system, which require application-level elasticity mechanisms. Elastic applications interact with elastic IaaS using the VM allocation and de-allocation APIs of the IaaS elasticity manager.

Elastic Complex Event Processing

Both complex event processing systems (CEP) and stream processing engines (SPE) process queries composed of directed acyclic graphs of stateful or stateless *operators* processing data in the form of *event streams*.

Fernandez et al. [46] propose application-level scale out for stateful operators, integrated with passive replication (checkpointing). The integration is made possible by explicit operator state management, an approach similar to that of e-StreamHub. The operator state is dynamically split to form new partitions deployed over multiple machines when scaling out, potentially requiring application support for partitioning. e-StreamHub supports full elasticity, i.e., scale *out* and *in*, while optimizing the overall system utilization, and without requiring specific application support. Its underlying runtime engine can support both passive [85] and active [84] replication.

In the context of IBM's System S [115], Schneider et al. [101] propose an extension of the SPADE domain-specific language to support parallel operators on a single host. e-StreamHub supports elastic parallelization within a single host as well as elastic distribution across multiple hosts.

ElasticStream [62] proposes to outsource parts of a System S SPE deployment to a public cloud. It periodically adjusts the deployment based on decision from a linear programming formulation of the allocation problem, where the goal is to minimize the monetary cost of the cloud usage. e-StreamHub elastically scales based on the *current* state of the system, immediately addressing load variations.

StreamCloud [52] monitors the average load per cluster in order to detect under- and overload situations. When an overload is detected, StreamCloud triggers pair-wise

re-balancing between most and least loaded hosts. StreamHub elasticity policies share this goal of rebalancing the load but with the additional goals of minimizing the amount of migrations and the support of migrations with minimal interruption of the flow.

Elastic Publish/Subscribe

Topic-based pub/sub engines used in the industry, such as Apache Kafka [73] or Hadoop HedWig [7], typically support *incremental scalability*. This refers to the ability to add/remove support servers in order to achieve linear scalability in throughput or number of topics. None of these systems supports automated addition and removal of servers based on the experienced workload, and therefore cannot be classified as elastic. e-StreamHub supports *incremental scalability* and *elasticity*, and considers the more general and challenging content-based model.
EQS [111] is a message queue architecture that can be classified as topic-based pub/sub. Elasticity in EQS is achieved by monitoring the load on each topic and migrating topics between hosts. This imposes a potential cost in terms of service interruption. In contrast to EQS, e-StreamHub targets content-based filtering where the destination of events can only be determined at runtime. It also migrates parts of the load to new hosts when overload is detected, but with the objective of minimizing service interruption thanks to an application-level migration mechanism.
Hoffert et al. [55] consider the problem of predicting the load requirements for an adaptive topic-based pub/sub service running in a cloud environment. Using supervised machine learning, the approach is able to dynamically configure the service (e.g., the transport protocols used) for different QoS requirement, but does not consider the dynamic addition or removal of hosts for supporting the service.
BlueDove [78] targets attribute-based pub/sub deployments on public cloud services. The message flow in BlueDove is strictly dependent on the attribute-based filtering model used for subscriptions: the attribute space is split in regions, and subscriptions and publications are dispatched to the matching servers that are in charge of an overlapping region. This disallows the use of filtering schemes that are not based on low-dimensionality attribute spaces or that do not allow examining the content of the subscriptions at the server side, such as with encrypted filtering. Finally, BlueDove supports only scale out, therefore, unlike e-StreamHub, it cannot be described as

fully elastic. Fang et al. [43] follow a similar approach to BlueDove and only scales out.

2.2 Online Trading

Strategies for optimizing the operation of matching platforms can be broadly divided into two categories: the reduction of latency and optimizations related to order processing. To achieve ultra-low latencies, high-frequency trading servers are typically housed in the same building as the matching engine servers [1]. Additionally, novel communication technologies, such as microwaves, are gaining popularity as they promise to convey orders faster than fibre optic [105].

Significant effort was also spent towards efficient middleware systems for order handling, besides the matching operation itself. *LMAX Disruptor* [110] is an integrated trading system running on the JVM. It implements the reception and pre-processing of orders. It stores the received orders in a queue with ordering guarantees similar to the *incoming* queue used in our algorithms. Disruptor features a simple single-threaded matching engine that fetches and process orders from the queue sequentially, but it also allows the implementation of more sophisticated matching or order processing engines including those using multiple-threads implementation. It is therefore complementary to our study, which concentrates on the internal of the matching engine.

Although, to the best of our knowledge, there does not exist concurrent lock-free implementations of matching engines, substantial effort has been dedicated to developing efficient single-threaded implementations. For instance, Shetty et al. [104] propose such an implementation for the .NET platform. The authors detail the steps required for locking the order book when accessing it from multiple threads concurrently, similarly to our baseline coarse-grain locking algorithm.

In addition to the models for generating orders that we mentioned in Section 6.4, several authors investigated the *dynamics* of order books. Huang et al. [58] propose a market simulator to help compute execution costs of complex trading strategies. They do so by viewing the order book as a Markov chain and by assuming that the intensities of the order flows depend only on the current state of the order book. Cont et al. [36] propose using a continuous-time stochastic model, capturing key empirical properties of order book dynamics. Alternatively, Kercheval et al. [70] use a machine

learning framework to build a learning model for each order book metric with the help of multi-class support vector machines.

"En vérité, le chemin importe peu,
la volonté d'arriver suffit à tout."

Albert Camus, *Le Mythe de Sisyphe*

3 Infrastructure Provisioning

This chapter is based on a paper that was previously published:
Infrastructure Provisioning for Scalable Content-Based Routing: Framework and Analysis. In *NCA 2012: 12th International Symposium on Network Computing and Applications*, pages 228-235, Cambridge, MA, USA, Aug 2012. IEEE.

3.1 Introduction

The publish/subscribe (Pub/Sub) communication model [41] is recognized as one of the approaches of choice for the construction of large-scale complex applications. By decoupling the communication between producers and consumers of information both in space and time, it hides the complexity of information routing between application components and presents a simpler, data-centric view to programmers. The publishers of information send publications, or *events*, to the system which is then responsible for dispatching them to the subscribers who previously expressed their interest in those events. One usually distinguishes between Pub/Sub systems based on the model used for expressing interests, by means of subscriptions sent to the system. *Topic-based* Pub/Sub, on the one hand, provides a set of predefined *topics*. Implementations are typically supported by a group membership protocol and application-level multicast within the group. Content-based Pub/Sub, on the other hand, allows expressing subscriptions as predicates on the *content* of events. This chapter focuses on the

content-based Pub/Sub model. This model, while more expressive, is also more complex to support. Unlike topic-based Pub/Sub, it is not possible to pre-compute the set of subscribers who will be interested in a given set of events: *filtering* of events against the subscriptions must be performed for each new event.

We assume in this chapter that the content-based filtering is supported by a set of dedicated servers or *brokers*, as implemented for instance in Siena [27], PADRES [63] and XNet [30]. The brokers need to establish and maintain communication channels with the publishers in order to receive new events, and with subscribers to dispatch matching events and receive subscriptions and cancellations. Each broker implements the *content filtering* operation by *matching* all the events against its set of stored subscriptions.

Providing scalable support for content-based Pub/Sub requires being able to scale three operations: (1) communication from the publishers, (2) communication from and to the subscribers, and (3) matching the publications against the subscriptions. The resources of each individual server composing the infrastructure are obviously constrained, and the bandwidth for handling communications to and from publishers and subscribers is also limited. Moreover, the number of connections any single server can handle is constrained by OS-specific limitations, and the limited processing power for the matching operation affects the filtering delay. Achieving scalability requires splitting the load of the three aforementioned operations so as to seamlessly scale as the number of publishers, subscribers, publications and subscriptions varies.

Contributions. In this chapter, we focus on the provisioning of a scalable infrastructure for content-based Pub/Sub systems. More precisely, we propose and describe a generic architecture that solves the scalability aspects of content-based routing systems and show how this infrastructure scales in a practical setting. We expect the results to help designers and engineers determine the requirements of the connection, routing, and filtering operators as the load varies. The work presented in this chapter is conceptual in nature and part of a larger project whose goal is the implementation of a fast, scalable and secure content-based Pub/Sub system.

Outline. The remaining of this chapter is organized as follows. Section 3.2 presents the infrastructure and defines its operators. Section 3.3 describes how to scale the components of our generic architecture for content-based routing. Section 3.4

generalizes our approach to a stage-driven design. Section 3.5 presents experimental results, Section 2.1.1 discusses related work, and finally Section 3.6 concludes.

3.2 Infrastructure Model and Assumptions

In this section, we describe our generic architecture as well as its basic operators required for a content-based system. Our Pub/Sub architecture model is organized with a collection of operators organized in stages, resulting in a directed acyclic graph traversed by every publication from a publisher to zero or more subscribers. We present our working assumptions in Sections 3.2.1 and 3.2.2, and describe the architecture starting from Section 3.2.3.

3.2.1 Communication Assumptions

In order to analyze the communication constraints of the Pub/Sub architecture, we use the *bounded multi-port* communication model [56], which limits the total amount of data that a node can *send* and/or *receive* per time unit. This is a realistic assumption, and intuitively the bound corresponds to the bandwidth capacity of the node's network card. The flow of data out of the card can be either directed to a single link or split among several links (likewise for the flow of data to the card) hence the multi-port hypothesis. Simultaneous sends and receives are allowed as long as the bound is satisfied. In the remaining of this chapter, the multi-port constraint is denoted as the bandwidth cap.

We also assume that the communication capacity between the operators is homogeneous (i.e., same network links, but connecting machines with heterogeneous speeds and network cards). This assumption is realistic if the targeted deployment environment is within a single data center. The study of a wide-area system over the Internet will be considered in future work.

3.2.2 Workload Assumptions

We assume that the throughput of incoming subscriptions and cancellations is negligible compared to the throughput of publications. This is representative of Pub/Sub applications and workloads, where subscriptions remain unchanged for long periods of times while publications are constantly flowing through the system. We also assume

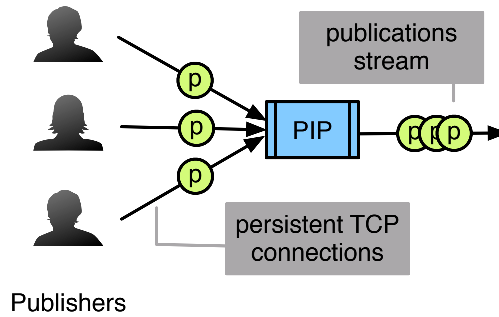


Figure 3.1 – A Publisher Interconnection Point operator (PIP).

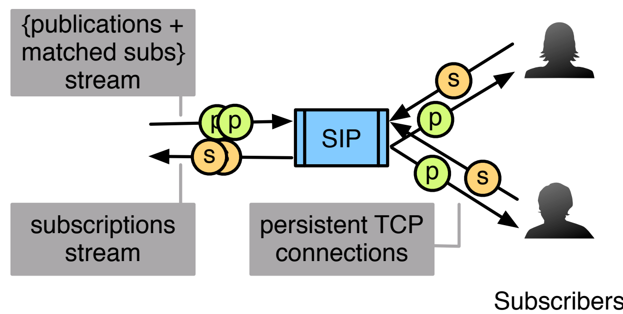


Figure 3.2 – A Subscriber Interconnection Point operator (SIP).

that the machines doing the filtering have enough incoming bandwidth, in other words that the filtering process is costlier than the routing of the publications and subscriptions. Finally, we realistically assume that the stream of publications to any given subscriber does not exceed the bandwidth cap of our system components.

3.2.3 Interconnection Points

Publisher Interconnection Points (PIPs) and *Subscriber Interconnection Points* (SIPs), illustrated in Figures 3.1 and 3.2, are the connection points to the Pub/Sub system for publishers and subscribers, respectively. We assume that the connection between a user (either a publisher or a subscriber) and its corresponding interconnection point is *persistent* (e.g., TCP). One can note that this is necessary for the SIPs as subscribers may be located behind a NAT or firewall, i.e., not reachable directly.
PIPs receive publications from connected publishers, which are then routed to the appropriate operators in the system. Each subscriber is connected to a single SIP to receive matching publications and send its subscriptions and cancellations. Moreover, SIPs are responsible for checking subscribers credentials for receiving publications, if any.

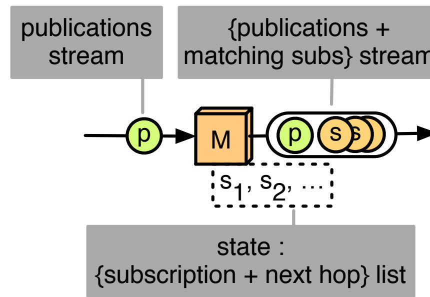


Figure 3.3 – A Matching operator (M).

It should be noted that the PIPs and SIPs could be merged into a single operator type maintaining persistent connections to and from the users. However, for simplicity and scalability purposes when the throughput from the publishers greatly differs from the throughput to the subscribers, we separate PIPs from SIPs.

3.2.4 Matching Operators

Matching operators (M), illustrated in Figure 3.3, receive streams of publications and match them against their local set of subscriptions. M operators output the matching publications and the list of subscribers who should receive them. The performance of an M operator depends on several factors like the size and complexity of events and subscriptions, the total number of subscriptions, and the relationships between subscriptions themselves (e.g., "containment" relations can be used for speeding up the filtering process). Such a complex behavior appears very difficult to analyze, but in practice with large systems we can model the processing speed of the M operators reasonably well as a function of the size of its dataset (set of stored subscriptions).

3.2.5 Dispatchers

Since we want to design a modular and scalable architecture, we want the PIPs, Ms and SIPs to scale independently. Therefore, we need intermediary operators, called *dispatchers* (D), to correctly route the messages between the interconnection points and the matching operators. Dispatchers ensure the transmission of messages independently of the number of operators, the number of transmitted messages, or the complexity of the matching operation. They can use different communication routines, such as *Anycast* and *Multicast*. Dispatchers are described in more details in the next section, and our entire framework, its operators, and its constraints for

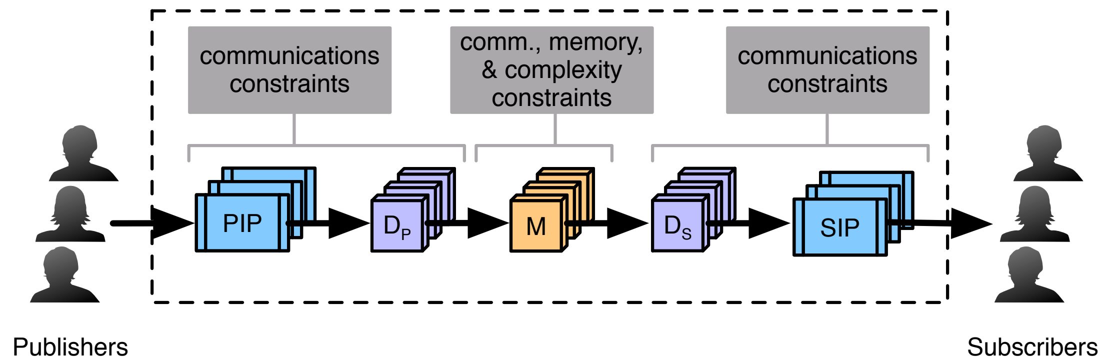


Figure 3.4 – Our generic architecture, its operators, and its constraints when routing publications.

routing the publications from the publishers to the subscribers are summarized in Figure 3.4.

3.3 Scaling the Infrastructure

Provisioning a scalable filtering infrastructure requires taking into account a large number of parameters like the number of publishers and subscribers, the throughput of incoming publications and subscriptions, and the filtering load on the matching operators. Provisioning requires solving the following problems. First, we must determine when to scale up or scale down each part of the infrastructure. Second, for a given workload, we must find the configuration that can sustain the filtering and communication costs but use as few operators (and thus, machines) as possible. In this chapter, we consider the throughput of the publications, the volume of the subscriptions, the memory of the matching operators, and the complexity of the filtering process. As mentioned in Section 3.2, we assume that the throughput of the subscriptions and cancellations is negligible and that subscriptions and cancellations can be routed to the matching operators at a negligible cost.
Our system architecture and its operators are shown in Figure 3.5. In the rest of the section, we describe how to scale each of the system components.

3.3.1 Scaling with Publishers and Subscribers

As explained in Section 3.2.3, the PIP and SIP operators are directly connected to publishers and subscribers, respectively. The number of open persistent connections from the users (publishers and subscribers) is limited to the number of TCP

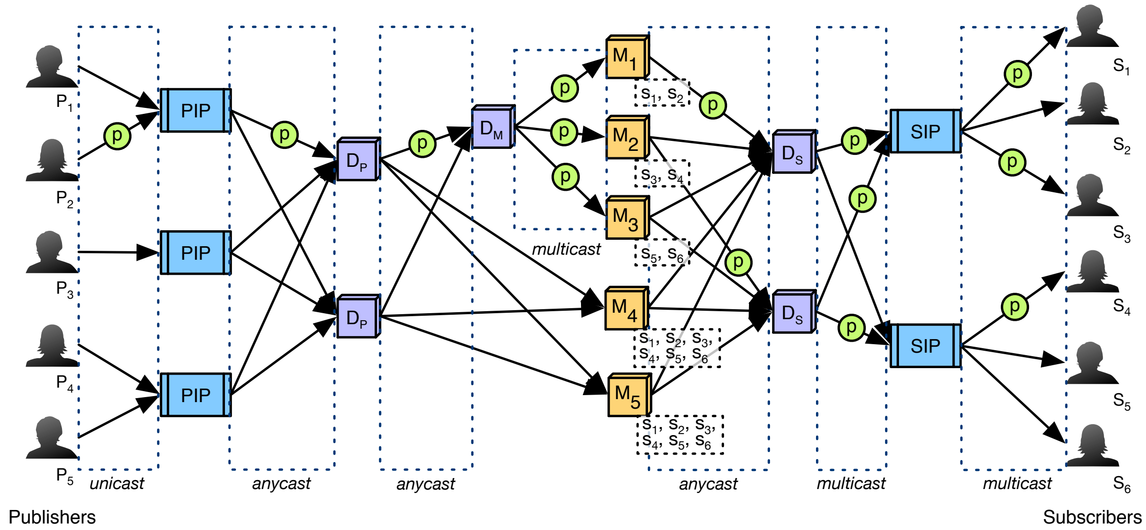


Figure 3.5 – Modular stage-driven Pub/Sub architecture and example path followed by a publication p sent by publisher P_2 and of interest to S_1, S_3, and S_4.

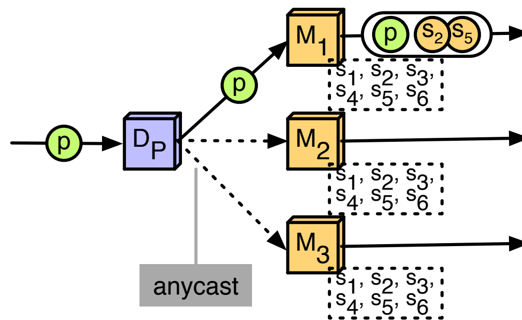


Figure 3.6 – Partitioning publications.

connections, typically a few hundreds on most systems. Thus, new Interconnection Points operators must be created if all the connections are taken, and Interconnection Points operators can be merged as the number of users goes down (assuming that the operators have enough available bandwidth).

3.3.2 Scaling with Publications (Matching Constraints)

In this section, we consider the scaling of the matching operators when they cannot accommodate all the incoming publications. Consider a single matching operator. As the publications throughput increases, it must be cloned if its queue increases and it can no longer filter the publications faster than they arrive. This is achieved by means of *publication partitioning*, as illustrated in Figure 3.6. The original matching operator is cloned, and all copies are connected by a dispatcher D_P. We say clones because

all the new matching operators must be provided with the full set of subscriptions managed by the original machine. The dispatcher splits the incoming stream of publications among the matching operators using anycast operations. Each event is filtered by a single matching operator, resulting in a filtering load reduced by a factor equal to the number of M operators (three in Figure 3.6) if they all have the same speed. It should be noted that subscriptions and cancellations must be *multicast* to all the matching operators in order to keep a consistent and complete view of the subscription set.

Let n_i be the number of publications that can be matched by the matching operator M_i per time unit for $i \in \{1, 2, 3, \ldots\}$, and let T be the publication throughput of all the publishers in the system per time unit. Without loss of generality, we assume that $n_1 \geq n_2 \geq n_3 \geq \ldots$, i.e., that the matching operators are sorted by speed in decreasing order. If $n_i = n_j$ for all $i \neq j$, then the number of matching operators required to split the publication load is at least $\lceil \frac{T}{n_1} \rceil$. If the matching speed of the M operators is heterogeneous, each of them will receive a different share of the incoming publication throughput. To minimize the number of matching operators in such a scenario, we sort them by their speed and select the fastest ones. The number of matching operators required to split the stream of publications is at least the smallest j such that $\sum_{i=1}^{j} n_i \geq T$.

Dynamic scaling: Scaling the number of matching operators as the publications throughput varies is done using a classical threshold mechanism. Since the creation of a new matching operator and the synchronization of its list of subscriptions can be a costly operation, one cannot wait for the saturation of an operator before setting up a new one. Therefore, if the incoming throughput exceeds the upper threshold (e.g., 75% of the achievable throughput), then one of the matching operators is cloned. Likewise, if the incoming throughput decreases below the lower threshold (e.g., 25% of the achievable throughput), then one of the matching operators is discarded.

3.3.3 Scaling with Subscriptions

As the number of subscriptions increases, the time needed to filter one publication increases. Furthermore, storing more subscriptions than can fit in the main memory of a matching operator can lead to a significant drop in performance. Thus, we limit the size of the subscription list kept at each matching engine using *subscription*

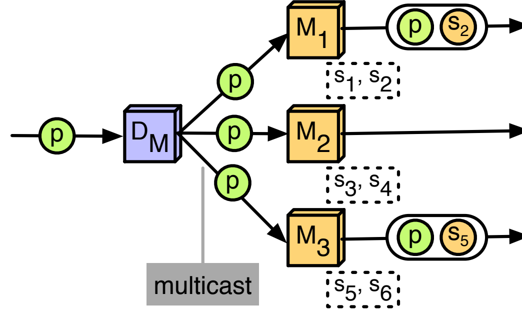


Figure 3.7 – Partitioning subscriptions.

partitioning, as illustrated in Figure 3.7. When a matching operator M reaches this memory limit, it is replaced by a new set of matching operators and the subscription set of the original operator is partitioned and split among the new operators. A dispatcher D_M must be used to multicast all the incoming publications to all the matching operators. Also, we point out that new subscriptions must be *anycast* to a single matching operator, or *unicast* to a specific machine if a clustering algorithm is used. Cancellations must be *broadcast* to all the matching operators, or *unicast* if their assigned operator is known.

Let $|S|$ be the number of subscriptions in the system, and let m_i be the number of subscriptions that can be stored by the matching operator M_i for $i \in \{1, 2, 3, \ldots\}$. Without loss of generality, we assume that $m_1 \geq m_2 \geq m_3 \geq \ldots$. If $m_i = m_j$ for $i \neq j$, then the set of subscriptions must be partitioned on at least $\lceil \frac{|S|}{m_1} \rceil$ machines. When simple linear filtering algorithms are used, partitioning the subscription set into k subsets of equal size will increase the filtering capacity by a factor of k. However, with sub-linear filtering algorithms, filtering a publication over k subscription lists of size α is in general more costly than filtering it over a single list of size $k \cdot \alpha$, thus the filtering capacity is multiplied by a factor smaller than the number of M operators. If the memory size is heterogeneous, then the minimum number of matching operators for the partitioned set of subscriptions is the smallest j such that $\sum_{i=1}^{j} m_i \geq |S|$.

3.3.4 Dispatching Filtered Messages

Once a publication is filtered by a matching operator, it is forwarded, along with its list of matching subscriptions, to one of several dispatchers D_S using an anycast operation. It is possible for the publication to be forwarded from more than one matching operator each choosing a different dispatcher D_S, nonetheless the lists of

matching subscriptions are disjoint. The selected dispatchers D_S then partition the subscription lists using the subscribers assigned to each SIP operator and forward the publication and its smaller lists of matching subscriptions to all the appropriate SIPs. Finally, the SIPs multicast the publication to matching subscribers.

3.3.5 Scaling with Publications (Communication Constraints)

In a Pub/Sub system with a large publication throughput, it is possible to saturate the bandwidth of the dispatchers and interconnection points, in which case the number of operators must be increased. When a dispatcher of type D_P must be added, all the PIPs must be able to anycast publications to it. Furthermore, the new dispatcher must be able to anycast publications to all the dispatchers of type D_M and to all the matching operators containing the complete set of subscriptions. When a dispatcher of type D_S must be added, all the matching operators must be able to anycast the publications and the lists of matching subscriptions to it. The new dispatcher D_S must also be able to forward the publications and the lists of subscriptions to all the SIP operators.

The dispatchers D_S and D_P must scale independently because the incoming and outgoing throughput of the matching operators can be quite different. More precisely, although the outgoing throughput depends on the incoming throughput, it also depends on the characteristics of the subscription set: a few general subscriptions can generate more outgoing bandwidth than a large set of very selective subscriptions that only match a tiny subset of the publications.

3.3.6 Putting Everything Together

Figure 3.5 shows the path followed by a publication p from publisher P_2 matching the set of subscriptions $\{S_1, S_3, S_4\}$. The publication is first routed to the PIP assigned to P_2. It is then forwarded to one of the dispatchers D_P, which then transmits it to D_M. The publication must then be filtered with the entire set of subscriptions, thus D_M multicasts it to the matching operators M_1, M_2, and M_3. The next step is the filtering process: since p matches $\{S_1, S_3, S_4\}$, M_1 anycasts $\{p, \{S_1\}\}$ to one of the dispatchers D_S and M_2 anycasts $\{p, \{S_3, S_4\}\}$. The top dispatcher D_S forwards $\{p, \{S_1\}\}$ to the SIP assigned to S_1. The bottom dispatcher D_S receives $\{p, \{S_3, S_4\}\}$, but since S_3

and S_4 are not assigned the same SIP, the dispatcher must split the subscriptions: it sends $\{p, \{S_3\}\}$ to the top SIP and $\{p, \{S_4\}\}$ to the bottom SIP. Finally, the top SIP conveys p to S_1 and S_3 while the bottom SIP conveys p to S_4.

3.4 A Stage-Driven Architecture

In the previous two sections, we have seen how to provision the interconnection points, matching operators, and dispatchers independently. Both partitioning approaches for scaling the matching operators, i.e., publication and subscription partitioning, have advantages and drawbacks and should therefore be combined to split the load of either subscriptions or publications at different stages of the filtering pipeline as it increases. Conversely, a reduction of the system load can result in parts of the system collapsing into a smaller number of operators. Such a stage-driven architecture, when combined with a monitoring system, can support the dynamic scaling of resources according to the actual load experienced by the filtering infrastructure. This is similar to the elastic computing model associated to cloud computing: the amount of resources is dynamically adapted to the current application needs.
We now explain using a simple example how to scale all the resources optimally while respecting the various constraints of the system. Consider first the simplest Pub/Sub system composed of one PIP, one D_P dispatcher, one matching operator, one D_S dispatcher, and one SIP. [1] Starting from this simple setup, the administrator of a new Pub/Sub system wants to scale it by adding machines (operators) so as to sustain a given number of connected publishers and subscribers. In some cases, this can be linked to QoS guarantees, for instance if subscribers pay for a minimum guaranteed throughput.

3.4.1 Interconnection Points Provisioning and Scalability

The first step is to provision the interconnection points to be able to scale up to the maximum number of expected users. When a new publisher wants to connect to the Pub/Sub system, the monitor searches for a PIP that has not reached its TCP limit nor its bandwidth cap. If one is found, then the new publisher connects to it. Otherwise, a new PIP is started. Note that publishers with very large throughput

[1] The dispatchers are not even necessary at the beginning.

requirements can be connected to more than one PIP. If the incoming publication throughput of a PIP increases to the point of saturation, a new PIP is also created, and the TCP connections of the saturated PIP are split with the new one.
When a publisher leaves the system, the monitor checks the usage of the PIPs (number of TCP connections and bandwidth). If possible, a PIP is removed and its TCP connections moved to other PIPs. As mentioned in the previous section, all the PIPs must be able to anycast the publications to any D_p dispatcher.
The subscription interconnection points are provisioned similarly. When a new subscriber wants to connect to the system, the monitor again looks for a SIP that can support the additional TCP connection and the additional bandwidth, and if none is available a new SIP is created. SIPs can also be created or removed as the incoming throughput coming from the matching operators varies.

3.4.2 Matchers, Dispatchers Provisioning and Scalability

We now describe how to scale the matching operators as the number of publications and subscriptions fluctuates.

Initialization: one matching operator.

At the beginning, only one matching operator M is activated and connected to the two dispatchers D_P and D_S. As the number of subscriptions increases, M stores them into its database and its filtering performance decreases. Furthermore, as the publication throughput increases, so does the filtering load. At some point, M is not longer able to filter the incoming publications and partitioning is required.

First partitioning level: publication partitioning.

When matching operators using sub-linear filtering algorithms are saturated, it is usually better to reduce the number of messages processed per operator than to reduce the number of subscriptions stored on each operator (see Section 3.5 for more details). Thus, we use publication partitioning as much as possible.
Given certain thresholds, when a matching operator cannot filter the publications faster than they arrive and its memory is not full, then publication partitioning is initiated. The use of thresholds is important to allow enough time for the deployment of the new operators. As the speed of the matching operators is not homogeneous, each operator receives a share of the incoming publications proportional to its computing power.

If the flow of incoming publications decreases, the system may become overused. Still given a threshold, the monitor compares the incoming throughput with the potential filtering speed of the matching operators and decides whether or not one of the operators can be removed.

Second partitioning level: subscription partitioning.
Publication partitioning alone cannot be done indefinitely, as even with complex filtering algorithms there exists a point after which the filtering performance stops to be sub-linear (again, see Section 3.5 for more details). Using a memory threshold (e.g., 75%), we thus monitor the memory of the matching operators and initiate subscription partitioning once one of them is saturated
The subscriptions must be split so that all the partitioned matching operators have roughly the same filtering speed. For instance, if a naive linear filtering algorithm is used and all the operators have the same processing power, then the subscriptions can simply be split equally in random fashion. If more complex filtering algorithms exploiting relationships between subscriptions are used and if the set of subscriptions is split randomly, it is possible to obtain a bad split, forcing the next subscription partitioning to be done sooner. To overcome this, a clustering algorithm can be used for splitting subscriptions, and dynamic reconfiguration can be used to move subscriptions from the slowest matching operators to the fastest ones.
As subscriptions are canceled, the monitor checks if the memory of the matching operators is underused. If the remaining subscriptions can be stored on less machines (always given a threshold), then one of the operators is discarded and its subscription set is dispatched to the others.
Our final architecture, again shown in Figure 3.5, can scale with the number of publishers, the number of publications, the number of subscribers, the number of subscriptions, and the number of matched publications.

3.5 Experimental Results

In this section, we study how the minimum number of required matching operators varies under different load scenarios, but first we discuss how the optimal partitioning depends on the filtering algorithm and the subscription set.

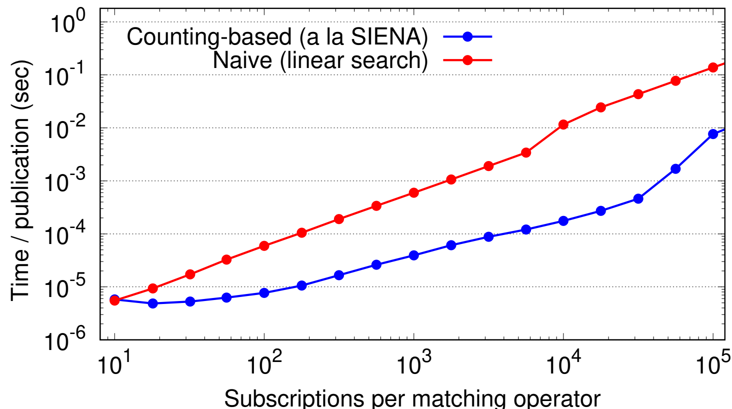


Figure 3.8 – Publication filtering time of the SIENA algorithm as the size of the subscription set varies.

If a naive filtering algorithm testing each publication against all stored subscriptions is used, or more generally if the time required to filter a publication by a matching operator is linearly dependent on the number of stored subscriptions, then any partitioning will result in roughly the same number of required matching operators. However, subscription partitioning requires to duplicate the stream of publications and increases the communication load in the system. It is therefore preferable to store as many subscriptions as possible per matching operator and to partition the publications. Subscription partitioning is only done if absolutely necessary.

Interestingly, this is not always the case when more complex filtering algorithms are used, for instance algorithms exploiting containment relationships between subscriptions. Such algorithms are sublinear with a small number of subscriptions but can become increasingly inefficient as the subscription set increases. This occurs when the matching memory saturates due to the use of paging at the OS level, which forces the operator to swap. Adding the complexity of the filtering algorithm, this results in a significant performance degradation. For such systems, it can be preferable not to exploit the entire memory of the matching operators, even if this increases the number of cloned publications in the system.

For our simulations, we assume that our content-based Pub/Sub system has 1000 publishers, each of them transmitting 10 publications per second. The publication throughput is therefore 10^4 publications per second. We also assume that the operators' bandwidth is sufficient, thus we disregard the number of dispatchers and interconnection points, and only focus on the minimum number of required matching

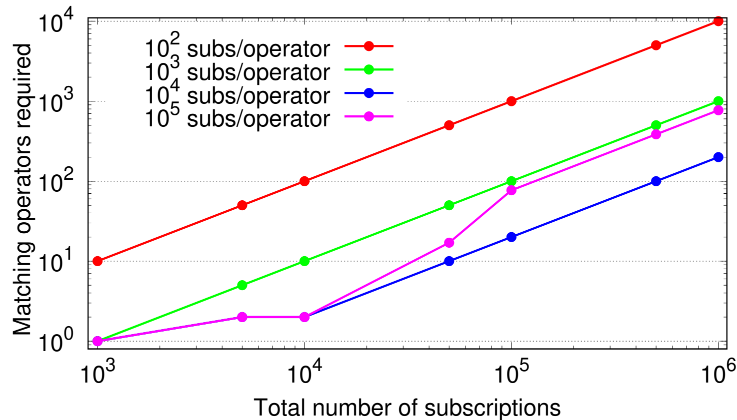


Figure 3.9 – Minimum number of matching operators required as the memory size and the total number of system subscriptions vary (part 1).

operators. We use the SIENA [27] content-based filtering algorithm. We generate subscription sets of various sizes and a collection of events using the workload generator provided by the authors of SIENA, and capture the average matching speed through experiments on our own cluster. The results are shown in Figure 3.8.

For our simulations, we assume that our content-based Pub/Sub system has 1000 publishers, each of them transmitting 10 publications per second. The publication throughput is therefore 10^4 publications per second. We also assume that the operators' bandwidth is sufficient, thus we disregard the number of dispatchers and interconnection points, and only focus on the minimum number of required matching operators. We use the SIENA [27] content-based filtering algorithm. We generate subscription sets of various sizes and a collection of events using the workload generator provided by the authors of SIENA, and capture the average matching speed through experiments on our own cluster. The results are shown in Figure 3.8.

Knowing the speed of the matching operators, Figure 3.9 presents the minimum required number of matching operators as a function of the number of subscriptions in the system. The four curves respectively use matching operators with subscription lists of size 10^2, 10^3, 10^4, and 10^5. Figure 3.10 shows the minimum number of matching operators as a function of the maximum number of stored publications per operator. The four curves use 10^3, 10^4, 10^5, and 10^6 subscriptions in the system. The main result is that for content-based Pub/Sub systems with non-naive filtering algorithms, designers should pay close attention to the number of stored subscription per matching operator. Using too few or too many subscriptions per operator can

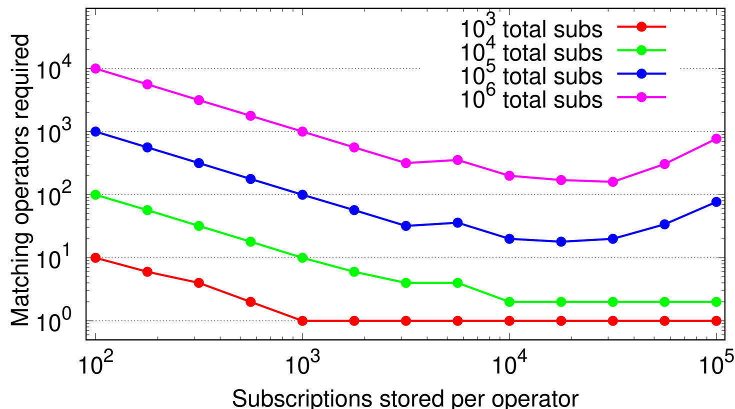


Figure 3.10 – Minimum number of matching operators required as the memory size and the total number of system subscriptions vary (part 2).

significantly increase the number of matching operators required to support the publication throughput.

3.6 Summary

Content-based Pub/Sub architectures are becoming increasingly important for various reasons. Such architectures can easily be deployed on clusters or private clouds within companies, but the main challenge is to scale them with application-specific requirements: throughput, latency, number of publishers and subscribers, etc.
In this chapter, we have proposed a flexible and generic stage-driven architecture for scaling a content-based Pub/Sub system combining dispatchers, interconnection points as well as publication and subscription partitioning. We have analyzed how to provision such an architecture in order to meet application-level constraints while minimizing the number of machines involved. Besides the framework itself, we have shown that when sub-linear filtering algorithms are used on large content-based Pub/Sub systems, developers must estimate the optimal size of the subscriptions lists as accurately as possible in order to minimize the total number of matching engines required. Our main goal for future work is to extend the architecture for the deployment of wide-area Pub/Sub systems over the Internet.

"Study hard what interests you the most in the most undisciplined, irreverent and original manner possible."

Richard P. Feynmann

4

Scalability and Performance

This chapter is based on a paper that was previously published:

StreamHub: a massively parallel architecture for high-performance content-based publish/subscribe. ***Best Paper Award.*** In *DEBS 2013: 7th ACM international conference on Distributed event-based systems*, pages 63–74 Arlington, TX, USA, July 2013. ACM.

4.1 Introduction

Content-based publish/subscribe (pub/sub) [41] is a strong contender for offering an efficient, yet *natural* communication paradigm to developers of large-scale applications. It supports decoupled interactions between the producers (*publishers*) and the consumers (*subscribers*) of information by the means of messages (*publications*). Decoupling occurs both in terms of space and time: publishers and subscribers do not need to know the existence or identity of one another, and no particular synchronization between them is necessary. They only communicate indirectly through a *pub/sub system*. It is the responsibility of this system to *route* publications from the publishers to interested subscribers. Routing is based on *subscriptions* registered by the subscribers to express their interest in specific content. The operation of *matching* the content of the publications against the subscriptions stored in the system is called *content filtering*.

A typical use of pub/sub systems is for composing a collection of independent applications running on different administrative domains or geographical locations. Communication between these applications takes place via a common pub/sub service running on a set of *dedicated servers*, typically set up in a *public cloud* or a *cluster equipped with a public address*, interconnected through a local area network and exposing access points to client applications.

The decoupled and data-centric nature of the pub/sub communication model allows for seamless integration and evolution of large-scale applications. A typical example is *QoS Monitoring as a Service* [98], where an application running on a private cloud is monitored and key performance indicators (KPIs) are generated as publications. These KPIs are propagated to a third-party monitoring service, based on subscriptions generated from a service level agreement (SLA) in order to detect violations of this SLA. Communication takes place via a pub/sub service deployed on a public cloud accessible by both parties. Other applications include e-Health systems [61] that bridge several medical and healthcare institutions sharing information about patients cases, or the canonical example of stock trading [53]. We note that for all these applications, the use of a third-party infrastructure for communication may raise concerns about privacy and data security: publications and subscriptions represent sensitive data that should not be leaked to a third party. As a result, *encrypted content filtering schemes* have gained interest in the recent years [13, 34, 60, 61, 94] as they support filtering of encrypted publications against encrypted subscriptions without needing decryption. Such approaches suffer, however, from a high computational cost and disallow some optimizations, in particular those based on containment relationships between subscriptions (i.e., the fact that a subscription will match a subset of the publications matching another subscription) or on the aggregation of a set of subscriptions into a single one.

Objectives. We argue that the key properties of a pub/sub system running on a public cloud or cluster and supporting large-scale application composition should be as follows.

(1) High throughput and low, predictable delays. The raw performance of the pub/sub service deployed on a public cloud or cluster must be sufficient to support demanding applications, such as high-frequency trading or network monitoring. This requires exploiting parallel processing of incoming subscriptions and publications

as much as possible. Since the filtering operation itself is costly, the design must avoid filtering an incoming publication against a given subscription multiple times, which typically happens in overlay brokers systems. Furthermore, delays between the generation of a publication and its dispatching to interested subscribers must remain of the same order as the delay a coupled communication between the producer and consumer of information would take. As a corollary, there should not be significant deviation in the notification time for all subscribers interested in a given publication.

(2) Scalability. The ability to support increasing numbers of publishers/publications, subscribers/subscriptions, and notifications, as well as more computationally intensive filtering schemes, requires several levels of scalability. *Vertical scalability* is required to take advantage of additional resources available on a given node, notably multi- and many-core architectures that can process the pub/sub traffic in parallel. *Horizontal scalability* allows supporting a higher load by adding more nodes to the cluster. Ideally, a linear increase in the number of nodes should result in a linear increase in maximum supported throughput.

(3) Filtering scheme agnosticism. The design and architecture of distributed pub/sub systems should not be dependent on a particular filtering scheme and in particular on the semantics and representations of publications and subscriptions. Most existing distributed pub/sub systems [27, 30, 33, 63, 118] support filtering schemes based on conjunctive predicates ($<, \leq, =, \dots$) over discrete attribute values (integers, strings, ...), and their designs are closely tied to the nature of this particular representation. This applies, for instance, to the construction and maintenance of routing tables between brokers that drive the flow of publications. To minimize inter-broker traffic, these systems typically rely on the ability to determine containment relationships between subscriptions and/or to construct aggregated subscriptions. Yet, such features are not available with all content-based filtering schemes, notably with encrypted approaches [13, 34, 60, 61, 94]. As a matter of fact, there exist no fundamental reasons why content-based routing should be restricted to attribute- and predicate-based filtering: a pub/sub service should be able to integrate virtually any filtering scheme operating on the content of exchanged data using stored filters, as required by the application. Examples include not only encrypted filtering for privacy preservation, but also statistical methods such as Bayesian filtering [100] or even template matching for digital images (for instance, for face recognition) [23].

The architecture of the pub/sub system should be independent of the nature of the filtering scheme, while still allowing for specific optimizations at the level of a single node.

Contributions. In this chapter, we revisit the design of a distributed content-based pub/sub engine for supporting high throughput, low latency, and horizontal and vertical scalability. We propose a novel approach based on a tiered architecture and inspired by dataflow programming techniques, which exploits parallelism in ways similar to MapReduce [39] and stream processing engines inspired from it [9, 22, 92, 107]. A set of independent operators, each spanning an arbitrary number of servers and taking advantage of multiple cores on individual servers, implement the three fundamental operations of content-based pub/sub: *subscription partitioning*, *publication filtering*, and *publication dispatching*. Interactions with the pub/sub system are managed by a set of independent *data converters and connection points* (DCCP) that maintain persistent connections with clients (publishers and subscribers).

We implement our approach in StreamHub, a pub/sub engine designed for operating on a public cluster or cloud. StreamHub leverages the runtime support of an existing stream processing engine such as S4 [92], Storm [107], or StreamMine [22]. We use the latter engine in our prototype implementation.

Our evaluation on a cluster with up to 384 cores on 48 physical machines indicates that StreamHub is able to sustain high-throughput workloads: up to 150 K subscriptions registered per second; and up to almost 2 K publications filtered per second with a population of 100 K stored subscriptions, resulting in an output flow of nearly 400 K notifications per second to interested subscribers.

We note that our contribution is not on the actual filtering scheme itself, which is supported by an independent library that can be chosen arbitrarily as long as it implements a simple and schema-oblivious API. We demonstrate the performance of StreamHub using the well-established counting algorithm of SIENA [26], and we leave the integration and comparison of other filtering libraries, and in particular those providing privacy-preserving encrypted matching [13, 34, 60, 61, 94], for future work. Similarly, while StreamHub is designed with elastic scalability in mind (i.e., the ability to dynamically adapt the number of servers associated with each operator according to the experienced workload), we leave the implementation of elastic server

Figure 4.1 – Architectural principles.

provisioning for future work and concentrate on the performance and scalability of the architecture with various static configurations.

Outline. The remainder of this chapter is organized as follows. We survey previous work on distributed pub/sub systems in Section 2.1, and we present and motivate our proposed architecture in Section 4.2. We describe the implementation of StreamHub in Section 4.3, as well as the libraries used in our evaluation for filtering publications and clustering subscriptions. We evaluate our approach and compare it to a broker-based pub/sub system in Section 4.4, before concluding in Section 4.5.

4.2 Architecture

Our design choices aim at maximizing pipeline, task, and data parallelism in order to support high-throughput and scalable pub/sub. The resulting architecture shares these design principles with big-data processing systems such as MapReduce [39], and in particular with the online models of computation on data *streams* that were inspired from it [9, 22, 92, 107].

Our architecture uses the base construction principles illustrated by Figure 4.1. It is composed of a set of *operators*, sharing the same code and supporting pipeline parallelism. Operators are organized as a directed acyclic graph (DAG). Commu-

nication takes place in the form of *events* flowing through the DAG of operators. Each operator can scale horizontally by using an arbitrary number of operator *slices*, each running on a different server and managing an independent state. Slices scale vertically by partitioning the received event load between all available cores on each server. There is no communication or shared state between the slices of an operator. Event forwarding between operators can use one the three primitives: *anycast* (sending to a random slice of an operator), *broadcast* (sending to all slices of an operator), or *unicast* (sending to a slice of an operator chosen according to a key).

We use a set of three operators. Each implements a different aspect of the pub/sub service: subscription *partitioning*, publication *filtering*, and publication *dispatching*. Thanks to the scalability properties of operators, one can easily adapt the number of physical machines and cores to the load experienced by each of these three operations. This load varies with the nature of the workload, such as the number, complexity, or selectivity of subscriptions. The load also varies with the nature of the filtering schemes. For instance, encrypted filtering requires more processing power than non-encrypted filtering. To sustain the same publication throughput, one should allocate more slices (servers) to the publication *encrypted filtering* operator.

We present in this section our operators and support mechanisms. We start by describing the endpoints used by external clients to access StreamHub. Afterwards, we present the operators that support content-based filtering, as well as the partition of the load onto different slices at each operator. The filtering operation itself is delegated to one or more *filtering libraries*. StreamHub also provides optional support for *clustering libraries*, which can partition the state of subscriptions in elaborate ways and speed up the filtering operation. As these libraries are pluggable components whose algorithms do not represent novel contributions of this chapter, we describe them in Section 4.3.

4.2.1 Connection To and From Clients

The pub/sub operators are typically deployed on a cluster or a cloud, i.e., a set of machines with limited hardware heterogeneity. In our implementation, StreamHub, operators are implemented using the same language (C++). As a result, the internal communication and serialization formats between the elements forming the architecture can be selected based on performance criteria. Our implementation uses the efficient

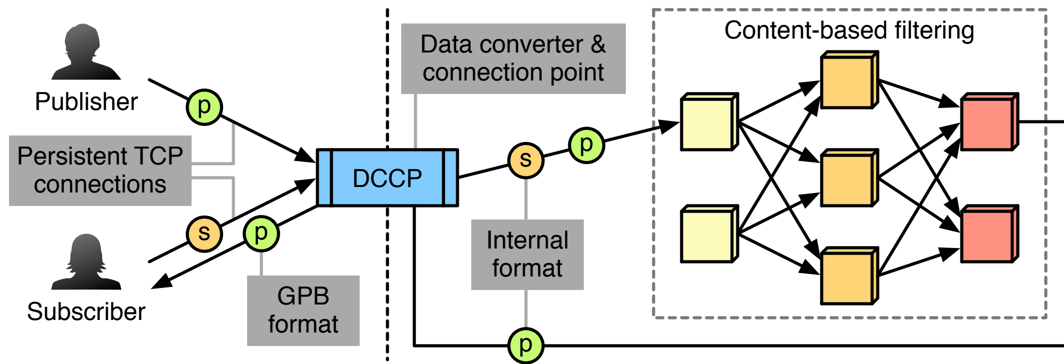


Figure 4.2 – User view of StreamHub.

binary format provided by Boost libraries [1] for internal propagation of events. In contrast, clients may execute on different platforms and use a variety of languages. The choice of the external format is thus driven by its hardware and language independence. Our implementation uses Google Protocol Buffers (GPB), [2] which provide efficient serialization primitives for subscriptions, unsubscriptions, and publications while hiding language and platform heterogeneity.

Publishers and subscribers need a persistent and public connection point to the cluster or cloud supporting the pub/sub service. Connecting to any of the nodes supporting the pub/sub service is impractical in clouds (due to VM migrations) and often impossible in clusters (as most nodes do not have a public IP address). Our design features components external to the operators implementing the pub/sub service, that act as such persistent connection points. These are also in charge of translating between the external and internal representation format, and henceforth named *Data-Converter & Connection-Points* or DCCPs. Figure 4.2 presents a user-centric view of the system. Clients connect to a DCCP via a *persistent* TCP connection to enable low end-to-end delay for communication with the pub/sub service and, more importantly, to support asynchronous notifications of matching publications, as clients may not be directly reachable (for instance, they may be located behind a NAT or firewall). We note that this practical impossibility to reach clients directly limits the applicability of rewiring schemes for solutions based on brokers overlays [31–33, 79].

Several DCCPs can be used for the same StreamHub deployment, e.g., when the number of opened connections or the necessary bandwidth becomes too high for a

[1] https://www.boost.org/

[2] https://developers.google.com/protocol-buffers/

Operator	Role	Description
AP Access Point	Subscription *partitioning*	• Receives subscription events and dispatches each to a single slice of an M operator. Optionally applies subscription clustering using a `libcluster` library. • Receives publications, forwards them to all slices of an M operator.
M Matching	Publication *filtering*	• Receives subscriptions and forwards them to the `libfilter` library. The `libfilter` library stores the subscription and corresponding subscriber identifier in the operator slice state. • Receives publication events and forwards them to the `libfilter` library, which returns a set of matching subscriber identifiers. The M operator slice forwards each publication and list of matching subscriber identifiers to the EP operator by unicast, using the publication identifier as key.
EP Exit Point	Publication *dispatching*	• Receives a publication and list of matching subscriber identifiers. When all lists are received, prepares the notifications, splits the list of matching identifiers, and dispatches them to corresponding DCCPs.

Table 4.1 – Operators supporting scalable CBR.

single machine, when the cost of conversion creates a bottleneck, or on the same host when several network adapters are available.

4.2.2 Content-Based Routing Operators

In this section, we present the three operators that form the *core engine* of our scalable pub/sub architecture. These three operators are organized as a pipeline. They are listed in Table 4.1 and illustrated by Figure 4.3, together with the communication primitives used for propagating events between them. A detailed example of the path taken by subscriptions and publications within the StreamHub engine is shown by Figure 4.4.

Access Point Operator

The *Access Point* (AP) operator plays the role of the input operator. It receives events from any of the DCCPs. The selection of an AP operator slice by a DCCP is done at random to guarantee good load balancing properties. The role of the AP

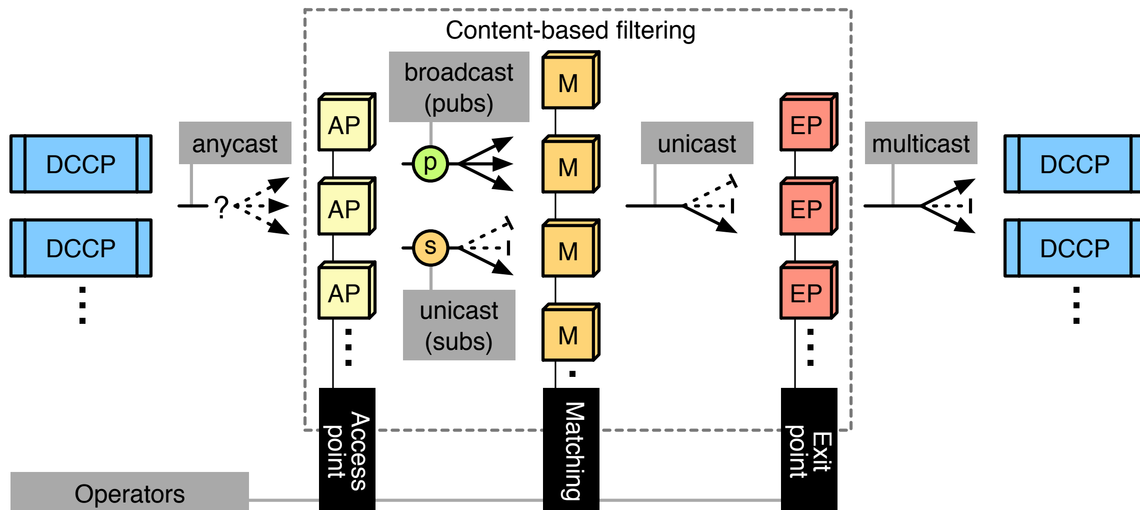


Figure 4.3 – StreamHub processing operators (libraries and states not shown for clarity).

operator is to *partition* incoming subscriptions among all slices of the *Matching* (M) operator as follows.

Each incoming event has a *key*, which is a data structure that indicates the type of the client request (i.e., a new subscription, an unsubscription, or a publication). Subscriptions are not stored by the AP operator slices but are instead forwarded to the M operator that implements the filtering operation as we describe next. Our architecture can simultaneously support different filtering schemes (such as flat vs. structured data, encrypted publications and/or subscriptions, declarative vs. executable filters). Each filtering scheme is supported by a separate M operator. The choice of the destination operator depends on the *filtering scheme identifier* embedded in subscriptions. The same applies to publications.

Only one of the slices of the M operator holds any given subscription. [3] AP slices hence use unicast communication to select the appropriate M operator slice that will be responsible for an incoming subscription. The default mechanism relies on unicast and routes the subscription based on the hashing of the *subscription identifier* specified in event keys. We note that this selection mechanism is *stateless* and *reproducible*: an unsubscription will be routed from the AP to the M operator using unicast and will arrive at the M operator slice that actually holds the subscription.

[3] Resilience of subscriptions in the presence of nodes faults can be handled at the level of the underlying stream processing engine, for instance using active replication or checkpoint/replay techniques.

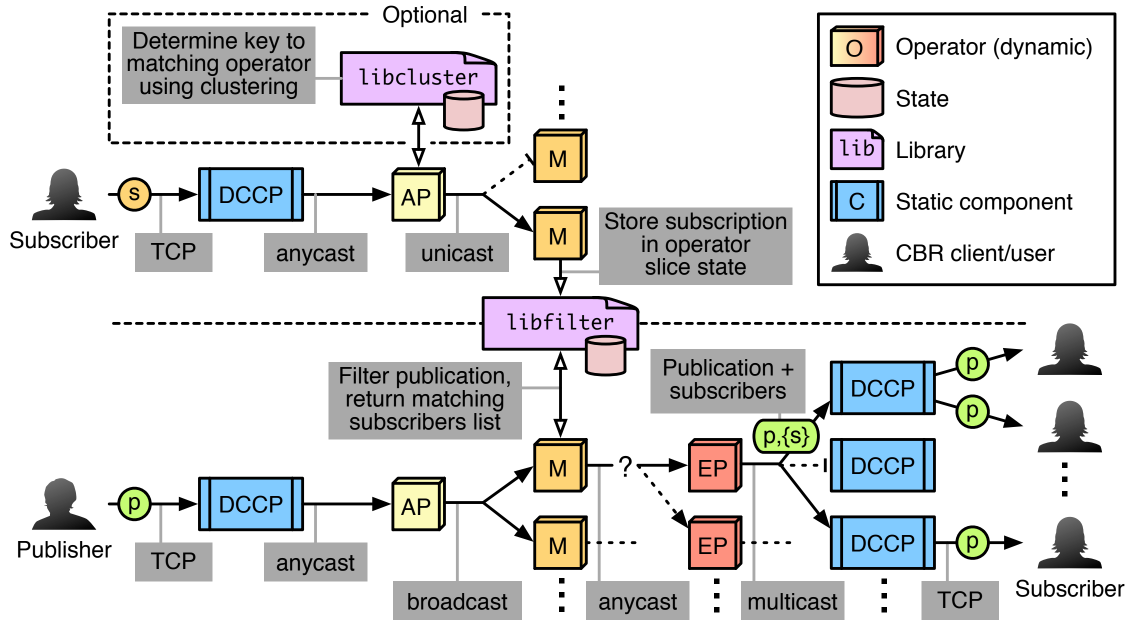


Figure 4.4 – Path taken by subscriptions (top) and publications (bottom) in the StreamHub architecture.

Alternatively to this default mechanism, the user can decide to defer selection to a library, denoted by `libcluster` in Figure 4.4, which supports more complex forms of subscription clustering (see Section 4.3.2). A clustering algorithm can maintain a slice-supported state, which allows for deciding on subscription placement based on the content of that subscription. This incurs additional costs at the AP operator level, in return for a better performance at the M operator level. In case the selection mechanism is not deterministic and reproducible, unsubscriptions need to be broadcast to all slices of the corresponding M operator.

Publications need to be matched against all subscriptions. They are thus broadcast from the AP operator to all slices in the corresponding M operator. Note that our architecture targets deployments in clouds or clusters, which are typically supported by dedicated, high-performance network infrastructures. The broadcast operation of publications in our architecture is designed to take advantage of the availability of IP multicast in such settings for dispatching publications, although our current evaluation does not exploit this feature.

Matching Operator

The *Matching* (M) operator supports publication *filtering.* An M operator is associated with a library, denoted by `libfilter` in Figure 4.4, operating on the independently-maintained state at each of its slices. This library matches incoming publications against registered subscriptions. Different filtering implementations can be used as `libfilter` for different M operators, but they must comply with a simple API supporting two main operations: (1) storing/removing subscriptions based on their identifiers; and (2) processing a publication and returning a list of matching subscriber identifiers. At this stage, the content of the subscriptions and publications themselves is only accessed by the filtering library, making our architecture oblivious to the nature of the matching operation. The default filtering library provided with StreamHub is based on the SIENA counting algorithm [26] and is described in Section 4.3.2. Privacy-preserving filtering can be easily implemented using asymmetric scalar-product preserving encryption [34] or other mechanisms [60, 61, 94]. Recent proposals to reduce the cost of privacy-preserving encrypted filtering through the use of a pre-filtering stage [13] can also trivially be integrated to `libfilter` libraries.
Subscriptions are stored in the state maintained for each slice. This state can be accessed concurrently using read and read-write locks (see Section 4.3 for implementation details). As filtering only requires reading the subscription set, and since most pub/sub workloads are dominated by publications, this allows for vertical scaling of the filtering operation for each slice of the M operator on multiple cores.
An M operator slice calls its `libfilter` for each incoming publication and generates an output event composed by the publication p and a list of matching subscriber identifiers, $s_1, s_2, \ldots, s_n$. When this list is empty, an output event indicating the lack of matching subscription is generated. The event is then sent to the next operator, the *Exit Point* (EP), using unicast. The routing key for selecting the slice of the EP operator is the identifier of the publication p. As a result, each slice of the M operator processing p will send its list of matching identifiers to the same slice of the EP operator. [4]

[4] If publications are of significant size, it is possible to have a single slice of the M operator send p and the others sending only their lists.

Exit Point Operator

The *Exit Point* (EP) operator acts as the output operator of the engine. It shares similarities with the *reduce* phase in the MapReduce terminology [39]. Each incoming publication will be filtered at all slices at the M operator level, but will be processed by a single slice at the EP operator level. An EP operator slice receives the publication and lists of matching subscriber identifiers from all slices of the M operator (or notifications of empty matching lists). Once lists have been received from all M operator slices (or after a timeout to avoid slow M operator slices to delay notifications), the EP operator proceeds with *publication dispatching*: it contacts each DCCP maintaining a connection to at least one interested subscriber and sends it a notification message together with the identifiers of matching subscribers connected to that DCCP. The latter is then in charge of propagating the notification to the actual subscribers.

4.3 Implementation

We implement our architecture on top of a stream processing engine, StreamMine [22]. Other frameworks also present the features and abstractions our implementation requires, such as S4 [92], Storm [107], or Continuous-MapReduce [9]. We present an overview of StreamMine in this section, as well as the `libfilter` and `libcluster` libraries supported by our prototype StreamHub.

4.3.1 StreamMine Stream Processing Engine

StreamMine is a framework that targets scalable processing of information flows in the form of streams of *events*. Its architecture follows the design principles presented at the beginning of Section 4.2 and illustrated in Figure 4.1. StreamMine allows defining operators organized in a DAG. Operators are composed of a number of *slices* that typically reside on separate machines in a processing cluster or a cloud, and have unique identifiers (*id*). Slices of the same operator share the same code, in the form of an *event handler*, a function that is called whenever a new event is received and may emit new events for operators downstream in the DAG (for instance, in Figure 4.1, slices of operator 1 may generate events for operator 2 or 3). The final operator is responsible for propagating the stream of resulting events to the clients of the event-based application.

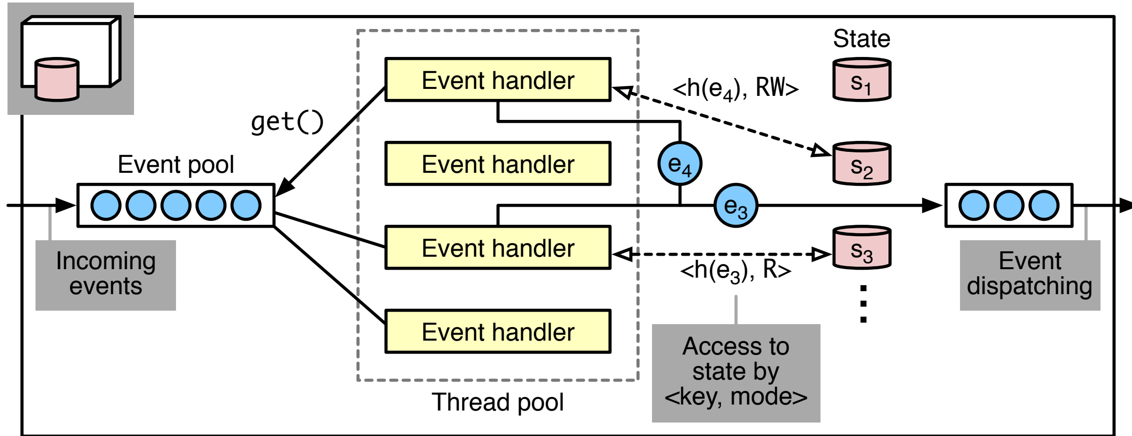


Figure 4.5 – Details of an operator slice from Figure 4.1 supported by 4 threads.

StreamMine provides support for communication among operators and for management of the state that may be maintained by operator slices. Communication between operators is exclusively conducted based on events. Each event is a <key,value> pair. StreamMine supports *unicast*, *anycast*, and *broadcast* communication primitives. The operator to which events are sent is chosen upon their creation. StreamMine communication primitives are oblivious to the content of events, which may only be used inside event handler functions. All communications take place on pre-established and persistent TCP connections: all slices of one operator are persistently connected to all slices of the next operator(s) in the DAG.

The *unicast* primitive determines the *id* of the slice in the next operator by using a key and a hash-function (a simple modulo by default). A specific hash-function can be specified by the event handler function, which may implement more sophisticated stream partitioning mechanisms. The *anycast* primitive sends the event to a random slice of the next operator. The *broadcast* primitive sends the event to all slices of the next operator.

StreamMine operators slices can be either stateless or stateful, and can be composed of several independent processing threads to support vertical scalability when running on a multi-core processor. All threads of all slices of a given operator support the execution of the same event handling function. Figure 4.5 presents a detailed view of the state of the greyed slice from Figure 4.1. Threads from a pool process incoming events from an input queue using a part of the state determined according to the event identifiers. The state of a slice (such as a window of events or summary

information from previously processed events) is managed by StreamMine, and can be partitioned using the same keys that are used for unicast routing between operators. Each thread accesses the state corresponding to the event to process using the appropriate read-only (`R`) or read-write (`RW`) mode. An event with key k_1 can be processed in parallel with another event with key k_2 as long as $k_1 \neq k_2$ or, if $k_1 = k_2$, only when both accesses are read-only.

4.3.2 Filtering and Clustering Libraries

In the following, we present the filtering and clustering libraries (`libfilter` and `libcluster`) that StreamHub currently supports. While these libraries are based on known algorithms and do not represent novel contributions *per se*, they contribute to the overall performance of StreamHub as studied in Section 4.4.

Filtering Libraries

StreamHub can support any filtering scheme if implemented through an appropriate `libfilter` library. We listed variants of filtering schemes in Section 2.1.1. We note that our architecture also supports filtering schemes that need to maintain a state for each of the subscription they store, across the processing of several publications. For instance, a subscriber might wish to receive only the nth publication that matches a given subscription, or be able to send subscriptions on the statistical evolution of publications attributes (e.g., over a window of publications). The corresponding state can be maintained by the `libfilter` in the slice-supported state.

StreamHub currently features an attribute-based filtering scheme library (`libfilter`) that uses a counting algorithm similar to that of SIENA [26]. It organizes predicates for received subscriptions in a forwarding table. An incoming publication will traverse and match in this forwarding table the predicates organized in conjunction sets. Each subscription is associated with a counter that specifies how many of its predicates have been matched so far. When a predicate is satisfied, the counters of all associated subscriptions are increased, and the whole subscription is marked as matched when all predicates of a subscription have been satisfied. Numerical predicates are indexed according to their type ($=$, $<$, $>$) and sorted by values in order to speed up traversals. Therefore, typically only a small part of the

graph is traversed by publications. The algorithm generally scales sublinearly in the number of evaluated conjunction sets.

Clustering Libraries

When using multiple slices in the matching operator, each of these slices holds a subset of all subscriptions and filters publications concurrently with other matchers. By default, we partition subscriptions in a simple and deterministic way using a hash. This splits the load among all M operator slices. However, for many filtering schemes, filtering performance can be improved when subscriptions are partitioned in a content-aware manner. These types of subscription clustering are more costly than hash-based partitioning but they result in gains for the publication filtering performance. This is the case of the attribute-based filtering scheme described previously. As similar subscriptions are stored in the same M operator slice, the filtering algorithm may be able to better factorize common predicates and achieve higher filtering performance (this typically depends on the ability of the filtering operator to support containment determination between subscriptions). For pub/sub systems that process more publications than subscriptions, the relative gain can be significant as we show in our evaluation.

StreamHub supports various clustering algorithms by the means of `libcluster` libraries, that can optionally maintain state about previous subscriptions. When multiple filtering schemes are supported by multiple M operators, each slice of the AP operator supports a different `libcluster` (or default hash-based unicasting) for each such M operator. Deterministic clustering allows unicasting unsubscriptions while non-deterministic clustering require broadcasting unsubscriptions. StreamHub features the two clustering libraries described below.

K-Means [114]: This clustering algorithm performs a partitioning of the subscriptions into K groups and a repetitive re-assignment based on the distance between subscriptions and groups until convergence. The algorithm is stateful and non-deterministic. We implement it in an online manner (*sequential* K-Means) for the dynamic clustering of subscriptions.

Event Space Partitioning (ESP) [112]: The space of subscriptions is represented as a d_s dimensional space, where d_s is the number of attributes. Each M operator slice is responsible for subscriptions that fall within its portion of the space. Subscriptions

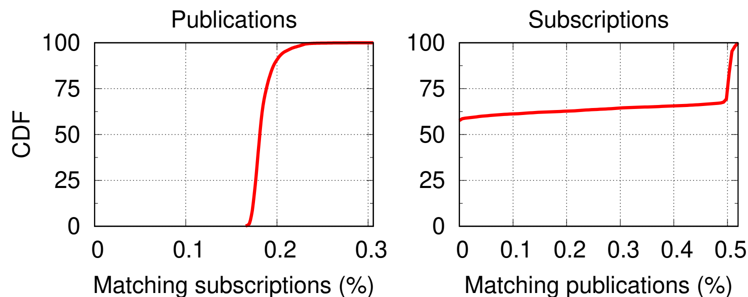


Figure 4.6 – Workload characteristics: cumulative distribution of matching set sizes for publications (left) and matching ratios for subscriptions (right).

that intersect multiple domains are managed by the M operator slice that hosts the first attribute in lexicographic order. As the value d_s cannot be known in advance with content-based routing, it will increase when encountering subscriptions with unknown attributes. This clustering mechanism is stateful but deterministic.

4.4 Evaluation

In this section, we present the experimental validation of StreamHub on a cluster of 48 nodes, each with two quad-core Intel Xeon (E5405) 2 GHz processors and 8 GB of RAM (384 cores total), interconnected with full-duplex 1 Gbps Ethernet. Our implementation uses the C++ language. We configure StreamMine to use batching between operators. Up to 16 KB of events can be stored in output buffers for each operator, and sent in batches or after a time limit. Batching allows increasing maximal supported throughput but has an impact on delays, as we demonstrate at the end of this Section.

We first present the characteristics of the pub/sub workload used for the evaluation, followed by the baseline performance of the counting algorithm (`libfilter`). We then proceed to a operator-by-operator evaluation of the StreamHub architecture, highlighting performance and scalability of each of the operators. We describe the impact of subscription clustering (`libcluster`) and evaluate how the system scales when using an increasing number of nodes. Finally, we present a comparison of our approach with a broker overlay solution running on the same cluster.

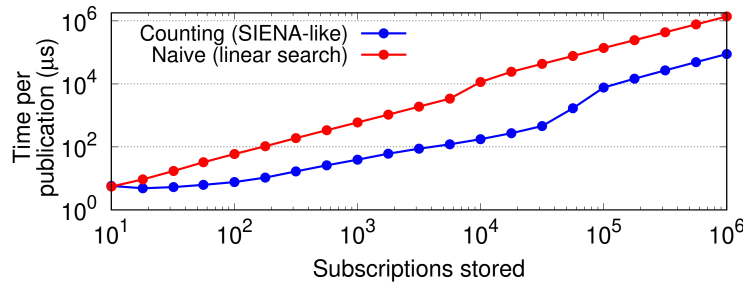


Figure 4.7 – Performance of the counting `libfilter` for filtering incoming publications with respect to the size of the stored subscriptions set.

4.4.1 Experimental Workload

We constructed an experimental workload similar to the one used for the evaluation of Meghdoot [53]. It targets an attribute-based filtering scheme. We gathered five years of quotes for 200 randomly selected stocks from Yahoo! Finance [116]. This corresponds to over 250,000 publications. We built synthetic subscriptions based on the same categories as in [53]. These subscriptions contain a variety of ranges and equality predicates on the attributes of stock quotes, namely the symbol, date, exchanged volume, and daily statistics on their price (open, close, high, low). The characteristics of the workload are detailed in Figure 4.6. They represent a moderately selective type of pub/sub workload: a publication needs to be dispatched to a median of 0.18% of all subscriptions (with 100,000 subscriptions, each publication generates a median of 180 notifications), while a large part of subscriptions do not find publications of interest in the workload but yet need to be processed by the M operator.

4.4.2 Baseline Filtering Performance

We first evaluate the raw performance of the `libfilter` filtering library based on the SIENA-like [26] counting algorithm. We compare it to a naive linear-search filtering mechanism acting as a baseline. Figure 4.7 indicates that the filtering operation cost evolves sublinearly and is at least an order of magnitude better than the naive algorithm above 500 stored subscriptions. Nonetheless, the cost of filtering can grow quite high with large sets of subscriptions, as can be observed on the right side of the graph. This highlights the importance of scaling the processing of incoming publications horizontally and vertically to sustain a high filtering throughput, and to

process an incoming publication against subsets of the overall set of subscriptions to reduce dispatching delays.

4.4.3 Performance of Operators

We now proceed to an operator-by-operator evaluation of StreamHub. Our methodology is to add one operator at a time, replacing the operators downstream the DAG by *sink* operators that receive the events but do not process them further. We denote such sink operators as S(AP), S(M), and S(EP). We focus our evaluation on the scalability aspects of each operator. All experiments are based on 180-seconds runs of StreamHub, during which the system is fed with subscriptions and/or publications as fast as possible to observe the maximal achievable throughput. When observing the performance of the publication filtering operation, subscriptions are registered before starting the measurement.

In our evaluation, the DCCPs are replaced by *generators* that inject the workload into the AP operator. We first verified that these generators can provide the system with a sufficient throughput of publications and subscriptions and do not represent a bottleneck. Our results (not shown) indicate that the generators are able to nearly saturate the input bandwidth capacity of the nodes that host the AP operator slices. This indicates they will not impair the remainder of the evaluation.

We present the complete operator-by-operator evaluation results in Figure 4.8. We look primarily at the throughput in terms of bandwidth and events processed.

AP Operator Scalability

The first column of two plots in Figure 4.8 presents the AP operator scalability. It depicts the maximal input and output throughput of the operator with a publications-only workload. This corresponds to the worst case scenario since publications, unlike subscriptions, need to be broadcast by each AP operator slice to all sink S(M) operator slices. As expected, the input throughput of the AP operator is inversely proportional to the number of sink S(M) operator slices: a copy of each publication needs to be made for every S(M) operator slice and the bottleneck becomes the output bandwidth of AP operators. The planned support for IP multicast between the AP and M operators would boost performance for this operation. We observe nonetheless that this output throughput nearly saturates the cross-bandwidth of the connections between the AP

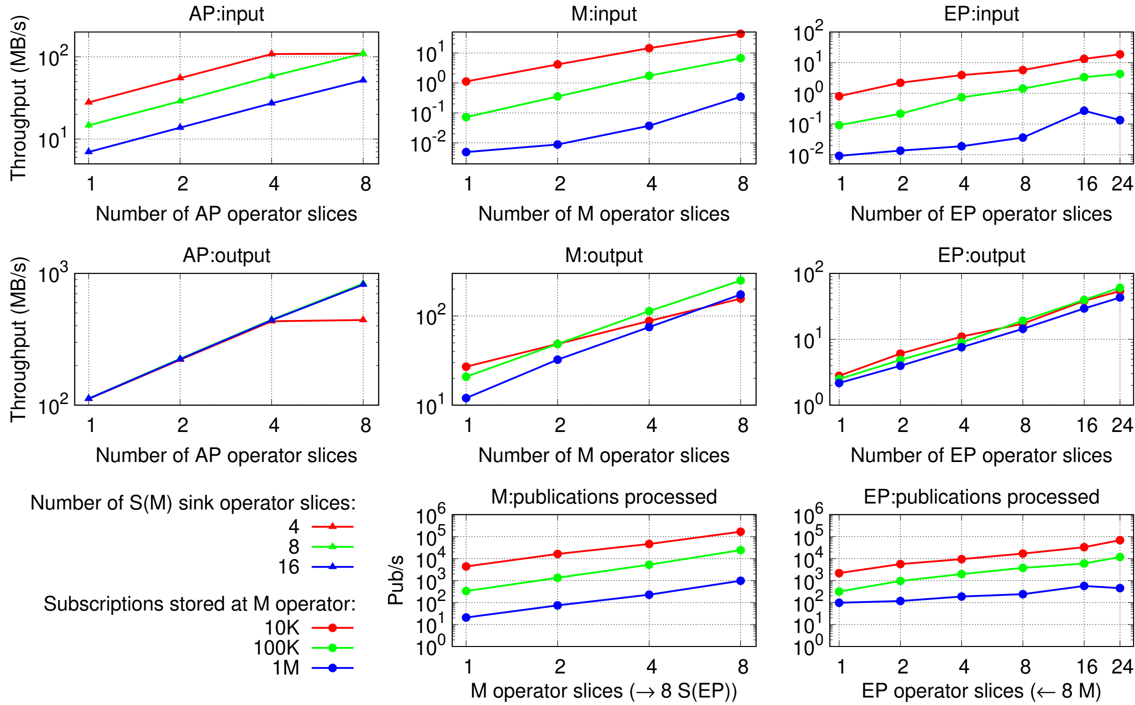


Figure 4.8 – Scaling of StreamHub operators: Input and output throughput for all operators when varying the number of slices and subscriptions. Each operator is evaluated with the downstream operator replaced by a sink. We use one physical machine per slice.

and S(M) operator slices, and is able to saturate the input bandwidth of the S(M) operator slices (serving 104 MB/s of publications to each of them). This indicates that the AP operator will not hinder scalability when the S(M) sink operator slices are replaced by real M operator slices that need to perform the computationally-intensive filtering operation, as confirmed by our next experiment.

M Operator: Subscription Storage Scalability

We start the evaluation of the M operator by assessing the scalability of the subscription storage with a subscriptions-only workload. Figure 4.9 presents the average bandwidth that the generators are able to push through the AP operator for storage at the M operator level. We use a set of 8 AP operator slices so that the AP operator does not form a bottleneck. We use 1 to 8 M operator slices. We observe that the scalability of the subscription storage is almost linear and StreamHub is able to register a flow of 35.4 MB/s with 8 M operator slices, which corresponds to a constant flow of 150,000 subscriptions stored per second. We observe a slight degradation of the throughput

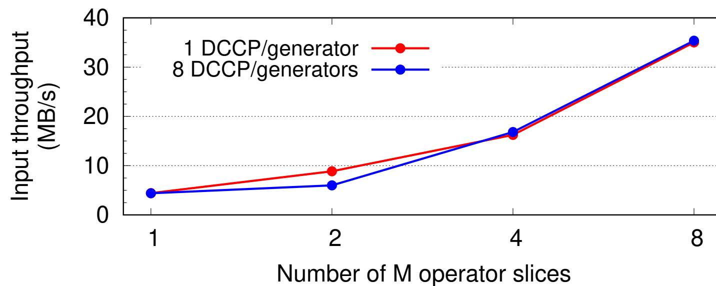


Figure 4.9 – Scaling of the M operator receiving and storing a subscriptions-only workload. The throughput corresponds to the traffic between the DCCP/generators and the AP operator, with 8 AP operator slices.

when using 8 generators and only 2 M operator slices. The reason was tracked down to overflows in input buffers of the M operator slices, leading to retransmissions of messages from the AP operator and some loss of bandwidth.

M Operator: Publication Matching Scalability

We now investigate the scalability of the filtering operation at the M operator level, i.e., matching each publication against the set of stored subscriptions. We use a set of 8 sink S(EP) operator slices as the downstream operator and 8 AP operator slices for the upstream operator. The second column of three plots in Figure 4.8 presents the achieved input/output throughput and the number of publications that the M operator filters per second, including transmission to the downstream S(EP) operator. We clearly observe that the architecture scales: the addition of new operator slices to the M operator results in linear increase of its processing capacity. Note that, as expected from the workload characteristics (median matching set of 0.18% of stored subscriptions), the bandwidth requirements are higher for output than for input because the publications are augmented with a potentially large list of matching identifiers.

EP Operator Scalability

We complete the operator-by-operator scalability evaluation by replacing the S(EP) sink operator slices with their real counterparts that perform publication dispatching. We use 8 generators, 8 AP, and 8 M operators slices. We observe in the third column of three plots of Figure 4.8 the input/output throughput of the EP operator and the number of publications that are effectively dispatched. With a configuration of 8

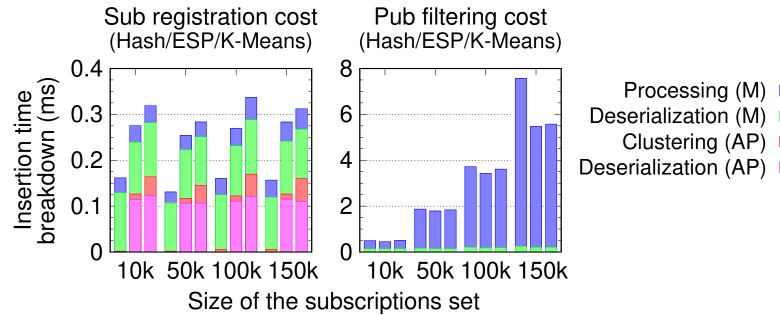


Figure 4.10 – Overhead of clustering for storing subscriptions (left) and impact on filtering efficiency (right). Each group of stacked bars shows the breakdowns of average costs for one subscription or one publication matched against the corresponding subscription set. Each group has three bars for hash-based (no clustering), ESP, and K-Means.

EP operator slices, StreamHub is already able to filter and dispatch from 3.8 K to 17 K publications per second, for respectively 100 K and 10 K stored subscriptions (corresponding to 684 K and 306 K notifications sent out to subscribers per second, respectively).

Discussion

The results of the operator-by-operator evaluation clearly show that the subscription registration and publication filtering operations scale by adding more slices (and thus physical machines) to the operator that supports them. Adequately provisioning the architecture allows handling an arbitrary number of publications and subscriptions. One should point out that the specific workload considered in our evaluation is costlier for the operators that deal with publication dispatching (EP) and matching (M) than for handling and forwarding incoming publications (AP).

4.4.4 Impact of Clustering

We now investigate the impact of using a `libcluster` library at the AP operator level for clustering subscriptions. Our objective is to evaluate if the additional cost for registering a subscription in the system using content-aware clustering is compensated by the subsequent performance gain when filtering publications against stored subscriptions. We present in Figure 4.10 the time required to store a subscription at the M operator level (left) and process an incoming publication against the set of stored subscriptions (right). Times are obtained by averaging over 10,000 events.

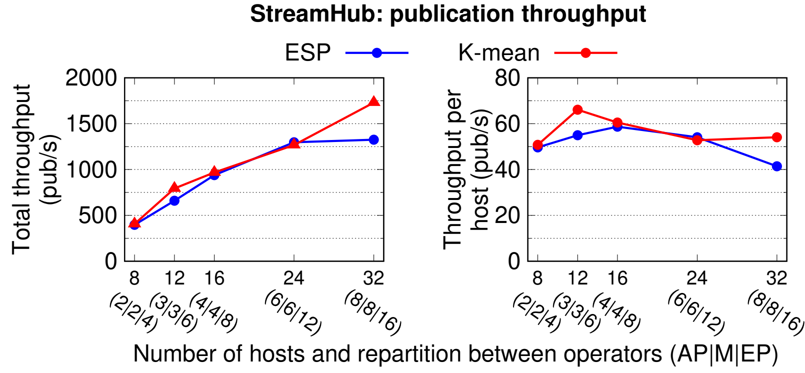


Figure 4.11 – Throughput of StreamHub with 100,000 subscriptions, using `libcluster` and the workload-optimal configurations for each number of available machines. The number of slices at each operator is indicated within parentheses.

Configuration	(AP **4**|M **4**|EP **8**)		(AP **8**|M **8**|EP **16**)	
Batching	16 K	None	16 K	None
Publications/s	500	200	1,000	500
Average delay	1.06 s	0.36 s	0.98 s	0.22 s
Std. dev.	0.28 s	0.17 s	0.3 s	0.15 s

Table 4.2 – End-to-end delays (settings as Figure 4.11).

The breakdown distinguishes between the different operations at the AP and M operators: deserializing the event at the AP operator level (for subscriptions when using clustering) and processing it (clustering, storing, or matching) at both the AP and M operators levels.

We observe that the cost of subscription insertion increases due to the additional deserialization and treatment at the AP operator level. On the other hand, the use of clustering yields significant performance gains when matching publications against a large set of subscriptions: for 150 K subscriptions matching is 25 to 27% faster when using K-Means or ESP. This supports our claim that using a `libcluster` library, when applicable to the filtering scheme being used, may significantly increase the filtering performance or reduce the number of M operator slices required to sustain a given publication filtering throughput requirement. At the same time, the use of a `libcluster` library does not break the separation of concerns and filtering schema agnosticism that underpins the complete architecture.

4.4.5 Overall Performance

Our last experiment with StreamHub relates to the overall provisioning and scaling of the complete architecture. We consider the case where a cluster or a cloud virtual environment needs to be scaled up to increase the throughput of the pub/sub process, with the objective of offering a linearly increasing performance as more physical nodes are added to support the service, and to sustain low and predictable end-to-end delays. As previously demonstrated, the optimal assignment of slices (and thus, physical machines) to operators depends on the nature of the workload. For the purpose of this evaluation, we determined the best configuration for a given budget of machines based on the operator-by-operator experiments. Our architecture is designed to easily support dynamic scaling by migrating slices between physical machines. We leave the integration of such mechanisms and the appropriate decision-making systems to future work. Figure 4.11 presents the evolution of the publication throughput with clusters of 8 to 32 machines (our other machines are used for 8 generators and 16 sink DCCPs). We observe that the scalability objectives of StreamHub are met: there is a linear gain in performance between 8 to 32 machines. The maximal supported throughput between the smaller and the larger configuration when using the ESP partitioning is actually 4.26x higher, which is mostly due to the reduced contention on the AP operator slices. We note that the impact of clustering is consistent with what we observed in Figure 4.10: throughput is from 10% to 28% better with clustering (see Figure 4.8). Table 4.2 presents the average end-to-end delays (between the reception of a publication and the reception of the corresponding notifications by the subscribers) observed by clients, and their variations. In this case, we selected the throughput to be around half of the maximal supported throughput in the 16 and 32 nodes configurations. We consider two settings, with batching and without. Batching increases throughput but also introduces extra delays. Disabling it divides the maximal supported throughput by approximately a factor of 2. In both cases though, the delays are low (around a second with batching, or a fraction of a second without it) and predictable as they show only slight variations between publications.

4.4.6 Comparison with PADRES

For completeness, we have also performed experiments with the same set of subscriptions and publications using the most recent version of PADRES [63] (v2.0).

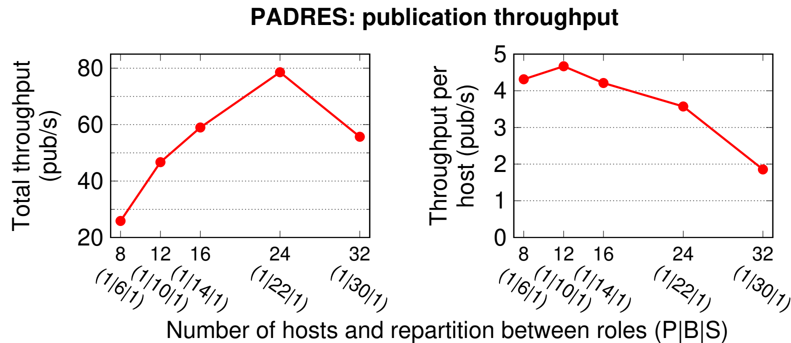


Figure 4.12 – Throughput of PADRES with 100,000 subscriptions. The number of hosts is indicated within parentheses: a single publisher (P) and a single subscriber (S) were sufficient to fully load the brokers (B).

As detailed in our introduction, PADRES is based on different design choices than StreamHub: it establishes and maintains a network of brokers that collectively implement pub/sub functionality and is specific to a particular filtering scheme, while our design dedicates different machines to each operation and is independent from the actual filtering scheme that is used.

Our PADRES setup consists of one publisher, one subscriber, and a varying number of brokers. We verified that neither the publisher nor the subscriber represent a bottleneck in the experiments. The publisher and the subscriber are connected to every broker. Subscriptions and publications are randomly partitioned among brokers. Every machine executes 4 broker instances, which corresponds to half of its available cores. Using all cores on each machines yielded lower performance figures, probably due to contention on resources. Results are averaged over 100 publications, measured after an initial warm-up phase of 2,000 messages to enable JIT optimizations.

Figure 4.12 shows the throughput achieved with the same 100,000 subscriptions as for Figure 4.11 and various numbers of brokers. We observe that PADRES scales well for up to 88 brokers (i.e., 22 machines, each running 4 broker processes) but seems to suffer when adding more brokers. Actually, each publication has to be filtered by multiple brokers for propagation to other brokers due to the use of routing tables that are constructed according to the filtering schema that is used. This adds to the overall load and reduces the contribution of each broker to overall throughput (Figure 4.12).

We finally observe that the maximal raw throughput achieved in our cluster is two orders of magnitude higher with our architecture than with PADRES. While a part of this difference can be accounted to language and implementation differences, the

higher parallelism and independence of operations in our architecture clearly helps improving the filtering throughputs.

4.5 Summary

We presented a novel design for high-throughput pub/sub services. We focused on the support of large-scale applications communicating through a managed environment providing the pub/sub service, such as a publicly available cluster or a public cloud deployment. Our architecture is highly parallel and scalable, and can readily support arbitrary complex filtering schemes, including encrypted or state-based filtering. We do so by departing from previous approaches based on broker overlays and by decoupling the architecture and communication flows of the pub/sub system from the filtering scheme(s) and the subscriptions workload. Our implementation, StreamHub, splits the pub/sub service into fundamental operations, allocated to horizontally and vertically scalable operators supported by a scalable stream processing engine. The evaluation of StreamHub on a cluster with up to 384 cores indicates that it can sustain high throughputs of subscription registrations and publication filtering: we filter thousands of publications against hundreds of thousands of registered subscriptions, resulting in hundreds of thousands notifications sent to clients every second.

This work opens several research perspectives that will be part of our future work on high-throughput pub/sub services for large-scale application compositions. The use of privacy-preserving encrypted filtering is a growing requirement for applications running on multiple private clouds and supported by untrusted public cloud infrastructures providing the pub/sub service. We plan on integrating such encrypted filtering approaches [34, 60, 61, 94]. We recently proposed techniques for lowering their computational cost [13], that we can easily integrate as `libfilter` libraries. Publications and subscriptions will need to be encrypted and decrypted as they enter and leave untrusted public clouds. This can be supported in a parallel and scalable manner at the AP and EP operator levels, and supported by pub/sub-specific key management techniques.

We also plan to support *elastic scaling* of the StreamHub architecture. This requires observing the workload experienced by each of the three operators and adapting the

number of physical machines for each operator according to the requirements of the corresponding pub/sub operation. To that end, we intend to extend StreamHub with a resource provisioner for stream processing.

"We can only see a short distance ahead, but we can see plenty there that needs to be done."

Alan Turing, *Computing machinery and intelligence*

5 Elasticity

This chapter is based on a paper that was previously published:

Elastic Scaling of a High-Throughput Content-Based Publish/Subscribe Engine. In *ICDCS 2014: 33th Internation Conference on Distributed Computing Systems*, Madrid, Spain, July 2014, IEEE.

5.1 Introduction

Cloud computing provides processing infrastructures, platforms, or software solutions *as a service*. One of the key promises of cloud computing is its support for *elasticity* [8], i.e., the ability to dynamically adapt the amount of resources allocated for supporting a service. Thanks to elasticity, cloud users no longer need to statically provision their infrastructure. Instead, resources can be added (scale out) or removed (scale in) on-demand in response to variations in the experienced load [88]. This allows a service to accommodate unpredictable growth or decay in popularity.

To support elasticity, an application or service deployed on a cloud must provide three key features. First, it should be intrinsically *scalable*: the addition or removal of resources must result in an increase, respectively decrease, of the processing capacity of the service. Second, it should support the *dynamic allocation of resources*. This allows modifying at runtime the processing capacity of a service without halting or restarting that service. Finally, it should provide *decision support*, in order to drive

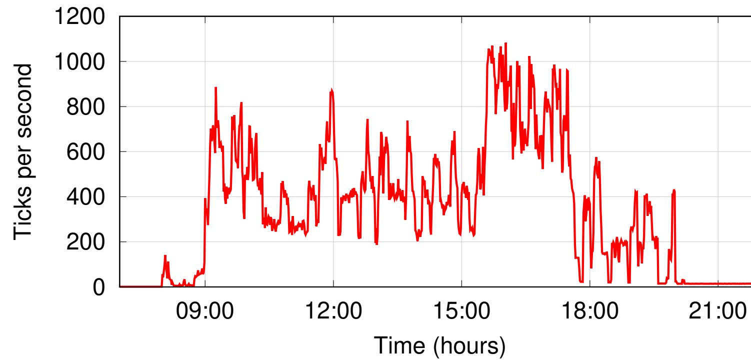


Figure 5.1 – Typical volume of ticks (Frankfurt Stock Exchange, 2011/11/18).

the dynamic allocation based on the experienced workload and according to *elasticity rules*. These rules trigger the allocation of new resources and the deallocation and reallocation of existing resources based on the observed load.

Publish/subscribe (pub/sub) [41] is a decoupled communication paradigm that is particularly well suited for cloud-based deployments [5]. In pub/sub systems, information producers, or *publishers*, send data to the pub/sub service in the form of *publications*. The pub/sub service is then in charge of dispatching these publications to all the parties, called *subscribers*, that previously expressed interest in them by the means of *subscriptions*. In the *topic-based* model [5, 7], publications and subscriptions are associated to pre-defined topics. *Content-based* pub/sub supports subscriptions that express interest as filters on the content of publications, typically by means of predicates over the attributes associated to each publication [4, 27, 29]. We consider the more general content-based model in this chapter.

The load experienced by a pub/sub system running as a service is by nature difficult to predict. The amount of subscribers and the volume of publications can vary significantly over time. A typical example is stock market monitoring [14]: stock exchanges (publishers) send real-time stock ticks or quotes, while clients (subscribers) register their interests in ticks and/or quotes related to their investment strategies, e.g., when the price for a given stock goes over or under a specific threshold. In such a setting, the volume of publications depends on the activity of the stock exchanges, which have specific periods of activity every work day and are closed on week-ends. Figure 5.1 shows a typical tick load recorded on the 18th of November 2011 at the Frankfurt Stock Exchange. The volume sharply rises when the trading opens at 9:00

and rapidly declines after market closes at 17:30. Static provisioning of cloud resources for a pub/sub system supporting the peak load of this application would be cost-ineffective. Conversely, the ability to elastically scale the amount of resources allows a better utilization of the infrastructure, translating into better return-on-investment for the service provider.

Contributions. In this chapter we present the design, implementation, and evaluation of an elastic pub/sub middleware service supporting high-throughput and low-delay content-based filtering. Our system, named e-StreamHub, extends a scalable but static pub/sub engine, StreamHub [14], with comprehensive elasticity support. It supports arbitrary types of filtering operations, including encrypted filtering for privacy preservation in public cloud settings. It leverages a stream processing engine, StreamMine3G [22], and relies on a set of stream processing *operators*, each implementing a specific aspect of the pub/sub service, allowing the system to scale *horizontally* and *vertically* with the number of allocated nodes.

In this work, we make the following contributions: (1) support for slicing of the state and computations of a high-performance and massively parallel pub/sub engine; (2) migration mechanisms for dynamically allocating resources to each operator of a pub/sub engine with low impact on delays and throughput; (3) decision support for elasticity and resource allocation based on system observation and enforcement of global and local elasticity policies; and (4) implementation of these mechanisms in e-StreamHub and their evaluation on a private cloud of 240 cores, using real load evolution traces from the Frankfurt Stock Exchange. Our evaluation allows adapting the number of machines used to the actual workload experienced by the pub/sub service, while maintaining continuous operation and with minimal impact on notification delays.

5.2 The StreamHub Scalable Pub/Sub Engine

We present in this section the high-level architecture of the pub/sub engine that we extend for elasticity support, StreamHub (see Chapter 4). Its components are shown in Figure 5.2. The design of StreamHub targets high throughput filtering and notification pub/sub. It also aims for low and stable latencies. For any given

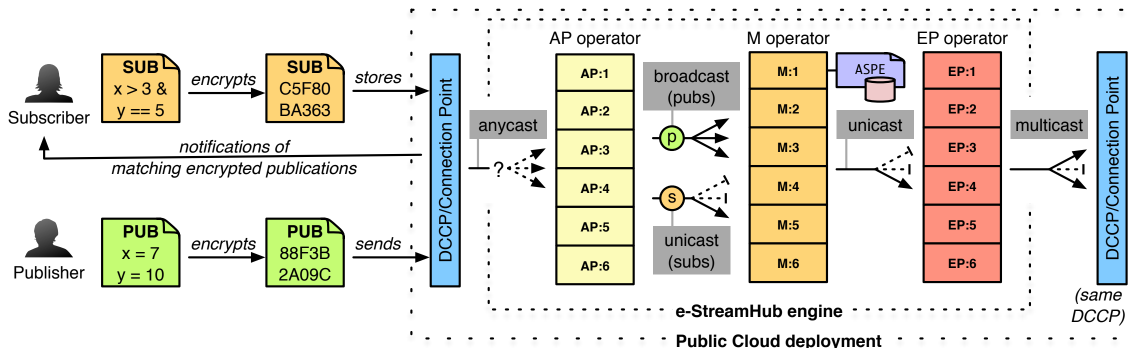


Figure 5.2 – Example of a StreamHub engine deployed on a public cloud. Each operator is supported by 6 slices. Events flow from left to right.

publication, the delay for notifying users shall not vary more than a small fraction of its average value.

The matching of incoming publications against stored subscriptions is performed in StreamHub by means of external filtering libraries. Deploying a content-based pub/sub system in a shared (public) cloud infrastructure poses security challenges. The values of the publications' attributes and the subscriptions' predicates may reveal confidential information that must be protected from potential attackers [106]. Encrypted filtering techniques [13, 34] allow matching encrypted publications against encrypted subscriptions, without revealing their original content. In the context of this chapter and in our evaluation, we focus on computationally intensive encrypted filtering using the ASPE algorithm [34]. Due to the independence of the architecture from the filtering scheme, our results would apply to any other scheme.

Most of the previous distributed content-based pub/sub approaches were based on an overlay of brokers, where each broker would implement all operations of the pub/sub service [27], and brokers would have to maintain routing tables between them whose efficiency and construction cost directly depend on the subscription model and workload. StreamHub departs from the use of broker overlays and adopts a tiered architecture with dedicated components for each operation. This yields better performance and allows the system to scale independently of the filtering scheme(s) being used. StreamHub splits the pub/sub service into three fundamental operations. Each of these operations is mapped to an *operator* spanning over an arbitrary number of processing entities, called *operator slices*.

The first operation is *subscription partitioning*. It is supported by an *Access Point* (AP) operator and splits the workload of subscriptions in non-overlapping sets, one for each slice of the next operator, using modulo hashing on subscription identifiers. This introduces data parallelism and allows processing of several incoming publications in parallel against all subscriptions. Incoming publications are broadcast to reach all slices of the next operator.

The second operation is *publication filtering*. Each slice of the corresponding *Matching* (M) operator is associated with an instance of the filtering library that stores incoming subscriptions in the state associated with the slice. Upon receiving a publication, the library returns the list of subscribers that have registered a matching subscription. The publication and this list are forwarded to the next operator. Note that there might be several M operators, one per filtering scheme supported by the platform (e.g., encrypted and non-encrypted).

Finally, the third operation is *publication dispatching*. It is supported by an *Exit Point* (EP) operator whose goal is to collect and combine the lists of matching subscribers sent by each of the M operator slices, for each publication. These lists are dispatched to EPs using modulo hashing on publication identifiers. As a result, the lists for a particular publication will all reach the same EP operator slice. Once all lists are collected by that slice, a notification message is prepared and sent to interested subscribers through the connection point(s).

StreamHub leverages the runtime support of a stream-based processing engine, StreamMine3G [22]. *Events* flow through a directed acyclic graph (DAG) of operators. All slices of an operator use the same processing code for handling incoming events. Synchronized access to operator state can take place *within a given slice* by using read (R) and read/write (R/W) locks. A slice has no access to the state of other slices, even within the same operator. Each physical node (*host*) supports vertical scalability for locally-deployed operator slices by using a thread pool whose size is adapted to the number of available cores. Each slice can be supported by multiple cores operating in parallel when the processing is stateless or requires only an R lock. StreamMine3G supports dependability through passive [85] or active [84] slice replication. This requires no modification to the application but the evaluation of these mechanisms is out of the scope of the present chapter.

The original StreamHub design assumes a static configuration: each operator slice is allocated to a single host and the number of slices for each operator must be determined manually based on the expected load for each of the pub/sub operations. Such manual provisioning is a challenging task as the workload is not necessarily known in advance. Even if the expected throughput could be estimated in certain cases, the intrinsic characteristics of its impact on the load each operator will have to support are hard to determine. Over-provisioning the operators is not desirable as the number of stored subscriptions and the rate of incoming publications will vary over time, thus leading to poor cost-effectiveness.

While scalability is a primary and necessary aspect of such a middleware solution, it is not sufficient *per se*. One also needs the ability to dynamically adapt the number of hosts allocated to each of the operators, based on the observed load on their slices. We present in the remaining of this chapter the mechanisms (Section 5.3) and policies (Section 5.4) required to provide elasticity within the pub/sub engine. These solutions provide minimal service interruption: they maintain a high filtering throughput and low delays even under dynamic reconfigurations.

5.3 Elasticity Mechanisms

Elasticity in e-StreamHub is supported by migrating operator slices across a varying number of hosts. The total number of active slices used by a single operator across all these hosts is fixed: we thus perform static partitioning of the operator state. A static partitioning is preferred as it does not require the application to be aware of scaling operations, avoiding specific up-calls for state management. Furthermore, using static partitioning relieves from using a restrictive storage model to allow state splitting and coalescing [46].

In the remainder of this section we describe the mechanisms for operator slice migration in StreamMine3G and the e-StreamHub manager component that orchestrates elasticity and system configuration.

5.3.1 Slice Migration with Low Impact on Delays

Migration of slices is supported at the level of StreamMine3G. The requirement is that the interruption of service must be as short as possible. We minimize the delay

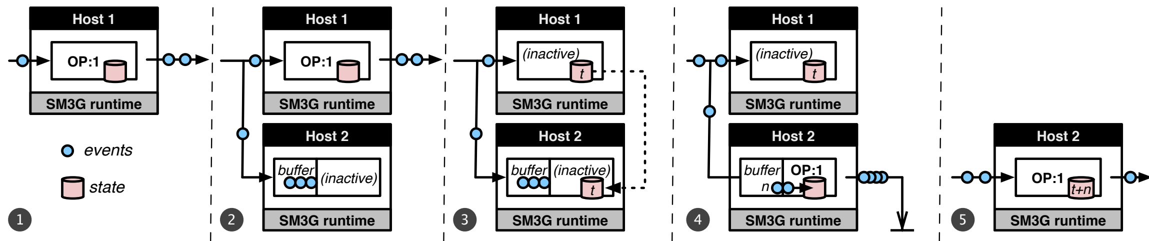


Figure 5.3 – The five main steps of a slice migration between two hosts.

introduced in the processing of events during migrations by using slice duplication and in-memory logging/buffering of events.

The migration mechanism is illustrated by Figure 5.3. The slice to migrate initially runs on a host 1 (Figure 5.3-❶). The e-StreamHub manager (described later in this section) orchestrates the migration of that slice to host 2 when requested by the e-StreamHub elasticity enforcer (described in the next section). The StreamMine3G runtime creates a new slice on host 2, which is initially inactive. The operator DAG is rewired in order to duplicate all incoming events for that slice (❷). Events still reach the original slice on host 1 where they are processed normally, but they also reach the new slice on host 2 where they are queued for later processing. There is one queue per originating slice of the previous operator. The copy of the state takes place when all queues contain events with sequence numbers that are lower than or equal to those of events already processed on host 1 for the same source. Before copying the state, processing is stopped on host 1 (❸). The state is associated with a timestamp vector. Processing resumes on host 2 using events following the migrated state's timestamp vector, filtering obsolete events and preventing duplicate processing (❹), and the original slice on host 1 is removed (❻).

The slice migration time, and hence the duration of the service interruption, depends on the state size. AP operator slices are stateless and there is no copy phase, therefore limited latency is expected during a slice migration. M operator slices have a persistent state consisting of the stored subscriptions. The migration delay is expected to mostly depend on this state size. EP operator slices also maintain a state, but it is transient and expected to be small: it consists of the lists of matching subscriber identifiers for publications being processed. Therefore, migrating EP operator slices is expected to have a small impact on the notification delay.

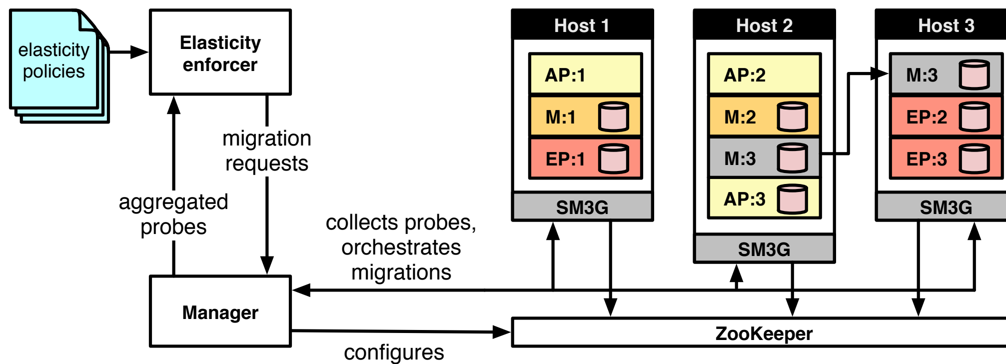


Figure 5.4 – Interaction of the e-StreamHub components. Each host runs an instance of the StreamMine3G runtime (SM3G) and several operator slices. The manager collects probes about the utilization of the hosts for each individual slice and maintains the configuration in ZooKeeper. Probes are dispatched to the elasticity enforcer, which takes as input a set of elasticity policies and issue migration requests to the manager.

5.3.2 The e-StreamHub Manager

The interaction of the components of the system is illustrated in Figure 5.4. The manager is in charge of the system configuration. It orchestrates the migrations according to the flow of operations described above, and updates the configuration accordingly. Migrations are not initiated by the manager itself but requested by the elasticity enforcer component, described in the next section. The manager collects probes from all participating hosts via heartbeats. Probes indicate for each slice the CPU utilization, memory utilization, and network usage. They are then aggregated on a per-slice and per-host basis, and sent to the elasticity enforcer in order to trigger changes to the slice placement according to the input elasticity policies.

The operation of e-StreamHub requires that all hosts supporting the system share a common configuration. At the application level, the static configuration allows, for instance, EP operator slices to know the amount of M operator slices from which they must await matching subscriber lists. At the runtime level, the configuration is dynamic and includes the location of the slices on the hosts, which is updated upon migrations. The orchestration of the migration and the update of the configuration must be reliable to tolerate, in particular, a failure of the master. To this end, the shared configuration and migration orchestration leverage an instance of the ZooKeeper [59] coordination kernel. ZooKeeper maintains a simple filesystem-like hierarchy of shared objects used to reliably store the configuration. At the core of

ZooKeeper, a reliable atomic broadcast with ordering guarantees on multiple support servers can tolerate failures and maintain configuration availability. The whole state of the manager is stored in ZooKeeper. This allows to easily restart the manager in case of failure.

5.4 Elasticity Enforcer and Policy

The elasticity enforcer decides on the placement of operator slices on hosts by requesting migrations to the manager (e.g., migration of M:3 from host 2 to host 3 in Figure 5.4). It bases its decisions on input *elasticity policies* and on probes from the manager. The goal of the enforcer is to match policies requirements while minimizing the number and cost of slices migrations and thus the impact on service degradation.

The e-StreamHub elasticity policy combines *global* and *local* rules. Rules are evaluated as soon as a set of probes for all slices has been received since their last evaluation. Violations of global rules cause the system to scale in or out, by adding or removing hosts. Scaling out implies moving slices from overloaded hosts to newly added ones. Scaling in implies re-dispatching slices from the least loaded hosts to other hosts, and releasing the former. Local rules violations result in a re-allocation of slices among existing hosts. Global rules have the highest priority: local rules are only evaluated when no global rule is violated.

In this work, we use as primary metric for the elasticity policy the CPU utilization. Network bandwidth and memory constraint are only used as constraint during the migration decisions.

The global rule used in our evaluation states that the average CPU load for existing hosts must remain in the [30%:70%] range. e-StreamHub scales out if the average load exceeds 70% for at least 30 seconds and scales in if it drops below 30% for the same duration. The local elasticity policy states that the individual CPU utilization for a given host shall remain in the [20%:80%] range, also measured over a 30 seconds period. Finally, the policy sets a target (ideal) average CPU utilization of 50% and specifies a grace period of at least 30 seconds between migration requests in order to reduce the probability of thrashing.

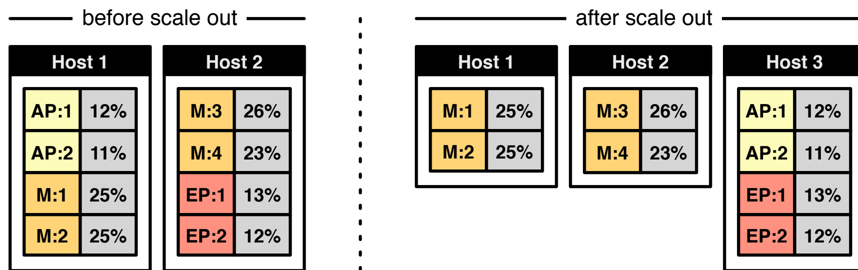


Figure 5.5 – Example of a slice placement decision.

The elasticity enforcer executes a two-step resolution algorithm when a global or local rule is violated. First, it decides on the set of slices to migrate. Second, it decides on a new placement (destination hosts) for these slices.

We start by describing the case of a scale out operation, upon a violation of the 70% average CPU utilization threshold of the global policy. The slice selection step consists in identifying a set of slices, whose summed CPU utilization is larger than, or equal to, the absolute difference between the current utilization and the average utilization target (50% in our case), starting from overloaded hosts. This corresponds to the *subset sum* problem [83], which is solved using dynamic programming with pseudo-polynomial complexity. This returns a set of valid solutions, among which we select the one incurring the minimal amount of state transfer (as reported by memory utilization probes). This allows us to minimize the cost and duration of migrations and to reduce service degradation.

The second step is the placement of the slices selected for migration. The enforcer uses a *First Fit* bin packing approach [35]. Hosts are modeled as bins with capacities reflecting the available CPU resources. Each slice is modeled as an item with weight equal to the CPU usage of the slice. A slice can only be assigned to a host when its load is smaller than the remaining CPU capacity of the host. The bin packing approach starts from the current placement of slices without the slices selected for migration. It greedily assigns migrated slices to hosts, in decreasing order of CPU utilization. Furthermore, the enforcer automatically derives allocation decisions for new hosts if the spare system capacity is not sufficient to accommodate a migrating slice without violating a local rule. The cost of the second step is linear in the number of hosts and slices in the system.

Figure 5.5 illustrates slice placement. The system starts with an average load of $\frac{74+73}{2} = 73.5\%$, violating the global rule and requiring the system to scale out. In order to reach a target utilization that is less than or equal to 50% for all hosts, the first step selects slices AP:1 and AP:2 for host 1 and slices EP:1 and EP:2 for host 2, among possible sets, as these have the lowest memory usage. The second step identifies that a new host is required for a placement that does not break the 50% max utilization rule. All selected slices are migrated to that new host.

When a global rule violation requires scaling in, the first step is to calculate the required number of hosts as the difference between the current number of hosts and the minimal number of hosts required for an average load equal to or higher than 50%. In the second step, the enforcer marks the least loaded host for release. Slices from that host are reassigned onto already running hosts and the host is released when it becomes empty. This procedure is repeated until the required number of hosts has been released.

Local policy enforcement uses the same two steps, except that only slices from the underloaded or overloaded host are considered by the bin packing algorithm.

5.5 Evaluation

We present in this section the experimental evaluation of e-StreamHub. We first evaluate the baseline performance without elasticity support. Then, we measure the cost of migrations and their impact on notification delays. We finally observe the behavior of e-StreamHub in terms of hardware resource utilization and notification delays, under synthetic and trace-based load evolution patterns.

5.5.1 Experimental Setup

We deploy e-StreamHub in a private cloud of 30 hosts running Debian Linux 6.0.7 Squeeze (kernel 2.6.32-48). Each host features two quad-core Xeon E5405 2.0 GHz processors and 8 GB of RAM. Hosts are interconnected by a 1 Gbps switched network. We use a fixed number of 8 slices for the AP and EP operators, and 16 slices for the M operator. The ZooKeeper service, the elasticity enforcer, and the e-StreamHub manager run on separate hosts. In addition to the AP, M and EP operators, we deploy on 4 dedicated hosts two convenience operators, the source and the sink, each with 4

slices. The source operator pushes subscriptions and publications to the system from pre-encrypted events stored on disk. The sink operator receives the notifications. We place the source and sink operator slices two-by-two onto the same nodes. We measure the notification delays between one source operator slice and the sink operator slices on the same host, to avoid the effect of potential clock drifts on delay measurements. The number of notifications is large enough for the measured average and standard deviation to be statistically significant.

5.5.2 Workload

Our evaluation focuses on the support of elastic pub/sub as a service running on a untrusted public cloud. We therefore use the ASPE [34] encrypted filtering scheme, and workloads of pre-encrypted subscriptions and publications. While the performance of plain-text filtering may depend on the characteristics of the workload, such as the possibility to leverage containment between subscriptions [4, 13, 27, 29, 41, 43, 78], encrypted filtering such as with ASPE requires filtering each incoming publication against all stored subscriptions. Each individual filtering operation cost is quadratic in number of attributes ($O(d^2)$). The static performance of e-StreamHub is thus impacted by two factors. First, the number of attributes d has an impact on the cost of each individual filtering operation at the M operator level. We use a ASPE schema with $d = 4$ attributes. Second, the matching rate is the probability that a publication will match each subscription. It impacts on the number of notifications an incoming publication generates, and therefore on the load at the EP operator level. We generate subscriptions with an average matching rate of 1%. Unless explicitly noted, we use a workload of 100,000 subscriptions. Each publication thus generates an average of 1,000 notifications. Since we use encrypted filtering, where each publication has to be matched against each one of the stored subscriptions, our experiments are independent of the nature of the workload as there is no support for containment in the ASPE scheme. We therefore do not need to use traces and rely on synthetic subscriptions and publications.

Our experiments always begin with a subscription storage phase where subscriptions are stored in the system prior to sending any publication. Then, the rate of publications is either the maximal one supported by the system (for the baseline evaluation in a non-elastic setting), based on synthetic descriptions of the rate evolution, or based

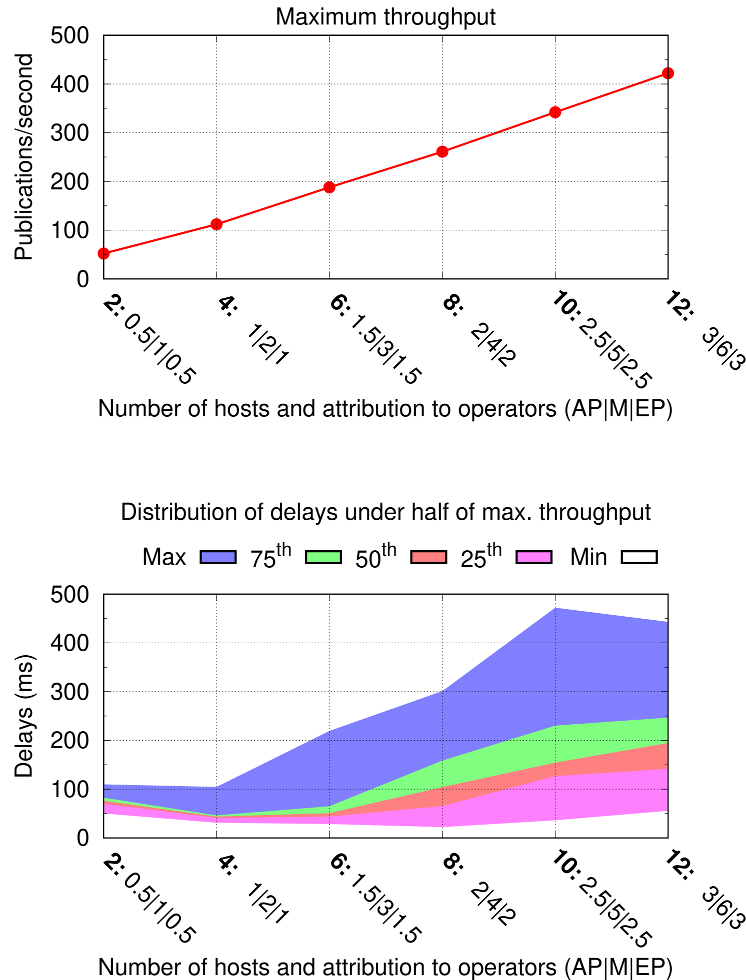


Figure 5.6 – Performance of static E-STREAMHUB (100 K subscriptions).

on the tick trace from the Frankfurt stock exchange described in our introduction (Figure 5.1).

5.5.3 Baseline StreamHub Performance

We start by evaluating the baseline performance of E-STREAMHUB with 100 K subscriptions. We consider static configurations of 2 to 12 hosts dedicated to the E-STREAMHUB engine and supporting AP, M, and EP operator slices. We used twice as many hosts for the M operator than for each of the two others. With 8 hosts, we deploy AP slices on 2 hosts (4 processors, 16 cores), M slices on 4 hosts (32 cores), and EP slices on 2 hosts (16 cores). Slices from two different operators can be placed on the same host (e.g., with 2 hosts, one host runs all AP and EP operators slices). This placement is not guaranteed to be optimal in terms of CPU utilization of the hosts but allows us to simply illustrate the linear scaling behavior.

	AP	**M** (12.5 K)	**M** (50 K)	**EP**
average	232 ms	1.497 s	2.533 s	275 ms
std. dev.	31 ms	354 ms	1.557 s	52 ms

Table 5.1 – Migration times, 100 publications/s, 12.5 K or 50 K subscriptions stored per M operator slice, for a total of 100 K and 500 K subscriptions, respectively.

Figure 5.6.up presents the maximal throughput supported by each configuration, before events start accumulating at the input of the AP operator. The throughput is perfectly linear as expected [14]. 12 hosts support a flow of 422 publications per second, corresponding to 42.2 millions encrypted filtering operations and 422,000 notifications sent per second.
We evaluate the delays by submitting to each configuration an incoming publication rate of half the maximal throughput, corresponding to the target load set in our elasticity policy. Figure 5.6.bottom presents the evolution of delays between the sending of each publication by the source operator and the reception of the last notification by the sink operator. For each configuration, stacked percentiles are represented as shades of grey. For instance, with 12 hosts, the minimal delay is 55 ms, while for 75% of the publications the notification is received by the last notified subscriber in less than 247 ms.

5.5.4 Operator Slice Migration Performance

We now proceed to the evaluation of the performance of slice migration. Table 5.1 presents the average and standard deviation of the migration time for the three operators, over 25 migrations. We consider larger slices: 4, 8, and 4 slices for the AP, M and EP operators, respectively deployed on 2, 4, and 2 hosts as the initial placement. Each migration picks a random slice of the corresponding operator and migrates it to another host. The system is under a constant flow of 100 publication/s, slightly less than half its maximal filtering throughput (Figure 5.6), and we consider the storage of 100 K subscriptions (12.5 K per M operator slice) and 500 K subscriptions (50 K per M operator slice). Migration times are very small for the AP operator, which is stateless, and for the EP operator, which has a very small state. The standard deviations are also small compared to the average. Migrations of M operator slices are more costly, as expected. The state size impacts on the migration time, but delays remain reasonable even for the very large number of subscriptions considered. We

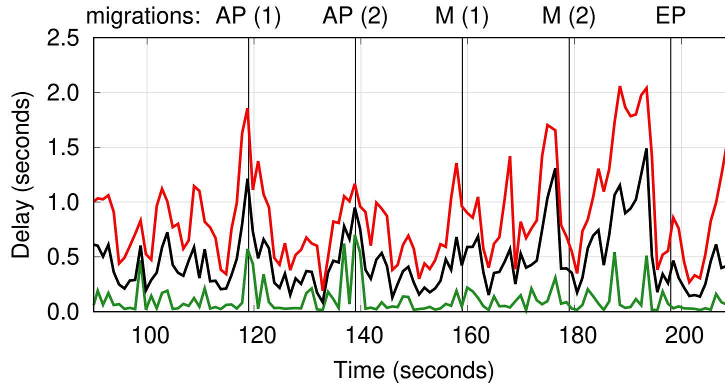


Figure 5.7 – Impact of migrations on delay (min, average, max).

deliberately evaluated small deployments (with 4 M operators) and large numbers of subscriptions (up to 500 K), to show that even disproportionately large migrations take place within a few seconds.

Our next experiment evaluates the impact of migrations on the experienced delays. We use the same configuration as previously, with 100 K stored subscriptions. Figure 5.7 indicates the delay average, deviation, and min/max values, and points the time when we perform migrations, respectively, for two AP operator slices consecutively, for two M operator slices consecutively, and finally for one of the EP operator slices. We observe that the notification delay increases from the steady state of 500 ms to a maximal value of less than two seconds after migrations take place, while the average notification delay remains below a second for the most part.

5.5.5 Elastic Scaling under Varying Workloads

We now evaluate the complete e-StreamHub with elasticity support. We start with a synthetic benchmark. Results are presented by Figure 5.8. The first plot depicts the load evolution. We start with a single host that initially runs all 32 slices (AP and EP: 8 slices, M: 16 slices). The system is initially fed with 100 K encrypted subscriptions. We gradually increase the rate of encrypted publications, until we reach 350 publications per second. After a rate stability period, we decrease the rate gradually back to no activity.

We observe the number of hosts used by the system, as dynamically provisioned by the enforcer, in the second plot. The minimum, average and maximum CPU load observed at these hosts are presented by the third plot. Finally the average delays and standard

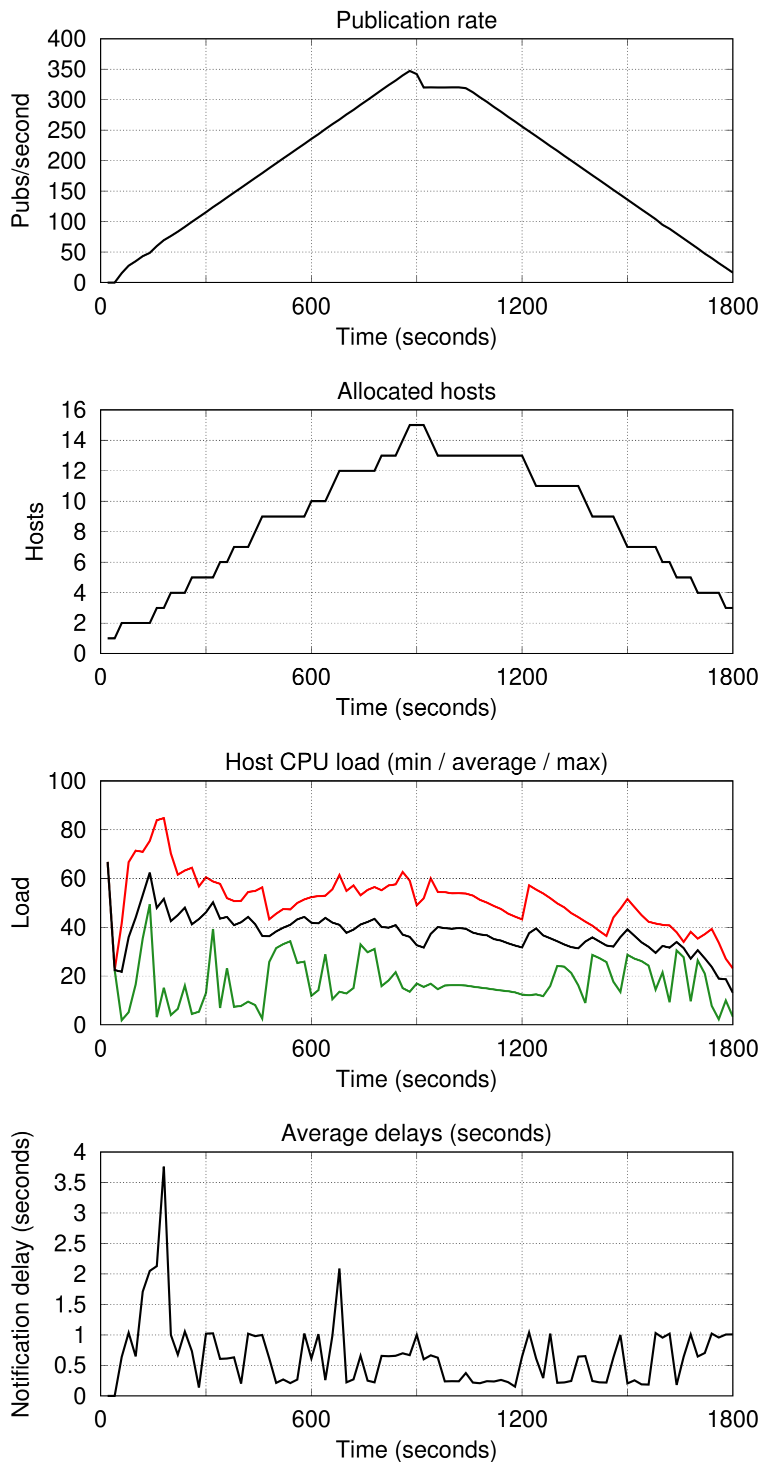


Figure 5.8 – Elastic scaling under a steadily increasing and decreasing synthetic workload and 20 K encrypted subscriptions.

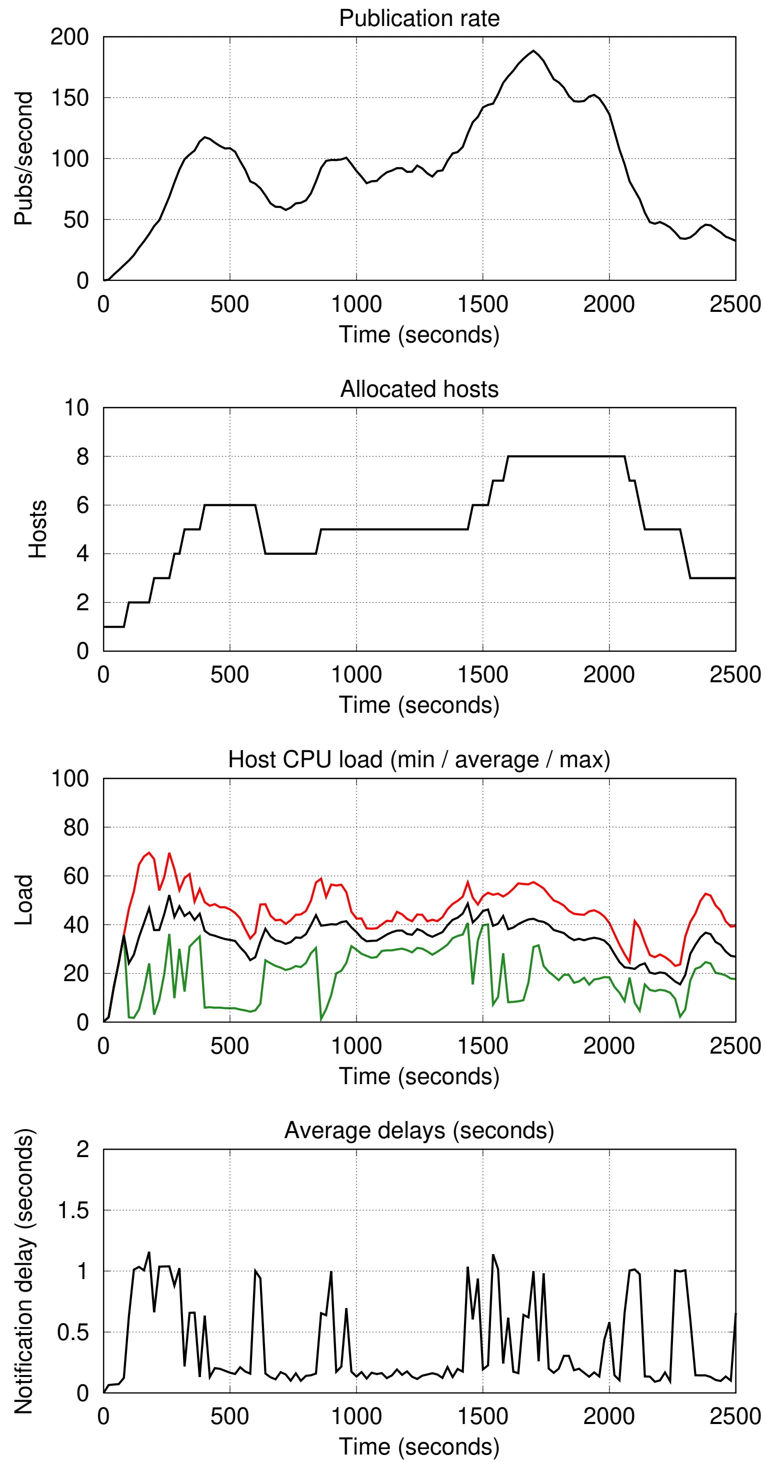


Figure 5.9 – Elastic scaling under load from the Frankfurt stock exchange.

deviations are shown on the fourth plot. For all plots, we present averages, standard deviations, minimum, or maximum values observed over periods of 30 seconds.

We clearly observe that the system meets the expectations set by the elasticity policy. The number of hosts gradually increases towards 15 hosts. This corresponds to the range of static performance expected (note that Figure 5.6 presents the maximal achievable throughput with more subscriptions, hence the deviation). As the publication rate decreases, hosts are released and we return to the initial configuration. The elasticity policy enforcement and corresponding 246 migrations during the experiment result in the load of the hosts remaining within an envelope of 40% to 70%, the average being close to the 50% target. Finally, we observe that the delays remain small despite migrations, which confirms the ability of the e-StreamHub approach to achieve elastic scale out and scale in with minimal degradation of the service performance. The initial migration from one to two hosts is the one that impacts most the delay. Indeed, the relative augmentation of load per host is more important in this case and roughly half of the operator slices have to be migrated to the new host, and the delays at the three operators may sum up if the three types are migrated as a same operation.

Our next and final experiment uses the same presentation as the previous, but instead of using a synthetic publication workload, we replay the trace-based publication activity from the Frankfurt stock exchange described in our introduction and in Figure 5.1. Results are presented in Figure 5.9 with the same format as for Figure 5.8. The publication rate from the source operator is based on the stock exchange ticks, while the set of 100,000 subscriptions is fixed. We consider a shorter time frame for the experiment. We speed up the tick trace by 10 times: one hour in the original trace corresponds to 3 minutes in the experiment (for a total duration of 40 minutes). We also reduce the maximum amount of publications per second to account for the limited size of our experimentation cluster: the flow is scaled down from 1,200 on Figure 5.1 to 190 publications per second (for a peak load of 19 millions evaluations and 19,000 notifications sent per second, 1,800 seconds after the beginning of the experiment). We observe that the number of machines for supporting the workload ranges from 1 to 8. The system reacts to changes in load, in particular at the beginning of day and for the spike observed in the afternoon, while lowering to 3 nodes for the lower load in the evening. We observe as with the previous experiment that the

load remains for all machines within the requested envelope. The effect of scale out operation (e.g., around 500 seconds) is to reduce the average and max load but may also be to let machines under-used: the subsequent scale-down experiment chooses the underloaded machine as a candidate for release, raising the minimal load observed. The average notification delay remain below one second for entire duration of the experiment despite migrations. This final trace-based experiment conveys the ability of e-StreamHub to quickly adapt to the experience load while keeping notification delays small and stable.

5.6 Summary

We presented the design, implementation and evaluation of an elastic content-based pub/sub engine, e-StreamHub. Pub/sub middleware eases the composition and integration of multiple applications, services and users across different administrative domains. Pub/sub is particularly suited as a cloud-provided service. The load experienced by such a service heavily depends on the application, the filtering scheme(s) used, and fluctuations of the service popularity. The unpredictability of the load poses a challenge for the appropriate dimensioning of the support infrastructure and for its evolution over time. We addressed this problem by describing mechanisms and techniques for building an elastic pub/sub engine that automatically scales out and in based on observation of the load experienced by the system. Elasticity takes place for each of the three operators forming the engine independently, allowing to adapt to the nature of the workload. Elastic scaling takes place through migrations of operator slices with minimal service degradation. The elasticity enforcer matches elasticity policies while reducing the cost and number of necessary migrations. Our evaluation using synthetic and trace-driven benchmarks indicates that e-StreamHub is able to react to dynamic changes in load, automatically adding and removing hosts as required by the elasticity policy.

"Non quia difficilia sunt non audemus,
sed quia non audemus difficilia sunt."
Lucius Annaeus Seneca, *Epistulae Morales ad Lucilium*

6 Application to Online Trading

This chapter is based on a paper that was previously published:
Exploiting Concurrency in Domain-Specific Data Structures: A Concurrent Order Book and Workload Generator for Online Trading. In *NETYS 2016: 4th International Conference on Networked Systems*, pages 16–31, Marrakech, Kingdom of Morocco, April 2016, Springer.

6.1 Introduction

Stock exchanges provide fully automated order matching platforms to their clients. For each security available on the market, a stock exchange broker maintains a structure called an *order book*, that agglomerates orders received from clients (see Figure 6.1). Orders can be of two kinds. *Bid* orders offer to buy a given security at a target (maximal) price, while *ask* orders propose to sell it, also at a target (minimal) price. A *matching engine* is in charge of comparing incoming bid and ask orders, triggering trade operations when a match exists.
With the advent of high-frequency trading, clients expect very low latencies from order matching platforms. The offered latency is actually a key commercial argument for stock exchange services [40]. Brokers and traders expect the latencies to be in the order of a few milliseconds. This means that the stock exchange matching service needs to have internal latencies that are at least one order of magnitude lower. To achieve such

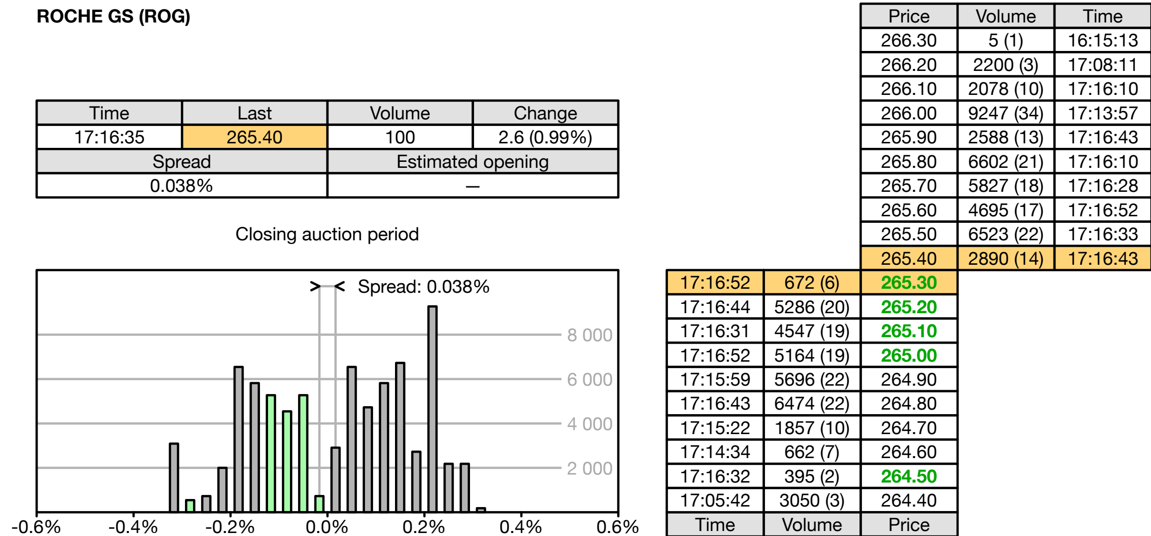


Figure 6.1 – An order book, as seen on real trading platform.

low latencies, designers of brokers started using new communication mechanisms [1] to gain advantages of a few milliseconds to even microseconds, sometimes resorting to dedicated hardware and custom algorithms running on FPGAs [77].

Instead of concentrating on communication mechanisms, we focus in this chapter on the effectiveness of the matching engine in order to minimize service latency and maximize throughput. The matching engine improvements are largely independent from those of communication mechanisms: as an incoming order must be processed by the matching engine before a response can be sent to the clients, a reduced matching time will improve end-to-end latency. State-of-the art matching engines thus far work sequentially [93], which means that, despite the system capacity to receive multiple orders concurrently, the processing of orders is handled one after the other. There is a great potential for obtaining performance gains for the matching operation, by taking advantage of the parallel processing capabilities of modern multi-core CPUs. We investigate in this chapter the support of concurrent order processing, and explore different design strategies to introduce parallelism in the non-trivial data structure that is the order book, starting from basic lock-based techniques to more sophisticated partially lock-free algorithms. The primary objective of this study is to demonstrate how one can turn a sequential data structure into a concurrent one by carefully combining different synchronization mechanisms and reasoning about concurrency under domain-specific constraints.

Order matching has interesting characteristics as it exhibits some clear potential for parallelism: there are multiple clients and two types of orders, and matching takes place only at the frontier between the two. At the same time, it is not trivial and presents a number of challenges that must be carefully addressed. First, we need to ensure that the output of the matching process in the concurrent case is the same as in the sequential case, notably when it comes to processing orders exactly once and according to arrival rank, [1] because clients are paying customers and real money is being traded. Second, as the system handles a variety of messages types (add/remove, sell/buy), it is not clear how to safely capture all message interactions in the concurrent case. Lastly, to fulfil an order, the matching engine can potentially access more than one existing order already stored in the book. In the concurrent case this might lead to several matching operations simultaneously accessing the same shared state, and special care needs to be taken to avoid possible data corruption associated with concurrency hazards. As such, the implementation need to be carefully designed so that the synchronization costs and algorithmic complexity do not outweigh the benefits associated with concurrent processing.
The first contribution of this work is the proposal and the evaluation of domain-specific strategies for processing orders concurrently in the order book. Specifically, the concurrent strategies we explore include: (1) a baseline thread-safe design based on a single global lock; (2) a fine-grained design for locking parts of the order book; and (3) several variants of partially lock-free designs, which trade runtime performance for weaker consistency guarantees. The second contribution of this work is the implementation of a synthetic workload generator that complies with widely-accepted models [86, 87]. We further use this workload generator to assess the effectiveness of our concurrent matching algorithms.

6.2 Online Trading and the Order Book

We first describe the principle and guarantees for trading operations. We start by defining some domain-specific terms. An **order** is an investor's instruction to a broker to buy (*bid*) or sell (*ask*) securities. There are two types of orders: *limit* and *market*

[1] To avoid possible confusion with the word "order" used to designate trading requests and for prioritizing operations (arrival and processing order), we will only use it in the former sense and resort to alternative expressions for the latter.

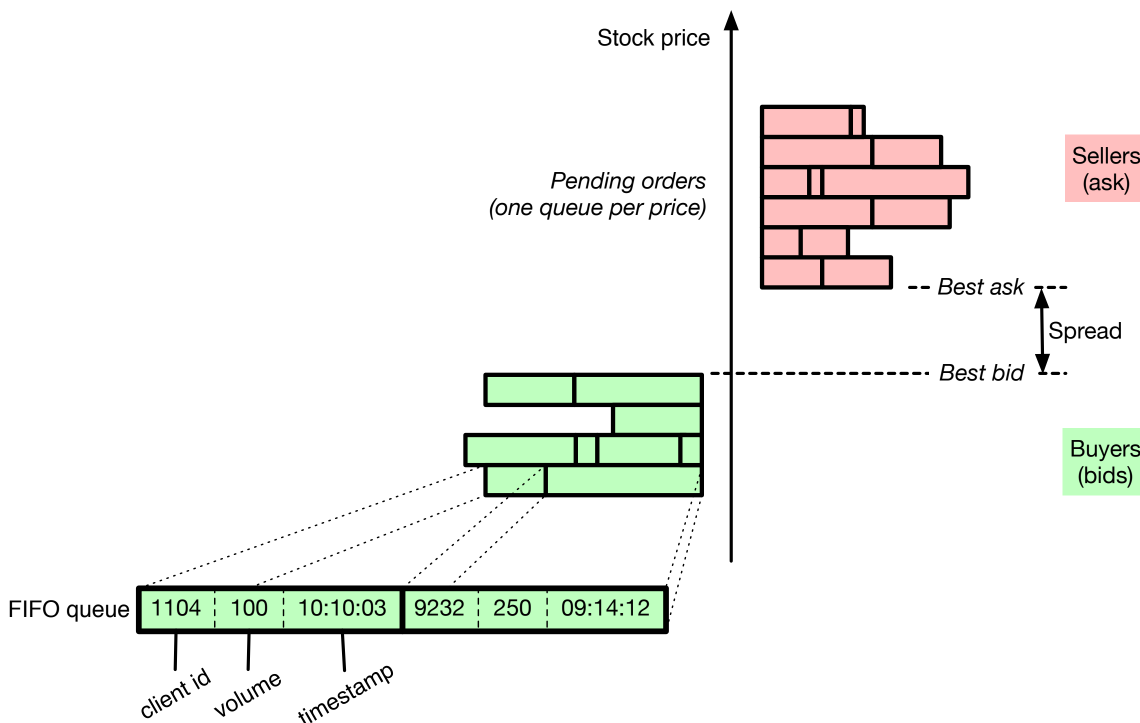


Figure 6.2 – Internal structure of the order book.

orders. A **limit order** specifies a maximum purchase price or minimum selling price. A **market order** does not specify a price and will be immediately matched with outstanding orders, at the best available price for this security. The **volume** indicates the amount of securities in an order as an integer value. The **order book** is a data structure for storing unfulfilled limit orders sent for a particular security. It features two queues, one for **asks** and one for **bids** orders. Orders stored in the book can be cancelled with a specific command. Finally, the **top of the book** consists of the ask with the lowest price and the bid with the highest price, and the difference between these two prices is the **spread**.

An order book maintains two separate queues, one for *bid* orders, and one for *ask* orders. Both data structures are organized in a way that facilitates the fast extraction of the best order as well as the quick insertion of new orders. In each of the queues, orders with the same price are aggregated, i.e., queues are organized as maps, where keys represent prices (with a granularity going up to the cent) and values are pending limit orders for a particular price. Pending orders for a particular price are sorted

according to arrival rank and, thus, upon arrival are stored in a first-in first-out (FIFO) queue (Figure 6.2).

Matching occurs only during order processing, when an incoming ask order satisfies some bid(s) or, vice versa, an incoming bid can be satisfied by some ask(s). When a match occurs, the associated existing orders are removed from the book. The priority of matching is driven by price, i.e., lowest selling prices (resp., highest) are sold (bought) first. If multiple orders have the same price, they are matched according to arrival time starting with the oldest. If there are no pending order to process, the system is in a *stable* state where the spread is positive and the two types of orders do not overlap.

To fulfil an order, the matching engine can "consume" more than one order on the other side of the book. This happens when an incoming order matches the best order on the opposite side of the order book, but it is not completely fulfilled and continues to match the next best order. This aggregation process stops once the incoming order has been filled completely, or when there are no more orders that can be consumed given the price and volume constraints. The remaining part of the incoming order is then added to the order book. Similarly, when the already existing order in the order book cannot be fully matched with the incoming order, the partially-matched order remains in the book with its volume decreased by the volume subsumed by the transactions.

The pseudo-code for the baseline sequential matching algorithm is shown in Algorithms 1 and 2. Sell and buy orders are stored it two separate *heaps*, each holding FIFO queues with orders of identical price sorted according to their arrival time. Queues are sorted by increasing price in the asks heap, and by decreasing price in the bids heap. The algorithm matches incoming orders against existing ones from the opposite heap, possibly adding them to the book if they are not completely fulfilled. To keep the pseudo-code as simple as possible, we assume that the heap at other side of the book is not empty when inserting an order and we do not explicitly handle the creation and removal of the queues in the heaps. This code is used as a basis for the concurrent variants presented in Section 6.3.

Algorithm 1 — Helper functions.

```
1: Type ORDER is:
2:    type: {LIMITED, MARKET}                              ▷ Limited or market price?
3:    operation: {BUY, SELL}                               ▷ Buy or sell?
4:    volume: integer                                      ▷ How many securities?
5:    price: float                                         ▷ At what price?
      ...
6:    id: integer                                          ▷ Timestamp (for concurrent algorithms)
7:    status: {IDLE, MATCHING, REMOVED}                    ▷ Status (for concurrent algorithms)

8: Type BOOK is:
9:    asks: heap of FIFO queues (orders of same price)     ▷ Sorted by increasing price
10:   bids: heap of FIFO queues (orders of same price)     ▷ Sorted by decreasing price
      ...

11: function CAN_MATCH(node, order)                        ▷ Can incoming order match node in book?
12:   if order.operation = node.operation then             ▷ Need order of opposite type
13:      return false
14:   if order.type = MARKET then                          ▷ Market orders always match
15:      return true
16:   if order.operation = SELL then
17:      return order.price ≤ node.price
18:   else
19:      return order.price ≥ node.price
```

6.3 A Concurrent Order Book

We now describe different strategies for supporting concurrency in the order book. This data structure is interesting because it is non-trivial and the matching operation may be time-consuming (e.g., when an incoming order matches and "consumes" many existing orders from the book). Hence, taking advantage of the parallel processing capabilities of recent multi-core architectures is obviously desirable.

Algorithm 2 — Sequential order insertion (single-threaded).

```
1: function HANDLE_ORDER_SEQ(order)
2:    sell ← (order.operation = SELL)
3:    while order.volume > 0 do
4:       q ← TOP(sell ? book.bids : book.asks)             ▷ Non-empty top queue on other side
5:       n ← FIRST(q)                                      ▷ Top order in the queue
6:       if ¬CAN_MATCH(n, order) then
7:          q ← GET(sell ? book.asks : book.bids, order.price)   ▷ Queue at price
8:          PUSH(q, order)                                 ▷ Store order in book (append to queue)
9:          break
10:      if n.volume > order.volume then
11:         n.volume ← n.volume − order.volume
12:         break
13:      order.volume ← order.volume − n.volume
14:      POP(q)                                            ▷ Remove top order from other heap
15:   return SUCCESS
```

It is, however, not easy to perform concurrent operations on the order book while at the same time preserving consistency. Some synchronization is necessary for correctness, but too much synchronization may hamper performance. We will start by discussing simple synchronization techniques and gradually move to more sophisticated strategies that achieve higher levels of concurrency.

In all concurrent approaches discussed below, requests to the order book are handled and processed by a pool of threads. As we would like the order book to yield the same output in a concurrent execution as when processing operations one at a time, we need to process requests in the same sequence as they have been received. We therefore insert incoming requests in a FIFO queue [2] and assign to each request a unique, monotonously increasing timestamp that we use to sort operations (see Algorithm 3, lines 1–8). We will discuss later scenarios where we can process some requests in a different sequence while still preserving the linearizability of the order book operations.

Before discussing our strategies for handling concurrent operations, let us first consider some observations about the specific properties of the order book. First, matching always occur at the top of the book. Therefore, the matching operation has interesting locality properties, and it will not conflict, for instance, with operations that are "far enough" from the top of the book. Second, an order that is matched upon insertion only needs to be inserted in the book if it is not fully matched. We can thus identify two interesting common cases: (1) an order is not inserted at the top of the book and hence no matching occurs, and (2) an order inserted at the top of the book is fully subsumed by existing orders and therefore does not need to be inserted in the book. Finally, there are several scenarios where we can straightforwardly determine that two concurrent limit orders do not conflict. For instance, insertions of an ask and a bid can take place concurrently if there is no price overlap between them, i.e., the ask has a higher price than the sell, as they cannot both yield a match. As another example, insertions of two limit asks or two limit bids can take place concurrently if they have different prices (i.e., they are in different queues) and they do not both yield a match. These observations will be instrumental for the design of advanced concurrency strategies.

[2] We assume that this queue is thread-safe as processing threads may dequeue orders concurrently with one another and with the (unique) thread that enqueues incoming orders.

6.3.1 Coarse-Grained Locking

Algorithm 3 — Coarse-grained locking and common functions.

```
 1: Variables:
 2:    incoming: FIFO queue (orders)          ▷ Thread-safe queue for incoming orders
 3:    ts: integer                            ▷ Timestamp for incoming orders (initially 0)
 4:    sgl: lock                              ▷ Single global lock (initially unlocked)
       ...
 5: upon RECEIVE(order):                      ▷ Reception of an order from a client (single thread)
 6:    order.id ← ts                          ▷ Assign unique timestamp
 7:    PUSH(incoming, order)                  ▷ Append order to queue
 8:    ts ← ts + 1
 9: function THREAD_PROCESS^SGL               ▷ Processing of an order by a thread
10:    order ← POP(incoming)                  ▷ Take next order from queue
11:    LOCK(sgl)                              ▷ Serialize processing
12:    r ← HANDLE_ORDER_SEQ(order)            ▷ Use sequential algorithm
13:    UNLOCK(sgl)
14:    return r
```

We first consider the trivial approach of using a single lock (SGL) to serialize accesses to the shared data structure. To simplify the presentation of concurrent algorithms, we assume that threads from the pool repeatedly execute the function THREAD_PROCESS to process one order from the incoming queue, and the result of this function is then returned to the corresponding client. The basic operating principle of the coarse-grained approach is shown in Algorithm 3, lines 9–14. Threads from the pool acquire the main lock, process the next order from the queue, and release the lock before the response is sent back to the client. Hence, the processing of orders is completely serialized and no parallelism takes place for this operation. The main advantage of this approach is its simplicity, which also makes the algorithm easy to prove correct. It will serve as a baseline for the rest of the chapter.

6.3.2 Two-Level Fine-Grained Locking

We now explore opportunities for finer-grained locking to increase the level of concurrency. We start from the observation that two threads accessing limit orders from the book with different prices, i.e., located in different queues in the heaps, can do so concurrently without conflicts. Therefore, it is only necessary to control access to the queues that are accessed by both threads.

The principle of the two-level locking strategy is shown in Algorithm 4. As before, threads first attempt to acquire the main lock (line 3). Once a given thread acquires

Algorithm 4 — Fine-grained locking.

```
 1: function THREAD_PROCESS^FGL                      ▷ Processing of an order by a thread
 2:   order ← POP(incoming)                           ▷ Take next order from queue
 3:   LOCK(sgl)                                       ▷ Serialize traversal of heap
 4:   sell ← (order.operation = SELL)
 5:   v ← order.volume
 6:   q ← TOP(sell ? book.bids : book.asks)           ▷ Top queue at other side
 7:   while v > 0 do
 8:     if ¬CAN_MATCH(FIRST(q), order) then
 9:       break
10:     LOCK(q)                                       ▷ Acquire lock on queue
11:     v ← v − VOLUME(q)                             ▷ Subtract volume of all orders in queue
12:     q ← NEXT(q)                                   ▷ Next queue from heap
13:   if (v > 0) then
14:     q ← GET(sell ? book.asks : book.bids, order.price)   ▷ Queue at price
15:     LOCK(q)                                       ▷ Acquire lock on queue
16:   UNLOCK(sgl)
17:   r ← HANDLE_ORDER_SEQ^FGL(order)                 ▷ Use sequential algorithm
18:   return r

   function HANDLE_ORDER_SEQ^FGL ≡ HANDLE_ORDER_SEQ^SGL        ▷ Algorithm 2
   ... UNLOCK(q) ...        ▷ All locks released once no longer needed (before line 15)
```

the main lock, it traverses the queues in the opposite heap of the book in sequence, starting from the top, and locks each visited queue individually (lines 7–12). This process stops as soon as the accumulated volume of orders in already traversed queues reaches the volume of the incoming order. In case of a limit buy or ask order, the process also stops whenever visiting a queue at a price that is higher, respectively lower, than the price of the incoming order. Finally, if the incoming order has not been fully matched, it needs to be inserted in the book and we also lock the queue associated with the price of the incoming order (lines 13–15). The algorithm then releases the main lock (line 16). It can now safely perform the actual matching operations on the previously locked queues, including the optional insertion of the incoming order in the book if some unmatched volume remains, and release the individual locks as soon as they are no longer needed. To that end, we simply reuse the sequential algorithms with the addition of lock release, which happen right before line 15 in Algorithm 2 or whenever a queue becomes empty. [3]

This approach provides higher concurrency than coarse-grained locking because the algorithm holds the main lock for a shorter duration, when determining which queues from the book will be accessed. To ensure consistency, it uses a second level of locks

[3] Some implementation details, such as avoiding a second traversal of the heap by keeping track of locked queues, are omitted for simplicity.

for concurrency control at the level of individual queues. Therefore, multiple orders can execute concurrently if they operate in different parts of the order book, but they are serialized if the sets of queues they access overlap.

6.3.3 Toward Lock-Free Algorithms

The final stage in our quest for concurrency is to try to reduce the dependencies on locks, whether coarse- or fine-grained, as they introduce serial bottleneck and may hamper progress. In particular, a thread that is slow, faulty, or preempted by the OS scheduler while holding a lock may prevent other threads from moving forward.
Our objective is thus to substitute locking operations by lock-free alternatives. To that end, we first need to remove the locks protecting the queues and permit threads to enqueue and dequeue orders concurrently. We do so by replacing the queues in Algorithm 1, lines 9–10, by a concurrent heap structure for backing the order book. Specifically, we use a concurrent map,[4] which imposes a custom sorting of the orders it contains. First, orders are sorted according to prices, and then, according to the timestamp.
To handle concurrent accesses explicitly, we also add in the order object an additional status flag that we use to indicate whether the order is being processed or has been removed by some thread. We modify this flag in a lock-free manner using an atomic compare-and-set (CAS) operation.
The $ordr.status$ flag (Algorithm 1, line 7) can be in one of three states: `IDLE` indicates that the order is not being processed by any thread; `MATCHING` specifies that some thread is processing the order; and `REMOVED` means that the order, although still present in the order book, has been logically deleted.
We have developed three variants of the concurrent, almost[5] lock-free algorithm, with each having different guarantees. The first algorithm, which we call LF-Greedy, provides the least guarantees in terms of the sequence in which orders are processed. The second algorithm, LF-Priority, prevents an incoming order from consuming new orders that have arrived later. The last algorithm, LF-FIFO, additionally prevents

[4] `java.util.concurrent.ConcurrentSkipListMap`.

[5] While the algorithms do not use explicit locks, they are not completely "lock-free" as in some situations a thread may be blocked waiting for the status flag to be updated by another thread. Techniques based on "helping" could be used to avoid such situations, at the price of increased complexity in the algorithms. We therefore slightly abuse the word "lock-free" in the rest of the chapter.

incoming orders arriving later from stealing existing orders from incoming orders arriving earlier.

For the sake of simplicity, the pseudo-code as presented further does not show the handling of market orders. Instead it only considers the more general case of limit orders. In the case of market orders, if there is no or only a partial match, the unmatched orders are returned back to the issuer. We furthermore omit obvious implementation-specific details, e.g., an incoming order is naturally matched against the opposite side of the order book.

Algorithm 5 — Greedy [and priority] order insertion algorithm.

```
 1: function THREAD_PROCESS^Greedy/Priority          ▷ Processing of an order by a thread
 2:   order ← POP(incoming)
 3:   while order.volume > 0 do
 4:     n ← 1^kt node from heap [ such that n.id < order.id ]
 5:     if n.status = REMOVED then                   ▷ Order logically removed?
 6:       heap ← heap \ {n}                          ▷ Yes: remove order from book
 7:       continue
 8:     wait until n.status ≠ MATCHING               ▷ Avoid useless CAS
 9:     if ¬CAS(n.status, IDLE, MATCHING) then       ▷ Take ownership of node
10:       continue
11:     if ¬CAN_MATCH(n, order) then                 ▷ Can we match order?
12:       heap ← heap ∪ {order}                      ▷ No: store order in book
13:       n.status ← IDLE
14:       break
15:     if n.volume > order.volume then              ▷ Order fully satisfied
16:       n.volume ← n.volume − order.volume
17:       n.status ← IDLE
18:       break
19:     order.volume ← order.volume − n.volume       ▷ Node fully consumed
20:     n.status ← REMOVED
21:   return SUCCESS
```

The LF-Greedy algorithm

The LF-Greedy algorithm (see Algorithm 5, omitting text within square brackets) works as follows. After the incoming order *ordr* has been received and scheduled for processing, the worker thread obtains the best order n from the order book's heap. Orders in our lock-free algorithms can be marked as `MATCHING` to indicate that they are being processed, or as `REMOVED` when logically deleted but still still physically present in the order book. As such, the thread first checks if n has been marked as removed (line 5). If so, it removes n from the order book (line 6) and continues to another iteration of the algorithm. Otherwise, we know that n has not been removed, and we need to check whether some other thread has already started processing n,

in which case we wait until the processing has finished by polling the *n.status* flag (line 8). Thereafter, we attempt to change the status of n from `IDLE` to `MATCHING` using CAS (line 9). If the CAS operation succeeds, then we know that the order was indeed idle (note that it could have been removed in the meantime, or taken over for matching by another thread) and the thread has successfully taken exclusive ownership over it. If the CAS operation fails, then some other thread must have just changed the order's status to either `MATCHING` or `REMOVED` and we continue to another iteration of the algorithm.

After taking ownership of n, we need to check if the price of the incoming order *ordr* could be matched with the price of n. If not, we store *ordr* in the order book and release n by setting its status to `IDLE` (line 13), effectively finishing the matching process of the incoming order. Otherwise, if the prices of n and *ordr* can be matched, we check if the incoming order *ordr* could fully consume n. If so, we decrease the incoming order's volume (line 19) and mark n as `REMOVED`. Note, that we do not physically remove n from the *heap* at this step; instead, we rely on other threads' help for removing it lazily (lines 5 and 6). If the volume of n is larger than *ordr* can consume, we decrease it and unlock n (line 17). This implies that the outstanding volume of n remains in the order book and can be consumed by other threads.

If the incoming order has a large volume, it can potentially consume multiple orders from the book. In this version of the algorithm we do not enforce any restrictions on which existing orders can be consumed by the incoming order, i.e., concurrent threads might consume existing orders that are interleaved in the heap.

The LF-Priority algorithm

The LF-Priority algorithm provides more guarantees in terms of sequence in which incoming orders consume existing orders stored in the book. Specifically, when matching an incoming order *ordr* with the content of the book, we want to only consider existing orders that have been received strictly before *ordr* has been received. To that end, we rely on the timestamp *ordr.id* assigned to each order upon arrival (Algorithm 3, lines 6). We then modify Algorithm 5 by adding an extra condition (line 4 between square brackets), which restricts *ordr* to only process orders from the order book having a smaller timestamp. This condition can be supported straightforwardly in our implementation because of the key we use to store orders in

the concurrent heap. Indeed, we use the same concurrent map as before and, when retrieving the best order, we apply an extra filter condition to select orders having keys with timestamps that are smaller than the currently processed order.

The LF-FIFO algorithm

To introduce the LF-FIFO algorithm, we first informally discuss where LF-Priority is lacking and how its shortcomings can be addressed. In Algorithm 5, if the thread processing an order is delayed (e.g., preempted by the OS scheduler), an order arriving later might consume the best outstanding orders in the book. This is a problem if one needs to enforce that orders arriving first are given precedence over orders arriving later. Furthermore, concurrent orders may consume interleaved orders from the book, i.e., an incoming order may be matched against a set of existing orders that does not represent a continuous sequence in the book, hence breaking atomicity. The main idea of LF-FIFO is therefore to prevent threads from consuming orders from the book before the processing of incoming orders received earlier has finished.

To provide these stricter guarantees, we employ ideas from the hand-over-hand locking [76] technique. The principle is that, when traversing a list, the lock for the next node needs to be obtained while still holding the lock of the current node. That way, threads cannot overtake one another. The pseudo-code of the LF-FIFO is show in Algorithm 6. A thread processing an incoming order $order$, which would consume multiple existing orders, first performs a CAS on the first best order (line 12), marking it as removed in the end (line 26). Then it saves the first best order in a local variable (line 28) and continues to another iteration, during which it select the second best node (line 5) and perform a CAS to atomically change its status to `MATCHING`. Upon success and only then do we physically remove the first best node from the heap (line 15).

The process describing order removals is distinctly different from that which was presented in prior algorithms. In LF-Greedy and LF-Priority algorithms, when an arbitrary thread detects that an order has been marked as `REMOVED` (Algorithm 5, line 5), it helps by removing that order (i.e., lazy removal with helping). In contrast, instead of assisting in the removal of n from the heap, the LF-FIFO algorithm restarts from the beginning (line 9), relying on the thread that has marked n as `REMOVED` to

Algorithm 6 — Order insertion algorithm with FIFO properties.

1: **function** THREAD_PROCESS$^{\text{FIFO}}$ ▷ Processing of an order by a thread
2: $\quad$ *order* $\leftarrow$ POP(*incoming*)
3: $\quad$ $p \leftarrow \bot$ ▷ Previous node fully matched by thread
4: $\quad$ **while** $order.volume > 0$ **do**
5: $\quad\quad$ $n \leftarrow 1^{\text{kt}}$ node $n \neq p$ from *heap* such that $n.id < order.id$
6: $\quad\quad$ **if** $n = \bot$ **then** ▷ Any matching order in book?
7: $\quad\quad\quad$ $heap \leftarrow heap \cup \{order\}$ ▷ No: store order in book
8: $\quad\quad\quad$ **break**
9: $\quad\quad$ **if** $n.status =$ `REMOVED` **then** ▷ Order logically removed?
10: $\quad\quad\quad$ **continue** ▷ Yes: wait until physically removed
11: $\quad\quad$ **wait until** $n.status \neq$ `MATCHING` ▷ Avoid useless CAS
12: $\quad\quad$ **if** $\neg$CAS($n.status$, `IDLE`, `MATCHING`) **then** ▷ Take ownership of node
13: $\quad\quad\quad$ **continue**
14: $\quad\quad$ **if** $p \neq \bot$ **then**
15: $\quad\quad\quad$ $heap \leftarrow heap \setminus \{p\}$ ▷ Delayed removal
16: $\quad\quad\quad$ $p \leftarrow \bot$
17: $\quad\quad$ **if** $\neg$CAN_MATCH(n, $order$) **then** ▷ Can we match order?
18: $\quad\quad\quad$ $heap \leftarrow heap \cup \{order\}$ ▷ No: store order in book
19: $\quad\quad\quad$ $n.status \leftarrow$ `IDLE`
20: $\quad\quad\quad$ **break**
21: $\quad\quad$ **if** $n.volume > order.volume$ **then** ▷ Order fully satisfied
22: $\quad\quad\quad$ $n.volume \leftarrow n.volume - order.volume$
23: $\quad\quad\quad$ $n.status \leftarrow$ `IDLE`
24: $\quad\quad\quad$ **break**
25: $\quad\quad$ $order.volume \leftarrow order.volume - n.volume$ ▷ Node fully consumed
26: $\quad\quad$ $n.status \leftarrow$ `REMOVED`
27: $\quad\quad$ $n.id \leftarrow order.id$ ▷ Prioritize concurrent insertions
28: $\quad\quad$ $p \leftarrow n$ ▷ Keep in book (to avoid being overtaken)
29: $\quad$ **if** $p \neq \bot$ **then**
30: $\quad\quad$ $heap \leftarrow heap \setminus \{p\}$ ▷ Remove last consumed order
31: $\quad$ **return** `SUCCESS`

also physically remove it from the heap (lines 15 and 30). Therefore, the LF-FIFO algorithms provides weaker progress guarantees but better fairness between threads.

6.4 Generating Workloads

Besides algorithms for exploiting concurrency in the order book operation, we contribute in this section a workload generator that allows evaluating the throughput of the matching operation under realistic workload assumptions.

The sensitive nature of financial data and the strict rights of disclosures signed between clients of stock quote operators typically prevent from using real datasets and call instead for appropriate models for synthetic data generation. Models emerged in economics and econophysics (i.e., physicists' approaches to tackle problems in economics) such as the ones by Maslov [86], Bartolozzi [15] and Bak et al. [10]. These

models allow understanding the properties of the order book in terms of the total volume of securities available or requested at each price point in the bid and ask queues. This aggregated information is enough for the targeted users of these models, who are interested in modelling and implementing investment strategies based on the total volume of securities at each price point, independently from their origin or destination. The distribution of individual order sizes has been studied separately, and shown to follow a power law by several authors [21, 48, 87]. Some models that consider individual orders nonetheless use a unit order size rather than a distribution in the interest of simplicity [71, 86].

We implement a variation of the model proposed by Maslov [86], which uses simple rules and which output has been shown to compare well with the behaviour of a real limit order-driven market. The original model assumes however, similarly to [71], that all orders have the same volume of one single security. This simplification is problematic for testing a matching engine, in particular for testing its behaviour and performance in the presence of partially matched orders. We therefore extend the model by allowing orders to feature arbitrary volumes and assign volumes following a power law distribution based on findings made by Maslov and Mills in [87]. We note that another limitation of this model is that it does not consider changes to existing orders stored in the order book, unlike for instance the Bak-Paczuski-Shubik model [10]. We choose not to address this limitation as it does not fundamentally limit the representativeness of the behaviour of clients using the order book for what concerns the matching algorithm itself. The expiry mechanism for existing orders proposed by the model, along with new insertions is indeed enough to model dynamics.

We now proceed to detailing the model itself. An average price p is fixed at the beginning or the generation, which starts by the generation of one bid and one ask limit order. Thereafter, orders are generated by first deciding on their operation (bid or ask), with equal priority. Each order is a *limit* order with priority q_{lo}, and a *market* order otherwise. The price attached to a limit order is generated based on the base price b of the best available order on the other side of the order book: the cheapest ask for a bid, and the largest bid for an ask. A random variation Δ, generated randomly in $\{1, 2, ..., \Delta_{max}\}$ is applied: the price for the order is set to $p(t) + \Delta$ for a bid, or to $p(t) - \Delta$ for an ask. The volume v for each order is generated according to the power law identified in [87]. For market orders, $P[v] \propto v^{-1-\mu_{\mathrm{maerlk}}}$ where $\mu_{\mathrm{e\ r\ \ st}} = 1.4$. For

limit orders, $P[v] \propto \frac{1}{v} e^{-\frac{(A-\ln(v))^2}{B}}$, where $A = 7$ and $B = 4$. These values for $\mu_{\text{e r st}}$, A and B are the ones suggested in the original paper [87], as are the values we use for the other parameters: $q_{lo} = \frac{1}{2}$, and $\Delta = 4$. We use an initial price of $p = 1,000$.

In order to prevent limit orders staying indefinitely in the order book, an expiry mechanism removes unmatched limit orders from the order book after λ_{max} time steps. The expiry mechanism prevents the accumulation of limit orders having prices that differ significantly from the current market price. In the real market, this operation is performed by either traders or by the stock exchange itself. For instance, the New York Stock Exchange purges all unmatched orders at the end of the day. Maslov indicates that for any reasonably large value of the cut-off parameter λ_{max}, the model produces the same scaling properties of price fluctuations. We use $\lambda_{max} = 1,000$ as in the original paper.

6.5 Evaluation

We experiment on two different architectures (Figure 6.3). The first is an Intel i7-5960X Haswell CPU (8 cores, 16 hardware threads with hyperthreading enabled) with 32 GB of RAM. The second is an IBM POWER8 S822 server (10 cores, 80 hardware threads) with 32 GB of RAM. We run our experiments in OpenJDK's Runtime Environment, build 1.8.0_40, with default options.

For all concurrent order book implementations considered, we process 100,000 orders in total. We vary the thread count from 1 to the maximum number of threads supported by each architecture. We run all experiments 10 times and present the average. The orders are generated offline using the model from Section 6.4, kept in memory and replayed directly to each of the order book implementations. For each of the experiments performed, we also plot the obtained speedup related to the baseline sequential matching engine running with a single thread.

For all the tests, we observe that the lock-free approaches outperform fine-grained locking. The latter approach does not scale beyond 4 threads for the Haswell architecture and 8 threads for POWER8. When more threads are used, however, its performance does not degrade significantly and remains relatively constant. In contrast, when looking at the lock-free approaches, we see that they scale almost linearly. Also, we see that the more guarantees in terms of the sequence in which orders are processed a

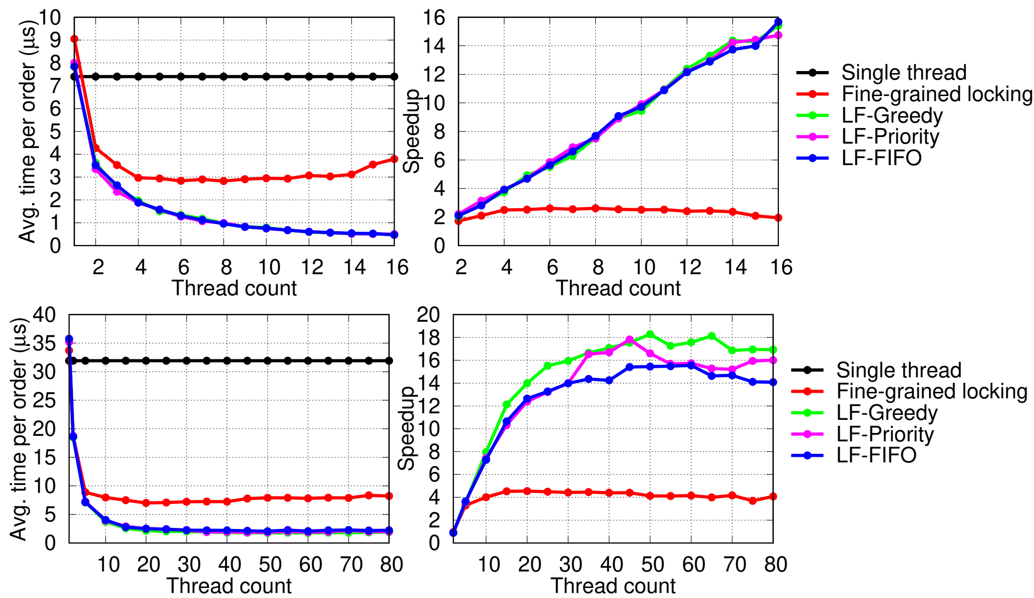


Figure 6.3 – Order processing time (average over 100,000) and speedup for different order book implementations on the Intel Haswell (top) and IBM POWER8 (bottom) architectures.

lock-free algorithm provides, the slower it performs. The variations in performance are, however, minimal.

6.6 Summary

We proposed in this chapter strategies for performing order matching in the order book in a concurrent manner. We started with two lock-based implementations using coarse- and fine-grained locking designs. We then proposed three algorithms that do not use explicit locks and provide different guarantees in terms of the sequence of order processing. We also contributed a workload generator that allows us to evaluate the throughput of order matching under realistic workload assumptions. Experimental results suggest that, although the fine-grained approach scales only up to a few cores, by carefully substituting locking operations by lock-free alternatives, we can achieve high performance and good scalability. Future work might target combining our concurrent matching engine with *LMAX Disruptor*, forming a cohesive framework where both, order dispatch and matching, are executed in an almost lock-free manner.

"Je suis de ceux qui pensent que la science est d'une grande beauté. Un scientifique dans son laboratoire est non seulement un technicien : il est aussi un enfant placé devant des phénomènes naturels qui l'impressionnent comme des contes de fées."

Marie Skłodowska Curie

7 Conclusion

Publish/subscribe is a popular approach that fulfills many of nowadays needs in distributed systems.
Chapters 3 and 4 discussed how a publish/subscribe engine could be provisioned and scale on a massively parallel infrastructure such as a cloud.
Chapter 5 focused on how to extend the aforementioned engine to allow it to cope with load variations that occur at runtime.
Chapter 6 presented an application to online trading, more specifically, to order book management, a problem that shares some common aspects with publish/subscribe.
In this chapter, I summarize the contributions of this thesis and conclude by presenting possible future directions, in which this work could be extended.

7.1 Summary of Contributions

The *first problem* that we decided to address was the **provisioning of a content-based publish/subscribe engine**. The outcome of this preliminary research was the establishment of a stage-driven architecture especially suitable for scaling. We also discussed how publications and subscriptions partitioning could be applied to cope with demanding workloads. We also evaluated the sub-linear cost of filtering algorithms on large content-based publish/subscribe systems. We highlighted metrics

and methods that are appropriate in the implementation and the deployment of a large-scale content-based publish/subscribe engine.

The *second problem* that we selected to address was **the design and the implementation of a scalable content-based publish/subscribe engine**. Our implementation, StreamHub, exhibited valuable scalability properties. StreamHub outperforms well-known content-based publish/subscribe engines that rely on broker overlays. StreamHub architecture splits the process into three scalable operators that are allocatable horizontally and vertically. On the testbench of 48 machines (for a total of 384 cores), StreamHub can sustain high throughputs of subscriptions and publications requests, filtering thousands of publications against hundreds of thousands of registered subscriptions, generating thousands of notifications to subscribers per second.

The *third problem* that we selected to address was **the addition of elasticity features to our content-based publish/subscribe engine**. Publish/subscribe is an excellent candidate to be provided as PaaS; ideally, the usage in terms of computing nodes should follow the load of the application. e-StreamHub is the successor of StreamHub designed to fulfill this requirement. Elasticity scaling is possible thanks to migrations of operator slices; an elasticity enforcer orchestrates migrations of slices in a smart way to match the elasticity policies. Our evaluation demonstrates that e-StreamHub scales up and down according to variations of the load; automatically provisioning and releasing nodes according to a predefined elasticity policy.

Finally, the *fourth problem* that on which we focused was **implementation of a highly concurrent matching engine**. Even though the matching engine of the stock exchange market shares many properties with the publish/subscribe paradigm discussed in the previous problems, it has some specific requirements that change the specter of possible solutions. Our first approaches are two lock-based implementations using coarse- and fine-grained locking. Then we propose three other algorithms that are not using explicit locks and provide various order processing guarantees. Experimental results show that the fine-grained approach scales up only to a few cores and that lock-free alternatives achieve higher performance with better scalability.

7.2 Future Directions

The presented outcomes of this thesis motivate additional ideas in the directions of improving content-based publish/subscribe. In the following sections, we list possible future directions that could be taken starting of our contributions.

7.2.1 Scalable Content-Based Publish/Subscribe Engine

Content-based publish/subscribe workloads require a lot of computing power on nodes. Various approaches address this issue by using specific hardware features.
By using *high-end graphics processing units (GPU)* Margera and Cugola [37, 81] built a content-based pub/sub system that benefit from the highly parallel and affordable processing power available in modern GPUs. Their research exhibits clear speedups with a higher number of attributes when compared to SIENA [27]. The authors emphasize on how it is hard to write efficient code on GPUs and about the various restrictions. While the use of GPUs is not exactly a silver bullet, the outcome of this research could be adapted and evaluated in StreamHub.
By taking advantage of *hardware supported networking filtering mechanisms* [17, 18] Bhowmik *et al.* propose a hybrid content-based pub/sub engine system in a manner that it combines filtering of events in the application and the network layers. A similar approach would be interesting as a pre-filtering step on StreamHub, deporting some of the filtering cost on the network infrastructure.

7.2.2 Elasticity

Our elastic content-based pub/sub system, e-StreamHub, is able to automatically scale out and in according to the load of the system.
Like Dynamoth [50], e-StreamHub's elasticity enforcer uses probes to measure the current system load as an input for scaling. We could investigate in *elasticity policy enforcer that complies to different type of SLAs, i.e. having a guarantee on latency.*
e-StreamHub evaluation was done on a homogenous cluster of nodes. Handling elastic scaling on a heterogeneous cluster could be a great improvement. On such a cluster, an *elasticity policy enforcer that tries to minimize the global power consumption* could be an interesting investigation. This work might be done using load

probes conjointly with power consumption probes (such that the one being used by Kurpicz [74]).

7.2.3 Application to Online Trading

Our various order book implementations confirmed our expectations, but there are still some open ways for further research.
One could be the use of specific CPU features; a good candidate could be the *hardware supported transactional memory*; available on many Intel processors since the generation codenamed Haswell with the instruction set TSX-NI [96] and available on IBM Power processors since generation 8 with instruction sets TLE/TLS [75]. Transaction memory allows batches of load and store operations to execute as a single atomic commit without the need for any explicit locks; it improves the overall performance and simplifies the parallel programming model. A Java/Scala bindings library for Intel TSX has been written [1] and could be used as an extension of our application. Note, on Intel TSX, the transactional memory code blocks should not alter memory than the level 1 cache [117]. The size of the L1 cache varies from 32kB to 896kB on the Intel CPU products range, TSX is an extremely promising instrument, but this limitation might prevent porting large data structures.
As argued in Chapter 6, high-frequency market making is done at a pace that forces users to move to high-speed algorithms and data representations. The port of our algorithms on FPGAs needs to be investigated in order to be compared with existing engines.

[1] TM4j on GitHub (P. Marlier and R. Barazzutti): `https://github.com/rbarazzutti/tm4j`

A
Publications

[1] Raphaël Barazzutti, Pascal Felber, Hugues Mercier, Emanuel Onica, Etienne Riviere. **Thrifty privacy: efficient support for privacy-preserving publish/subscribe**. In *DEBS 2012: Sixth ACM International Conference on Distributed Event-Based Systems*, pages 225-236, Berlin, Germany, July 2012. ACM.

[2] Raphaël Barazzutti, Pascal Felber, Hugues Mercier, Emanuel Onica, Jean-Francois Pineau, Etienne Rivière, Christof Fetzer. **Infrastructure Provisioning for Scalable Content-Based Routing: Framework and Analysis**. In *NCA 2012: 12th International Symposium on Network Computing and Applications*, pages 228-235, Cambridge, MA, USA, Aug 2012. IEEE.

[3] Raphaël Barazzutti, Pascal Felber, Christof Fetzer, Emanuel Onica, Jean-François Pineau, Marcelo Pasin, Etienne Rivière, Stefan Weigert. **StreamHub: a massively parallel architecture for high-performance content-based publish/subscribe. *Best Paper Award***. In *DEBS 2013: 7th ACM international conference on Distributed event-based systems*, pages 63–74 Arlington, TX, USA, July 2013. ACM.

[4] Raphaël P. Barazzutti, Pascal Felber, Emanuel Onica, Marcelo Pasin, Etienne Rivière. **StreamHub : Une architecture massivement parallèle pour un système pub/sub haute-performance**. In *ComPAS 2014: Conférence en Parallélisme, Architecture et Système*, Neuchâtel, Switzerland, April 2014.

[5] Raphaël Barazzutti, Thomas Heinze, Andre Martin, Emanuel Onica, Pascal Felber, Christof Fetzer, Zbigniew Jerzak, Marcelo Pasin, Etienne Riviere. **Elastic Scaling of a High-Throughput Content-Based Publish/Subscribe Engine**. In *ICDCS 2014: 33th Internation Conference on Distributed Computing Systems*, Madrid, Spain, July 2014, IEEE.

[6] Raphaël Barazzutti, Pascal Felber, Hugues Mercier, Emanuel Onica, Etienne Riviere. **Efficient and Confidentiality-Preserving Content-Based Publish/Subscribe with Prefiltering**. In *IEEE Transactions on Dependable and Secure Computing*, vol. 14 pages 308-325, June 2015, IEEE.

[7] Raphaël P. Barazzutti, Yaroslav Hayduk, Pascal Felber, Etienne Rivière. **Exploiting Concurrency in Domain-Specific Data Structures: A Concurrent Order Book and Workload Generator for Online Trading**. In *NETYS 2016: 4th International Conference on Networked Systems*, pages 16–31, Marrakech, Kingdom of Morocco, April 2016, Springer.

B

Bibliography

[1] Jerry Adler. Raging bulls: How wall street got addicted to light-speed trading. *Wired Magazine*, 20(9), 2012.

[2] Marcos K. Aguilera, Robert E. Strom, Daniel C. Sturman, Mark Astley, and Tushar D. Chandra. Matching events in a content-based subscription system. In *PODC*, 1999.

[3] Marcel Altherr, Martin Erzberger, and Silvano Maffeis. iBus-a software bus middleware for the Java platform. In *Proceedings of the International Workshop on Reliable Middleware Systems*, pages 43–53, 1999.

[4] Mehmet Altinel and Michael J. Franklin. Efficient filtering of XML documents for selective dissemination of information. In *VLDB*, 2000.

[5] Amazon Simple Notification Service. `https://aws.amazon.com/sns/`.

[6] Amazon Web Services Auto Scaling. `https://aws.amazon.com/autoscaling/`.

[7] Apache Hadoop Hedwig. `https://wiki.apache.org/hadoop/HedWig`.

[8] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, April 2010.

[9] Nathan Backman, Karthik Pattabiraman, Rodrigo Fonseca, and Ugur Cetintemel. C-MR: continuously executing MapReduce workflows on multi-core processors. In *MapReduce workshop*, 2012.

[10] Per Bak, Maya Paczuski, and Martin Shubik. Price variations in a stock market with many agents. *Physica A: Statistical Mechanics and its Applications*, 246 (3–4):430–453, 1997.

[11] Roberto Baldoni, Carlo Marchetti, Antonio Virgillito, and Roman Vitenberg. Content-based publish-subscribe over structured overlay networks. In *ICDCS*, June 2005.

[12] Roberto Baldoni, Roberto Beraldi, Leonardo Querzoni, and Antonino Virgillito. Efficient publish/subscribe through a self-organizing broker overlay and its application to SIENA. *the Computer Journal*, 2007.

[13] Raphaël Barazzutti, Pascal Felber, Hugues Mercier, Emanuel Onica, and Etienne Rivière. Thrifty privacy: Efficient support for privacy-preserving publish/subscribe. In *Proceedings of the 6th ACM International Conference on Distributed and Event-based Systems (DEBS)*, Berlin, Germany, July 2012. ACM.

[14] Raphaël Barazzutti, Pascal Felber, Christof Fetzer, Emanuel Onica, Marcelo Pasin, Jean-François Pineau, Etienne Rivière, and Stefan Weigert. StreamHub: A massively parallel architecture for high-performance content-based publish/subscribe. In *7th ACM International Conference on Distributed and Event-based Systems (DEBS)*, Arlington, TX, USA, July 2013.

[15] Marco Bartolozzi. Price variations in a stock market with many agents. *The European Physical Journal B*, 78(2):265–273, 2010.

[16] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The R*-tree: an efficient and robust access method for points and rectangles. In *SIGMOD*, 1990.

[17] Sukanya Bhowmik, Muhammad Adnan Tariq, Lobna Hegazy, and Kurt Rothermel. Hybrid content-based routing using network and application layer filtering.

In *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, pages 221–231. IEEE, 2016.

[18] Sukanya Bhowmik, Muhammad Adnan Tariq, Boris Koldehofe, Frank Durr, Thomas Kohler, and Kurt Rothermel. High performance publish/subscribe middleware in software-defined networks. *IEEE/ACM Transactions on Networking (TON)*, 25(3):1501–1516, 2017.

[19] Kenneth P. Birman and Thomas A. Joseph. Exploiting virtual synchrony in distributed systems. *SIGOPS Oper. Syst. Rev.*, 21(5):123–138, November 1987.

[20] Sven Bittner and Annika Hinze. The arbitrary boolean publish/subscribe model: making the case. In *DEBS*, 2007.

[21] Jean-Philippe Bouchaud, Marc Mézard, and Marc Potters. Statistical properties of stock order books: empirical results and models. *Quantitative finance*, 2(4), 2002.

[22] Andrey Brito, Andre Martin, Thomas Knauth, Stephan Creutz, Diogo Becker, Stefan Weigert, and Christof Fetzer. Scalable and low-latency data processing with StreamMapReduce. In *CloudCom*, 2011.

[23] Roberto Brunelli. *Template Matching Techniques in Computer Vision: Theory and Practice*. Wiley, 2009.

[24] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 15(4):412–447, 1997.

[25] Edouard Bugnion, Scott Devine, Mendel Rosenblum, Jeremy Sugerman, and Edward Y. Wang. Bringing virtualization to the x86 architecture with the original VMWare workstation. *ACM Transactions on Computer Systems (TOCS)*, 30(4):12, 2012.

[26] Antonio Carzaniga and Alexander L. Wolf. Forwarding in a content-based network. In *SIGCOMM*, 2003.

[27] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3):332–383, August 2001.

[28] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, and Antony I. T. Rowstron. Scribe: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in communications*, 20(8):1489–1499, 2002.

[29] Chee-Yong Chan, Pascal Felber, Minos Garofalakis, and Rajeev Rastogi. Efficient filtering of XML documents with XPath expressions. *VLDB Journal*, 11, 2002.

[30] Raphael Chand and Pascal Felber. Scalable distribution of XML content with XNet. *IEEE TPDS*, 19, 2008.

[31] Alex King Yeung Cheung and Hans-Arno Jacobsen. Load balancing content-based publish/subscribe systems. *ACM Trans. Comput. Syst.*, 28(4):9:1–9:55, December 2010.

[32] Alex King Yeung Cheung and Hans-Arno Jacobsen. Publisher placement algorithms in content-based publish/subscribe. In *ICDCS*, 2010.

[33] Alex King Yeung Cheung and Hans-Arno Jacobsen. Green resource allocation algorithms for publish/subscribe systems. In *ICDCS*, 2011.

[34] Sunoh Choi, Gabriel Ghinita, and Elisa Bertino. A privacy-enhancing content-based publish/subscribe system using scalar product preserving transformations. In *DEXA*, 2010.

[35] Edward G. Coffman Jr, Michael R. Garey, and David S. Johnson. Approximation algorithms for bin packing: A survey. In *Approximation algorithms for NP-hard problems*, pages 46–93. PWS Publishing Co., 1996.

[36] Rama Cont, Sasha Stoikov, and Rishi Talreja. A stochastic model for order book dynamics. *Journal of the American Statistical Association*, 58(3), 2010.

[37] Gianpaolo Cugola and Alessandro Margara. High-performance location-aware publish-subscribe on GPUs. In *Proceedings of the 13th International Middleware Conference*, pages 312–331. Springer-Verlag New York, Inc., 2012.

[38] Nigel Deakin. JSR 914: Java Message Service (JMS) API. *Sun Microsystems, Inc*, 2003.

[39] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. In *OSDI*, 2004.

[40] Bartholomäus Ende, Tim Uhle, and Moritz C. Weber. The impact of a millisecond: Measuring latency effects in securities trading. In *Wirtschaftsinformatik Proceedings (Paper 116)*, 2011.

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

[42] Françoise Fabret, Hans-Arno Jacobsen, François Llirbat, João Pereira, Kenneth A. Ross, and Dennis Shasha. Filtering algorithms and implementation for very fast publish/subscribe systems. In *SIGMOD*, 2001.

[43] Wenjing Fang, Beihong Jin, Biao Zhang, Yuwei Yang, and Ziyuan Qin. Design and evaluation of a pub/sub service in the cloud. In *CSC*, 2011.

[44] Amer Farroukh, Elias Ferzli, Naweed Tajuddin, and Hans-Arno Jacobsen. Parallel event processing for content-based publish/subscribe systems. In *DEBS*, 2009.

[45] Pascal Felber, Chee-Yong Chan, Minos Garofalakis, and Rajeev Rastogi. Scalable filtering of XML data for web services. *IEEE Internet Computing*, 7:49–57, 2003.

[46] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. In *SIGMOD*, 2013.

[47] Marcus Fontoura, Suhas Sadanandan, Jayavel Shanmugasundaram, Sergei Vassilvitski, Erik Vee, Srihari Venkatesan, and Jason Zien. Efficiently evaluating complex boolean expressions. In *SIGMOD*, 2010.

[48] Xavier Gabaix. Power laws in economics and finance. Technical report, National Bureau of Economic Research, 2008.

[49] Anshul Gandhi, Mor Harchol-Balter, Ram Raghunathan, and Michael A. Kozuch. Autoscale: Dynamic, robust capacity management for multi-tier data centers. *ACM Trans. Comput. Syst.*, 30(4):14:1–14:26, 2012.

[50] Julien Gascon-Samson, Franz-Philippe Garcia, Bettina Kemme, and Jörg Kienzle. Dynamoth: A scalable pub/sub middleware for latency-constrained applications in the cloud. In *2015 IEEE 35th International Conference on Distributed Computing Systems*, pages 486–496. IEEE, 2015.

[51] Dominique Guinard, Vlad Trifa, Friedemann Mattern, and Erik Wilde. From the Internet of things to the Web of things: Resource-oriented architecture and best practices. *Architecting the Internet of things*, pages 97–129, 2011.

[52] Vincenzo Gulisano, Ricardo Jimenez-Peris, Marta Patino-Martinez, Claudio Soriente, and Patrick Valduriez. Streamcloud: An elastic and scalable data streaming system. *IEEE TPDS*, 23(12), 2012.

[53] Abhishek Gupta, Ozgur D. Sahin, Divyakant Agrawal, and Amr El Abbadi. Meghdoot: content-based publish/subscribe over P2P networks. In *Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*, pages 254–273, New York, NY, USA, 2004. Springer-Verlag New York, Inc.

[54] Daniela Hernandez. Tech time warp of the week: Watch AT&T invent cloud computing in 1994. *Wired Magazine*, 2014.

[55] Joe Hoffert, Douglas C. Schmidt, and Aniruddha Gokhale. Adapting distributed real-time and embedded pub/sub middleware for cloud computing environments. In *Middleware*, 2010.

[56] Bo Hong and Viktor K. Prasanna. Distributed adaptive task allocation in heterogeneous computing environments to maximize throughput. In *IPDPS*, 2004.

[57] Jue Hong, Pavan Balaji, Gaojin Wen, Bibo Tu, Junming Yan, Chengzhong Xu, and Shengzhong Feng. Container-based job management for fair resource sharing. In *International Supercomputing Conference*, pages 290–301. Springer, 2013.

[58] Weibing Huang, Charles-Albert Lehalle, and Mathieu Rosenbaum. Simulating and analyzing order book data: The queue-reactive model. *Journal of the American Statistical Association*, 110(509), 2013.

[59] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: wait-free coordination for internet-scale systems. In *USENIX ATC*, 2010.

[60] Mihaela Ion, Giovanni Russello, and Bruno Crispo. Supporting publication and subscription confidentiality in pub/sub networks. In *SecureComm*, 2010.

[61] Mihalea Ion, Giovanni Russello, and Bruno Crispo. An implementation of event and filter confidentiality in pub/sub systems and its application to e-health. In *CCS*, 2010.

[62] Atsushi Ishii and Toyotaro Suzumura. Elastic stream computing with clouds. In *IEEE CLOUD*, 2011.

[63] Hans-Arno Jacobsen, Alex King Yeung Cheung, Guoli Li, Balasubramaneyam Maniymaran, Vinod Muthusamy, and Reza Sherafat Kazemzadeh. The PADRES publish/subscribe system. In *Handbook of Research on Adv. Dist. Event-Based Sys., Pub./Sub. and Message Filtering Tech.*, 2009.

[64] Kallapalayam R. Jayaram and Patrick Eugster. Split and subsume: Subscription normalization for effective content-based messaging. In *ICDCS*, 2011.

[65] Zbigniew Jerzak and Christof Fetzer. Bloom filter based routing for content-based publish/subscribe. In *DEBS*, 2008.

[66] Zbigniew Jerzak, Christof Fetzer, and Robert Wójcicki. Soft state in the XSiena publish/subscribe system. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, pages 37:1–37:2, New York, NY, USA, 2009. ACM.

[67] Yuhui Jin and Rob Strom. Relational subscription middleware for internet-scale publish-subscribe. In *Proceedings of the 2nd international workshop on Distributed Event-Based Systems*, pages 1–8. ACM, 2003.

[68] Satyen Kale, Elad Hazan, Fengyun Cao, and Jaswinder Pal Singh. Analysis and algorithms for content-based event matching. In *DEBS*, 2005.

[69] Reza Sherafat Kazemzadeh and Hans-Arno Jacobsen. Opportunistic multipath forwarding in content-based publish/subscribe overlays. In *Middleware*, 2012.

[70] Alec N. Kercheval and Yuan Zhang. Modelling high-frequency limit order book dynamics with support vector machines. *Quantitative Finance*, 15(8), 2015.

[71] Kunal Khanna, Michael Smith, David Wu, and Tony Zhang. Reconstructing the order book. Technical report, Stanford University, 2009.

[72] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy*, 2019.

[73] Jay Kreps, Neha Narkhede, and Jun Rao. Kafka: a distributed messaging system for log processing. In *NetDB*, 2011.

[74] Mascha Kurpicz, Anita Sobe, and Pascal Felber. Using power measurements as a basis for workload placement in heterogeneous multi-cloud environments. In *Proceedings of the 2nd International Workshop on CrossCloud Systems*, page 6. ACM, 2014.

[75] Hung Q. Le, Guy L. Guthrie, Derek Williams, Maged M. Michael, Brad G. Frey, William J. Starke, Cathy May, Rei Odaira, and Takuya Nakaike. Transactional memory support in the IBM Power8 processor. *IBM Journal of Research and Development*, 59(1):8–1, 2015.

[76] Doug Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley, Boston, MA, USA, 1996.

[77] Christian Leber, Benjamin Geib, and Heiner Litz. High frequency trading acceleration using FPGAs. In *Int. Conf. on Field Programmable Logic and Applications*, 2011.

[78] Ming Li, Fan Ye, Minkyong Kim, Han Chen, and Hui Lei. A scalable and elastic publish/subscribe service. In *IPDPS*, 2011.

[79] Wei Li, Songlin Hu, Jintao Li, and Hans-Arno Jacobsen. Community clustering for distributed publish/subscribe systems. In *CLUSTER*, 2012.

[80] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium*, 2018.

[81] Alessandro Margara and Gianpaolo Cugola. High performance content-based matching using gpus. In *Proceedings of the 5th ACM international conference on Distributed event-based system*, pages 183–194. ACM, 2011.

[82] Paul Marshall, Kate Keahey, and Tim Freeman. Elastic site: Using clouds to elastically extend site resources. In *CCGRID*, 2010.

[83] Silvano Martello and Paolo Toth. *Knapsack problems: algorithms and computer implementations*. John Wiley & Sons, Inc., 1990.

[84] Andre Martin, Christof Fetzer, and Andrey Brito. Active replication at (almost) no cost. In *SRDS*, 2011.

[85] Andre Martin, Thomas Knauth, Stephan Creutz, Diogo Becker, Stefan Weigert, Christof Fetzer, and Andrey Brito. Low-overhead fault tolerance for high-throughput data processing systems. In *ICDCS*, 2011.

[86] Sergei Maslov. Simple model of a limit order-driven market. *Physica A: Statistical Mechanics and its Applications*, 278(3):571–578, 2000.

[87] Sergei Maslov and Mark Mills. Price fluctuations from the order book perspective – empirical facts and a simple model. *Physica A: Statistical Mechanics and its Applications*, 299(1):234–246, 2001.

[88] Peter Mell and Timothy Grance. The NIST definition of cloud computing. *NIST special publication*, 800(145):7, 2011.

[89] Dirk Merkel. Docker: lightweight Linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2, 2014.

[90] J. Willard Milnor and George A. Randall. The Newfoundland-Azores high-speed duplex cable. *Transactions of the American Institute of Electrical Engineers*, 50 (2):389–396, June 1931.

[91] Jeffrey C. Mogul, Jayaram Mudigonda, Jose Renato Santos, and Yoshio Turner. The NIC is the hypervisor: Bare-metal guests in IaaS clouds. In *Presented as part of the 14th Workshop on Hot Topics in Operating Systems*, Santa Ana Pueblo, NM, 2013. USENIX.

[92] Leonardo Neumeyer, Bruce Robbins, Anish Nair, and Anand Kesari. S4: Distributed stream computing platform. In *KDCloud*, Sidney, Australia, 2010.

[93] Tobias Preis. *Ökonophysik - Die Physik des Finanzmarktes*. Springer, Wiesbaden, Germany, 2011.

[94] Costin Raiciu and David S. Rosenblum. Enabling confidentiality in content-based publish/subscribe infrastructures. In *Securecomm*, 2006.

[95] Costin Raiciu, David S. Rosenblum, and Mark Handley. Revisiting content-based publish/subscribe. In *ICDCS workshops*, 2006.

[96] Ravi Rajwar and Martin Dixon. Intel transactional synchronization extensions. In *Intel Developer Forum San Francisco*, volume 2012, 2012.

[97] John Reumann. Pub/Sub at Google. CANOE and EuroSys Summer School, 2009.

[98] Luigi Romano, Danilo De Mari, Zbigniew Jerzak, and Christof Fetzer. A novel approach to QoS monitoring in the cloud. In *CCP*, 2011.

[99] Ian Rose, Rohan Murty, Peter Pietzuch, Jonathan Ledlie, Mema Roussopoulos, and Matt Welsh. Cobra: Content based filtering and aggregation of blogs and RSS feeds. In *NSDI*, 2007.

[100] Mehran Sahami, Susan Dumais, David Heckerman, and Eric Horvitz. A bayesian approach to filtering junk e-mail. In *AAAI Workshop on Learning for Text Categorization*, 1998.

[101] Scott Schneider, Henrique Andrade, Bugra Gedik, Alain Biem, and Kun-Lung Wu. Elastic scaling of data parallel operators in stream processing. In *IPDPS*, 2009.

[102] Mark Seaborn and Thomas Dullien. Exploiting the DRAM rowhammer bug to gain kernel privileges. *Black Hat*, 15, 2015.

[103] Vinay Setty, Gunnar Kreitz, Roman Vitenberg, Maarten Van Steen, Guido Urdaneta, and Staffan Gimåker. The hidden pub/sub of spotify:(industry article). In *Proceedings of the 7th ACM international conference on Distributed event-based systems*, pages 231–240. ACM, 2013.

[104] Yogesh Shetty and Samir Jayaswal. The order-matching engine. In *Practical .NET for Financial Markets*, pages 41–103. Apress, 2006.

[105] Ankit Singla, Balakrishnan Chandrasekaran, P. Brighten Godfrey, and Bruce Maggs. The Internet at the speed of light. In *13th ACM Wshop on Hot Topics in Networks*, 2014.

[106] Juraj Somorovsky, Mario Heiderich, Meiko Jensen, Jörg Schwenk, Nils Gruschka, and Luigi Lo Iacono. All your clouds are belong to us: security analysis of cloud management interfaces. In *CCSW*, 2011.

[107] Storm: distributed and fault-tolerant realtime computation. `https://storm.apache.org/`.

[108] Daryll Strauss and David Bilkus (Wook). Linux helps bring Titanic to life. *Linux Journal*, 1998(46es):6, 1998.

[109] Ellen Terrell. History of the American and NASDAQ stock exchanges. In *Library of Congress–Business Reference Services*, 2010.

[110] Martin Thompson, Dave Farley, Michael Barker, Patricia Gee, and Andrew Stewart. Disruptor: High performance alternative to bounded queues for exchanging data between concurrent threads. *Technical paper. LMAX, May*, page 206, 2011.

[111] Nam-Luc Tran, Sabri Skhiri, and Esteban Zimányi. Eqs: An elastic and scalable message queue for the cloud. In *CLOUDCOM*, 2011.

[112] Yi-Min Wang, Lili Qiu, Dimitris Achlioptas, Gautam Das, Paul Larson, and Helen J. Wang. Subscription partitioning and routing in content-based publish/subscribe networks. In *DISC*, 2002.

[113] Yi-Min Wang, Lili Qiu, Chad Verbowski, Dimitris Achlioptas, Gautam Das, and Paul Larson. Summary-based routing for content-based event distribution networks. *SIGCOMM Comput. Commun. Rev.*, 34:59–74, October 2004.

[114] Tina Wong, Randy Katz, and Steven McCanne. An evaluation of preference clustering in large-scale multicast applications. In *INFOCOM*, 2000.

[115] Kun-Lung Wu, Kirsten W. Hildrum, Wei Fan, Philip S. Yu, Charu C. Aggarwal, David A. George, Buğra Gedik, Eric Bouillet, Xiaohui Gu, Gang Luo, and Haixun Wang. Challenges and experience in prototyping a multi-modal stream analytic and monitoring application on System S. In *VLDB*, 2007.

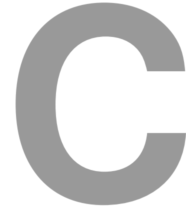
[116] Yahoo! Finance. `https://finance.yahoo.com/`.

[117] Richard M. Yoo, Christopher J. Hughes, Konrad Lai, and Ravi Rajwar. Performance evaluation of Intel transactional synchronization extensions for high-performance computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 19. ACM, 2013.

[118] Young Yoon, Vinod Muthusamy, and Hans-Arno Jacobsen. Foundations for highly available content-based publish/subscribe overlays. In *ICDCS*, 2011.

[119] Yaxiong Zhao and Jie Wu. Towards approximate event processing in a large-scale content-based network. In *ICDCS*, 2011.

[120] Shelley Q. Zhuang, Ben Y. Zhao, Anthony D. Joseph, Randy H. Katz, and John D. Kubiatowicz. Bayeux: An architecture for scalable and fault-tolerant wide-area data dissemination. In *Proceedings of the 11th international workshop on Network and operating systems support for digital audio and video*, pages 11–20. ACM, 2001.



Curriculum Vitae

Education

2019 **Université de Neuchâtel**
Ph.D. in Computer Science
Thesis: *"Scalable Content-Based Publish/Subscribe and Application to Online Trading"* supervised by Prof. Pascal Felber

2009 **École Polytechnique Fédérale de Lausanne**
M.Sc. Eng. in Computer Science
Thesis: *"Special halftoning techniques for printing with custom inks"* supervised by Prof. Roger D. Hersch

Professional experience

2016 – *now* **Haute École d'Ingénieurie et de Gestion de Vaud**
Lecturer and Supervisor
Responsible for the following theoretical and practical courses entitled "Object Oriented Programming" and "Distributed Architectures". Supervisor of several diploma works

2016 – *now* **Various companies**
Freelancer
Several projects with small, medium-sized enterprises, and startups. Consultancy in strategic choices related to IT. Software development and integration with existing software solutions

2011 – 2015 **Université de Neuchâtel**
Teaching Assistant
For the following courses:
– Imperative programming with C
– Object oriented programming with Java

2010 – 2013 **SRT15 Project**
Scientific Collaborator
European Research project driven by SAP, Yahoo!, Epsilon S.R.L, Université de Neuchâtel, and TU Dresden

2009 – 2010 **Swissquote Bank**
Development Engineer
Development of software retrieving financial instruments on several stock exchanges

Awards

2018 **Excellence in Teaching Award**
Haute École d'Ingénieurie et de Gestion de Vaud

2013 **Best Paper Award**
The International Conference on Distributed Event-Based Systems

Interests

professional: software development, large scale computing, concurrent computing, networks, computer security

personal: climbing/bouldering, ski alpinism, running, salsa, guitar, chess