UNIVERSIDAD POLITÉCNICA DE MADRID

Escuela Técnica Superior de Ingenieros Informáticos

Departamento de Lenguajes y Sistemas Informáticos
e Ingeniería de Software

A Novel Approach to Context-Aware Content-Based Middleware

Thesis submitted for the degree of Doctor of Philosophy

Advisor: Prof. Francisco Javier Soriano Camino

Co-Advisor: Prof. David Lizcano Casas

Author:

Sergio Vavassori

M.Sc. in Computer Science
and Engineering

2016

Departamento de Lenguajes y Sistemas Informáticos
e Ingeniería de Software

Escuela Técnica Superior de Ingenieros Informáticos

**A Novel Approach to Context-Aware
Content-Based Middleware**

Author: Sergio Vavassori
M.Sc. in Computer Science and Engineering

Advisor: Prof. Francisco Javier Soriano Camino
Ph.D. in Computer Science

Co-Advisor: Prof. David Lizcano Casas
Ph.D. in Computer Science

Tribunal nombrado por el Magnífico y Excelentísimo Señor Rector de la
Universidad Politécnica de Madrid el día de de 2016

Presidente	Juan Pazos Sierra
Vocal	Mariano Fernández López
Vocal	Juan Manuel Ares Casal
Vocal	Santiago Rodríguez Yáñez
Secretaria	Genoveva López Gómez
Suplente	Jaime Lloret Mauri
Suplente	Sandra Sendra Compte

Realizado el acto de defensa y lectura de la Tesis el día de de
2016 en la E.T.S.I. Informáticos

EL PRESIDENTE

LOS VOCALES

LA SECRETARIA

Acknowledgements

I would like to express my gratitude to all those people who directly, indirectly or unconsciously have pushed me to improve myself continuously.

Abstract

The application of pervasive computing is extending from field-specific to everyday use. The Internet of Things (IoT) is the shiniest example of its application and of its intrinsic complexity compared with classical application development.

The main characteristic that differentiates pervasive from other forms of computing lies in the use of contextual information. Some classical applications do not use any contextual information whatsoever. Others, on the other hand, use only part of the contextual information, which is integrated in an ad hoc fashion using an application-specific implementation. This information is handled in a one-off manner because of the difficulty of sharing context across applications. As a matter of fact, the application type determines what the contextual information is. For instance, for an imaging editor, the image is the information and its meta-data, like the time of the shot or camera settings, are the context, whereas, for a file-system application, the image, including its camera settings, is the information and the meta-data external to the file, like the modification date or the last accessed timestamps, constitute the context.

This means that contextual information is hard to share. A communication middleware that supports context decidedly eases application development in pervasive computing. However, the use of context should not be mandatory; otherwise, the communication middleware would be reduced to a context middleware and no longer be compatible with non-context-aware applications.

SilboPS, our implementation of content-based publish/subscribe inspired by SIENA [11, 9], solves this problem by adding two new elements to the paradigm: the *context* and the *context function*. Context represents the actual contextual information specific to the message to be sent or that needs to be notified to the subscriber, whereas the context function is evaluated using the publisher's context and the subscriber's context to decide whether the current message and context are useful for the subscriber. In this manner, context logic management is decoupled from context management, increasing the flexibility of communication and usage across different applications.

Since the default context is empty, context-aware and classical applications can use the same SilboPS, resolving the syntactic mismatch that there is between the two categories. In any case, the possible semantic mismatch is still present because it depends on how each application interprets the data, and it cannot be resolved by an agnostic third party.

The IoT environment introduces not only context but scaling challenges too. The number of sensors, the volume of the data that they produce and the number of applications that could be interested in harvesting such data are growing all the time. Today’s response to the above need is cloud computing. However, cloud computing applications need to be able to scale elastically [22].

Unfortunately there is no *slicing*, as distributed system primitives that support internal state partitioning [33] and hot swapping and current cloud systems like OpenStack or OpenNebula do not provide elastic monitoring out of the box. This means there is a two-sided problem: 1) how to scale an application elastically and 2) how to monitor the application and know when it should scale in or out.

E-SilboPS is the elastic version of SilboPS. It is the solution for the monitoring problem thanks to its content-based publish/subscribe nature and, unlike other solutions [5], it scales efficiently so as to meet workload demand without overprovisioning or underprovisioning. Additionally, it is based on a newly designed algorithm that shows how to add elasticity in an application with different state constraints: stateless, isolated stateful with external coordination and shared stateful with general coordination.

Its evaluation shows that it is able to achieve remarkable speedups where the network layer is the main limiting factor: the calculated efficiency (see Figure 5.8) shows how each configuration performs with respect to adjacent configurations. This provides insight into the actual trending of the whole system in order to predict if the next configuration would offset its cost against the resulting gain in notification throughput. Particular attention has been paid to the evaluation of same-cost deployments in order to find out which one is the best for the given workload demand.

Finally, the overhead introduced by the different configurations has been estimated to identify the primary limiting factor for throughput. This helps to determine the intrinsic sequential part and base overhead [26] of an optimal versus a suboptimal deployment. Depending on the type of workload, this can be as low as 10% in a local optimum or as high as 60% when an overprovisioned configuration is deployed for a given workload demand. This

Karp-Flatt metric estimation is important for system management because it indicates the direction (scale in or out) in which the deployment has to be changed in order to improve its performance instead of simply using a scale-out policy.

Resumen

La computación ubicua está extendiendo su aplicación desde entornos específicos hacia el uso cotidiano; el Internet de las cosas (IoT, en inglés) es el ejemplo más brillante de su aplicación y de la complejidad intrínseca que tiene, en comparación con el clásico desarrollo de aplicaciones.

La principal característica que diferencia la computación ubicua de los otros tipos está en como se emplea la información de contexto. Las aplicaciones clásicas no usan en absoluto la información de contexto o usan sólo una pequeña parte de ella, integrándola de una forma ad hoc con una implementación específica para la aplicación. La motivación de este tratamiento particular se tiene que buscar en la dificultad de compartir el contexto con otras aplicaciones. En realidad lo que es información de contexto depende del tipo de aplicación: por poner un ejemplo, para un editor de imágenes, la imagen es la información y sus metadatos, tales como la hora de grabación o los ajustes de la cámara, son el contexto, mientras que para el sistema de ficheros la imagen junto con los ajustes de cámara son la información, y el contexto es representado por los metadatos externos al fichero como la fecha de modificación o la de último acceso.

Esto significa que es difícil compartir la información de contexto, y la presencia de un middleware de comunicación que soporte el contexto de forma explícita simplifica el desarrollo de aplicaciones para computación ubicua. Al mismo tiempo el uso del contexto no tiene que ser obligatorio, porque si no se perdería la compatibilidad con las aplicaciones que no lo usan, convirtiendo así dicho middleware en un middleware de contexto.

SilboPS, que es nuestra implementación de un sistema publicador/subscriptor basado en contenido e inspirado en SIENA [11, 9], resuelve dicho problema extendiendo el paradigma con dos elementos: el *Contexto* y la *Función de Contexto*. El contexto representa la información contextual propiamente dicha del mensaje por enviar o aquella requerida por el subscriptor para recibir notificaciones, mientras la función de contexto se evalúa usando el contexto del publicador y del subscriptor. Esto permite desacoplar la lógica de gestión del contexto de aquella de la función de contexto, incrementando de esta forma la flexibilidad de la comunicación entre varias aplicaciones.

De hecho, al utilizar por defecto un contexto vacío, las aplicaciones clásicas y las que manejan el contexto pueden usar el mismo SilboPS, resolviendo de esta forma la incompatibilidad entre las dos categorías. En cualquier caso la posible incompatibilidad semántica sigue existiendo ya que depende de la interpretación que cada aplicación hace de los datos y no puede ser solucionada por una tercera parte agnóstica.

El entorno IoT conlleva retos no sólo de contexto, sino también de escalabilidad. La cantidad de sensores, el volumen de datos que producen y la cantidad de aplicaciones que podrían estar interesadas en manipular esos datos está en continuo aumento. Hoy en día la respuesta a esa necesidad es la computación en la nube, pero requiere que las aplicaciones sean no sólo capaces de escalar, sino de hacerlo de forma elástica [22].

Desgraciadamente no hay ninguna primitiva de sistema distribuido de *slicing* que soporte un particionamiento del estado interno [33] junto con un cambio en caliente, además de que los sistemas cloud actuales como OpenStack u OpenNebula no ofrecen directamente una monitorización elástica. Esto implica que hay un problema bilateral: cómo puede una aplicación escalar de forma elástica y cómo monitorizar esa aplicación para saber cuándo escalarla horizontalmente.

E-SilboPS es la versión elástica de SilboPS y se adapta perfectamente como solución para el problema de monitorización, gracias al paradigma publicador/subscriptor basado en contenido y, a diferencia de otras soluciones [5], permite escalar eficientemente, para cumplir con la carga de trabajo sin sobre-provisionar o sub-provisionar recursos. Además está basado en un algoritmo recientemente diseñado que muestra como añadir elasticidad a una aplicación con distintas restricciones sobre el estado: sin estado, estado aislado con coordinación externa y estado compartido con coordinación general.

Su evaluación enseña como se pueden conseguir notables speedups, siendo el nivel de red el principal factor limitante: de hecho la eficiencia calculada (ver Figura 5.8) demuestra cómo se comporta cada configuración en comparación con las adyacentes. Esto permite conocer la tendencia actual de todo el sistema, para saber si la siguiente configuración compensará el coste que tiene con la ganancia que lleva en el throughput de notificaciones. Se tiene que prestar especial atención en la evaluación de los despliegues con igual coste, para ver cuál es la mejor solución en relación a una carga de trabajo dada.

Como último análisis se ha estimado el overhead introducido por las distintas configuraciones a fin de identificar el principal factor limitante del throughput. Esto ayuda a determinar la parte secuencial y el overhead de base [26] en un despliegue óptimo en comparación con uno subóptimo. Efectivamente, según el tipo de carga de trabajo, la estimación puede ser tan baja como el 10 % para un óptimo local o tan alta como el 60 %: esto ocurre cuando se despliega una configuración sobredimensionada para la carga de trabajo. Esta estimación de la métrica de Karp-Flatt es importante para el sistema de gestión porque le permite conocer en que dirección (ampliar o reducir) es necesario cambiar el despliegue para mejorar sus prestaciones, en lugar que usar simplemente una política de ampliación.

Contents

1	Introduction	1
1.1	Statement of the problem	1
1.2	Definitions of terms	2
1.3	Description of the remaining chapters	5
1.4	Derived publications	6
2	State of the Art	7
2.1	Distributed Event-Based Systems	7
2.1.1	Topic-based publish/subscribe systems	10
2.1.2	Content-based publish/subscribe systems	11
2.1.3	Record-based publish/subscribe systems	11
2.2	Message format	12
2.2.1	XML	13
2.2.2	Key-Value	16
2.2.3	Structureless	17
3	Goals and Contributions	19
3.1	Objectives	19
3.1.1	Conceptual	19
3.1.2	Technical	20
3.2	Hypothesis and contribution	21
3.2.1	Limitations	23
4	Managing Context Explicitly: SilboPS	25
4.1	Shortcomings of implicit context management	25
4.2	Proposed design principles	27
4.3	Proof of concept	29
4.3.1	Adding publishers with changing context	30
4.3.2	Adding subscribers with changing contexts	31
4.3.3	Notification delivery	33
4.3.4	Subscriber context update	34
4.3.5	Publisher context update	34
4.4	Prototype Evaluation	35

5	Bringing it to the cloud: Elastic SilboPS	43
5.1	Concept	43
5.2	The Algorithm	44
5.2.1	Access Point Scaling	44
5.2.2	Matcher Scaling	45
5.2.3	Exit Point Scaling	49
5.3	Evaluation	51
6	Conclusions	61
6.1	About Context-Awareness	61
6.2	About Elasticity	62
6.3	Future Work	62
6.3.1	Cloud development	64
	Bibliography	66
	Appendix A Experiments' Raw Data Sets	77
A.1	Context-Aware performance tests for SilboPS	77
A.1.1	Notification matching time for SIENA	77
A.1.2	Context change time for a single broker	86
A.1.3	Network test	106
A.2	Elastic performance tests for E-SilboPS	107
A.2.1	Throughput tests	107
A.2.2	Speedup test	116
A.2.3	Scale-out test	120

List of Figures

2.1	Composite Event Detection (CED)	8
2.2	Cambridge Event Architecture for publish/subscribe systems	9
4.1	Exchange of messages with implicit context management	26
4.2	Exchange of messages with explicit context management	28
4.3	Scenario broker network	29
4.4	Adding a new publisher to the scenario	30
4.5	Adding a new subscriber to the scenario	31
4.6	Context change of a subscriber	34
4.7	Normalized time for 50 context changes	36
4.8	Context change time with 10 k and 100 k subscriptions	37
4.9	Single context change time with increasing connection	38
4.10	Context change time in a broker network	39
4.11	Notification throughput with 10 k and 80 subscriptions	41
5.1	Messages exchange during Matcher layer scale-out	47
5.2	Exit Point scale-out schema	49
5.3	Proof of the consistent cut of notifications at Matcher layer	50
5.4	Notification throughput with elastic deploy	52
5.5	Notification throughput speedup dominance	53
5.6	Notification throughput speedup	54
5.7	Notification throughput speedup sliced on 1- x -2 configurations	55
5.8	Notification throughput speedup efficiency	56
5.9	Karp-Flatt estimation of e	57
5.10	Notification throughput during scale-out	58
6.1	Adding a new publisher to the scenario	63
6.2	Clients change their context	64

List of Tables

4.1	Numerical result of test with 100 k subscriptions	40
-----	---	----

List of Listings

2.1	Publication in type-based Publish/Subscribe	12
2.2	Subscription in type-based Publish/Subscribe	12
2.3	Subscriber sending one subscription to a specific node	13
2.4	System reply to subscriber's subscription	13
2.5	Publisher publishes a new weblog entry	14
2.6	Service notifies subscribers	15
2.7	Google Cloud Pub/Sub notification request	16
2.8	Google Cloud Pub/Sub notification response	16
2.9	Google Cloud Pub/Sub subscription request	17
2.10	Google Cloud Pub/Sub notification delivery	17
A.1	SIENA notification throughput	77
A.2	SilboPS notification throughput	81
A.3	SIENA single broker 50 context changes	86
A.4	SilboPS single broker 50 context changes	89
A.5	SIENA vs SilboPS context change with up to 100 k subs	92
A.6	SilboPS broker context change with up to 100 k subs	94
A.7	SIENA vs SilboPS context change time in a broker network	106
A.8	Notification throughput for 1-1-1 deploy	107
A.9	Notification throughput for 1-2-1 deploy	108
A.10	Notification throughput for 1-2-2 deploy	108
A.11	Notification throughput for 1-3-1 deploy	109
A.12	Notification throughput for 1-4-1 deploy	110
A.13	Notification throughput for 1-4-2 deploy	111
A.14	Notification throughput for 1-5-1 deploy	112
A.15	Notification throughput for 1-5-2 deploy	113
A.16	Notification throughput for 1-6-1 deploy	114
A.17	Notification throughput for 1-6-2 deploy	115
A.18	Notification Speedup and Karp-Flatt metric	116
A.19	Throughput during scale-out	120

Introduction

*It doesn't matter how good you are;
you can't get an API right until you've tried to code to it.*

— Joshua Bloch

1.1 Statement of the problem

As a consequence of the wider availability of broadband connections, the adoption of distributed event-based systems has increased over the last decade, leading to the provision of more and more services built on top of others. Since these systems are intrinsically loosely coupled, however, the provided API is based on message passing.

Message passing places greater demands on developers than classical method invocation, which is a well-known and well-understood mechanism for transferring control and data within a program [7], because not only does it have to reason about the static structure but it also has to deal with the time dimension of the messages. This means that concurrency and race conditions have to be taken in account during development. This ultimately leads to the creation of middleware. Middleware is software that offers local invocation semantics for each node of a distributed system and hides not only the network but even the distributedness of other components.

Middleware technology is now used on the same computer to hide differences between components for use without tight integration. An example of this is the use of middleware as a bus for the graphical system or between devices. However, the adoption of middleware as a basis of communication between components challenged the technology to move from tens or hundreds to millions of components, bringing with it the problem of middleware scalability.

The recent advent of cloud computing expanded the application field of middleware and added the need for an automated means to cost-effectively handle the variability of the workload to the above scalability demand. Nonetheless, middleware technology continues to serve its main purpose as a communication layer, represented primarily by monitoring nodes and components.

Over the last few years, a multitude of systems have been created to satisfy the above demand, but most have been confined to academia and have not been used in an enterprise environment.

Nevertheless, the gap between professional and academic use has narrowed within cloud computing and IoT environments, since professionals have need of the tools built by academia to efficiently and cost-effectively harness their investment in hardware, software and data, whereas academia has need of the professionals to provide the field in which to experiment with technology.

This is the main reason of why distributed event-based systems, and in particular publish/subscribe systems, have experienced such growth in recent years.

1.2 Definitions of terms

This thesis presents the latest evolutions in the field of distributed event-based systems. However, this class of systems has some properties [13, 45] that are intrinsic to their distributed nature, like:

Loose coupling

each component has little or no knowledge about the other separate components.

Concurrency

in a centralized system, concurrency is a design choice, whereas, in distributed systems, it is a fact of life that has to be dealt with because there is no shared memory or physical clock.

Absence of global clock

there are many clocks which are not necessarily synchronized. Independent failures: communication is the first thing that fails but any component can fail independently of the others. This complicates failure detection because there is no yes/no response, and sometimes the failure is simply impossible to identify.

Heterogeneity

there are several systems, built by different people at different times, that have to be dealt with.

Openness

a distributed system is intrinsically open because of the unknown number or type of components that will use the system at design time.

Security

malicious components can be attached to the system because of openness.

Scalability

increased workload leads to a smoother performance drop than in a centralized system because of the different component saturation rates within the distributed system.

Failure handling

unreliable links and host failure are indistinguishable, and no one machine has all the information about the whole system state, making failure handling something that should be taken into account as of design time.

Transparency

the details are hidden to simplify programmer interaction; this can lead to inefficiency or unintuitive behavior.

These properties change the classical working/failed model of a centralized system with the following three types of failures:

Omission channels drop messages or process crashes.

Byzantine processes omit, alter or add processing steps; channels deliver corrupted, manipulated, non-existent messages or simply deliver messages multiple times.

Timing the time limits for a synchronous system are violated.

The focus will be on a subcategory of distributed event-based systems, that is, publish/subscribe systems (P/S) that implement the distributed version of the observer pattern.

These systems have loose coupling within space, time and synchronization. This favors scalability and failure recovery but, at the same time, makes it harder to synchronize applications or provide the same level of guarantees as synchronous systems. Luckily, cloud and IoT applications are

already distributed environments. This means that the applications that run in these environments already take into account the above properties and failure model. Nonetheless, there is still a compelling need for a *general* and *uniform* communication system with higher level semantics than sockets.

In this kind of systems, the client part is divided in two categories: *publishers*, which send *notifications*, and *subscribers*, which send *subscriptions*. *Notifications* are messages containing the actual information to be delivered, whereas *subscriptions* are filters used to receive *notifications* published by *publishers*. According to the kind of P/S system, the actual implementation of this concept, as well as its management differs.

The primary example of this need is monitoring: a common layer that is capable of exchanging information at different stack levels, such as virtual machine (VM) parameters like CPU or RAM consumption or service metrics like response time, waiting time, throughput or usage up to a general cloud-wide view, is the foundation for managing complexity. On the other hand, the IoT environment has slightly different requirements since it focuses on the *openness* and *flexibility* of the system and has to interact with heterogeneous devices and with changing topology. Nevertheless, scalability is still a top-ranking feature because of the number of sensors involved. In addition to the *reading* capability, recent trends show how *writing* capability is increasingly required since embedded devices have evolved from simple sensors to actuators. Likewise, the application of changes to the VM or the settings of a set of services in a distributed and groupwise manner is a requirement for cloud computing. This brings with it a whole new dimension with regard to how to target a single element of a group, while at the same time having to perform operations on a potentially massive number of elements in a simple way.

Publish/Subscribe systems have proven capable of considerable scaling, but the *elasticity* to operate in a cloud environment was missing until recently. It is shown that *dynamic state partitioning* is a requirement for proper *elasticity* [22]. Besides, contextual information handling is of primary importance, especially in IoT environments, since it adds semantic power without increasing verbosity. This brings it closer to the classical human communication style. On top of all this, the same middleware must be easy to administer and possibly automated for most tasks.

It is shown that our system, named **SilboPS**¹, and its cloud evolution **E-SilboPS** is a potential reference model for future development.

1.3 Description of the remaining chapters

The thesis is structured as follows:

Chapter 2

introduces the state of the art with respect to current distributed event-based systems, showing the different types that exist and their intrinsic and extrinsic characteristics viewed with regard to both architecture and usage trends. Additionally, the specific limitations of each category are detailed to give an understanding of which is a technological limitation and which is a design limitation.

Chapter 3

shows the objectives and the contribution of this thesis to the current state of the art from both a conceptual and a technical point of view. Specifically, an explanation is given of the *slicing as a distributed primitive* concept, as well as of the extent to which it can be achieved and subject to which limitations. Also, an example of what is required to remove the above limitations is given.

Chapter 4

details *context-awareness*, stating which model has been used and its semantics and reporting its evaluation and performance using data from the published experiments.

Chapter 5

addresses how to move from a generic *content-based publish/subscribe* system to a cloud-enabled system: the objective is to add *elasticity*, but this can only be done using dynamic state partitioning, without which no proper elasticity is achieved. It shows how the limitations of the model presented in Chapter 4 can be resolved and which new limitations emerge. It reports a comparison with the most interesting

¹**SilboPS** is the contraction of *Silbo Publish/Subscribe*, named after the Gómeran whistle, also known as *el silbo* in Spanish. It is a whistled language that is used by the inhabitants of the island of La Gomera to communicate across valleys or ravines. It has a range of up to 5 km [8] and is used for public communication due to its loud nature. These are the characteristics that led to the use of its name for this publish/subscribe system

architectures, highlighting the differences and stressing the implications for their employment. Finally, the model is described in detail, together with a proof of correctness of the algorithm used to scale the system in or out. Finally, an evaluation of the performance is reported to qualitatively and quantitatively show the behavior of the system.

Chapter 6

summarizes the remaining architectural limitations and lists the open questions for future research on the topic.

1.4 Derived publications

Some of the research on which this thesis is based was published in the following co-authored articles:

Conferences

- David Lizcano, Sebastian Ortega, Javier Soriano, and Sergio Vavassori. Explicitly context-aware publish/subscribe with context-invariant subscriptions. In *Proceedings of the 13th International Conference on Information Integration and Web-based Applications and Services*, iiWAS '11, pages 367–370, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0784-0. doi: 10.1145/2095536.2095606. URL <http://doi.acm.org/10.1145/2095536.2095606>

Journals

- Sergio Vavassori, Javier Soriano, David Lizcano, and Miguel Jiménez. Explicit context matching in content-based publish/subscribe systems. *Sensors*, 13(3):2945, 2013. ISSN 1424-8220. doi: 10.3390/s130302945. URL <http://www.mdpi.com/1424-8220/13/3/2945>

State of the Art

*For a successful technology, reality must take precedence
over public relations, for Nature cannot be fooled.*

— Richard Feynman

2.1 Distributed Event-Based Systems

Distributed event-based systems have evolved from the traditional message passing to a more abstracted form in which the semantic of the message has changed from a simple self-contained information to a more structured version. As a matter of fact the first kind of distributed event-based systems were those that used sockets to connect to each others. However, in those systems, the messaging using sockets was only a way to reach other computers but it did not represent a direct paradigm to programming application [17].

With the spread of wide band access, there has been an increase in the necessity to reach a growing number of users in a more scalable way. In the past years radio and television technology allowed this kind of wide audience but in the Internet was still missing a system that allowed one-to-many massive decoupled communication. In centralized applications the *observer pattern* allowed this kind of decoupling but it was limited to a single computer; bringing this paradigm to the distributed environment would have allowed the kind of scaling required, like in sensors reading or other IoT environments or in more classical applications like *content delivery networks* (CDN).

The distributed implementation of the observer pattern is what is called *publish/subscribe* systems. This family has the same characteristics of the centralized pattern, that is *decoupling* in *space* and *synchronization* but it adds the *time* dimension to this decoupling. In addition it allows to

Figure 2.1: Composite Event Detection (CED)

anonymize the production and the consumption of the message, so producers can be effectively hidden from consumers and vice-versa, and to be more flexible than the *request-reply* paradigm allowing asynchronous processing. Another interesting feature of these types of system is the possibility to aggregate low level messages as higher level events: this is called *Composite Event Detection* [25, 43] and it allows to use the same system as communication middleware, that means a single communication layer can support various applications that have dependency in the semantic of the messages they exchanges as shown in Figure 2.1.

Today's publish/subscribe systems can be differentiated in three main categories: *topic-based*, *record based* and *content-based* but all of them shares some basic concept about clients' nomenclature and functionality.

Before starting explaining the characteristics of the three kind of publish/subscribe systems it is useful to list the type of entities theses system have; this is the *Cambridge Event Architecture* (CEA) that is an extension of the object-oriented middleware suitable for applications like multimedia presentation control, pervasive environments, tracking mobile entities [38].

Publisher

advertise its message publication and publish messages called *notifications* in the system through the *Broker*.

Subscriber

receives all kind of *advertise* from the *Broker* and *subscribes* to the *Broker* to receive published *notifications*.

Broker

handles connection from *publishers* and *subscribers* allowing anonymisation and according to its internal model, sends notifications received from *publishers* to matching *subscribers*.

Figure 2.2: Cambridge Event Architecture for publish/subscribe systems

Advertise

in some systems it is an optional message that contains a general description of the notification the connected publisher will send. In those systems that requires this message, sending a wrong version blocks notification delivery but not system's functionality.

Notification

it is the message that transport the actual information. Publishers simply send this message to the connected broker without knowing how many clients will receive it or where they are; this imply that possibly no clients will receive the message and it will be discarded by the broker.

Subscription

it is the message with whom subscribers express their interest to the broker in receiving notification messages that comply with some kinds of requirements. It can be thought like a pass-filter applied to a notification stream.

As can be seen in Figure 2.2 publishers only sends two types of messages, advertises if needed and notifications but didn't receive any, while subscribers sends one type of message, namely subscriptions and they receive two type of messages: advertise, if required, and notifications. According to the type of publish/subscribe system, advertise and subscription messages could be avoided since they are implicitly defined by the broker.

2.1.1 Topic-based publish/subscribe systems

Topic-based publish/subscribe, sometimes known as *Channel-based publish/subscribe* [24] or *Subject-based P/S* [1, 18, 10] depending if the *herarchical* disposition is viewed as a differentiating characteristics or not, is the simplest distributed form of P/S. It has *topics* that are essentially logical named channels into which publishers writes their advertises and notifications. From the subscriber's point of view, the broker could require them to send their subscriptions to receive the messages or simply deliver every message that is written on that topic. In the case that subscribers must send subscriptions to receive alerts, these subscriptions are limited to filter on messages' meta-data or simple standard fields, without the possibility to apply complex operators. In order to avoid routing algorithm, this type of filtering is usually done by the brokers that have subscribers connected to them. Being the model so simple, it have the best throughput delivery performance compared to the other models, because it does not require complex matching, however network utilization is pretty high since the filtering is done far from the publishers.

Another drawback of this model is that publishers and subscribers have to reach an agreement on which topic connect to publish, and thus receive, notifications. This can lead to an explosion of topics if each one of them is strictly selective because the possible combination of topics that can be compatible with a notification increase exponentially with respect to notification's content. To solve the explosion of topics, the *herarchical* approach has been proposed [10] but it helped on reducing the number only when topics were super-sets or partitions one of the others.

Moreover publishers and subscribers have the burden to select the correct topic and this could be very cumbersome if topic creation is dynamic. At the same time, dynamic topic creation is requested in a real-world scenario since, except in very few environments, a defined set of topic cannot cover all application's needs; on the other hand, *automatically* create topics is a very complex task, both on the side of naming and on the side of notify publishers and subscribers about the new topic, something that normally is done out-of-band. There have been some proposal to reduce the impact of these drawback, mainly on the network utilization side, as in [37, 49], but a fundamental limitation of this paradigm is the explicit division of the event space on a single dimension; that means adding more dimensions will imply a combinatorial explosion of the number of topics, since by definition are orthogonally to each others.

2.1.2 Content-based publish/subscribe systems

Content-based P/S address the aforementioned dimension problem of topic-based P/S, avoiding the *explicit addressing* present in the topic-based by using *dynamic delivery tree* created by the broker using clients subscriptions. This removes the burden from publishers and subscribers to choose which topics to connect to and simplify to receive messages from publishers without making them directly aware of the interest of clients. According to the topology structure of the delivery network, advertise messages can be required.

As a matter of fact, using advertises in a broker's network allows to perform a smarter routing of the subscriptions instead of broadcasting to all neighbors [11, 9] and at the same time reduces the amount of work each broker has to do to match incoming notifications; however it requires publishers send the advertise before the relative notifications. One possible workaround is to let the broker to generate automatically the needed advertises, and to do that it has two main strategies: be as much specific as possible to let subscribers be aware of the new type of message or be as general as possible. The first strategy is the most computational intensive and it loads the broker possibly for details in the message that are not really important while the second is the lighter but can lead to a too-generic advertise that is no more useful from subscribers' point of view.

This type of P/S has a greater computational cost compared to the topic-based but the simplicity of its usage and the benefits of not having to choose or manager which topic has to be created are worthwhile the added cost.

2.1.3 Record-based publish/subscribe systems

Record-based P/S [38], also known as *type-based publish/subscribe* [16], uses a message structure that describes the type of object to be retrieved, that means instead of focusing on lower level, they offer directly the object abstraction to the subscriber.

It improves the semantics of subscriptions that can be sent, since it allows to be type-specific and introduces type-safety directly in the P/S system; however the performance penalties that it experience rarely are worth the added type-safety. This is due because the type checking can be viewed as additional constraints on the attributes that are performed automatically, but the lack of optimization lead to missing the additional value, comparing to the content-based P/S in the case that attribute's names does not collide.

Listing 2.1 and Listing 2.2 show respectively how a notification can be published and the construction of a subscription in type-based Publish/Subscribe. It is worth note that these kind of systems are not restricted to a specific set of types, but can handle natively every user-defined types and apply operators even to collections.

```
1 Adapter<StockQuote> quoteAdapter = new Adapter<StockQuote>();
2 StockQuote quote = new StockQuote("TelcoOperators", 100, 90.0);
3 quoteAdapter.publish(quote);
```

Listing 2.1: Publication in type-based Publish/Subscribe

```
1 Adapter<StockQuote> quoteAdapter = new Adapter<StockQuote>();
2 Subscription<StockQuote> quoteSubscription = quoteAdapter
3                                     .subscribe(
4     new Subscriber<StockQuote>() {
5
6         public void notify(StockQuote quote) {
7
8             System.out.println("Got offer: " + quote.getPrice());
9         }
10    }
11 );
12
13 Stockquote formalQuote = quoteSubscription.constrain();
14 formalQuote.getCompany().equals("Telco");
15 formalQuote.cheaperThan(100.0);
16 quoteSubscription.activate();
```

Listing 2.2: Subscription in type-based Publish/Subscribe

2.2 Message format

Orthogonal to the model, there is the message format that creates other categories in which publish/subscribe systems can be classified: XML, Key-Value and Structureless.

2.2.1 XML

Notifications and advertises are sent as XML messages while subscriptions could be expresses in XML or as XPATH queries [23, 39]. Advertise messages can carry XML Schema Definitions (XSD) to help subscribers to create more specific subscriptions. This type of message format allows subscribers to have a lot of semantic expressiveness but it is paid on broker's complexity and delivery performance [46, 48].

The following listings shown show the publish/subscribe extension of XMPP protocol¹ [41, 42]. In detail Listing 2.3 shows the subscription created by a subscriber, targeting a specific node and Listing 2.4 shows the system response to confirm that the sent subscription has been accepted. When a publisher sends a notification, like Listing 2.5, the subscribers that have sent subscriptions matching the notification, will receive it, as shown in Listing 2.6.

```
1 <iq type="set" from="francisco@denmark.lit/barracks"
2   to="pubsub.shakespeare.lit" id="sub1">
3   <pubsub xmlns="http://jabber.org/protocol/pubsub">
4     <subscribe node="princely_musings"
5               jid="francisco@denmark.lit"/>
6   </pubsub>
7 </iq>
```

Listing 2.3: Subscriber sending one subscription to a specific node

```
1 <iq type="result" id="sub1" from="pubsub.shakespeare.lit"
2   to="francisco@denmark.lit/barracks">
3   <pubsub xmlns="http://jabber.org/protocol/pubsub">
4     <subscription node="princely_musings"
5                  jid="francisco@denmark.lit"
6                  subid="ba49252aaa4f5d320c24d3766f0bdcade78c78d3"
7                  subscription="subscribed"/>
8   </pubsub>
9 </iq>
```

Listing 2.4: System reply to subscriber's subscription

¹<http://www.xmpp.org/extensions/xep-0060.html>

```
1 <iq type="set" id="publish1" from="hamlet@denmark.lit/blogbot"
2   to="pubsub.shakespeare.lit">
3   <pubsub xmlns="http://jabber.org/protocol/pubsub">
4   <publish node="princely_musings">
5     <item id="bnd81g37d61f49fgn581">
6       <entry xmlns="http://www.w3.org/2005/Atom">
7         <title>Soliloquy</title>
8         <summary>
9           To be, or not to be: that is the question:
10          Whether 'tis nobler in the mind to suffer
11          The slings and arrows of outrageous fortune,
12          Or to take arms against a sea of troubles,
13          And by opposing end them?
14        </summary>
15        <link rel="alternate" type="text/html"
16          href="http://denmark.lit/2003/12/13/atom03"/>
17        <id>tag:denmark.lit,2003:entry-32397</id>
18        <published>2003-12-13T18:30:02Z</published>
19        <updated>2003-12-13T18:30:02Z</updated>
20      </entry>
21    </item>
22  </publish>
23 </pubsub>
24 </iq>
```

Listing 2.5: Publisher publishes a new weblog entry

```
1 <message from="pubsub.shakespeare.lit" to="francisco@denmark.lit" id="foo">
2   <event xmlns="http://jabber.org/protocol/pubsub#event">
3     <items node="princely_musings">
4       <item id="ae890ac52d0df67ed7cfd51b644e901">
5         [ ... ENTRY ... ]
6       </item>
7     </items>
8   </event>
9 </message>
10
11 <message from="pubsub.shakespeare.lit" to="bernardo@denmark.lit" id="bar">
12   <event xmlns="http://jabber.org/protocol/pubsub#event">
13     <items node="princely_musings">
14       <item id="ae890ac52d0df67ed7cfd51b644e901">
15         [ ... ENTRY ... ]
16       </item>
17     </items>
18   </event>
19 </message>
20
21 <message from="pubsub.shakespeare.lit" to="horatio@denmark.lit" id="baz">
22   <event xmlns="http://jabber.org/protocol/pubsub#event">
23     <items node="princely_musings">
24       <item id="ae890ac52d0df67ed7cfd51b644e901">
25         [ ... ENTRY ... ]
26       </item>
27     </items>
28   </event>
29 </message>
30
31 <message from="pubsub.shakespeare.lit" to="bard@shakespeare.lit" id="fez">
32   <event xmlns="http://jabber.org/protocol/pubsub#event">
33     <items node="princely_musings">
34       <item id="ae890ac52d0df67ed7cfd51b644e901">
35         [ ... ENTRY ... ]
36       </item>
37     </items>
38   </event>
39 </message>
```

Listing 2.6: The publish/subscribe service notifies subscribers about the new blog entry

2.2.2 Key-Value

Messages are based on key-value list, advertises can be simply a list of values, like attribute name, or a more complex list of attribute names as key and types as value. Notifications are typically a list of attribute name as key and attribute content as value, while subscriptions are, in their simpler form, a list of attribute names but they can be even attribute names, values and an operator over the values. Here below there are some JSON extracted from the Google Cloud Pub/Sub:² Listing 2.7 shows how a publisher can send a notification with the message Hello Cloud Pub/Sub! Here is my message! to the topic `iana.org/language_tag` and Listing 2.8 report the server response. In the same way subscribers can send subscription, as shown by Listing 2.9.

```
1 {
2     "messages": [{
3         "attributes": [{
4             "key": "iana.org/language_tag",
5             "value": "en"
6         }],
7         "data": "SGVsbG8gQ2xvdWQgUHVlL1N1YiEgSGVyZSBpcyBteSBtZXNzYWdlIQ=="
8     }]
9 }
```

Listing 2.7: Google Cloud Pub/Sub notification request

```
1 {
2     "messageIds": [
3         "19916711285"
4     ]
5 }
```

Listing 2.8: Google Cloud Pub/Sub notification response

²<https://cloud.google.com/pubsub/overview>

```
1 {
2     "topic": "projects/someproject/topics/sometopic",
3     "pushConfig": {
4         "pushEndpoint": "https://myproject.appspot.com/myhandler"
5     }
6 }
```

Listing 2.9: Google Cloud Pub/Sub subscription request

```
1 {
2     "message": {
3         "attributes": {
4             "key": "value"
5         },
6         "data": "SGVsbG8gQ2xvdWQgUHVlL1N1YiEgSGVyZSBpcyBteSBtZXNzYWdlIQ==",
7         "message_id": "136969346945"
8     },
9     "subscription": "projects/myproject/subscriptions/mysubscription"
10 }
```

Listing 2.10: Google Cloud Pub/Sub notification delivery

2.2.3 Structureless

The content of the message does not have any particular structure but subscriptions have the capacity to apply operators (like `contains()`, etc.) on it. The content is typically represented as a `String` since it allows an easier matching without requiring Turing-complete power for subscriptions.

Goals and Contributions

Simplicity and elegance are unpopular because they require hard work and discipline to achieve and education to be appreciated.

— Edsger Dijkstra

3.1 Objectives

This thesis aims to show how a context-aware middleware can improve not only the performances of the applications that uses but even to resolve the syntactic mismatch for context handling and exchanging between environments. In addition, the proposed solution is first presented as a standalone application and then is explained how it has been moved to a cloud environment, in such a way to scale according to the workload and using elasticity to precisely provision the needed computing power.

A precise description of the scaling algorithm is detailed as well as the reason of its design to help understand possible hidden problems and unintuitive shortcomings. Then the *slice as a distributed system primitive* concept is introduced to show how it could ease porting applications to the cloud environment and the common problems that have to be resolved.

3.1.1 Conceptual

The first problem to resolve is how to model the context in a middleware in such a way that it can be used by different applications in a similar manner and the same context can be exchanged without incurring in syntactic problems. This thesis will show that using two different object to model the context, namely the *ContextFunction* and the *Context* allows to do context matching with a reasonable performances and a minimal number of messages.

A key concept to understand why this distinction in two different object is necessary, is about how to manage the context in the system. Classical applications have an ad-hoc form to model the context and implies that exchanging between applications requires some kind of import/export filters. In addition modeling the context with only one component doesn't allow to differentiate the *use* of the context from its value.

As a matter of fact what is message's context for an application could be interpreted as the very message, so there is the need to store the *value* of the context in an explicitly separated structure. In addition the *use* of this value can be different according to the current state or to the current application's semantics: this is the reason to have a *ContextFunction* object, so to be able to map different logic operating on the same context.

The second problem is how to add *elasticity* to a general application. Since cloud environment is focused on *horizontal* scalability, a dynamic repartitioning of application state is required, otherwise it should be done on an *a priori* base leading to wasted resources and major overhead when the workload demand is low and possibly limiting the maximum scalability not because of hardware or architecture limitations but due to the incorrect deploy forecast.

However *dynamically* repartition the application state cannot be done in an application-agnostic manner since, depending of the current application state and architecture, cloud be required some coordination between its components to ensure a consistent final application state. To resolve these limitations a case-by-case solution will be detailed together with the explanation of the proposed algorithm.

3.1.2 Technical

Conversely of the proof of concept that have the only goal to prove the feasibility of the theory, this work has been implemented and tested to provide the required validation; however its scope is not limited to just verify the correctness of our hypothesis but to provide a reference implementation for future development. That means, during the whole development of the application, the design and the implementation has been driven by the final goal of have a first prototype that can be evolve into a final product.

Unfortunately, the only test that really matters is the production environment, so to offer a wider understanding of how the selected technologies can affect the performances and how much they can be improved changing a component is still to be completed. In the following chapters will be detailed the uses cases that drive the development the solution but depending on the environment in which they will be used it could be that some adjustment will be needed.

As an example, in the elastic implementation build, the design choice to use classical sockets instead for the network component and thus simulate multicast even if it is not used was aimed to give an insight of how the system is behaving not only for the case where the actual hardware network could not offer multicast, but for the case of a virtualized environment. As a matter of fact, even if the virtualized environment can offer multicast, it could be that that actual deployment cannot benefit of it because of the VM allocation or of its implementation. Nonetheless where a real multicast is present, the system should use it, so to avoid being itself a limiting point for scalability.

Another goal of the technical part is to verify the easiness of a theoretical solution: it could be that a very simple theoretical solution requires a complicated or resource-expensive implementation due to the limits of current technology. An example of it can be seen in the first solution developed in the 60s to virtualize resources and support different OS through the concept of the virtual machine [14]: the VM concept is the same but required almost 40 years before viewing the first implementation available for the general public.

3.2 Hypothesis and contribution

The first hypothesis of this thesis is that context can be modeled with using two different object: one to store the actual value of the context in an opaque fashion, called the *Context* and another to capture context usage, called the *ContextFunction*. This distinction between the representation of the context value and its usage is necessary, as it will be explained with more details in the next chapter, to separate what is context from what each application wants to do with it. In fact different application can agree on what is context but they use it according to their needs, so it must be present in the model a way to reflect this difference.

The second hypothesis is how to port a middleware application to a cloud environment and add elasticity in order to make the most of its natural horizontal scalability. That means dynamic state partitioning is required, otherwise no fine grade adaptation can be made and the system will be over or under provisioned. The difficulty of this dynamic state partitioning depends of the type of stored information that makes the state and the constraints and invariants that has to be respected between each component. For instance a stateless component is easy to replicate since it has state constraints with other components but it could be that its presence require some form of coordination with the components that have a state that depends on it. This means that in addition to a local analysis, a global analysis has to be done to see the inter-component dependencies.

Obviously the ultimate goal would be the creation of a *Slice as distributed system primitive* [33] that can be invoked by the application to scale and thus separate and externalize all the complexity of the scaling process the resource provisioning system instead of handling internally.

The proposed algorithm, as it will detailed in chapter 5, operates on three different cases:

Stateless component

This component has no state, so its replication is easy and other components does not depends on it in their state.

Stateful component without coordination needs

The component has a state but it is only for internal operations and does not needs any coordination with others components during the scaling; this is typically the case of end-chain components. The replication of this kind of components can be done in parallel and be externalized using minor or no-component modifications, depending on the ability to track its state or state modifications.

Stateful component with coordination needs

The component has a state and its state depends on other components, that means changing others components requires some degree of coordination through message exchange to keep the consistency of component's state.

The base on which operate is to collect a consistent state of the distributed application using the Chandy-Lamport algorithm [12] and applying buffers to temporary hold state-changing messages during application reconfiguration. In addition to this algorithm has been designed to allow seamless application usage by its clients, providing only partially and temporally bounded suspension of some service that are not directly detectable by clients, thanks to the use of hot stand-by swap strategies.

These characteristics made it a good candidate as a reference *template* to look at when moving applications to a cloud environment, but at the same time highlighted some limitations of the concept of *Slice as distributed system primitive* that will be discussed in the next section.

3.2.1 Limitations

The main limitations that arose during the development of the aforementioned algorithm are the capacity to provide uninterrupted service for the clients and, at the same time, dynamically repartition application's state.

These limitations come in the form of possibly huge application modification to repartition the state according to the new configuration because each application has its own way to store information and it cannot be split in an agnostic way and that the new deployment could require some merge of information in addition to the split, independently if it is a scale-out or a scale-in operation. This means that some application-state tracking or some internal modifications has to be done to support a dynamic state partitioning.

As a matter of fact, if the uninterrupted service requirement is given up, an expensive but feasible strategy would be saving all application state and then re-deploy the whole application to match the new workload demand, and finally loading the previously saved state. Of course this strategy is very simple and could be easily implemented since the only requirements it has are the availability of an API to save and load application's for each component and the partitioning function.

On the other hand, if the dynamic state partitioning requirement would dropped, an *a priori* state partitioning could be applied, as in [5], but in that case no proper *elasticity* would be present since no real workload demand adjustment is present by only system's live migration: a feature that nowadays is offered by all hypervisors because it can be done in an application-agnostic manner.

Managing Context Explicitly: SilboPS

*Thinking doesn't guarantee that we won't make mistakes.
But not thinking guarantees that we will.*

— Leslie Lamport

4.1 Shortcomings of implicit context management

Taking into account roles, notifications and subscriptions, the problem is necessarily strongly affected by the entity context, in particular, their location. A naïve application of CBPS to such a scenario would be to merge content and context in both messages and filters, making it implicitly context-aware.

- **Matching inversion**

Classic CBPS systems model messages as key-value sets, $m \in \{\mathbb{K} \times \mathbb{V}\}$, where $\mathbb{K}$ is the attribute name and $\mathbb{V}$ is its value. Those messages can be matched by filters issued by subscribers, $f \in F \subseteq \{\mathbb{K} \times \mathbb{V}\}$. If m is in the scope of a filter f , it can be stated that f covers m or $f \succeq m$. The purpose of the message is to encode data, whereas the filter represents the interests in this information.

Applying CBPS to context-aware scenarios violates this separation of concerns: subscriber filters contain constraints over the context mixed with constraints over the notification; symmetrically published notifications include constraints for subscribers that are enforced by brokers.

Only a few CBPS systems, which implement filtering using Turing-complete languages, can afford this out-of-the-box change. Anyway, such systems are difficult to optimize and exploit in the first place.

- **Message flooding in changing environments**

Published messages bundle the notification content with contextual information ($\mathbb{K} = \mathbb{K}_{content} \cap \mathbb{K}_{ctx}$), for example, the events detected by cameras include the camera location. This approach has a comparatively small overhead with respect to a context-agnostic scenario since the rate of change of the context is small compared with the rate of publication.

If as in Figure 4.1 that assumption is not met, constant context updates will create a massive overhead because existing subscription filters ① are constantly invalidated ② and then updated ③. In terms of messages, this implies a waterfall of unsubscriptions and new updated subscriptions flooding the whole network as illustrated. In our scenario, driver subscriptions pose this problem since their position changes constantly, and they are encoded as part of the filter.

Figure 4.1: Exchange of messages with implicit context management

- **Unrealized potential performance**

If properly exploited, contextual information can improve efficiency by minimizing networking overhead in two senses: (a) messages could be efficiently routed; (b) the overlay network topology could be optimized. For instance, it is of no use to cluster drivers interested in *nearby traffic alerts* (same interest) without considering the context that permeates their interest. Similarly, clustering drivers at nearby locations (same context) is plagued by the same problem, since it ignores their interests.

Using implicit context, however, the broker will have to handle more complexity in the form of bigger routing tables and longer routing computation.

Finally, any advantage from context exploitation requires explicit context management. Implicit context-aware solutions require the broker to parse both messages and filters, which is an additional overhead.

- **Separation of concerns within entities**

Context management and subscription or publication are very different concerns that must be handled by different components at the architectural level. Implicit context-awareness favors coupling and forces rigid designs in which the P/S component is also responsible for detecting, managing and communicating context changes.

For instance, drivers can use a mobile application in which location context is managed by a GPS component decoupled from the subscriber component in charge of handling incoming notifications.

- **Undermined publisher/subscriber decoupling**

The main property of CBPS systems is decoupling in terms of space, time and synchronization between publisher and subscriber. The only thing that the subscriber needs to know is how the events of interest are represented.

Subscribers and publishers are obliged to share a unified and homogeneous representation of context, as context is embedded in notifications for implicitly context-aware systems. In the example scenario, location is represented uniformly as a coordinate point for all entities. A more flexible approach, such as ontology-based mediation, is beyond the scope of the scheme.

4.2 Proposed design principles

There is a clear motivation to overcome the weaknesses of implicit context-aware management, which degrade as the application scenario context becomes more changeable.

Our approach is to decouple context and content throughout the CBPS with the aim of spatially, temporally and synchronously decoupling publishers from subscribers within a *contextual scope*.

The design principles that guide our vision of a context-aware content-based publish/subscribe (CA-CBPS) system are as follows:

1. *Explicit separation of context and content*

The scope of both messages and their filters must be restricted and deal exclusively with content. Therefore, $m \in \{\mathbb{K}_{content} \times \mathbb{V}\}$.

2. Generalized P/S model for contextual scoping

The reversal of the matching procedure demonstrates an impedance mismatch between content-based filtering and any other type of filtering that does not fit the content-based model. Symmetric context-scoping filters will be attached to both subscriptions and publications, restricting publisher contexts and subscriber contexts, respectively. Such filters will be attached ($||$) to publications, $m||f_{cs}$, $f_{cs} \in F_{ctx}$, and to subscriptions $f||f_{cp}$, $f_{cp} \in F_{ctx}$ given $F_{ctx} \subseteq \{\mathbb{K}_{context} \times \mathbb{V}\}$. Note how publisher and subscriber can restrict their actions symmetrically through each other's contextual scope. This scoping mechanism can be seen as a role-based access control by the publisher [3] and as a new filtering dimension by the subscriber.

3. Context-invariant subscriptions and advertisements

Subscriptions will be context-invariant to avoid context change flooding. Context variable references will be used instead of embedding context values (cv) as part of the filters. In fact, a subscription f_s defined as $f||f_{cp}$ can be regarded as a function on the context in which the filter f_{cp} is the context-aware part:

$$f_s : \mathbb{C} \rightarrow F$$

$$f_s(ctx) \subseteq \{\mathbb{K} \times \mathbb{V}\}.$$

Such functions become regular filters when applied to a given context $ctx \in \mathbb{C}$ within the brokers. Figure 4.2 illustrates how flooding is avoided as subscriptions (①) are updated by propagating context updates (②).

Figure 4.2: Exchange of messages with explicit context management

To apply context update and bind it to its correct filters, a subscriber-unique UUID, created by the broker, is added to filters and to context messages. This is necessary to distinguish the same filter if issued by different subscribers and to bind the filter to its correct context when the context function is called. This does not clash with the decoupling offered by P/S systems since it is used only internally by brokers and is comparable to the procedure enacted by other systems that use a multipath topology to resolve the duplicate message problem [30, 27].

4.3 Proof of concept: proposed CA-CBPS applied to the use case

This section shows the application of our proposal to the domain problem presented in section 4.1. Our scenario includes a set of cameras sending traffic information (publishers) and a set of cars receiving relevant (context-aware) information (subscribers). Cameras are automatically connected to the brokers thanks to the infrastructure in which they are installed whereas cars can connect to brokers using the access points deployed along the road. This is an instance of a V2I scenario and is a simple sub-problem covering all the relevant details. At the same time this example fits the assumptions about environments with highly changeable context. Let us examine the state of the routing tables and the messages exchanged when adding publishers and subscribers to the network. Figure 4.3 shows the network of brokers, each of which has its own *contextManager*.

Figure 4.3: Scenario broker network

4.3.1 Adding publishers with changing context

A camera is connected to the broker network, specifically to broker B_1 whose *advertisement* and *subscription* tables are initially empty (see Figure 4.4). This camera establishes its initial context, namely its position, by invoking $setContext(CamPos)$ with $CamPos \equiv cv_p = \langle x_p = CamPos.x, y_p = CamPos.y \rangle$. However context updates from Cam_1 will not be sent to the broker since their value is meaningful only when bound to a notification. Instead the last context will be held by the local API and appended to the sending message on each *publish* method invocation. As a matter of fact, our assumption in this scenario is that context changes are more frequent than notification/subscription messages.

Figure 4.4: Adding a new publisher to the scenario

Cam_1 is a publisher and, as such, must send an advertisement declaring what content and context data it can provide. This information will be useful for finding out what kind of information each entity provides and what contextual variables can be used to contextualize the messages received. Cam_1 sends $advertise(A, A_{cp})$, where:

- A is an advertisement filter as in traditional CBPS systems. Cam_1 can notify accidents: $A \equiv \{msg = accident\}$.
- A_{cp} models the context of the publisher, x_p and y_p , and shows what contextual information could be exploited by subscribers to enrich their subscriptions. Cam_1 has a location context: $A_{cp} \equiv \{\exists x_p, \exists y_p\}$.

This advertisement is propagated throughout the broker network to B_2, B_3 and B_4 , producing changes in all their advertisement tables:

- B_1 , *advertisement* table: $\langle Cam_1; A, A_{cp} \rangle$

- B_2 , advertisement table: $\langle B_1; A, A_{cp} \rangle$
- B_3 , advertisement table: $\langle B_2; A, A_{cp} \rangle$
- B_4 , advertisement table: $\langle B_2; A, A_{cp} \rangle$

Cam_1 can now send notifications to the P/S infrastructure. B_1 receives $publish(m, cxt)$, where m is the notification, which is context-free content, and cxt , is the last context sent by Cam_1 . B_1 queries the *subscription* table to forward the message. Since it is empty, B_1 drops the message m until it receives a subscription following the path marked by previous advertisements. If the traffic camera is a mobile sensor and changes its position, it will use *setContext* to set the new position, but this will not be sent to broker B_1 until *publish* is invoked. Although for this scenario the context could be directly passed at publication time, interaction is cleaner if the API offers two different methods because there is no need for clients (publishers or subscribers) to remember the last context.

4.3.2 Adding subscribers with changing contexts

Now a first-time subscriber joins the scenario (as in Figure 4.5). The subscriber is a vehicle, V_1 , using a context-aware mobile device to receive alerts created within a 20 km radius of its position, which, for simplicity's sake, we model as a square with 40 km sides. Initially, V_1 's location will be in the range of Cam_1 .

Figure 4.5: Adding a new subscriber to the scenario

First of all, V_1 accesses the P/S network as a client of B_2 and sends its initial context, $setContext(CarPos)$. B_2 receives this information as a *context message*, extracts the cv_s values and creates a UUID (V_1^{id}) for V_1 , which it stores to its *contextManager* table as $\langle V_1^{id}; x_s = CarPos.x, y_s = CarPos.y \rangle$. Then B_2 checks, using the *subscription* table, to see if V_1 context needs to be forwarded. In this case, no action is taken since the table is empty. Next, B_2 receives the subscription of V_1 to nearby traffic alerts, $subscribe(f, f_{cp})$, which is mandatory since it has sent a $setContext(CarPos)$ call, where f is a content-based filter $f \equiv \{msg = accident\}$ and f_{cp} is a function that models a filter based on restrictions about publisher context, and receives dynamic subscriber context values:

$$\begin{aligned} f_{cp} \equiv & \{x_p \geq this.x - 20, \\ & x_p \leq this.x + 20, \\ & y_p \geq this.y - 20, \\ & y_p \leq this.y + 20\}. \end{aligned}$$

The broker first adds V_1^{id} to the new subscription f' , which is then stored into B_2 's *subscription* table as $\langle V_1; f', f_{cp} \rangle$. The next step is to check if there is any advertisement that matches the subscription. To do this, B_2 queries its *advertisement* table, which contains row $\langle B_1; A, A_{cp} \rangle$ and then checks if there exists an $A \succeq f' \wedge A_{cp} \succeq f_{cp}$. Given that A and f contains $msg = accident$ and that A_{cp} is $\exists x_p \wedge \exists y_p$ and f_{cp} uses exactly x_p and y_p as the publisher context, the match is positive, and B_2 sends first the context message, if needed, and then the subscription message to B_1 . To send the context message, B_2 checks whether the *subscription* table already contains an outbound entry for V_1^{id} . If it does not, it sends the context message, namely $context(cv_s)$ with cv_s the *context value* containing $\langle V_1^{id}; x_s = CarPos.x, y_s = CarPos.y \rangle$, to B_1 . Then it sends a subscription message with $subscribe'(f, f_{cp})$.

When B_1 receives both messages it adds the subscriber's context cv_s to the *contextManager* as $\langle V_1^{id}, cv_s \rangle$ and modifies its *subscription* table with the subscriber's data, writing row $\langle B_2; f', f_{cp} \rangle$. B_2 and B_1 then stop propagating the subscription because there are no more entries in their *advertisement* tables.

4.3.3 Notification delivery

To illustrate how new publications are to be delivered to interested subscribers, imagine that there is an accident close to Cam_1 . Cam_1 sends a new $publish(m, cv_p)$, where $m \equiv \{msg = accident\}$. The publication includes the contextual information because the previous $setContext$ called by publisher did not actually send the changes to the broker.

B_1 's *subscription* table contains row $\langle B_2; f', f_{cp} \rangle$ and therefore B_1 must check if the publish message matches these filters, dropping it otherwise. Because $m \succeq f$, B_1 must check the contextual filter. This is a lazy process led by usually more restrictive and relevant content filters.

At this point, the context-invariant filter f'_{cp} , which is the key component of our proposal, must be evaluated using the values fetched from the *contextManager* table. In particular, B_1 uses V_1^{id} that searches f' to look up context information of the associated $cv'_s \equiv \langle x_s = CarPos.x', y_s = CarPos.y' \rangle$ and then passes both context (cv_s and cv_p) to evaluate f'_{cp} :

$$\begin{aligned} f'_{cp}(cv_p, cv_s) = f''_{cp} \equiv & \{CamPos.x \geq CarPos.x - 20, \\ & CamPos.x \leq CarPos.x + 20, \\ & CamPos.y \geq CarPos.y - 20, \\ & CamPos.y \leq CarPos.y + 20\}. \end{aligned}$$

Note that the use of contextual values to evaluate context-aware restrictions has been delayed until now, improving performance in scenarios like this where context changes (car positions) occur at a faster rate than new notifications (traffic accidents). In addition, the context function is more expressive than the one available in subscriptions since it is able to perform simple computations and is not confined only to comparison.

In this case, the car is close enough to the camera, and the result of evaluating f'_{cp} is a positive match. Since both filters are satisfied, the message is sent to B_2 as the *subscription* table prescribes. To do so, B_1 generates an internal publish message, $publish'(m, cv_p)$ with the contextual variables of the publisher used in the previous step, namely $cv_p \equiv \{x_p = CamPos.x, y_p = CamPos.y\}$. Note again that the contextual data, in this case information about the publisher, is not propagated until strictly necessary.

This publication reaches B_2 . First, B_2 accesses the *subscription* table and checks that the publication matches row $\langle V_1; f', f_{cp} \rangle$. This process is identical to the above: $m \succeq f$ so f'_{cp} must be evaluated as $f'_{cp}(cv_p, cv_s) \rightarrow f''_{cp}$. B_2 ultimately obtains the same contextual filter, and so will still match.

As the publication matches V_1 's range, the message m carrying information about the nearby traffic incident is finally delivered to car V_1 .

4.3.4 Subscriber context update

First, let us consider that V_1 is driving along the highway and changes its context (see Figure 4.6) by sending $setContext(CarPos)$. B_2 processes this change of context with the help of the context manager, which replaces the values of x_s and y_s previously stored in the *contextManager* table.

Figure 4.6: Context change of a subscriber

B_2 receives the new context values $cv'_s \equiv \langle x_s = CarPos.x', y_s = CarPos.y' \rangle$, creates the V_1^{id} and checks if it is different from the last value entered in the *contextManager* table. If it is, it updates the table with the new values, then looks up the outbound entries for V_1^{id} used to send the context message in the *subscription* table and finds B_1 . If the new and old values are the same, no action is taken. Then B_1 receives this context message and modifies its *contextManager* table accordingly; next it queries the *subscription* tables, repeating the process carried out by B_2 .

Subsequent publications by Cam_1 will be subject to a new contextual match, using the new data. f''_{cp} may no longer match, thus no messages will reach B_2 . This way, tables are updated elsewhere in the broker network for every context they receive.

4.3.5 Publisher context update

Now camera position changes. Again, Cam_1 invokes the local API with $setContext(CamPos)$ but no message is sent to the broker; instead the new values $cv'_p \equiv \langle x_s = CamPos.x', y_s = CamPos.y' \rangle$ are held locally and sent together with the following *publish* call.

4.4 Prototype Evaluation

SilboPS evolved from our Java implementation of SIENA [9] and extends its interface by adding context functionality. In this way we were able to test the two systems in the same environment and match their differences without incurring in any semantic mismatch API penalization. As usual for P/S systems, our prototype does not support transactional messages, but offers FIFO channels with reliable delivery. This implies that no message loss will be observed unless there is a link failure; however both channel endpoints are aware of such failure. Messages are processed in the same order as they arrive at broker irrespectively of their sending time.

The experiment has been run under Linux 3.2 and OpenJDK 7u3 on an Intel i7 860 at 2.80 GHz machine equipped with 4 GiB of RAM. Due to our scenario assumption, the focus is on context reconfiguration cost, rather than notification delivery performance. Nevertheless we ran a notification delivery performance test to verify the overhead introduced by this extension.

To assure that our results were meaningful, we used the same distribution parameters as in [9], adding the context part as an attribute of notifications and subscriptions for the SIENA version, and as a context function/context message in SilboPS. We used a uniform distribution applied on a square with 1000 km sides to generate position values. The choice if this distribution gives SIENA a slight edge since it can exclude filters that do not match the notification position value whereas SilboPS has to wait the context function result.

Thus, a context change in SIENA requires two messages to be sent: one message to remove the old filter, if any, and the second message to add the new filter. This is necessary because otherwise subscriptions will pile up leading to poor router performance and, above all, the delivery of unwanted messages.

The first experiment is meant to compare the context change cost for the two systems. It performs 50 changes per subscriber on a single broker previously loaded with 10 k subscriptions (equivalent to 1250 connections). As shown in Figure 4.7, the operation has a fixed cost in both systems. However, SilboPS outperforms SIENA by three orders of magnitude. The reason is that SIENA has to modify its index entries (removing and adding the filters), instead of updating only the subscriber's context, whereas SilboPS only updates context values.

Figure 4.7: Normalized time on connection to perform 50 context change on each new subscriber.

Figure 4.8 shows the total time taken by our prototype to perform context changes when it is loaded with an increasing number of connections. The router is loaded with a maximum of 10 k subscriptions in the top chart and a maximum of 100 k in the bottom chart. Each connection has an average of 10 subscriptions, that is, a load of 1200 and 12000 connections per router respectively. This highlights how processing time is proportional to the number of context changes and depends on neither the number of connections nor the number of subscriptions.

Figure 4.9 shows the cost of one context change with respect to the number of subscriptions issued by a single subscriber. It stands to reason that the cost increases linearly for SIENA, whereas it is constant in SilboPS. In addition to a faster processing time, this will reduce the number of messages sent in the network by a linear factor. So, the more subscriptions

(a) Test executed with 10 k subscriptions.

(b) Test executed with 100 k subscriptions.

Figure 4.8: Time taken to change context with 10 k (a) and 100 k (b) subscriptions on a single broker in our prototype.

Figure 4.9: Time for a single context change with an increasing number of subscription per connection.

that are loaded to the broker, the bigger the saving will be. As a matter of fact, SilboPS behaves like SIENA when there is only one subscription per subscriber, but, even so, has the advantage of sending one instead of two messages, and message size is smaller.

Note that the data in Figure 4.9 and 4.7 are consistent: the time required for a single context change in SIENA at 50 subscriptions per connection is equal to the value shown in Figure 4.7, that is, 10 ms. In this test SilboPS performance for a single context change appeared to be equal to its performance rate for 50 context changes. This is because, even though it has nanosecond precision, the available API does not guarantee nanosecond accuracy¹. Even so we were able to calculate the slope for SIENA version and to get an estimate for the cost of a single context change of 105 $\mu\text{s}/\text{msg}$. For SilboPS we can either use Figure 4.7 where the time is high enough to be measured accurately, or use Figure 4.8 to calculate the slope, which is a fairly straight line. Thus we estimated an upper-bound of 0.2 $\mu\text{s}/\text{msg}$.

¹As described in [http://docs.oracle.com/javase/7/docs/api/java/lang/System.html#nanoTime\(\)](http://docs.oracle.com/javase/7/docs/api/java/lang/System.html#nanoTime())

Figure 4.10: Time required for changing the context in a 5-broker network with 1 publisher and 100 subscribers each.

Figure 4.10 reports the time to process context change messages in a network of five brokers each with one publisher and 100 subscribers, where each subscriber sends on average 10 subscriptions for a total of 1000 subscriptions per broker.

Our implementation is clearly about five times faster than SIENA. However, network transport protocol overhead reduces the three order of magnitude gain shown earlier. Another network effect is buffering introduced by sockets. Note that the slope of the two curves changes at around 200 and 1000 context changes for SIENA and our prototype respectively; this is due to selected channel socket buffer filling. As a matter of fact, both systems use a queue to hold and then a single thread to process incoming messages in the broker. Consequently, incoming sockets do not slow down the routing since they each have their own thread. However, outgoing sockets do slow down the process because they use the broker's thread.

Figure 4.11 shows notification throughput when the broker is loaded with 10 k subscriptions in the top chart, and with 80 k subscriptions in the bottom chart. It is clear that SilboPS introduces a small overhead with respect to SIENA and that maintains the same shape through all the range; this proves our first assumption, stating that filters applied to notifications are stricter than the ones applied to context. As expected, the more notifications we send, the more stable the value is, since the broker queue always has an element and thus the broker saturation point can be reached.

These data are useful for calculating the matching time difference between the two systems. This reveals the trade-off point between the number of notifications and the number of changing subscriptions in the system, namely when it is convenient to switch from SIENA to our approach. We chose these values (see Table 4.1) because they represent the biggest difference between the two systems throughputs above the broker saturation point.

Notifications	SIENA		Explicit Context		Difference	
	[msg/s]	[µs/msg]	[msg/s]	[µs/msg]	[msg/s]	[µs/msg]
3000	1083	923	984	1016	99	93
4000	1101	908	1016	984	85	76
Average	1092	915	1000	1000	92	85

Table 4.1: Numerical result of test with 100 k subscriptions

The trade-off point can be found by comparing the matching and reconfiguration time for the two systems. More formally:

$X_{notification}$ is the number of notifications

$X_{context}$ is the number of context changes

T_{SIENA}^{not} is the average time to process a notification with SIENA, namely 915 µs/notification

T_{SIENA}^{cxt} is the average time to process a context change with SIENA, namely 105 µs/context change

$T_{SilboPS}^{not}$ is the average time to process a notification with SilboPS, namely 1000 µs/notification

$T_{SilboPS}^{cxt}$ is the average time to process a context change with SilboPS, namely 0.2 µs/context change

(a) 10 k subscriptions

(b) 80 k subscriptions

Figure 4.11: Notification throughput with 10 k (a) and 80 k (b) subscriptions in a single broker

Thus the inequality to be applied to determine when switch to SIENA is

$$\begin{aligned}
 &X_{notification}T_{SIENA}^{not} + X_{context}T_{SIENA}^{cxt} \\
 &\quad < \\
 &X_{notification}T_{SilboPS}^{not} + X_{context}T_{SilboPS}^{cxt} \quad (4.1)
 \end{aligned}$$

which is solved with respect to the ratio of X 's ratio:

$$\frac{X_{notification}}{X_{context}} > \frac{T_{SilboPS}^{cxt} - T_{SIENA}^{cxt}}{T_{SIENA}^{not} - T_{SilboPS}^{not}} \quad (4.2)$$

and substituting the above values, we found as trade-off point:

$$\frac{X_{notification}}{X_{context}} \approx 1.23$$

Consequently, if the ratio between $X_{notification}$ and $X_{context}$ is greater than 1.23, SIENA processing time is better, whereas SilboPS performs better when the number of context changes is greater than the number of notifications. Note, additionally, that $X_{notification}$ and $X_{context}$ represents the *total* number, so the more subscribers in the system there are, the more likely our system is to perform better. Moreover, this supports our assumption for the scenario described, where the number of notifications (traffic issues) is lower than the number of context changes (car's position updates).

Bringing it to the cloud: Elastic SilboPS

*Complicated is not a sign that you are clever.
It is a sign that you failed.*
— Gilad Bracha

5.1 Concept

In order to overcome the weakness of sizing a scalable monitoring system with an a priori deployment, namely wasting resources or underperforming monitoring because of the changing load, the system must have a global elasticity property. In our approach we have added this property to all operators. In this way, we can adapt each layer to the specific environment without worrying about providing a setup that fits from the start. Besides, temporary deviations from the standard usage pattern, like load spikes, can be handled without having to restart the system.

E-SilboPS is divided into four layers, similar to the engine described in [4, 5], and each operator layer can scale in and out independently of the others. Some orchestration between layers is needed, however, to maintain state consistency. These four layers are:

Access Point Dispatches incoming subscriptions to the correct *matcher* and broadcasts notifications to all matchers.

Matchers Store incoming subscriptions in the internal structure, match notifications against subscriptions, and dispatch the local result set to the *Exit Point* layer.

Exit Point Collects all partial results sets coming from the *matcher* layer, which it merges in the final results set and then sends the notification to all *Connection Points* found in this set.

Connection Point Represents the system entry and exit point; it handles the connection with clients and sends the received notifications to the interested subscribers; it forwards subscriber subscriptions and publisher notifications to the *Access Point*.

To perform proper scaling [22, 29], the system must be able to match the variable demand of workload and, at the same time, control the cost of the deployed infrastructure. Our solution uses the *Chandy-Lamport algorithm* [12], also known as the *snapshot algorithm*, to create a consistent cut of the global state of the system. It then executes the algorithm that we designed to transfer the state throughout the system, repartitioning it according to the new available slices. One noteworthy feature of our algorithm is that it is capable of matching notifications while restructuring its internal state and has a very short no-matching time before switching to the new state.

5.2 The Algorithm

In this section we explain our scaling algorithm: there are three possible layer reconfigurations, one for each operator except for *Connection Point*. Depending on the layer and operator state, coordination with others layers may be required.

5.2.1 Access Point Scaling

Access point scaling is straightforward since it is a stateless operator, and no coordination is required because it is usually bound with a *Connection Point*. This means that layer reconfiguration is limited to adding or removing target operators and opening or closing connections to the *Matcher* layer.

Access Point start or stop instances according to the new configuration

Matcher no action

Exit Point no action

5.2.2 Matcher Scaling

As shown in Figure 5.1, the scaling of the *Matcher* layer requires coordination with the other two layers since *Access Point* will need to use a new selector and buffering subscriptions and notifications, whereas *Exit Point* will have to change their internal state to correctly collect notifications coming from the new *Matcher* configuration.

The scaling process starts by sending a **Start Scale-In/Out** message to all Access Points ①; upon receiving this message each access point starts buffering subscriptions and forwards the same message, together with the new *selector* that they have to use, to all existing matchers ①. This renders the state of the *Matcher* layer temporarily unchangeable, although, at the same time, it is still able to process notifications. Thus, this phase is transparent for *publishers*, and *subscribers* will still receive notifications as usual, although they will not see any updates of the subscriptions that have changed in the meantime.

As soon as a matcher receives the first **Start Scale-In/Out** message, it connects to all matchers of the new configuration and partitions its internal state using the *selector* included in the received message. It then sends this new state to the respective matchers, itself included ②. In this way, all matchers know in advance the matcher from which they will receive the new state, that is, from the matchers present in the *new* configuration in the event of a *scale-out* or from the matchers in the *old* configuration in the event of a *scale-in*.

When a matcher has received all state information, it sends a **Change-Ready** message to ZooKeeper to notify the *Access Point* layer ③. This layer will be unlocked with a **Change-Matcher-Ready** when all matchers have finished ④. The **Change-Matcher-Ready** message signals to the *Access Point* layer to enqueue notifications and to send a **Change-Matcher-On-Exit Point** message to all old matchers ⑤ and a **Change-Matcher** to all (old+new) matchers ⑥, so they can swap their internal state using the new one.

The **Change-Matcher-On-Exit Point** message must precede **Change-Matcher** because, when it arrives at the *Matcher* layer, it is forwarded to the *Exit Point* layer in order to create a consistent change of the view of the *Matcher* layer state from the *Exit Point* layer perspective, without direct synchronization between matchers. This message must be sent to all matchers in order to prevent subsequent notifications reaching a newly added matcher and being processed faster than the last notification sent before the state change. Otherwise, the exit point would associate this new notification with a wrong number of matchers, which, in the event of *scale-out*, will lead to the

received subset of subscribers being notified too early. This will leave the remaining set of subscribers starving at the same exit point. In the event of *scale-in*, the notification will never be delivered to subscribers because the exit point will not receive all the expected sets for this purpose.

When an exit point receives the **Change-Matcher-On-Exit Point** message, it reads the current number of matchers from ZooKeeper and notifies its new state. The number read from the configuration will be immediately used as the new threshold before pushing notifications to subscribers.

Once all exit points have been updated, ZooKeeper will send an **Exit Point-Matcher-Updated** to *Access Point* layer ⑦ to flush buffered notifications if *scaling-out* and wait for all access points to finish the operation. It then sends a **Scale-End** message to the *Access Point* layer ⑧, which will be forwarded to the *Matcher* layer and will close all unnecessary connections. On receiving the **Scale-End** message, each matcher will close its connections to others but will not forward this message to the *Exit Point* layer since no operation is needed for this purpose.

Access Point layer

- When receiving a **Start Scale-In/Out** message with a new matcher selector:
 1. Continue to broadcast notifications to all current matchers
 2. Buffer subscriptions in a queue
 3. Send **Start Scale-In/Out** message to all (old+new) matchers with the new matcher selector to be used
- When receiving a **Change-Ready** message:
 1. If in the *scale-out* phase, the notifications are no longer sent¹
 2. The **Change-Matcher-On-Exit Point** message is sent to all old matchers
 3. The **Change-Matcher-Ready** is sent to all (old+new) matchers
 4. The current selector is exchanged for the new selector received from *Access Point*
 5. While waiting **Exit Point Matcher-Updated** sends enqueued subscriptions to matchers
- When receiving a **Exit Point Matcher-Updated** message:

¹The notifications sent after *change view* could enter the new matcher instance, being matched sooner than other notifications from other matchers and reaching the Exit Point before the last notification of the previous group.

Figure 5.1: Messages exchanged during *Matcher* layer scale-out: solid lines represent connections between operators used by the system to send notifications and subscriptions. Dashed lines represent ZooKeeper-mediated messages used to synchronize layers, typically to unlock all operators from a layer waiting for another layer to complete its task.

1. If it is in *scale-out* phase, flush enqueued notifications
2. Resume sending notifications
3. Signal **Scale-End** to ZooKeeper

Matcher layer

- When receiving a **Start Scale-In/Out** message with a new matcher selector from all *Access Points*:
 1. Notifications continue to pass through the matcher using current matching algorithm²
 2. Snapshot matcher's *Subscription Map*
 3. map snapshotted subscription set with new matcher selector and store the result into `Map<Slice, Map<ConnectionID, Set<Subscription>>>`
 4. Send to each matcher, including itself, the new mapping
- When receiving a **Add-Matcher-State** message:
 1. The subscription set is added to the new matcher state
 2. When all **Add-Matcher-State** messages have been received, the **Change-Ready** message is sent to *Access Point* through ZooKeeper
- When receiving **Change-Matcher-On-Exit Point**:
 1. Forward it to all Exit Points
- When receiving a **Change-Matcher-Ready** message:
 1. Change matcher state with new one
 2. Start processing notifications and subscriptions

Exit Point layer

- When receiving a **Change-Matcher-On-Exit Point** message:
 1. The new number of matchers is read from ZooKeeper and used as the current limit for notifications
 2. **Exit Point-Changed** is sent to ZooKeeper

²No subscriptions can arrive because they are buffered at Access Point layer.

Figure 5.2: *Exit Point* layer scale-out: exit points from $j + 1$ to k are added to the current configuration.

5.2.3 Exit Point Scaling

Exit Point layer scaling (see Figure 5.2) has to be coordinated with the *Matcher* layer because a consistent cut of notification must be done. Even though no direct mechanism of synchronization between matchers is required, each matcher has to perform a set of operations such that the added (or removed) exit point instances will be used only after a consistent *change view*.

An **Exit-Point-Scaling** message is first sent to access points which they then broadcast to matchers. The matchers use this message to create the new exit point *selector* and buffering notification on an access point basis. Once it has received the respective message from all access points, the matcher processes all buffered notifications using the new exit point *selector* and sends a message to ZooKeeper to notify its reconfiguration. After collecting all reconfiguration messages from matchers, Zookeeper sends a **Scale-End** message to the *Access Point* layer to clean up connections in the same manner as for *Matcher* layer scaling.

Figure 5.3: *Matcher* layer notifications cut: AP_1 sends odd notifications ($N_1 \dots N_5$), whereas AP_n sends even notifications ($N_2 \dots N_4$). EPS_1 and EPS_n represent the **Exit Point-Scaling** messages broadcast by each access point to all matchers.

As shown in Figure 5.3, AP_1 sends notifications starting with N_1 and ending with N_5 , whereas AP_2 sends notifications starting with N_2 and ending with N_4 . However, due to the independent FIFO channels that connect access points to matchers, the order of delivery is N_1, N_3 and N_2 , as viewed by M_1 , as opposed to N_2, N_1 and N_3 , as viewed by M_m , before reception of the two **Exit Point-Scaling** messages. Besides, M_m receives N_4 before ESP_1 , but it knows that this notification will be in the *Post* set because it has already received ESP_n from the same channel.

This proves that, even if the order of the messages received by matchers differs, their *Pre* and *Post* sets contain the same elements, so they can process notifications according to a *consistent cut*. Besides, the relative ordering of notifications coming from each access point is preserved, so, from a *publisher's* point of view, no out-of-order matching is performed.

Access Point layer

- When receiving **Exit Point-Scaling**:
 1. Forward **Exit Point-Scaling** to all matchers

Matcher layer

- When receiving first **Exit Point-Scaling**:
 1. Read from ZooKeeper the number of *Access Point* and *Exit Point* instances
 2. Create the new *Exit Point* selector

3. Start enqueueing notification from that *Access Point* slice
- When receiving subsequent **Exit Point-Scaling**:
 1. Notifications from that *Access Point* slice are enqueued
- When receiving last **Exit Point-Scaling**:
 1. The new *Exit Point* selector is adopted and all enqueued notifications are flushed
 2. DONE message is sent to ZooKeeper to signal the end of current matcher reconfiguration

Exit Point layer

- Instances are added or removed according to the new configuration

5.3 Evaluation

E-SilboPS is the elastic version derived from SilboPS, our Java implementation of SIENA [9, 47]. As usual for P/S systems, our software does not support transactional messages, but offers FIFO channels with reliable delivery. This implies that no message loss will be observed, unless there is a link failure, of which both channel endpoints are aware. Messages are processed in the same order as they arrive at the operator instances irrespective of their sending time.

The experiment has been run on an Intel i7 860 @ 2.80 GHz machine equipped with 12 GiB of RAM running Linux 4.2 and OpenJDK 7u91.

At first we tested throughput performance with different configuration deployments to find out where the processing time was dominated by matching time instead of network time. We started our tests with a 1-1-1 configuration, that is, one access point, one matcher and one exit point with the addition of one connection point bound to the access point. Before measuring performance, we created 100 k notifications and one million subscriptions in order to assure that object creation time did not affect the measured throughput. In addition, the notifications were created to match at least one subscription in order to be able to constantly measure notification throughput.

This is a worst-case scenario setup since the system has to deliver all the notifications, whereas, in a real environment, some notifications will not be delivered because no matching subscriptions have been sent. As shown in Figure 5.4, when E-SilboPS was deployed with a 1-1-1 configuration, notification throughput started to slow down just after the system was loaded

Figure 5.4: Notification throughput with different configurations: the triplet in the legend represents the number of instances deployed for *Access Point*, *Matcher* and *Exit Point*

with 10 k subscriptions. It descended at a constant rate, and processing time started to be dominated by matching as of 20 k subscriptions. This means that the constant throughput on the left side of the graph is due to network saturation. This limits the amount of messages that an operator can process [28].

This is proven by the fact that there is a decrease in notification throughput when the number of operators, and consequently the number of messages passing to the network layer, increases but the workload is unchanged. Unsurprisingly, minimum notification throughput is observed with the configuration deployment that has the maximum number of elements (1-6-2). In the same way, we find that increasing the numbers of matchers on the right side of the graph, where the matching time is dominant, improves throughput, because, on a per operator basis, the state is smaller, and thus it takes less time to calculate the partial set of subscribers. This is really important since it proves that the current architecture is able to scale out, which is a prerequisite for elasticity [22].

Figure 5.5: Notification throughput speedup dominance of $1-x-2$ over $1-x-1$: the red line with filled circles plots the speedup for the $1-x-2$ configuration, whereas the blue line with empty circles plots the speedup for the $1-x-1$ configuration; plots (a) to (e) represent matcher instances, namely x , scaling from 2 to 6.

Another interesting point is that the addition of more matchers not only increases the throughput but also changes the slope of the curve. This is consistent with the region in the center of the graph where each matcher is working. If there are fewer subscriptions, the processing time is divided more equally between network time and matching time, and, ideally, this division will be such that the two are balanced leading to 100% usage of both resources [28, 15].

Note also that it is in the region between 50 k and 300 k subscriptions where performance differences between configurations that have only one exit point and others that have two start to appear. As a matter of fact, the $1-x-2$ configuration always outperforms the $1-x-1$ configuration, and the difference is greater the more matchers it has, as shown in Figure 5.5.

Figure 5.6: Notification throughput speedup for different configurations with an increasing subscriptions load

This means that the *Exit Point* layer does a good job of offloading work from matchers in this region, namely by reducing the partial sets of subscribers to a single set. On the contrary, network saturation limits the total throughput on the left of the graph, hiding any potential gain, whereas the matchers are the factor limiting notification throughput on the right.

Finally, note for Figure 5.4 that, on the far right, the speedup gained by adding more and more matchers is downward, that is sub-linear. This applies to total efficiency, as shown in Figure 5.6, Figure 5.7 and Figure 5.8 too. This is consistent with Amdahl's law, Gustafson's law [2, 21] and Eager [15]. Even so, the notification throughput provided using the 1-6-2 was 3.4 times higher than using the 1-1-1 configuration.

Figure 5.7 comes in useful for analyzing the cost-worthiness of the scaling, since it shows where, for a given configuration deployment and load, it is worth adding another matcher. We find that the top speedup for a 100 k load is achieved with a 1-3-2 configuration. This means that adding more resources simply results in a worse throughput because of message overhead. Similarly, a workload increase pushes the curve's peak to the right of the graph.

Figure 5.7: Notification throughput speedup sliced on 1- x -2 configurations

For clarity's sake, Figure 5.7 shows the speedup for 1- x -2 deployments only, since, as shown in Figure 5.5 above, the 1- x -2 configuration always outperforms 1- x -1. However, the same does not apply with respect to efficiency, which is computed as a ratio of speedup to the number of instances (processors) used, that is, $E(n) = \frac{S(n)}{n}$ [15]. Figure 5.8 shows how efficiency decreases as the number of slices increases and the 1- x -1 configuration is more efficient than the 1- x -2 configuration in most cases and especially with deployments with not many instances. Additionally, this graph indicates to which layer it is better to add a new slice since deployments with the same number of slices (highlighted regions) can be compared and the cost-worthiness of adding a second exit point (white regions) analyzed. As a matter of fact, we find that the most efficient deployment for the three-slice region for all loads except 100 k is the configuration with only one exit point. This means that matcher requirements are greater than exit point utility. The same applies for the four-slice region, although the threshold is raised to a 400 k subscriptions load. This is consistent with expectations since the speedup *contribution margin* of each new added matcher will be smaller than its predecessor's. This could be modeled in an *open* cloud using the Shapley Value [44, 40].

Figure 5.8: Notification throughput speedup efficiency: a horizontal line means equal efficiency, the highlighted region represents an equal number of slices where a horizontal line also denotes an equal speedup, whereas equal speedup is represented by the local hyperbola in the white region.

Likewise, we observe that adding an exit point in the five- and six-matcher region will help to improve both efficiency and speedup: this suggests that it is better to add an exit point than a matcher in such deployments. Thus we can infer, as a rule of thumb, that one exit point is necessary for every four matcher instances, irrespective of the number of loaded subscriptions.

Finally, Figure 5.9 shows the estimated Karp-Flatt metric $e = \frac{\frac{1}{s} - \frac{1}{p}}{1 - \frac{1}{p}}$ of configuration deployments, where s represents the speedup and p the number of slices (processors) used. This metric is used to estimate the serial part and the overhead from experimental data [26]. It is, therefore, a valuable metric for use in conjunction with efficiency, since it gives an insight into whether the decreased efficiency is due to the limits of parallelism or overhead. As we can see, we are experiencing *network* overhead with a 100 k load since

Figure 5.9: Estimation e of the serial part and overhead using the Karp-Flatt metric: the smallest load (100 k) clearly yields a larger overhead whereas the biggest load (1 M) reduces overhead to 10%. This reduction is present in all configurations deployed and is directly proportional to the number of matchers. However, the number of exit points has a positive effect in that it lowers e . Therefore, the estimation of e can be attributed to *network* overhead rather than the *intrinsic* serial part of matching notifications.

adding more matchers increases the estimated e , especially with the $1-x-1$ configuration. By looking at how the same deployment behaves with a larger subscriptions load, we find the proof that the overhead is related to the network: the estimated e drops down to the 10-20% region.

In order to measure system behavior during scale-out, we loaded the system with 100 k subscriptions and set up a constant stream of 2000 notifications/s and 1000 subscriptions/s. Thus we can examine the throughput of notifications delivered to the subscriber to see how the different *Matcher* layer scale-out phases affect the throughput of notifications. We ran this test with all deployments, finding that all the results were very similar.

Figure 5.10: Notification throughput with subscriptions during scale-out of *Matcher* layer: the deployed system configuration was 1-1-1, scaling out to 1-3-1 and pre-loaded with 100 k subscriptions. The red line is the notification throughput delivered to subscribers while publishing 2000 notifications/s and adding 1000 subscriptions/s. The highlighted region shows when the system was scaling out.

Figure 5.10 shows the scale-out from 1-1-1 to 1-3-1 configuration. The red line represents notification throughput received from the subscriber and the highlighted region represents the time in which the system was scaling out. Clearly, the system continues to process notifications while the subscriptions are buffered in the access points and matchers are switching to their new state, after which the access point starts buffering notifications and flushes all its buffered subscriptions. This phase is represented in the graph seconds 11.1 and 11.9. The matchers then have their new state updated so that they can reprocess the notifications. During the last phase, which takes place from seconds 11.9 to 12.3, all buffered notifications

are flushed. Obviously, since each matcher operates independently of the others, throughput is highly variable. The normal throughput of 2000 notifications/s is restored at the end of the scale-out, meaning that all internal buffers have been emptied.

We found that notification and subscription throughput has a linear impact on the duration of the scaling window. However, the system always completes the scale-out in a finite amount of time since the size of the buffers that it has to empty is determined by the time taken to complete each phase and the buffers are not updated with new subscriptions or notifications once a phase is complete. In the test that we performed, the time taken to scale out the system ranged from 1 to 8 seconds with notification and subscription throughput ranging from 1000 to 8000 messages per second.

This result shows how dynamic state partitioning can be achieved in an acceptable amount of time, thereby solving the problem of forecasting the maximum number of slices required during peak loads and providing the system with the most efficient configuration for the actual load. This is the major contribution of our research on implementing a proper *elastic* content-based publish/subscribe system. It differs from other systems [5] that use pre-state partitioning and simply move slices around. These systems offer no more than application-specific optimized versions of the VM migration provided by all major hypervisors today.

Conclusions

So I have just one wish for you — the good luck to be somewhere where you are free to maintain the kind of integrity I have described, and where you do not feel forced by a need to maintain your position in the organization, or financial support, or so on, to lose your integrity.

May you have that freedom.

— Richard Feynman

6.1 About Context-Awareness

This thesis presented a novel context-aware P/S model in which the context is managed explicitly. Our model adds a new dimension to CBPS flexibility: context scoping based on context-invariant filters, i.e. the notification filtering is separated from context filtering, allowing SIENA-like forwarding index to be updated only on new subscriptions and at the same time enabling a quick subscriber's context update without rebuilding the whole structure thanks to the indirection provided by the context functions and the *contextManager* table.

The experiments have shown that the approach presented introduces a small overhead with respect to SIENA in terms of notifications throughput, but at the same time outperforms SIENA by three orders of magnitude on subscriber context updates. In addition, using this solution, clients do not have to hold the whole list of sent filters and manage filter life cycle (unsubscribe previous subscriptions, create filters with the new context and send the new subscriptions).

It has also been demonstrated that the solution is extremely valuable in domains where context is highly dynamic and the number of context changes exceeds the number of notifications, such as inventorying, stock portfolios, people or vehicle locators that have with high or “bursty” context update rates.

6.2 About Elasticity

E-SilboPS implements a proper *elastic* content-based publish/subscribe system that supports transparent, *publisher-wise* dynamic state repartitioning without client disconnection and with minimal notification delivery interruption for subscribers.

Unlike other proposals such as [31], which focuses on fault tolerance and thus requires more space to store the redundant state and also a more complex algorithm for execution by the *Access Point* layer, or [5], which is really a reimplement of an application-specific version of the VM migration capability offered by today’s hypervisors, we described the full algorithm which we validated with qualitative and quantitative measurements. This demonstrated that dynamic state repartitioning is not only possible but also achievable even from the performance point of view.

6.3 Future Work

Of the design principles presented in section 4.2, the focus will be now on the *generalized P/S model for contextual scoping*. According to this model, symmetric context-scoping filters are attached to both subscriptions and publications. This allows publishers to restrict notification delivery to subscribers (i.e. by checking subscriber filter validity) based on both contexts. This scoping mechanism can be seen as role-based access control by the publisher [3, 6] and as a new filtering dimension by the subscriber.

This design principle will be accomplished in two stages. First, a third parameter A_{cs} (i.e. $advertise(A, A_{cp}, A_{cs})$) will be added to the advertisement $advertise(A, A_{cp})$ sent by the publisher (Cam_1), see Figure 6.1), where A_{cs} models the context information that the publisher is going to exploit to restrict the dissemination of published messages, resembling the above role-based access control.

This advertisement is propagated through the broker network to B_2 , B_3 and B_4 , leading to changes in all their advertisement tables, which now manage the publisher advertisement context for subscribers, A_{cs} :

Figure 6.1: Adding a new publisher to the scenario

- B_1 , advertisement table: $\langle Cam_1; A, A_{cp}, A_{cs}; \emptyset \rangle$
- B_2 , advertisement table: $\langle B_1; A, A_{cp}, A_{cs}; \emptyset \rangle$
- B_3 , advertisement table: $\langle B_2; A, A_{cp}, A_{cs}; \emptyset \rangle$
- B_4 , advertisement table: $\langle B_2; A, A_{cp}, A_{cs}; \emptyset \rangle$

The second stage is concerned with enhancing the *publish* message, changing context values with a contextual scope filter F_{cs} (i.e. a function of the subscriber context) that publishers (i.e. cameras) use to restrict the propagation of m , and second context values in a similar way as for subscribers with *setContext(CamPos)* message. As Cam_1 does not restrict the scope, $F_{cs} = \emptyset$.

This way, Cam_1 can send notifications to the P/S infrastructure. B_1 receives *publish*(m, F_{cs}) where m is the notification, which is context-free content, and F_{cs} is a contextual scope filter that the camera uses to restrict the propagation of m .

If the traffic camera is a mobile sensor and changes position, it will send new *setContext* messages to B_1 . The context manager of this broker will just update its local variables x_p and y_p with the new data and forward the message to its neighbors (see Figure 6.2). Whether filter removal should be applied at each context change, timed or advertisement based requires evaluation.

The next steps are to evaluate the overall system performance overhead introduced by this approach, and the trade-off between this overhead and benefits for publishers that intend to take advantage of the added functions.

Figure 6.2: Clients change their context

6.3.1 Cloud development

With E-SilboPS we were able to add elasticity to a content-based publish subscribe system. However, state partitioning is not enough for efficient elasticity, and thus scalability, in a cloud system. As a matter of fact, one limitation of our current model is that it does not provide for slice *replacement*. This means that once a slice is assigned to a machine, the machine cannot be removed unless the slice is deleted. Due to current system constraints, the only option is to scale in the system. This does not, however, solve the problem for the first slice of each operator layer. Nevertheless, this hitch is mitigated by the VM migration that is offered by today's hypervisors.

A second improvement is to take full advantage of the availability of network multicast: this will reduce the number of messages to be sent and CPU usage. However, this requires thorough validation since it could perform as well as the current implementation in a virtualized environment, where, unlike high-end routers, there are no physical processors dedicated to each endpoint.

Another interesting feature warranting support is multilayer scalability: the capacity to scale all three layers in and out independently of each other within a single scaling transaction. However, this should be weighted with the increased complexity of the algorithm. Apart from in-field validation of the use case, it should also be validated to see if it delivers the expected improvements.

E-SilboPS currently depends on an external system to decide when it has to scale in or out. This system has to make this decision based on current resource usage, as well as delivered performance. The use of metrics that measure only CPU or memory usage is not very suitable for a cloud system because they are only helpful for the technicians that manage the cloud. Service end users neither know nor care about how much CPU or how many machines are used, they are concerned about service quality such as SLA or their user experience. This means that the software that decides when to scale E-SilboPS in or out should have access to the above metrics and possibly predict peaks of usage with respect to the SLA. At the same time, it should have knowledge of the performance efficiency and monetary cost of the current configuration, which should be optimum [34, 35, 36, 20, 19].

The last feature that we think would add real value to the system is the capacity to handover connections between connection points without disconnecting clients. Using this feature it will be possible to balance the load repartitioning connection looking at producer or consumer throughput.

Bibliography

- [1] M. Altherr, M. Erzberger, and S. Maffeis. iBus - a software bus middleware for the Java platform. In *Proceedings of the Workshop on Reliable Middleware Systems of IEEE SRDS'99*, pages 43–53, 1999.
- [2] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference, AFIPS '67 (Spring)*, pages 483–485, New York, NY, USA, 1967. ACM. doi: 10.1145/1465482.1465560. URL <http://doi.acm.org/10.1145/1465482.1465560>.
- [3] Jean Bacon, David M. Eysers, Jatinder Singh, and Peter R. Pietzuch. Access control in publish/subscribe systems. In *Proc. of the 2nd int. conference on Distributed event-based systems, DEBS*, pages 23–34, NY, USA, 2008. ISBN 978-1-60558-090-6. doi: <http://doi.acm.org/10.1145/1385989.1385993>. URL <http://doi.acm.org/10.1145/1385989.1385993>.
- [4] Raphaël Barazzutti, Pascal Felber, Christof Fetzer, Emanuel Onica, Jean-François Pineau, Marcelo Pasin, Etienne Rivière, and Stefan Weigert. Streamhub: A massively parallel architecture for high-performance content-based publish/subscribe. In *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems, DEBS '13*, pages 63–74, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1758-0. doi: 10.1145/2488222.2488260. URL <http://doi.acm.org/10.1145/2488222.2488260>.
- [5] Raphaël Barazzutti, Thomas Heinze, André Martin, Emanuel Onica, Pascal Felber, Christof Fetzer, Zbigniew Jerzak, Marcelo Pasin, and Etienne Rivière. Elastic scaling of a high-throughput content-based publish/subscribe engine. In *Proceedings of the 2014 IEEE 34th International Conference on Distributed Computing Systems, ICDCS '14*, pages 567–576, Washington, DC, USA, 2014. IEEE Computer Society. ISBN 978-1-4799-5169-7. doi: 10.1109/ICDCS.2014.64. URL <http://dx.doi.org/10.1109/ICDCS.2014.64>.

- [6] András Belokosztolszki, David M. Eysers, Peter R. Pietzuch, Jean Bacon, and Ken Moody. Role-based access control for publish/subscribe middleware architectures. In *Proc. of the 2nd int. workshop on Distributed event-based systems*, DEBS, pages 1–8, NY, USA, 2003. ISBN 1-58113-843-1. doi: <http://doi.acm.org/10.1145/966618.966622>. URL <http://doi.acm.org/10.1145/966618.966622>.
- [7] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Trans. Comput. Syst.*, 2(1):39–59, February 1984. ISSN 0734-2071. doi: 10.1145/2080.357392. URL <http://doi.acm.org/10.1145/2080.357392>.
- [8] René Guy Busnel and André Classe. *Whistled languages*. Communication and cybernetics (Book 13). Springer-Verlag, Berlin; New York, 1976. ISBN 0387077138, 978-0387077130. URL <https://library.soas.ac.uk/Record/113815>.
- [9] Antonio Carzaniga and Alexander L. Wolf. Forwarding in a content-based network. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '03, pages 163–174, New York, NY, USA, 2003. ACM. ISBN 1-58113-735-4. doi: 10.1145/863955.863975. URL <http://doi.acm.org/10.1145/863955.863975>.
- [10] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Achieving scalability and expressiveness in an internet-scale event notification service. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '00, pages 219–227, New York, NY, USA, 2000. ACM. ISBN 1-58113-183-6. doi: 10.1145/343477.343622. URL <http://doi.acm.org/10.1145/343477.343622>.
- [11] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Trans. Comput. Syst.*, 19(3):332–383, aug 2001. ISSN 0734-2071. doi: 10.1145/380749.380767. URL <http://doi.acm.org/10.1145/380749.380767>.
- [12] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, February 1985. ISSN 0734-2071. doi: 10.1145/214451.214456. URL <http://doi.acm.org/10.1145/214451.214456>.

- [13] George Coulouris, Jean Dollimore, Tim Kindberg, and Gordon Blair. *Distributed Systems: Concepts and Design*. Addison-Wesley Publishing Company, USA, 5th edition, 2011. ISBN 0132143011, 9780132143011.
- [14] R. J. Creasy. The origin of the vm/370 time-sharing system. *IBM J. Res. Dev.*, 25(5):483–490, September 1981. ISSN 0018-8646. doi: 10.1147/rd.255.0483. URL <http://dx.doi.org/10.1147/rd.255.0483>.
- [15] D. L. Eager, J. Zahorjan, and E. D. Lozowska. Speedup versus efficiency in parallel systems. *IEEE Trans. Comput.*, 38(3):408–423, March 1989. ISSN 0018-9340. doi: 10.1109/12.21127. URL <http://dx.doi.org/10.1109/12.21127>.
- [16] Patrick Eugster. Type-based publish/subscribe: Concepts and experiences. *ACM Trans. Program. Lang. Syst.*, 29(1), January 2007. ISSN 0164-0925. doi: 10.1145/1180475.1180481. URL <http://doi.acm.org/10.1145/1180475.1180481>.
- [17] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, June 2003. ISSN 0360-0300. doi: 10.1145/857076.857078. URL <http://doi.acm.org/10.1145/857076.857078>.
- [18] PatrickTh. Eugster, Rachid Guerraoui, and Joe Sventek. Distributed asynchronous collections: Abstractions for publish/subscribe interaction. In Elisa Bertino, editor, *ECOOOP 2000 — Object-Oriented Programming*, volume 1850 of *Lecture Notes in Computer Science*, pages 252–276. Springer Berlin Heidelberg, 2000. ISBN 978-3-540-67660-7. doi: 10.1007/3-540-45102-1_13. URL http://dx.doi.org/10.1007/3-540-45102-1_13.
- [19] Jeffrey Galloway, Karl Smith, and Jeffrey Carver. An empirical study of power aware load balancing in local cloud architectures. In *Proceedings of the 2012 Ninth International Conference on Information Technology - New Generations*, ITNG '12, pages 232–236, Washington, DC, USA, 2012. IEEE Computer Society. ISBN 978-0-7695-4654-4. doi: 10.1109/ITNG.2012.171. URL <http://dx.doi.org/10.1109/ITNG.2012.171>.
- [20] Jeffrey M Galloway, Karl L Smith, and Susan S Vrbsky. Power aware load balancing for cloud computing. In *Proceedings of the World Congress on Engineering and Computer Science*, volume 1, pages 19–21, 2011.

- [21] John L. Gustafson. Reevaluating amdahl's law. *Commun. ACM*, 31(5): 532–533, May 1988. ISSN 0001-0782. doi: 10.1145/42411.42415. URL <http://doi.acm.org/10.1145/42411.42415>.
- [22] Nikolas Roman Herbst, Samuel Kounev, and Ralf Reussner. Elasticity in cloud computing: What it is, and what it is not. In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)*, pages 23–27, San Jose, CA, 2013. USENIX. ISBN 978-1-931971-02-7. URL <https://www.usenix.org/conference/icac13/technical-sessions/presentation/herbst>.
- [23] Mingsheng Hong, Alan J. Demers, Johannes E. Gehrke, Christoph Koch, Mirek Riedewald, and Walker M. White. Massively multi-query join processing in publish/subscribe systems. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, SIGMOD '07, pages 761–772, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-686-8. doi: 10.1145/1247480.1247564. URL <http://doi.acm.org/10.1145/1247480.1247564>.
- [24] Hans-Arno Jacobsen. Channel-based publish/subscribe. In LING LIU and M.TAMER ÖZSU, editors, *Encyclopedia of Database Systems*, pages 321–323. Springer US, 2009. ISBN 978-0-387-35544-3. doi: 10.1007/978-0-387-39940-9_1207. URL http://dx.doi.org/10.1007/978-0-387-39940-9_1207.
- [25] K. R. Jayaram and Patrick Eugster. Scalable efficient composite event detection. In *Proceedings of the 12th International Conference on Coordination Models and Languages*, COORDINATION'10, pages 168–182, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-13413-0, 978-3-642-13413-5. doi: 10.1007/978-3-642-13414-2_12. URL http://dx.doi.org/10.1007/978-3-642-13414-2_12.
- [26] Alan H. Karp and Horace P. Flatt. Measuring parallel processor performance. *Commun. ACM*, 33(5):539–543, May 1990. ISSN 0001-0782. doi: 10.1145/78607.78614. URL <http://doi.acm.org/10.1145/78607.78614>.
- [27] Reza Sherafat Kazemzadeh and Hans-Arno Jacobsen. Partition-tolerant distributed publish/subscribe systems. In *Proceedings of the 2011 IEEE 30th International Symposium on Reliable Distributed Systems*, SRDS '11, pages 101–110, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-0-7695-4450-2. doi: 10.1109/SRDS.2011.21. URL <http://dx.doi.org/10.1109/SRDS.2011.21>.

- [28] Edward D Lazowska, John Zahorjan, G Scott Graham, and Kenneth C Sevcik. *Quantitative system performance: computer system analysis using queueing network models*. Prentice-Hall, Inc., 1984.
- [29] Sebastian Lehrig, Hendrik Eikerling, and Steffen Becker. Scalability, elasticity, and efficiency in cloud computing: A systematic literature review of definitions and metrics. In *Proceedings of the 11th International ACM SIGSOFT Conference on Quality of Software Architectures, QoSA '15*, pages 83–92, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3470-9. doi: 10.1145/2737182.2737185. URL <http://doi.acm.org/10.1145/2737182.2737185>.
- [30] Guoli Li, Vinod Muthusamy, and Hans-Arno Jacobsen. Adaptive content-based routing in general overlay topologies. In *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware, Middleware '08*, pages 1–21, New York, NY, USA, 2008. Springer-Verlag New York, Inc. ISBN 3-540-89855-7. URL <http://dl.acm.org/citation.cfm?id=1496950.1496952>.
- [31] Ming Li, Fan Ye, Minkyong Kim, Han Chen, and Hui Lei. A scalable and elastic publish/subscribe service. In *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 1254–1265, May 2011. doi: 10.1109/IPDPS.2011.119.
- [32] David Lizcano, Sebastian Ortega, Javier Soriano, and Sergio Vavassori. Explicitly context-aware publish/subscribe with context-invariant subscriptions. In *Proceedings of the 13th International Conference on Information Integration and Web-based Applications and Services, ii-WAS '11*, pages 367–370, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0784-0. doi: 10.1145/2095536.2095606. URL <http://doi.acm.org/10.1145/2095536.2095606>.
- [33] F. Maia, M. Matos, R. Oliveira, and E. Riviere. Slicing as a distributed systems primitive. In *Dependable Computing (LADC), 2013 Sixth Latin-American Symposium on*, pages 124–133, April 2013. doi: 10.1109/LADC.2013.21.
- [34] Ming Mao and Marty Humphrey. Auto-scaling to minimize cost and meet application deadlines in cloud workflows. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, SC '11*, pages 49:1–49:12, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0771-0. doi: 10.1145/2063384.2063449. URL <http://doi.acm.org/10.1145/2063384.2063449>.

- [35] Ming Mao and Marty Humphrey. A performance study on the vm startup time in the cloud. In *Proceedings of the 2012 IEEE Fifth International Conference on Cloud Computing, CLOUD '12*, pages 423–430, Washington, DC, USA, 2012. IEEE Computer Society. ISBN 978-0-7695-4755-8. doi: 10.1109/CLOUD.2012.103. URL <http://dx.doi.org/10.1109/CLOUD.2012.103>.
- [36] Ming Mao and Marty Humphrey. Scaling and scheduling to maximize application performance within budget constraints in cloud workflows. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing, IPDPS '13*, pages 67–78, Washington, DC, USA, 2013. IEEE Computer Society. ISBN 978-0-7695-4971-2. doi: 10.1109/IPDPS.2013.61. URL <http://dx.doi.org/10.1109/IPDPS.2013.61>.
- [37] Tova Milo, Tal Zur, and Elad Verbin. Boosting topic-based publish-subscribe systems with dynamic clustering. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data, SIGMOD '07*, pages 749–760, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-686-8. doi: 10.1145/1247480.1247563. URL <http://doi.acm.org/10.1145/1247480.1247563>.
- [38] Gero Mühl, Ludger Fiege, and Peter Pietzuch. *Distributed Event-Based Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006. ISBN 3540326510.
- [39] Yuan Ni, Chee-Yong Chan, Changqing Li, and Tok Ling. *Content-Based Publish/Subscribe for XML Data*, pages 207–226. IGI Global. doi: 10.4018/978-1-61520-727-5.ch010. URL <http://dx.doi.org/10.4018/978-1-61520-727-5.ch010>.
- [40] Alvin E Roth. *The Shapley value: essays in honor of Lloyd S. Shapley*. Cambridge University Press, Cambridge, 1988.
- [41] Peter Saint-Andre. Extensible messaging and presence protocol (xmpp): Core. RFC 6120, RFC Editor, March 2011. URL <http://www.rfc-editor.org/rfc/rfc6120.txt>. <http://www.rfc-editor.org/rfc/rfc6120.txt>.
- [42] Peter Saint-Andre. Extensible messaging and presence protocol (xmpp): Instant messaging and presence. RFC 6121, RFC Editor, March 2011. URL <http://www.rfc-editor.org/rfc/rfc6121.txt>. <http://www.rfc-editor.org/rfc/rfc6121.txt>.

- [43] Enrico Seib, Helge Parzyjegl, and Gero Mühl. Adaptive distributed composite event detection. In *Proceedings of the 11th International Workshop on Adaptive and Reflective Middleware*, ARM '12, pages 2:1–2:6, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1609-5. doi: 10.1145/2405679.2405681. URL <http://doi.acm.org/10.1145/2405679.2405681>.
- [44] L. S. Shapley. A value for n-person games. contribution to the theory of games. *Annals of Mathematics Studies*, 2:28, 1953.
- [45] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2nd edition, 2006. ISBN 0132392275.
- [46] Feng Tian, Berthold Reinwald, Hamid Pirahesh, Tobias Mayr, and Jussi Myllymaki. Implementing a scalable xml publish/subscribe system using relational database systems. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, pages 479–490, New York, NY, USA, 2004. ACM. ISBN 1-58113-859-8. doi: 10.1145/1007568.1007623. URL <http://doi.acm.org/10.1145/1007568.1007623>.
- [47] Sergio Vavassori, Javier Soriano, David Lizcano, and Miguel Jiménez. Explicit context matching in content-based publish/subscribe systems. *Sensors*, 13(3):2945, 2013. ISSN 1424-8220. doi: 10.3390/s130302945. URL <http://www.mdpi.com/1424-8220/13/3/2945>.
- [48] Sanghyun Yoo, Jin Hyun Son, and Myoung Ho Kim. An efficient subscription routing algorithm for scalable xml-based publish/subscribe systems. *J. Syst. Softw.*, 79(12):1767–1781, December 2006. ISSN 0164-1212. doi: 10.1016/j.jss.2006.03.039. URL <http://dx.doi.org/10.1016/j.jss.2006.03.039>.
- [49] Ye Zhao, Kyungbaek Kim, and Nalini Venkatasubramanian. Dynatops: A dynamic topic-based publish/subscribe architecture. In *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*, DEBS '13, pages 75–86, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1758-0. doi: 10.1145/2488222.2489273. URL <http://doi.acm.org/10.1145/2488222.2489273>.

Experiments' Raw Data Sets

In God we trust; all others must bring data.
— William Edwards Deming

In this chapter are listed the raw data files generated by the tests. The aim is to provide the source data, from which the graphs used in this thesis have been generated, in order to offer a more detailed understanding and the possibility to compare these data with other systems.

A.1 Context-Aware performance tests for SilboPS

These data files are the results of the tests that have been run on an Intel i7 860 @ 2.80 GHz machine equipped with 4 GiB of RAM running Linux 3.2 and OpenJDK 7u3.

A.1.1 Notification matching time for SIENA

Listing A.1: Measured matching time for notification using and increasing number of subscriptions for SIENA implementation.

```

1  # SIENA
2  # Matching time for Notification w.r.t. Subscription number
3  # <# of Subscription>    <# of Notification> Time[usec.]
4  # Time/<# of Notification> <msg/sec>    Time/<# of Subscription>
5  #
6  # Sent a total of 9 subscriptions on 1 connections.
7
8  10  10      1923      192 5200      192
9  10  20      627       31 31897      63
10 10  30      849       28 35335      85

```

11	10	40	995	25	40201	100
12	10	50	1387	28	36049	139
13	10	60	1636	27	36674	164
14	10	70	1652	24	42372	165
15	10	80	2359	29	33912	236
16	10	90	2301	26	39113	230
17	10	100	2960	30	33783	296
18	10	200	5278	26	37893	528
19	10	300	7285	24	41180	729
20	10	400	9133	23	43797	913
21	10	500	10254	21	48761	1025
22	10	600	12825	21	46783	1283
23	10	700	13926	20	50265	1393
24	10	800	9621	12	83151	962
25	10	900	8228	9	109382	823
26	10	1000	8657	9	115513	866
27	10	2000	16259	8	123008	1626
28	10	3000	22509	8	133280	2251
29	10	4000	23485	6	170321	2349
30	10	5000	29198	6	171244	2920
31	10	6000	29286	5	204876	2929
32	10	7000	69165	10	101207	6917
33	10	8000	19703	2	406029	1970
34	10	9000	18883	2	476619	1888
35	10	10000	20190	2	495294	2019
36	10	12500	24523	2	509725	2452
37						
38						
39	# Sent a total of 91 subscriptions on 9 connections.					
40						
41	100	10	306	31	32679	3
42	100	20	240	12	83333	2
43	100	30	387	13	77519	4
44	100	40	393	10	101781	4
45	100	50	594	12	84175	6
46	100	60	751	13	79893	8
47	100	70	699	10	100143	7
48	100	80	772	10	103626	8
49	100	90	45162	502	1992	452
50	100	100	1053	11	94966	11

51	100	200	2371	12	84352	24
52	100	300	3144	10	95419	31
53	100	400	4146	10	96478	41
54	100	500	4605	9	108577	46
55	100	600	5925	10	101265	59
56	100	700	6834	10	102429	68
57	100	800	7221	9	110787	72
58	100	900	7696	9	116943	77
59	100	1000	5678	6	176118	57
60	100	2000	10932	5	182949	109
61	100	3000	15957	5	188005	160
62	100	4000	14398	4	277816	144
63	100	5000	67564	14	74003	676
64	100	6000	21804	4	275178	218
65	100	7000	26040	4	268817	260
66	100	8000	75517	9	105936	755
67	100	9000	33653	4	267435	337
68	100	10000	36919	4	270863	369
69	100	12500	90620	7	137938	906
70						
71						
72	# Sent a total of 991 subscriptions on 117 connections.					
73						
74	1000	10	602	60	16611	1
75	1000	20	943	47	21208	1
76	1000	30	1424	47	21067	1
77	1000	40	1772	44	22573	2
78	1000	50	2949	59	16954	3
79	1000	60	4897	82	12252	5
80	1000	70	2615	37	26768	3
81	1000	80	3537	44	22618	4
82	1000	90	3561	40	25273	4
83	1000	100	4059	41	24636	4
84	1000	200	8444	42	23685	8
85	1000	300	12411	41	24172	12
86	1000	400	16069	40	24892	16
87	1000	500	20225	40	24721	20
88	1000	600	17223	29	34837	17
89	1000	700	13229	19	52914	13
90	1000	800	64239	80	12453	64

91	1000	900	14512	16	62017	15
92	1000	1000	16759	17	59669	17
93	1000	2000	33023	17	60563	33
94	1000	3000	49397	16	60732	49
95	1000	4000	116720	29	34270	117
96	1000	5000	80534	16	62085	81
97	1000	6000	142777	24	42023	143
98	1000	7000	160795	23	43533	161
99	1000	8000	173600	22	46082	174
100	1000	9000	145808	16	61725	146
101	1000	10000	201582	20	49607	202
102	1000	12500	243347	19	51366	243
103						
104						
105	# Sent a total of 9999 subscriptions on 1236 connections.					
106						
107	10000	10	1925	193	5194	0
108	10000	20	3042	152	6574	0
109	10000	30	4802	160	6247	0
110	10000	40	5887	147	6795	1
111	10000	50	6972	139	7171	1
112	10000	60	7345	122	8168	1
113	10000	70	9023	129	7757	1
114	10000	80	11927	149	6707	1
115	10000	90	10752	119	8370	1
116	10000	100	12817	128	7802	1
117	10000	200	24119	121	8292	2
118	10000	300	38795	129	7732	4
119	10000	400	50065	125	7989	5
120	10000	500	62990	126	7937	6
121	10000	600	74652	124	8037	8
122	10000	700	86313	123	8110	9
123	10000	800	101749	127	7862	10
124	10000	900	112096	125	8029	11
125	10000	1000	122443	122	8167	12
126	10000	2000	281234	141	7111	28
127	10000	3000	375382	125	7991	38
128	10000	4000	504698	126	7925	50
129	10000	5000	650034	130	7691	65
130	10000	6000	755807	126	7938	76

131	10000	7000	882858	126	7928	88
132	10000	8000	1021675	128	7830	102
133	10000	9000	1155889	128	7786	116
134	10000	10000	1274738	127	7844	127
135	10000	12500	1590843	127	7857	159
136						
137						
138	# Sent a total of 79999 subscriptions on 9979 connections.					
139						
140	80000	10	8334	833	1199	0
141	80000	20	27918	1396	716	0
142	80000	30	29590	986	1013	0
143	80000	40	36147	904	1106	0
144	80000	50	49300	986	1014	1
145	80000	60	59726	995	1004	1
146	80000	70	70513	1007	992	1
147	80000	80	76428	955	1046	1
148	80000	90	84195	935	1069	1
149	80000	100	91961	920	1087	1
150	80000	200	182217	911	1097	2
151	80000	300	309341	1031	969	4
152	80000	400	392186	980	1019	5
153	80000	500	492075	984	1016	6
154	80000	600	570767	951	1051	7
155	80000	700	663767	948	1054	8
156	80000	800	779985	975	1025	10
157	80000	900	833842	926	1079	10
158	80000	1000	967479	967	1033	12
159	80000	2000	1918431	959	1042	24
160	80000	3000	2769090	923	1083	35
161	80000	4000	3631150	908	1101	45
162	80000	5000	4571951	914	1093	57
163	80000	6000	5519459	920	1087	69
164	80000	7000	6310873	902	1109	79
165	80000	8000	7220933	903	1107	90
166	80000	9000	8142604	905	1105	102
167	80000	10000	9019380	902	1108	113
168	80000	12500	11299150	904	1106	141

Listing A.2: Measured matching time for notification using and increasing number of subscriptions for SilboPS implementation.

```

1 # Explicit C-A
2 # Matching time for Notification w.r.t. Subscription number
3 # <# of Subscription>    <# of Notification> Time[usec.]
4 # Time/<# of Notification> <msg/sec>    Time/<# of Subscription>
5 #
6 # Sent a total of 5 subscriptions on 2 connections.
7
8 10 10      1757      176 5691      176
9 10 20      553       28 36166     55
10 10 30      925       31 32432     93
11 10 40     1087       27 36798    109
12 10 50     1375       28 36363    138
13 10 60     1766       29 33975    177
14 10 70     1720       25 40697    172
15 10 80     1944       24 41152    194
16 10 90     2540       28 35433    254
17 10 100    2640       26 37878    264
18 10 200    5035       25 39721    504
19 10 300    6749       22 44451    675
20 10 400    8506       21 47025    851
21 10 500    9725       19 51413    973
22 10 600   11534       19 52020   1153
23 10 700   13685       20 51150   1369
24 10 800   16101       20 49686   1610
25 10 900   17350       19 51873   1735
26 10 1000  8002        8 124968   800
27 10 2000  15488       8 129132  1549
28 10 3000  21850       7 137299  2185
29 10 4000  27710       7 144352  2771
30 10 5000  32236       6 155106  3224
31 10 6000  33658       6 178263  3366
32 10 7000  69500      10 100719  6950
33 10 8000  22017       3 363355  2202
34 10 9000  23572       3 381808  2357
35 10 10000 23941       2 417693  2394
36 10 12500 26712       2 467954  2671
37

```



```

38
39 # Sent a total of 93 subscriptions on 10 connections.
40
41 100 10      289      29 34602  3
42 100 20      364      18 54945  4
43 100 30      422      14 71090  4
44 100 40      589      15 67911  6
45 100 50      666      13 75075  7
46 100 60      801      13 74906  8
47 100 70     34521     493 2027   345
48 100 80     1160      15 68965  12
49 100 90     1251      14 71942  13
50 100 100     1455      15 68728  15
51 100 200     3016      15 66312  30
52 100 300     4560      15 65789  46
53 100 400     6722      17 59506  67
54 100 500     7857      16 63637  79
55 100 600     7204      12 83287  72
56 100 700     7155      10 97833  72
57 100 800     8463      11 94529  85
58 100 900     9867      11 91213  99
59 100 1000    11299      11 88503  113
60 100 2000    23659      12 84534  237
61 100 3000    28554      10 105064 286
62 100 4000    24015      6 166562 240
63 100 5000    55330      11 90366  553
64 100 6000    26230      4 228745 262
65 100 7000    26886      4 260358 269
66 100 8000    62891      8 127204 629
67 100 9000    34625      4 259927 346
68 100 10000   85676      9 116718 857
69 100 12500   50468      4 247681 505
70
71
72 # Sent a total of 999 subscriptions on 124 connections.
73
74 1000      10      702      70 14245  1
75 1000      20     1024      51 19531  1
76 1000      30     1534      51 19556  2
77 1000      40     1966      49 20345  2

```

78	1000	50	2440	49	20491	2
79	1000	60	2486	41	24135	2
80	1000	70	3549	51	19723	4
81	1000	80	3972	50	20140	4
82	1000	90	4431	49	20311	4
83	1000	100	4484	45	22301	4
84	1000	200	9197	46	21746	9
85	1000	300	10324	34	29058	10
86	1000	400	9613	24	41610	10
87	1000	500	48403	97	10329	48
88	1000	600	10820	18	55452	11
89	1000	700	13064	19	53582	13
90	1000	800	14270	18	56061	14
91	1000	900	16448	18	54717	16
92	1000	1000	17511	18	57106	18
93	1000	2000	37332	19	53573	37
94	1000	3000	88892	30	33748	89
95	1000	4000	68227	17	58627	68
96	1000	5000	117760	24	42459	118
97	1000	6000	137392	23	43670	137
98	1000	7000	118686	17	58979	119
99	1000	8000	167537	21	47750	168
100	1000	9000	186874	21	48160	187
101	1000	10000	203238	20	49203	203
102	1000	12500	246777	20	50653	247
103						
104						
105	# Sent a total of 10000 subscriptions on 1293 connections.					
106						
107	10000	10	3032	303	3298	0
108	10000	20	5342	267	3743	1
109	10000	30	6698	223	4473	1
110	10000	40	8054	201	4966	1
111	10000	50	7766	155	6438	1
112	10000	60	8095	135	7411	1
113	10000	70	9406	134	7442	1
114	10000	80	9944	124	8045	1
115	10000	90	12109	135	7432	1
116	10000	100	13368	134	7480	1
117	10000	200	25470	127	7852	3

118	10000	300	38170	127	7859	4
119	10000	400	51874	130	7710	5
120	10000	500	67264	135	7433	7
121	10000	600	78178	130	7674	8
122	10000	700	89091	127	7857	9
123	10000	800	109307	137	7318	11
124	10000	900	148172	165	6074	15
125	10000	1000	130093	130	7686	13
126	10000	2000	267058	134	7489	27
127	10000	3000	398510	133	7528	40
128	10000	4000	531996	133	7518	53
129	10000	5000	682538	137	7325	68
130	10000	6000	805467	134	7449	81
131	10000	7000	969547	139	7219	97
132	10000	8000	1060890	133	7540	106
133	10000	9000	1197667	133	7514	120
134	10000	10000	1365331	137	7324	137
135	10000	12500	1649771	132	7576	165
136						
137						
138	# Sent a total of 79994 subscriptions on 10000 connections.					
139						
140	80000	10	14472	1447	690	0
141	80000	20	25757	1288	776	0
142	80000	30	39397	1313	761	0
143	80000	40	43012	1075	930	0
144	80000	50	46628	933	1072	1
145	80000	60	57585	960	1041	1
146	80000	70	66982	957	1045	1
147	80000	80	74709	934	1070	1
148	80000	90	83743	930	1075	1
149	80000	100	92777	928	1077	1
150	80000	200	192568	963	1038	2
151	80000	300	324814	1083	923	4
152	80000	400	415906	1040	961	5
153	80000	500	499151	998	1001	6
154	80000	600	615162	1025	975	8
155	80000	700	728181	1040	961	9
156	80000	800	812932	1016	984	10
157	80000	900	940199	1045	957	12

158	80000	1000	1029887	1030	970	13
159	80000	2000	2050947	1025	975	26
160	80000	3000	3047813	1016	984	38
161	80000	4000	3937015	984	1015	49
162	80000	5000	4962471	992	1007	62
163	80000	6000	5896480	983	1017	74
164	80000	7000	6857454	980	1020	86
165	80000	8000	7778825	972	1028	97
166	80000	9000	8758694	973	1027	109
167	80000	10000	9708571	971	1030	121
168	80000	12500	12041335	963	1038	151

A.1.2 Context change time for a single broker

Listing A.3: Measured time to perform 50 context changes with a single broker using SIENA.

```

1 # Time to perform 50 context changes with SIENA implementation
2 #
3 # <# of Subscription changes>    <# of connection changed>    Time[usec.]
4 # Time/# connection Time[ms]/#connection
5 #
6 # Sent a total of 95 subscriptions on 11 connections.
7
8 50 1 7661 7661 7.661
9 50 2 15371 7686 7.686
10 50 3 66073 22024 22.024
11 50 4 19826 4957 4.957
12 50 5 23320 4664 4.664
13 50 6 27281 4547 4.547
14 50 7 31438 4491 4.491
15 50 8 33668 4209 4.209
16 50 9 27572 3064 3.064
17 50 10 23676 2368 2.368
18 50 11 19562 1778 1.778
19
20
21 # Sent a total of 997 subscriptions on 130 connections.
22

```

23	50	1	19617	19617	19.617
24	50	2	43255	21628	21.628
25	50	3	62726	20909	20.909
26	50	4	48216	12054	12.054
27	50	5	9064	1813	1.813
28	50	6	10479	1747	1.747
29	50	7	12921	1846	1.846
30	50	8	16129	2016	2.016
31	50	9	16629	1848	1.848
32	50	10	16366	1637	1.637
33	50	20	29092	1455	1.455
34	50	30	77896	2597	2.597
35	50	40	41066	1027	1.027
36	50	50	41267	825	0.825
37	50	60	50426	840	0.84
38	50	70	95029	1358	1.358
39	50	80	65274	816	0.816
40	50	90	75280	836	0.836
41	50	100	118554	1186	1.186
42	50	130	107408	826	0.826

43

44

45 # Sent a total of 9999 subscriptions on 1243 connections.

46

47	50	1	8565	8565	8.565
48	50	2	16067	8034	8.034
49	50	3	26014	8671	8.671
50	50	4	33033	8258	8.258
51	50	5	40065	8013	8.013
52	50	6	48277	8046	8.046
53	50	7	56497	8071	8.071
54	50	8	64449	8056	8.056
55	50	9	73535	8171	8.171
56	50	10	81502	8150	8.15
57	50	20	162221	8111	8.111
58	50	30	232823	7761	7.761
59	50	40	245128	6128	6.128
60	50	50	308816	6176	6.176
61	50	60	372369	6206	6.206
62	50	70	426897	6099	6.099

63	50	80	527306	6591	6.591
64	50	90	546080	6068	6.068
65	50	100	611111	6111	6.111
66	50	200	1249750	6249	6.249
67	50	300	1864250	6214	6.214
68	50	400	2517219	6293	6.293
69	50	500	3147011	6294	6.294
70	50	600	3852492	6421	6.421
71	50	700	4457503	6368	6.368
72	50	800	4981629	6227	6.227
73	50	900	6097595	6775	6.775
74	50	1000	6240186	6240	6.24
75	50	1243	7778866	6258	6.258
76					
77					
78	# Sent a total of 79991 subscriptions on 9922 connections.				
79					
80	50	1	156752	48983	48.983
81	50	2	97108	48554	48.554
82	50	3	146091	48697	48.697
83	50	4	194727	48682	48.682
84	50	5	242714	48543	48.543
85	50	6	291744	48624	48.624
86	50	7	340683	48669	48.669
87	50	8	409023	51128	51.128
88	50	9	438498	48722	48.722
89	50	10	487101	48710	48.71
90	50	20	973565	48678	48.678
91	50	30	1459884	48663	48.663
92	50	40	1948755	48719	48.719
93	50	50	2434781	48696	48.696
94	50	60	2930600	48843	48.843
95	50	70	3406816	48669	48.669
96	50	80	3894122	48677	48.677
97	50	90	4470133	49668	49.668
98	50	100	4831391	48314	48.314
99	50	200	9673281	48366	48.366
100	50	300	14595201	48651	48.651
101	50	400	19463300	48658	48.658
102	50	500	24379200	48758	48.758

103	50	600	29017040	48362	48.362
104	50	700	33955967	48509	48.509
105	50	800	38878413	48598	48.598
106	50	900	44008967	48899	48.899
107	50	1000	50641700	50642	50.642
108	50	2000	99949583	49975	49.975
109	50	3000	150180073	50060	50.06
110	50	4000	200483219	50121	50.121
111	50	5000	250648077	50130	50.13
112	50	6000	301567065	50261	50.261
113	50	7000	352057336	50294	50.294
114	50	8000	402016756	50252	50.252
115	50	9000	452358045	50262	50.262
116	50	10000	503244188	50324	50.324

Listing A.4: Measured time to perform 50 context changes with a single broker using SilboPS.

```

1  # Context change test with single broker with Explicit context management
2  #
3  # <# of Context changes>      <# of connection changed>   Time[usec.]
4  # Time/# connection Time[ms]/#connection
5  #
6  # Sent a total of 100 subscriptions on 12 connections.
7
8  50  1   302      302 0.302
9  50  2   356      178 0.178
10 50  3   526      175 0.175
11 50  4   696      174 0.174
12 50  5   870      174 0.174
13 50  6  1070      178 0.178
14 50  7  1216      174 0.174
15 50  8  1401      175 0.175
16 50  9  1702      189 0.189
17 50 10  1762      176 0.176
18 50 12  2569      214 0.214
19
20
21 # Sent a total of 990 subscriptions on 129 connections.
```

22

23	50	1	220	220	0.22
24	50	2	407	204	0.204
25	50	3	593	198	0.198
26	50	4	785	196	0.196
27	50	5	981	196	0.196
28	50	6	1249	208	0.208
29	50	7	1499	214	0.214
30	50	8	1478	185	0.185
31	50	9	1633	181	0.181
32	50	10	1939	194	0.194
33	50	20	3463	173	0.173
34	50	30	4222	141	0.141
35	50	40	5576	139	0.139
36	50	50	6437	129	0.129
37	50	60	7819	130	0.13
38	50	70	9143	131	0.131
39	50	80	10518	131	0.131
40	50	90	41804	464	0.464
41	50	100	1402	14	0.014
42	50	129	1058	8	0.008

43

44

45 # Sent a total of 9991 subscriptions on 1284 connections.

46

47	50	1	19	19	0.019
48	50	2	15	8	0.008
49	50	3	22	7	0.007
50	50	4	29	7	0.007
51	50	5	36	7	0.007
52	50	6	42	7	0.007
53	50	7	53	8	0.008
54	50	8	59	7	0.007
55	50	9	115	13	0.013
56	50	10	83	8	0.008
57	50	20	161	8	0.008
58	50	30	215	7	0.007
59	50	40	348	9	0.009
60	50	50	358	7	0.007
61	50	60	430	7	0.007

62	50	70	571	8	0.008
63	50	80	719	9	0.009
64	50	90	622	7	0.007
65	50	100	690	7	0.007
66	50	200	1376	7	0.007
67	50	300	2189	7	0.007
68	50	400	3314	8	0.008
69	50	500	3742	7	0.007
70	50	600	4619	8	0.008
71	50	700	4992	7	0.007
72	50	800	5885	7	0.007
73	50	900	6548	7	0.007
74	50	1000	7274	7	0.007
75	50	1284	9788	8	0.008
76					
77					
78	# Sent a total of 79458 subscriptions on 10000 connections.				
79					
80	50	1	14	14	0.014
81	50	2	16	8	0.008
82	50	3	23	8	0.008
83	50	4	30	8	0.008
84	50	5	37	7	0.007
85	50	6	44	7	0.007
86	50	7	52	7	0.007
87	50	8	59	7	0.007
88	50	9	66	7	0.007
89	50	10	73	7	0.007
90	50	20	145	7	0.007
91	50	30	242	8	0.008
92	50	40	291	7	0.007
93	50	50	356	7	0.007
94	50	60	416	7	0.007
95	50	70	485	7	0.007
96	50	80	556	7	0.007
97	50	90	739	8	0.008
98	50	100	695	7	0.007
99	50	200	1385	7	0.007
100	50	300	2400	8	0.008
101	50	400	2703	7	0.007

102	50	500	3397	7	0.007
103	50	600	4336	7	0.007
104	50	700	4860	7	0.007
105	50	800	5424	7	0.007
106	50	900	6234	7	0.007
107	50	1000	7087	7	0.007
108	50	2000	84976	24	0.024
109	50	3000	20767	7	0.007
110	50	4000	27370	7	0.007
111	50	5000	34343	7	0.007
112	50	6000	107925	18	0.018
113	50	7000	48576	7	0.007
114	50	8000	121867	15	0.015
115	50	9000	129334	14	0.014
116	50	10000	68328	7	0.007

Listing A.5: Single context change with respect to the number of subscriptions per connection using SIENA and SilboPS implementation.

1	# Single context change w.r.t Subscriptions per each connection			
2	#			
3	# SIENA			
4	# <# of Subscription per connection> Time[usec.] Time[ms]			
5				
6	1	200	0.2	
7	2	247	0.247	
8	3	300	0.3	
9	4	450	0.45	
10	5	612	0.612	
11	6	665	0.665	
12	7	776	0.776	
13	8	981	0.981	
14	9	992	0.992	
15	10	1098	1.098	
16	20	2225	2.225	
17	30	3388	3.388	
18	40	4573	4.573	
19	50	5316	5.316	
20	60	6374	6.374	

21	70	7453	7.453
22	80	9100	9.1
23	90	10763	10.763
24	100	9678	9.678
25	200	20233	20.233
26	300	28567	28.567
27	400	33115	33.115
28	500	35774	35.774
29	600	43785	43.785
30	700	38771	38.771
31	800	29279	29.279
32	900	31969	31.969
33	1000	40770	40.77
34	2000	103960	103.96
35	3000	86953	86.953
36	4000	88599	88.599
37	5000	139225	139.225
38	6000	144090	144.09
39	7000	41198	41.198
40	8000	75005	75.005
41	9000	82449	82.449
42	10000	79437	79.437
43			
44			
45	# Explicit Context-Aware		
46	# <# of Subscription per connection>		Time[usec.]
47			
48	1	12	0.012
49	2	12	0.012
50	3	11	0.011
51	4	10	0.01
52	5	10	0.01
53	6	6	0.006
54	7	6	0.006
55	8	6	0.006
56	9	6	0.006
57	10	7	0.007
58	20	7	0.007
59	30	10	0.01
60	40	7	0.007

61	50	7	0.007
62	60	7	0.007
63	70	6	0.006
64	80	11	0.011
65	90	4	0.004
66	100	5	0.005
67	200	5	0.005
68	300	5	0.005
69	400	6	0.006
70	500	7	0.007
71	600	7	0.007
72	700	6	0.006
73	800	7	0.007
74	900	7	0.007
75	1000	7	0.007
76	2000	7	0.007
77	3000	8	0.008
78	4000	8	0.008
79	5000	9	0.009
80	6000	9	0.009
81	7000	10	0.01
82	8000	10	0.01
83	9000	10	0.01
84	10000	10	0.01

Listing A.6: Context change test with single broker and explicit management in SilboPS.

```

1 # Context change test with single broker and explicit management.
2 #
3 # <# of Context changes>      <# of connection changed>    Time[usec.]
4 # Time/# connection[usec]    Time/#connection [ms]
5 #
6 # Sent 100 subscriptions, index [0-4]
7
8 100      1    505      505      0.505
9 200      1    829      829      0.829
10 300      1   1227     1227     1.227
11 400      1   1750     1750     1.75

```

12	500	1	2077	2077	2.077
13	600	1	2461	2461	2.461
14	700	1	2932	2932	2.932
15	800	1	3880	3880	3.88
16	900	1	4043	4043	4.043
17	1000	1	4137	4137	4.137
18	2000	1	7815	7815	7.815
19	3000	1	12443	12443	12.443
20	4000	1	14022	14022	14.022
21	5000	1	43558	43558	43.558
22	6000	1	6182	6182	6.182
23	7000	1	3479	3479	3.479
24	8000	1	3962	3962	3.962
25	9000	1	3558	3558	3.558
26	10000	1	4017	4017	4.017
27					
28					
29	100	10	543	54	0.054
30	200	10	1250	125	0.125
31	300	10	1549	155	0.155
32	400	10	2048	205	0.205
33	500	10	3038	304	0.304
34	600	10	3562	356	0.356
35	700	10	3026	303	0.303
36	800	10	1145	115	0.115
37	900	10	1278	128	0.128
38	1000	10	1430	143	0.143
39	2000	10	2905	291	0.291
40	3000	10	4302	430	0.43
41	4000	10	5696	570	0.57
42	5000	10	7863	786	0.786
43	6000	10	34648	3465	3.465
44	7000	10	12444	1244	1.244
45	8000	10	13563	1356	1.356
46	9000	10	15069	1507	1.507
47	10000	10	40794	4079	4.079
48					
49					
50	100	100	1243	12	0.012
51	200	100	2542	25	0.025

52	300	100	3819	38	0.038
53	400	100	5355	54	0.054
54	500	100	7418	74	0.074
55	600	100	9444	94	0.094
56	700	100	9869	99	0.099
57	800	100	11610	116	0.116
58	900	100	36980	125	0.125
59	1000	100	14628	146	0.146
60	2000	100	31027	310	0.31
61	3000	100	70013	700	0.7
62	4000	100	61530	615	0.615
63	5000	100	91636	916	0.916
64	6000	100	109749	1097	1.097
65	7000	100	122974	1230	1.23
66	8000	100	142112	1421	1.421
67	9000	100	158866	1589	1.589
68	10000	100	178098	1781	1.781
69					
70					
71	100	1000	13077	13	0.013
72	200	1000	48538	49	0.049
73	300	1000	39136	39	0.039
74	400	1000	53879	54	0.054
75	500	1000	91250	91	0.091
76	600	1000	85460	85	0.085
77	700	1000	122648	123	0.123
78	800	1000	116137	116	0.116
79	900	1000	147861	148	0.148
80	1000	1000	163658	164	0.164
81	2000	1000	296537	297	0.297
82	3000	1000	443362	443	0.443
83	4000	1000	603661	604	0.604
84	5000	1000	744623	745	0.745
85	6000	1000	898907	899	0.899
86	7000	1000	1075406	1075	1.075
87	8000	1000	1184551	1185	1.185
88	9000	1000	1383804	1384	1.384
89	10000	1000	1611690	1612	1.612
90					
91					

92	100	10000	135629	14	0.014
93	200	10000	249953	25	0.025
94	300	10000	389873	39	0.039
95	400	10000	527364	53	0.053
96	500	10000	660247	66	0.066
97	600	10000	827406	83	0.083
98	700	10000	949980	95	0.095
99	800	10000	1106864	111	0.111
100	900	10000	1217617	122	0.122
101	1000	10000	1366848	137	0.137
102	2000	10000	2872510	287	0.287
103	3000	10000	4353263	435	0.435
104	4000	10000	5900898	590	0.59
105	5000	10000	7475423	748	0.748
106	6000	10000	8977743	898	0.898
107	7000	10000	10579925	1058	1.058
108	8000	10000	11979688	1198	1.198
109	9000	10000	13139423	1314	1.314
110	10000	10000	14599070	1460	1.46

111

112

113 # Sent 1000, index [5-9]

114

115	100	1	35	35	0.035
116	200	1	49	49	0.049
117	300	1	69	69	0.069
118	400	1	92	92	0.092
119	500	1	111	111	0.111
120	600	1	130	130	0.13
121	700	1	151	151	0.151
122	800	1	173	173	0.173
123	900	1	194	194	0.194
124	1000	1	216	216	0.216
125	2000	1	526	526	0.526
126	3000	1	800	800	0.8
127	4000	1	880	880	0.88
128	5000	1	1074	1074	1.074
129	6000	1	1279	1279	1.279
130	7000	1	1641	1641	1.641
131	8000	1	1609	1609	1.609

132	9000	1	1865	1865	1.865
133	10000	1	2002	2002	2.002
134					
135					
136	100	10	126	13	0.013
137	200	10	264	26	0.026
138	300	10	380	38	0.038
139	400	10	536	54	0.054
140	500	10	674	67	0.067
141	600	10	822	82	0.082
142	700	10	959	96	0.096
143	800	10	1106	111	0.111
144	900	10	1247	125	0.125
145	1000	10	1388	139	0.139
146	2000	10	3037	304	0.304
147	3000	10	4572	457	0.457
148	4000	10	6120	612	0.612
149	5000	10	7637	764	0.764
150	6000	10	9411	941	0.941
151	7000	10	11108	1111	1.111
152	8000	10	18488	1849	1.849
153	9000	10	20318	2032	2.032
154	10000	10	26972	2697	2.697
155					
156					
157	100	100	1187	12	0.012
158	200	100	2458	25	0.025
159	300	100	3794	38	0.038
160	400	100	5206	52	0.052
161	500	100	7118	71	0.071
162	600	100	8115	81	0.081
163	700	100	9534	95	0.095
164	800	100	11010	110	0.11
165	900	100	12052	121	0.121
166	1000	100	15239	152	0.152
167	2000	100	29240	292	0.292
168	3000	100	52229	522	0.522
169	4000	100	58969	590	0.59
170	5000	100	83619	836	0.836
171	6000	100	109567	1096	1.096

172	7000	100	121002	1210	1.21
173	8000	100	134473	1345	1.345
174	9000	100	161514	1615	1.615
175	10000	100	183663	1837	1.837
176					
177					
178	100	1000	12553	13	0.013
179	200	1000	31461	31	0.031
180	300	1000	38526	39	0.039
181	400	1000	53056	53	0.053
182	500	1000	69758	70	0.07
183	600	1000	83250	83	0.083
184	700	1000	98709	99	0.099
185	800	1000	113832	114	0.114
186	900	1000	124932	125	0.125
187	1000	1000	142673	143	0.143
188	2000	1000	300722	301	0.301
189	3000	1000	444395	444	0.444
190	4000	1000	599375	599	0.599
191	5000	1000	759484	759	0.759
192	6000	1000	932639	933	0.933
193	7000	1000	1052988	1053	1.053
194	8000	1000	1246244	1246	1.246
195	9000	1000	1374004	1374	1.374
196	10000	1000	1515151	1515	1.515
197					
198					
199	100	1000	12553	13	0.013
200	200	1000	31461	31	0.031
201	100	10000	123757	12	0.012
202	200	10000	247874	25	0.025
203	300	10000	378961	38	0.038
204	400	10000	521489	52	0.052
205	500	10000	670314	67	0.067
206	600	10000	810780	81	0.081
207	700	10000	958208	96	0.096
208	800	10000	1094301	109	0.109
209	900	10000	1255302	126	0.126
210	1000	10000	1364749	136	0.136
211	2000	10000	2886793	289	0.289

212	3000	10000	4377010	438	0.438
213	4000	10000	5882028	588	0.588
214	5000	10000	7308553	731	0.731
215	6000	10000	8873859	887	0.887
216	7000	10000	10268580	1027	1.027
217	8000	10000	11664412	1166	1.166
218	9000	10000	13032010	1303	1.303
219	10000	10000	14510761	1451	1.451

220

221

222 # Sent 9993, index [10-14]

223

224	100	1	33	33	0.033
225	200	1	45	45	0.045
226	300	1	62	62	0.062
227	400	1	82	82	0.082
228	500	1	100	100	0.1
229	600	1	122	122	0.122
230	700	1	136	136	0.136
231	800	1	155	155	0.155
232	900	1	172	172	0.172
233	1000	1	245	245	0.245
234	2000	1	481	481	0.481
235	3000	1	664	664	0.664
236	4000	1	874	874	0.874
237	5000	1	1073	1073	1.073
238	6000	1	1341	1341	1.341
239	7000	1	1551	1551	1.551
240	8000	1	1785	1785	1.785
241	9000	1	1802	1802	1.802
242	10000	1	2067	2067	2.067

243

244

245	100	10	125	13	0.013
246	200	10	249	25	0.025
247	300	10	382	38	0.038
248	400	10	524	52	0.052
249	500	10	668	67	0.067
250	600	10	884	88	0.088
251	700	10	958	96	0.096

252	800	10	1100	110	0.11
253	900	10	1252	125	0.125
254	1000	10	1500	150	0.15
255	2000	10	3168	317	0.317
256	3000	10	4769	477	0.477
257	4000	10	24687	620	0.62
258	5000	10	7709	771	0.771
259	6000	10	9363	936	0.936
260	7000	10	12294	1229	1.229
261	8000	10	23630	2363	2.363
262	9000	10	21465	2147	2.147
263	10000	10	17233	1723	1.723
264					
265					
266	100	100	1234	12	0.012
267	200	100	2370	24	0.024
268	300	100	3728	37	0.037
269	400	100	5103	51	0.051
270	500	100	6431	64	0.064
271	600	100	8294	83	0.083
272	700	100	9959	100	0.1
273	800	100	12430	124	0.124
274	900	100	13441	134	0.134
275	1000	100	14032	140	0.14
276	2000	100	28956	290	0.29
277	3000	100	47927	479	0.479
278	4000	100	67683	677	0.677
279	5000	100	74437	744	0.744
280	6000	100	100151	1002	1.002
281	7000	100	107772	1078	1.078
282	8000	100	142656	1427	1.427
283	9000	100	158155	1582	1.582
284	10000	100	176956	1770	1.77
285					
286					
287	100	1000	19332	19	s 0.019
288	200	1000	23913	24	0.024
289	300	1000	39912	40	0.04
290	400	1000	52334	52	0.052
291	500	1000	66141	66	0.066

292	600	1000	84545	85	0.085
293	700	1000	94086	94	0.094
294	800	1000	122095	122	0.122
295	900	1000	121906	122	0.122
296	1000	1000	136079	136	0.136
297	2000	1000	296044	296	0.296
298	3000	1000	453339	453	0.453
299	4000	1000	615000	615	0.615
300	5000	1000	751621	752	0.752
301	6000	1000	893495	893	0.893
302	7000	1000	1073103	1073	1.073
303	8000	1000	1196428	1196	1.196
304	9000	1000	1314008	1314	1.314
305	10000	1000	1498883	1499	1.499

306

307

308	100	10000	119756	12	0.012
309	200	10000	244253	24	0.024
310	300	10000	382628	38	0.038
311	400	10000	518648	52	0.052
312	500	10000	656048	66	0.066
313	600	10000	796881	80	0.08
314	700	10000	924625	92	0.092
315	800	10000	1068339	107	0.107
316	900	10000	1213104	121	0.121
317	1000	10000	1343689	134	0.134
318	2000	10000	2865296	287	0.287
319	3000	10000	4354514	435	0.435
320	4000	10000	5825235	583	0.583
321	5000	10000	7323965	732	0.732
322	6000	10000	8788597	879	0.879
323	7000	10000	10260866	1026	1.026
324	8000	10000	11680254	1168	1.168
325	9000	10000	13057445	1306	1.306
326	10000	10000	14482482	1448	1.448

327

328

329

330 # Sent a total of 99994 subscriptions on 12496 connections, index [15-20]

331

332	100	1	33	33	0.033
333	200	1	46	46	0.046
334	300	1	69	69	0.069
335	400	1	113	113	0.113
336	500	1	142	142	0.142
337	600	1	174	174	0.174
338	700	1	159	159	0.159
339	800	1	155	155	0.155
340	900	1	173	173	0.173
341	1000	1	192	192	0.192
342	2000	1	422	422	0.422
343	3000	1	651	651	0.651
344	4000	1	805	805	0.805
345	5000	1	998	998	0.998
346	6000	1	1189	1189	1.189
347	7000	1	1445	1445	1.445
348	8000	1	1587	1587	1.587
349	9000	1	1858	1858	1.858
350	10000	1	2038	2038	2.038
351					
352					
353	100	10	136	14	0.014
354	200	10	290	29	0.029
355	300	10	420	42	0.042
356	400	10	563	56	0.056
357	500	10	714	71	0.071
358	600	10	879	88	0.088
359	700	10	1074	107	0.107
360	800	10	1158	116	0.116
361	900	10	1305	131	0.131
362	1000	10	1475	148	0.148
363	2000	10	3011	301	0.301
364	3000	10	4814	481	0.481
365	4000	10	6253	625	0.625
366	5000	10	7786	779	0.779
367	6000	10	10487	1049	1.049
368	7000	10	11452	1145	1.145
369	8000	10	13119	1312	1.312
370	9000	10	16169	1617	1.617
371	10000	10	17332	1733	1.733

372

373

374	100	100	1277	13	0.013
375	200	100	2702	27	0.027
376	300	100	4048	40	0.04
377	400	100	5595	56	0.056
378	500	100	7002	70	0.07
379	600	100	8446	84	0.084
380	700	100	10250	103	0.103
381	800	100	11484	115	0.115
382	900	100	13420	134	0.134
383	1000	100	14861	149	0.149
384	2000	100	31184	312	0.312
385	3000	100	48751	488	0.488
386	4000	100	61027	610	0.61
387	5000	100	76743	767	0.767
388	6000	100	98907	989	0.989
389	7000	100	166212	1662	1.662
390	8000	100	132104	1321	1.321
391	9000	100	156819	1568	1.568
392	10000	100	207435	2074	2.074

393

394

395	100	1000	12851	13	0.013
396	200	1000	26163	26	0.026
397	300	1000	42306	42	0.042
398	400	1000	55957	56	0.056
399	500	1000	70469	70	0.07
400	600	1000	86352	86	0.086
401	700	1000	145117	145	0.145
402	800	1000	114404	114	0.114
403	900	1000	131943	132	0.132
404	1000	1000	193580	194	0.194
405	2000	1000	360937	361	0.361
406	3000	1000	546883	547	0.547
407	4000	1000	657385	657	0.657
408	5000	1000	925640	926	0.926
409	6000	1000	1021663	1022	1.022
410	7000	1000	1191935	1192	1.192
411	8000	1000	1348374	1348	1.348

412	9000	1000	1419954	1420	1.42
413	10000	1000	1604253	1604	1.604
414					
415					
416	100	10000	133784	13	0.013
417	200	10000	267399	27	0.027
418	300	10000	428978	43	0.043
419	400	10000	571403	57	0.057
420	500	10000	719935	72	0.072
421	600	10000	865214	87	0.087
422	700	10000	1035471	104	0.104
423	800	10000	1161148	116	0.116
424	900	10000	1327190	133	0.133
425	1000	10000	1461667	146	0.146
426	2000	10000	2975175	298	0.298
427	3000	10000	4620928	462	0.462
428	4000	10000	6148067	615	0.615
429	5000	10000	7604845	760	0.76
430	6000	10000	9232936	923	0.923
431	7000	10000	10881295	1088	1.088
432	8000	10000	12422676	1242	1.242
433	9000	10000	13919791	1392	1.392
434	10000	10000	15496131	1550	1.55
435					
436					
437	100	12496	165324	13	0.013
438	200	12496	344778	28	0.028
439	300	12496	531366	43	0.043
440	400	12496	723699	58	0.058
441	500	12496	903990	72	0.072
442	600	12496	1123801	90	0.09
443	700	12496	1271271	102	0.102
444	800	12496	1459166	117	0.117
445	900	12496	1640860	131	0.131
446	1000	12496	1827605	146	0.146
447	2000	12496	3679205	294	0.294
448	3000	12496	5669354	454	0.454
449	4000	12496	7544933	604	0.604
450	5000	12496	9498185	760	0.76
451	6000	12496	11527894	923	0.923

452	7000	12496	13556969	1085	1.085
453	8000	12496	15506724	1241	1.241
454	9000	12496	17394135	1392	1.392
455	10000	12496	19281353	1543	1.543

A.1.3 Network test

Listing A.7: Measured total context change time for a 5 broker network with one publisher per router, 10 filter per subscribers and a total of 100 subscribers.

```

1  # Network Test - Context Change
2  #
3  # 100 subcriptors. 1 publisher per router. 5 routers network.
4  # 10 Filters per subcriptors.
5  #
6  # <# context change> Time[usec] Time[ms]
7  #
8  # SIENA version
9
10 50      1747    1.747
11 100     2219    2.219
12 200     2567    2.567
13 500     4697    4.697
14 1000    6684    6.684
15 2000    15861   15.861
16 5000    36552   36.552
17 10000   78032   78.032
18
19
20 # Explicit Context-Aware
21
22 50      1348    1.348
23 100     1570    1.57
24 200     1592    1.592
25 500     1635    1.635
26 1000    2664    2.664
27 2000    3892    3.892
28 5000    9237    9.237

```

```

29 10000    20811    20.811

```

A.2 Elastic performance tests for E-SilboPS

These data files are the results of the tests that have been run on an Intel i7 860 @ 2.80 GHz machine equipped with 12 GiB of RAM running Linux 4.2 and OpenJDK 7u91.

A.2.1 Throughput tests

Listing A.8: Measured notifications time and throughput for 1-1-1 deploy.

```

1 # Subscriptions Notifications Time [ms] Throughput [notifications/sec]
2 # Layers configuration: 1 1 1
3
4 1000      100000  4485      22296
5 2000      100000  4618      21654
6 3000      100000  3770      26525
7 4000      100000  3743      26716
8 5000      100000  3944      25354
9 6000      100000  3698      27041
10 7000      100000  4624      21626
11 8000      100000  4106      24354
12 9000      100000  5035      19860
13 10000     100000  5032      19872
14 20000     100000  7378      13553
15 30000     100000  10417     9599
16 40000     100000  12486     8008
17 50000     100000  15331     6522
18 60000     100000  17751     5633
19 70000     100000  19763     5059
20 80000     100000  22291     4486
21 90000     100000  24782     4035
22 100000    100000  26926     3713
23 200000    100000  50357     1985
24 300000    100000  73930     1352
25 400000    100000  98600     1014
26 500000    100000  123532    809
27 600000    100000  149177    670

```

28	700000	100000	176761	565
29	800000	100000	203374	491
30	900000	100000	230193	434
31	1000000	100000	258271	387

Listing A.9: Measured notifications time and throughput for 1-2-1 deploy.

1	#	Subscriptions	Notifications	Time [ms]	Throughput	[notifications/sec]
2	#	Layers	configuration:	1	2	1
3						
4	1000	100000	5314	18818		
5	2000	100000	5268	18982		
6	3000	100000	4826	20721		
7	4000	100000	5486	18228		
8	5000	100000	5806	17223		
9	6000	100000	5085	19665		
10	7000	100000	5002	19992		
11	8000	100000	5292	18896		
12	9000	100000	5095	19627		
13	10000	100000	5152	19409		
14	20000	100000	6756	14801		
15	30000	100000	8104	12339		
16	40000	100000	9293	10760		
17	50000	100000	10983	9104		
18	60000	100000	12699	7874		
19	70000	100000	13810	7241		
20	80000	100000	15146	6602		
21	90000	100000	16675	5997		
22	100000	100000	18168	5504		
23	200000	100000	30769	3250		
24	300000	100000	43341	2307		
25	400000	100000	56625	1766		
26	500000	100000	70628	1415		
27	600000	100000	83139	1202		
28	700000	100000	97386	1026		
29	800000	100000	110986	901		
30	900000	100000	124906	800		
31	1000000	100000	138874	720		

Listing A.10: Measured notifications time and throughput for 1-2-2 deploy.

1	# Subscriptions	Notifications	Time [ms]	Throughput [notifications/sec]
2	# Layers configuration: 1 2 2			
3				
4	1000	100000	5179	19308
5	2000	100000	4853	20605
6	3000	100000	5120	19531
7	4000	100000	5358	18663
8	5000	100000	5030	19880
9	6000	100000	5428	18422
10	7000	100000	5111	19565
11	8000	100000	5314	18818
12	9000	100000	5476	18261
13	10000	100000	5352	18684
14	20000	100000	6888	14518
15	30000	100000	8175	12232
16	40000	100000	9568	10451
17	50000	100000	10333	9677
18	60000	100000	11321	8833
19	70000	100000	13010	7686
20	80000	100000	13639	7331
21	90000	100000	15144	6603
22	100000	100000	16326	6125
23	200000	100000	28542	3503
24	300000	100000	41616	2402
25	400000	100000	54920	1820
26	500000	100000	68887	1451
27	600000	100000	81229	1231
28	700000	100000	95571	1046
29	800000	100000	109122	916
30	900000	100000	123569	809
31	1000000	100000	138014	724

Listing A.11: Measured notifications time and throughput for 1-3-1 deploy.

1	# Subscriptions	Notifications	Time [ms]	Throughput [notifications/sec]
2	# Layers configuration: 1 3 1			
3				

4	1000	100000	7907	12647
5	2000	100000	7612	13137
6	3000	100000	7683	13015
7	4000	100000	7802	12817
8	5000	100000	7582	13189
9	6000	100000	7775	12861
10	7000	100000	7576	13199
11	8000	100000	7716	12960
12	9000	100000	7793	12832
13	10000	100000	7587	13180
14	20000	100000	7481	13367
15	30000	100000	8432	11859
16	40000	100000	9994	10006
17	50000	100000	12100	8264
18	60000	100000	13354	7488
19	70000	100000	14857	6730
20	80000	100000	15703	6368
21	90000	100000	16816	5946
22	100000	100000	17379	5754
23	200000	100000	25969	3850
24	300000	100000	34732	2879
25	400000	100000	43733	2286
26	500000	100000	52913	1889
27	600000	100000	62646	1596
28	700000	100000	72454	1380
29	800000	100000	82284	1215
30	900000	100000	91870	1088
31	1000000	100000	102658	974

Listing A.12: Measured notifications time and throughput for 1-4-1 deploy.

1	# Subscriptions	Notifications	Time [ms]	Throughput [notifications/sec]
2	# Layers	configuration:	1 4 1	
3				
4	1000	100000	9878	10123
5	2000	100000	9627	10387
6	3000	100000	9864	10137
7	4000	100000	9960	10040
8	5000	100000	9809	10194

9	6000	100000	9982	10018
10	7000	100000	9975	10025
11	8000	100000	9901	10099
12	9000	100000	9986	10014
13	10000	100000	10018	9982
14	20000	100000	10115	9886
15	30000	100000	10712	9335
16	40000	100000	11454	8730
17	50000	100000	13017	7682
18	60000	100000	14844	6736
19	70000	100000	16540	6045
20	80000	100000	18308	5462
21	90000	100000	19373	5161
22	100000	100000	20313	4922
23	200000	100000	27274	3666
24	300000	100000	33755	2962
25	400000	100000	40540	2466
26	500000	100000	47648	2098
27	600000	100000	55103	1814
28	700000	100000	62464	1600
29	800000	100000	69298	1443
30	900000	100000	76681	1304
31	1000000	100000	84870	1178

Listing A.13: Measured notifications time and throughput for 1-4-2 deploy.

1	# Subscriptions	Notifications	Time [ms]	Throughput [notifications/sec]
2	# Layers	configuration:	1 4 2	
3				
4	1000	100000	9283	10772
5	2000	100000	9306	10745
6	3000	100000	9206	10862
7	4000	100000	9351	10694
8	5000	100000	9434	10599
9	6000	100000	9393	10646
10	7000	100000	9545	10476
11	8000	100000	9474	10555
12	9000	100000	9596	10421
13	10000	100000	9632	10382

14	20000	100000	10224	9780
15	30000	100000	10730	9319
16	40000	100000	11585	8631
17	50000	100000	12281	8142
18	60000	100000	12815	7803
19	70000	100000	12749	7843
20	80000	100000	13835	7228
21	90000	100000	15561	6426
22	100000	100000	15830	6317
23	200000	100000	22224	4499
24	300000	100000	28561	3501
25	400000	100000	36264	2757
26	500000	100000	43234	2312
27	600000	100000	50551	1978
28	700000	100000	59353	1684
29	800000	100000	66604	1501
30	900000	100000	74346	1345
31	1000000	100000	82161	1217

Listing A.14: Measured notifications time and throughput for 1-5-1 deploy.

1	# Subscriptions	Notifications	Time [ms]	Throughput [notifications/sec]
2	# Layers configuration: 1 5 1			
3				
4	1000	100000	11982	8345
5	2000	100000	11970	8354
6	3000	100000	11885	8413
7	4000	100000	12122	8249
8	5000	100000	12076	8280
9	6000	100000	12396	8067
10	7000	100000	12189	8204
11	8000	100000	12089	8271
12	9000	100000	12388	8072
13	10000	100000	12450	8032
14	20000	100000	12624	7921
15	30000	100000	12942	7726
16	40000	100000	13853	7218
17	50000	100000	15095	6624
18	60000	100000	16660	6002

19	70000	100000	18518	5400
20	80000	100000	19763	5059
21	90000	100000	21889	4568
22	100000	100000	23693	4220
23	200000	100000	32073	3117
24	300000	100000	37537	2664
25	400000	100000	44349	2254
26	500000	100000	51015	1960
27	600000	100000	57681	1733
28	700000	100000	64461	1551
29	800000	100000	70877	1410
30	900000	100000	77991	1282
31	1000000	100000	85157	1174

Listing A.15: Measured notifications time and throughput for 1-5-2 deploy.

1	# Subscriptions	Notifications	Time [ms]	Throughput [notifications/sec]
2	# Layers configuration: 1 5 2			
3				
4	1000	100000	11823	8458
5	2000	100000	11748	8512
6	3000	100000	11721	8531
7	4000	100000	11714	8536
8	5000	100000	12028	8313
9	6000	100000	11908	8397
10	7000	100000	12030	8312
11	8000	100000	12228	8177
12	9000	100000	12268	8151
13	10000	100000	12184	8207
14	20000	100000	12890	7757
15	30000	100000	13788	7252
16	40000	100000	14293	6996
17	50000	100000	15231	6565
18	60000	100000	15443	6475
19	70000	100000	16249	6154
20	80000	100000	17107	5845
21	90000	100000	17692	5652
22	100000	100000	17920	5580
23	200000	100000	24937	4010

24	300000	100000	29777	3358
25	400000	100000	36428	2745
26	500000	100000	41834	2390
27	600000	100000	49115	2036
28	700000	100000	54722	1827
29	800000	100000	63364	1578
30	900000	100000	70127	1425
31	1000000	100000	76632	1304

Listing A.16: Measured notifications time and throughput for 1-6-1 deploy.

1	# Subscriptions	Notifications	Time [ms]	Throughput [notifications/sec]
2	# Layers configuration: 1 6 1			
3				
4	1000	100000	13896	7196
5	2000	100000	14288	6998
6	3000	100000	14150	7067
7	4000	100000	14363	6962
8	5000	100000	14047	7118
9	6000	100000	14239	7022
10	7000	100000	14286	6999
11	8000	100000	14272	7006
12	9000	100000	14426	6931
13	10000	100000	14391	6948
14	20000	100000	14604	6847
15	30000	100000	15251	6556
16	40000	100000	16757	5967
17	50000	100000	18690	5350
18	60000	100000	19957	5010
19	70000	100000	21840	4578
20	80000	100000	22964	4354
21	90000	100000	25442	3930
22	100000	100000	28074	3562
23	200000	100000	43518	2297
24	300000	100000	47984	2084
25	400000	100000	54029	1850
26	500000	100000	60095	1664
27	600000	100000	66196	1510
28	700000	100000	72560	1378

29	800000	100000	78898	1267
30	900000	100000	84640	1181
31	1000000	100000	91013	1098

Listing A.17: Measured notifications time and throughput for 1-6-2 deploy.

1	# Subscriptions	Notifications	Time [ms]	Throughput [notifications/sec]
2	# Layers configuration: 1 6 2			
3				
4	1000	100000	14295	6995
5	2000	100000	14255	7015
6	3000	100000	14354	6966
7	4000	100000	14427	6931
8	5000	100000	14448	6921
9	6000	100000	14568	6864
10	7000	100000	14764	6773
11	8000	100000	14735	6786
12	9000	100000	15028	6654
13	10000	100000	14911	6706
14	20000	100000	15929	6277
15	30000	100000	16723	5979
16	40000	100000	17527	5705
17	50000	100000	19236	5198
18	60000	100000	19085	5239
19	70000	100000	19829	5043
20	80000	100000	20406	4900
21	90000	100000	20677	4836
22	100000	100000	21982	4549
23	200000	100000	27959	3576
24	300000	100000	34190	2924
25	400000	100000	38821	2575
26	500000	100000	45647	2190
27	600000	100000	49673	2013
28	700000	100000	57521	1738
29	800000	100000	63128	1584
30	900000	100000	68919	1450
31	1000000	100000	75827	1318

114

Appendix A | Experiments' Raw Data Sets

#	Notification matching time in [ms] for 100000 notifications											
#	with different deploy configuration											
#												
#	subs	1-1-1	1-2-1	1-2-2	1-3-1	1-3-2	1-4-1	1-4-2	1-5-1	1-5-2	1-6-1	1-6-2
100000	26926	18168	16326	17379	14624	20313	15830	23693	17920	28074	21982	
200000	50357	30769	28542	25969	22868	27274	22224	32073	24937	43518	27959	
300000	73930	43341	41616	34732	31474	33755	28561	37537	29777	47984	34190	
400000	98600	56625	54920	43733	40916	40540	36264	44349	36428	54029	38821	
500000	123532	70628	68887	52913	50262	47648	43234	51015	41834	60095	45647	
600000	149177	83139	81229	62646	59743	55103	50551	57681	49115	66196	49673	
700000	176761	97386	95571	72454	69980	62464	59353	64461	54722	72560	57521	
800000	203374	110986	109122	82284	80065	69298	66604	70877	63364	78898	63128	
900000	230193	124906	123569	91870	89436	76681	74346	77991	70127	84640	68919	
1000000	258271	138874	138014	102658	99842	84870	82161	85157	76632	91013	75827	
# Notifications throughput speedup w.r.t. deployed instances												

```

20 # it's basically the transpose of the previous block with the computed division
21 #
22 # subs->100 k    200 k    300 k    400 k    500 k    600 k    700 k    800 k    900 k    1 M
23 #
24
25 1-1-1    1    1    1    1    1    1    1    1
26 1-2-1    1.482    1.637    1.706    1.741    1.749    1.794    1.815    1.832    1.843    1.860
27 1-2-2    1.649    1.764    1.776    1.795    1.793    1.836    1.850    1.864    1.863    1.871
28 1-3-1    1.549    1.939    2.129    2.255    2.335    2.381    2.440    2.472    2.506    2.516
29 1-3-2    1.841    2.202    2.349    2.410    2.458    2.497    2.526    2.540    2.574    2.587
30 1-4-1    1.326    1.846    2.190    2.432    2.593    2.707    2.830    2.935    3.002    3.043
31 1-4-2    1.701    2.266    2.588    2.719    2.857    2.951    2.978    3.053    3.096    3.143
32 1-5-1    1.136    1.570    1.970    2.223    2.421    2.586    2.742    2.869    2.952    3.033
33 1-5-2    1.503    2.019    2.483    2.707    2.953    3.037    3.230    3.210    3.283    3.370
34 1-6-1    0.959    1.157    1.541    1.825    2.056    2.254    2.436    2.578    2.720    2.838
35 1-6-2    1.225    1.801    2.162    2.540    2.706    3.003    3.073    3.222    3.340    3.406
36
37
38 # Efficiency for the previous speedup: Speedup / n. processors (a.k.a slices for us)
39 # see http://www.cs.utah.edu/~kirby/classes/cs6230/EagerSlides.pdf
40 # It has been calculated using the normalized value for slices since it has to be
41 # compared with the basic case 1-1-1 instead of the single-threaded version.
42 #
43 # subs->100 k    200 k    300 k    400 k    500 k    600 k    700 k    800 k    900 k    1 M Slices    Normalized

```

```

44 #
45
46 1-1-1  1  1  1  1  1  1  1  1  1  1  3  1
47 1-2-1  0.741  0.818  0.853  0.871  0.875  0.897  0.908  0.916  0.921  0.930  4  2
48 1-2-2  0.550  0.588  0.592  0.598  0.598  0.612  0.617  0.621  0.621  0.624  5  3
49 1-3-1  0.516  0.646  0.710  0.752  0.778  0.794  0.813  0.824  0.835  0.839  5  3
50 1-3-2  0.460  0.551  0.587  0.602  0.614  0.624  0.631  0.635  0.643  0.647  6  4
51 1-4-1  0.331  0.462  0.548  0.608  0.648  0.677  0.707  0.734  0.750  0.761  6  4
52 1-4-2  0.340  0.453  0.518  0.544  0.571  0.590  0.596  0.611  0.619  0.629  7  5
53 1-5-1  0.227  0.314  0.394  0.445  0.484  0.517  0.548  0.574  0.590  0.607  7  5
54 1-5-2  0.250  0.337  0.414  0.451  0.492  0.506  0.538  0.535  0.547  0.562  8  6
55 1-6-1  0.160  0.193  0.257  0.304  0.343  0.376  0.406  0.430  0.453  0.473  8  6
56 1-6-2  0.175  0.257  0.309  0.363  0.387  0.429  0.439  0.460  0.477  0.487  9  7
57
58
59 # Karp{Flatt metric to estimate the serial part and the overhead:  $e = (1/s - 1/p) / (1 - 1/p)$ 
60 # with s the speedup and p the number of processors
61 # https://en.wikipedia.org/wiki/Karp%E2%80%93Flatt\_metric
62 #
63 # It can't be applied to the case 1-1-1 because it's an indetermination of 0/0
64 # subs->100 k  200 k  300 k  400 k  500 k  600 k  700 k  800 k  900 k  1 M Slices  Normalized
65 #
66
67 1-2-1  0.350  0.222  0.172  0.149  0.144  0.115  0.102  0.092  0.085  0.075  4  2

```

68	1-2-2	0.410	0.350	0.345	0.336	0.337	0.317	0.311	0.305	0.305	0.302	5	3
69	1-3-1	0.468	0.274	0.205	0.165	0.142	0.130	0.115	0.107	0.099	0.096	5	3
70	1-3-2	0.391	0.272	0.234	0.220	0.209	0.201	0.195	0.192	0.185	0.182	6	4
71	1-4-1	0.672	0.389	0.275	0.215	0.181	0.159	0.138	0.121	0.111	0.105	6	4
72	1-4-2	0.485	0.302	0.233	0.210	0.188	0.174	0.170	0.159	0.154	0.148	7	5
73	1-5-1	0.850	0.546	0.385	0.312	0.266	0.233	0.206	0.186	0.173	0.162	7	5
74	1-5-2	0.598	0.394	0.283	0.243	0.206	0.195	0.172	0.174	0.166	0.156	8	6
75	1-6-1	1.051	0.837	0.579	0.458	0.384	0.332	0.293	0.265	0.241	0.223	8	6
76	1-6-2	0.786	0.481	0.373	0.293	0.264	0.222	0.213	0.195	0.183	0.176	9	7

A.2.3 Scale-out test

Listing A.19: Measured throughput during scale out from 1-1-1 to 1-3-1 while sending 2000 notifications/s and 1000 subscriptions/s. Start time is 10.01 s and End time is 12.27 s.

```

1 # Scale out from 1-1-1 to 1-3-1
2 # Input: 2000 notifications/s and 1000 subscriptions/s
3 # Start 10.014090263, End 12.275181866
4 # Time [s] Throughput [notifications/s]
5
6 0.0      1400
7 0.1      2600
8 0.2      2400
9 0.3      1800
10 0.4      2000
11 0.5      1000
12 0.6      1400
13 0.7      2200
14 0.8      2200
15 0.9      2000
16 1.0      1800
17 1.1      2000
18 1.2      2000
19 1.3      1800
20 1.4      1000
21 1.5      2000
22 1.6      2000
23 1.7      2000
24 1.8      2000
25 1.9      2000
26 2.0      1800
27 2.1      2000
28 2.2      2000
29 2.3      1800
30 2.4      2000
31 2.5      2000
32 2.6      2000
33 2.7      2000
34 2.8      2000

```

35	2.9	1800
36	3.0	2000
37	3.1	2000
38	3.2	1800
39	3.3	2000
40	3.4	2000
41	3.5	1800
42	3.6	2000
43	3.7	2000
44	3.8	2000
45	3.9	2000
46	4.0	2000
47	4.1	1800
48	4.2	2000
49	4.3	2000
50	4.4	2000
51	4.5	1800
52	4.6	2000
53	4.7	2000
54	4.8	2000
55	4.9	2000
56	5.0	2000
57	5.1	1600
58	5.2	1800
59	5.3	2000
60	5.4	2000
61	5.5	2000
62	5.6	2000
63	5.7	2000
64	5.8	1800
65	5.9	2000
66	6.0	1800
67	6.1	2000
68	6.2	2000
69	6.3	2000
70	6.4	2000
71	6.5	1800
72	6.6	2000
73	6.7	2000
74	6.8	2000

75	6.9	2000
76	7.0	2000
77	7.1	1800
78	7.2	2000
79	7.3	2000
80	7.4	2000
81	7.5	2000
82	7.6	1800
83	7.7	2000
84	7.8	2000
85	7.9	2000
86	8.0	1800
87	8.1	2000
88	8.2	2000
89	8.3	1800
90	8.4	2000
91	8.5	2000
92	8.6	2000
93	8.7	2000
94	8.8	2000
95	8.9	1400
96	9.0	2200
97	9.1	1800
98	9.2	2000
99	9.3	2000
100	9.4	2000
101	9.5	2000
102	9.6	2000
103	9.7	1800
104	9.8	2000
105	9.9	1800
106	10.0	1800
107	10.1	2000
108	10.2	200
109	10.3	2000
110	10.4	3800
111	10.5	2000
112	10.6	1600
113	10.7	2000
114	10.8	2000

115	10.9	2000
116	11.0	2000
117	11.1	1200
118	11.2	1600
119	11.3	1400
120	11.4	2800
121	11.5	2000
122	11.6	1600
123	11.7	1000
124	11.8	1400
125	11.9	800
126	12.0	3000
127	12.1	2600
128	12.2	1600
129	12.3	2000
130	12.4	1600
131	12.5	1000
132	12.6	2400
133	12.7	2000
134	12.8	2000
135	12.9	2000
136	13.0	2000
137	13.1	1800
138	13.2	2000
139	13.3	2000
140	13.4	2000
141	13.5	800
142	13.6	2000
143	13.7	2000
144	13.8	2000
145	13.9	2000
146	14.0	2000
147	14.1	2000
148	14.2	1800
149	14.3	2000
150	14.4	2000
151	14.5	1000
152	14.6	1800
153	14.7	2000
154	14.8	2000

155	14.9	2000
156	15.0	2000
157	15.1	1800
158	15.2	2000
159	15.3	2000
160	15.4	2000
161	15.5	800
162	15.6	1400
163	15.7	2600
164	15.8	2000
165	15.9	2000
166	16.0	1800
167	16.1	2000
168	16.2	2000
169	16.3	2000
170	16.4	2000
171	16.5	600
172	16.6	2000
173	16.7	2000
174	16.8	2000
175	16.9	1800
176	17.0	2200
177	17.1	1800
178	17.2	2000
179	17.3	2000
180	17.4	2000
181	17.5	1200
182	17.6	1600
183	17.7	2000
184	17.8	2000
185	17.9	1800
186	18.0	2000
187	18.1	2000
188	18.2	2000
189	18.3	2000
190	18.4	2000
191	18.5	1000
192	18.6	1600
193	18.7	2000
194	18.8	2000

195	18.9	2000
196	19.0	2000
197	19.1	1800
198	19.2	2000
199	19.3	2000
200	19.4	2000
201	19.5	2000
202	19.6	800
203	19.7	2000
204	19.8	2000
205	19.9	1800
206	20.0	2000
207	20.1	2000
208	20.2	2000
209	20.3	2000
210	20.4	2000
211	20.5	2000
212	20.6	800
213	20.7	1800
214	20.8	2000
215	20.9	2000
216	21.0	2000
217	21.1	2000
218	21.2	2000