UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

MASTER THESIS num. 592

A Dynamic and Elastic Publish-Subscribe Service for the Cloud Environment

Eugen Rožić

Zagreb, March 2017.

Umjesto ove stranice umetnite izvornik Vašeg rada.
Da bi ste uklonili ovu stranicu obrišite naredbu `\izvornik`.

Hvala mentorici Ivani Podnar Žarko na vremenu, dostupnosti i entuzijazmu bez kojeg ovaj rad nebi bio ni približno ovoliko koristan i zanimljiv.

Hvala također i znanstvenom novaku Aleksandru Antoniću na dobrom društvu i savjetima te znanstvenom novaku Valteru Vasiću na tehničkoj podršci.

Najviše od svega hvala Mariji Brbić bez koje bi izrada ovog rada bila beskonačno dosadnija i tmurnija.

CONTENTS

1. Introduction

In today's world, people with as much as a computer and a connection to the Internet have access to huge amounts of information that are being created every second, mostly as users of various services and applications. Not only do users have access to vast amounts of information, but they are often either bombarded with it or it is presented to them in a manner that doesn't match their interests, which then generates a negative user opinion of those services and applications. The interest of information providers is, naturally, to have satisfied users, and for that they have to try to offer personalized information to each of them.

Another side of the information supply and demand problem is that today almost anyone can afford a mobile device that only resembles a cell phone, but is actually more of a general-purpose computer (a so-called *smartphone*). The speed of today's mobile networks, combined with the low cost of bandwidth and the capabilities of mobile devices, gave rise to a trend of users being online all the time. Users now expect information to be delivered to their mobile devices as soon as possible (in real-time), and they also expect information providers to take into account current user context, e.g., a user's current location, when delivering the content.

A user's mobile device also became a useful source of information for other users or the applications and services themselves. That information ranges from user location to anything that a sensor in the user's BAN[1] network can measure and send to the user's mobile device via a communication protocol like Bluetooth. The result of all these circumstances and technological advances is a huge amount of users that can be online all the time, but can also go offline at any time, a huge amount of data that flows through the system, both to and from the users, and streams of data that can be generated in real-time and have to be processed in real-time.

One of the more popular approaches to the aforementioned problem of personalized information is the publish-subscribe communication model, which enables users

[1] BAN - Body Area Network

to define their interests, and receive, in real-time, the information that satisfies their interests whenever they are online. It also enables users to easily be both receivers and publishers of information. The problems of processing huge amounts of information and managing large numbers of users are often solved with the currently popular cloud computing paradigm. The purpose of this master thesis is to explore the publish-subscribe communication model and the cloud computing paradigm, and to develop an elastic and dynamic service, based on the two technologies, that would be a model for efficient handling of the described challenges.

The thesis is organised as follows. Chapter 2 gives an overview and describes the basic features of the publish-subscribe communication model. It also provides a more detailed section on the BE-Tree algorithm which was used in the developed system. Chapter 3 explains the basics of the cloud computing paradigm. Chapter 4 previews the state-of-the-art in cloud-based publish-subscribe systems. Chapter 5 gives a detailed description of the architecture of the developed system, along with the algorithms used in it, while Chapter 6 explains the implementation details. The thesis ends with an analysis of the performed experimental evaluation followed by a conclusion.

2. Publish-Subscribe communication

The Publish-subscribe communication model is, in essence, a messaging pattern that is based on the idea that information destinations are not interested in all the information available, but only information that satisfies certain constraints, and that information sources do not necessarily need to know exactly who receives the information they are sending, as long as it is received by the ones who need and/or want it. Information sources are called publishers, because they give out information into the system without knowing who exactly will receive it. In one word, they publish it. Information destinations are called subscribers, because they express their interests for certain types of information they want to receive by making simple, continuous queries over the system. In one word, subscriptions. The communication between publishers and subscribers is persistent and asynchronous. This enables a subscriber to be disconnected while remaining registered, which means that messages matching the subscriber's subscriptions will be saved and delivered to it as soon as it reconnects.

The process of comparing messages to defined subscriptions is called publication matching, and is explained in more detail in Section 2.1. Publish-subscribe systems can be categorized by the type of constraints that subscriptions define. The two most common types of publish-subscribe systems are: topic-based and content-based. In topic-based systems, publications are published to named logical channels, i.e. topics, while subscribers are subscribed to certain topics, and receive all the publications that belong to those topics. In content-based systems, subscribers define subscriptions as collections of constraints on the attributes and/or content of publications, and they receive all publications that satisfy those constraints. Publishers can publish two different types of publications: structured and unstructured. Structured publications are essentially data that has values assigned to named attributes, while unstructured publications consist of "raw" data like, for example, a piece of text.

A common entity in the topologies of publish-subscribe systems is a broker - an intermediary to which publishers post publications and subscribers register subscriptions. Usually brokers perform the task of publication matching, and are used to store

publications in queues for local subscribers that are temporarily disconnected. In distributed systems they also route publications from one broker to the other on their path from the originating publishers to all the subscribers. Brokers are the main components of a publish-subscribe system, and they enable the scalability of the system.

The main advantages of the publish-subscribe messaging pattern are loose coupling and scalability. Loose coupling of system components that are communicating means that they do not have to be aware of each other or the system topology in order to communicate. That allows the system topology to be dynamic, and information sources and destinations to be independent of each other. The loose coupling property enables publish-subscribe systems to be scalable, because they can be seamlessly distributed over different kinds of large networks. Moreover, in such architectures parallelization of operations can be easily implemented, and message caching is natively supported. However, the scalability deteriorates when publish-subscribe communication is used in a tightly-coupled, high-volume enterprise environment where the system is, for example, a data center with thousands of servers sharing the publish-subscribe infrastructure. In those cases problems like instability in the throughput, slowdowns, and IP broadcast storms may appear. The use of the publish-subscribe architecture in these circumstances is currently a research challenge.

Systems based on publish-subscribe also lack stronger properties on an end-to-end communication basis, like the guarantee of delivery (there is no way a subscriber can notify a publisher) and information about other entities (a publisher can't find out if a subscriber has failed or crashed), but there are a lot of systems where those properties are traded for the advantages of a loosely coupled system, or a service that enables users to specify their interests and receive only information that satisfies them, reducing the overall communication cost.

The publish-subscribe communication model is a sibling to the message queue model. They are usually both supported by message-oriented middleware like the Java Messaging System (JMS), which was among the first to offer those communication models more than a decade ago. Today, publish-subscribe is a well established communication model for push-based dissemination of data. It is most often used for efficient messaging in real-time, because it has the ability to process large amounts of data due to simple subscription languages and efficient matching algorithms. This is why publish-subscribe is almost exclusively used as the communication platform for the increasingly popular and important Data Stream Processing, which is defined as "a novel computing paradigm particularly suited for application scenarios where massive amount of data must be processed with small delay," in [3]. The authors of the arti-

cle add that: “Rather than processing stored data like in traditional database systems, SPEs[1] process tuples on-the-fly. This is due to the amount of input that discourages persistent storage and the requirement of providing prompt results,” which perfectly fits the publish-subscribe communication model.

2.1. Matching

Matching a publication against a set of subscriptions is computationally the most demanding task in a publish-subscribe system, and therefore limits its runtime performance and throughput, especially in centralised systems. Due to that fact, matching algorithms and supporting data structures are one of the focuses of research when it comes to publish-subscribe systems and data stream processing.

There are different approaches to the matching problem. Some of the approaches focus on finding better subscription indexing structures that would support faster matching algorithms, while other approaches try to optimize algorithms for specific systems or situations, building their solutions on assumptions which make the matching process simpler, or focus on taking advantage of the abilities of special hardware like graphical processing units [6], [1]. Also, a special case that is of high current interest is taking into account the information about the location of a subscriber. A location-based publish-subscribe system should support subscriptions that use location information and that are constantly changing because the subscriber is mobile. For this use structures like R-trees [5] and its variations can be used for subscription indexing. An example of location-based publish-subscribe matching is proposed in [1].

Current matching algorithms can be roughly divided into two main categories: counting-based and tree-based approaches. Another possible categorization can be to key-based and non key-based approaches, where the former use sets of subscription constraints as identifiers of individual subscriptions.

Counting-based approaches are based on the idea of reducing subscription constraint evaluations by building an inverted index over all unique subscription constraints, while tree-based methods try to achieve the same thing by recursively dividing the search space on encountering unsatisfiable constraints, and therefore eliminating evaluation of whole subsets of subscriptions.

Most of the currently developed matching algorithms have problems with either

[1] SPE - Stream Processing Engine

scaling to large numbers of subscriptions, high dimensionality of subscription space, or dynamicity of the subscriptions. Most of them successfully deal with two out of the three stated matching aspects, which makes their performance satisfiable for use in a wide variety of applications where the third aspect is not very relevant. However, in a general-purpose publish-subscribe system all three aspects should be addressed and dealt with. There are perhaps only a few algorithms that rise to that challenge, and BE-Tree is one that has been very recently developed, and has seemingly generated some very good results and an interest among researchers.

2.2. The BE-Tree algorithm

The BE-Tree data structure and matching algorithm, described in [8], is based on the idea of exploiting the discreteness and finiteness of the many-dimensioned attribute space which is usually present in (stream) data processing systems for which the BE-Tree is intended, e.g., publish-subscribe or complex event processing (CEP) systems. Each possible subscription attribute represents one dimension, and the domain of each dimension is the set of all possible values the attribute that represents that dimension can assume. BE-Tree is a dynamic structure, which means that it supports inserting and removing of expressions during it's use, and even adaptation to the workload changes. The only restrictive requirement of the BE-Tree design is that, in order for it to exploit the discreteness and finiteness of the attribute space, all possible attributes used in the system, along with all their allowed values, have to be known and listed before using the system, and have to be known to all the clients of the system.

The **BE** in BE-Tree stands for "**Boolean Expression**". Boolean expressions are universal entities which can be used to accurately represent objects of various (stream) data processing systems, and which allow a lot of expressiveness in defining relations among them. Generally, a Boolean expression is a conjunction of Boolean predicates, and a Boolean predicate is a triplet consisting of a variable name (an attribute), a value and an operator which defines the relation of the variable to the given value. For example, in publish-subscribe systems publications represent events, actualisations of some sort, and are as such represented by a Boolean expression in which the attribute name of each predicate is related to its respective value by the *equals* operator. Subscriptions are also represented by a Boolean expression, whose predicates define conditions on the values of the corresponding attributes, i.e. a multi-dimensional subspace.

BE-Tree is an n-ary tree structure, in which a leaf node contains a set of Boolean expressions, and an internal node contains partial predicate information about the ex-

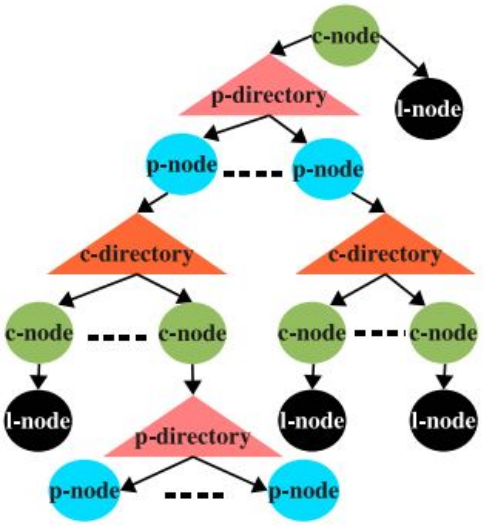

Figure 2.1: The structure of a BE-Tree

pressions in its descendant leaf nodes. The general structure of the BE-Tree is shown in Figure 2.1. In essence, it consists of 2 types of nodes: c-nodes and p-nodes. A third type of node is an l-node which is the only possible leaf node of the tree, and only an l-node contains the actual Boolean expressions stored in the tree, but since every c-node has to have exactly one l-node it is implementationally logical to put the expressions in a list, or a similar structure, within a c-node, and not in a separate node structure/class.

A p-node contains a single attribute name, which represents space partitioning information since all the expressions below that p-node will have an expression defined over that attribute. A c-node is defined by a range of values of the attribute represented by the c-node's parent p-node. It represents space clustering information in the sense that all the expressions below a c-node will have a predicate such that values satisfying that predicate will be from the range defined by that c-node. The p-nodes and c-nodes alternate in the BE-Tree structure, and are organized in directories, namely p-directories and c-directories. The organization of the directories is shown in Figure 2.2.

Since each c-node can only have one p-node for one attribute, the logical choice for p-directory organization is a hash table. This greatly speeds up operations over the BE-Tree because accessing a hash table is $O(1)$, which means that the dimensionality of the space is irrelevant, and is exactly why BE-Tree is perfectly suited for high-dimensional matching. The organization of c-directories is more complex because expressions (predicates) can both span over the entire domain of an attribute or just a single element. A c-directory is therefore organized as a binary tree structure of

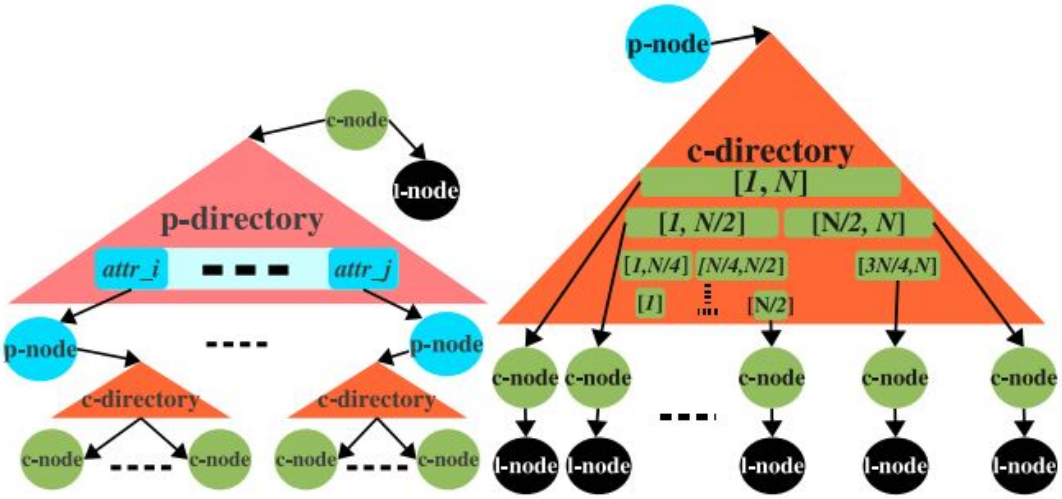

Figure 2.2: The structure of the BE-Tree directories

buckets. Each of the buckets represent a segment of the p-node attribute's domain and is joined with a c-node. The root bucket of a c-directory spans over the entire domain of the attribute, and each bucket can have two children buckets, left and right, that span over the first and second half of the parent bucket's range.

The key feature of the BE-Tree is it's two-phase space-cutting technique of building the tree. That approach significantly reduces the complexity and the level of uncertainty of choosing an effective criterion to recursively cut the space, and to identify highly dense subspaces. The two phases are: *space partitioning* and *space clustering*.

Space partitioning is the first space-cutting phase, and is based on the discriminative power of an attribute in a set of Boolean expressions. The space partitioning phase is triggered when an l-node overflows (the number of expressions in it exceeds a given threshold level). It uses a scoring function to determine the best attribute for partitioning, among the ones used in the Boolean expressions of the overflowed l-node, and then constructs a new p-node over that attribute, adds it to the p-directory, and transfers all the expressions, that have a predicate defined over the chosen attribute, from the overflowing l-node to the newly created p-node (which then places them in the appropriate c-nodes in its c-directory). In order to avoid cyclic partitioning, an attribute can be used for partitioning at most once in any path from the root node to a leaf node. However, the space-clustering phase makes sure that a single choice of an attribute for partitioning is enough to fully exploit its domain, and that reselection of the same attribute lower in the tree would provide no additional benefits.

Space clustering is the second phase in the space-cutting technique, and the key part of the BE-Tree idea, because it takes advantage of the discreteness and finiteness of a

dimension, and successfully deals with the "cascading split problem" described in [2]. It is basically an interval indexing algorithm that, due to the space partitioning phase, has to deal with only a single dimension. The key issues this phase solves are making a decision on how to cluster the domain of an attribute, and how to alternate the two phases. The policy for making those decisions is deterministic and independent of the insertion sequence, therefore it is completely predictable and ensures avoiding the cascading split problem. Otherwise, dilemmas would arise over whether to further pursue one phase or switch to the other, and whether there would be further gain in partitioning the space over an attribute that was already chosen higher in the tree.

The basic idea is to pursue space clustering as much as possible before switching back to space partitioning, and this is possible only because of the finiteness and the discreteness of the attribute domains. So when the l-node of a c-node of a leaf c-directory bucket overflows, two children buckets are created and are given the first and second half of the parent bucket's range. All the expressions from the overflowing l-node, that "fit" into either the left or the right child bucket, are then transferred to the c-node's l-node of the child bucket they fit into. This process is repeated until a bucket is reached that cannot be split further simply because it's range is a single value in the attribute domain (and that can always be achieved because the attribute domains have to be discrete). Every new Boolean expression being inserted is always inserted in the smallest possible existing enclosing c-directory bucket (*the insertion rule*). This way maximal space clustering is achieved before resorting to space partitioning (*the forced split rule*), which ensures that there can be no further gain from partitioning and clustering over the same attribute lower in the tree. Space partitioning is therefore triggered in two cases: 1) When an l-node of a c-node of an *atomic* c-directory's bucket overflows, and 2) When an l-node of a c-node of an already clustered (already has children buckets) c-directory's bucket overflows. Once a c-node of c-directory bucket is partitioned, it cannot be removed nor be merged with its children buckets (*the merge rule*). These three rules, along with the cost-based ranking function used for the BE-Tree self-adjustment (explained in detail in [8]), are the basis of the BE-Tree, and what makes it one of the currently best and most popular Boolean expression indexing structures and matching algorithms.

3. The “Cloud computing” paradigm

Cloud computing doesn’t have a precise technical definition, but it can perhaps be perceived as a common name for every service that offers any form of computational resource to a user, while hiding from him/her the hardware on which it resides and the manner in which it is achieved. This, of course, implies that access to the “cloud”, whatever service it might offer, has to be over a network. The cloud is usually thought of as something in the network that has an infinite amount of processing power and other resources for which, in order to get as much of them as you want or need, you just have to ask and maybe pay.

Clouds can be divided by type based on the network in which they are located and, consequently, by the people who have access to their resources. Public clouds can be accessed from a public network, usually the Internet, and everyone has access to them. They are typically backed by server farms of huge companies, like Google, Microsoft or Amazon, and offer various general-purpose cloud services. Private clouds can only be accessed from private networks of the organizations that own them. A private cloud is used inside an organization as a centralized resource for everyone, usually meant for computationally demanding tasks. Such centralization of resources enables them to be far more effectively used and managed than if, for example, everyone in the organization that occasionally needs a lot of computational power buys an expensive machine. The third basic type of clouds are the hybrid clouds. A cloud can be hybrid in a few ways. One type of hybrid cloud is a private cloud that when in need for extra resources gets them from a public cloud. Another type of hybrid cloud is a private cloud that can offer some of its resources publicly or to another organization.

There are three basic cloud service models: infrastructure as a service (IaaS), platform as a service (PaaS) and software as a service (SaaS). Infrastructure as a service is based on providing the users with virtual “raw” resources, like a virtual machine, which then the user uses any way he/she wants, and can ask for more resources or give up using any amount of them at any time. This enables the user to have any amount

of general-purpose computational resources on demand, while paying for just the time and the amount he/she used. One of the first, and certainly the most important, to offer this kind of service model was Amazon. The company realised it had lots of computational power which was mostly only about 10% used for their purposes, and decided to offer the rest through the Elastic Cloud Compute (EC2) web service using the IaaS model. Later, other big companies offered similar services, like the Google's Compute Engine and Microsoft's Azure Services Platform. In 2008 Eucalyptus and OpenNebula, the first open-source software for deploying private and hybrid IaaS clouds, were released.

Platform as a service model usually offers a lot of preconfigured applications and tools, and often a programming language and a special API for it. This kind of service makes achieving specific purposes, for which it was made, extraordinarily easy, while offering performance that would otherwise be either impossible or extremely expensive to have, and at the same time charging them little in comparison to the capital costs that would otherwise be required, and only as much as the user uses. An example of such a service is Google's App Engine, which is used to deploy web applications. It offers an API for Java and Python for writing web applications, which enables very simple use of various complex services, e.g., mail or data storage. Web applications are run inside the cloud with only a single click of a mouse, and their requested performances are maintained regardless of the workload using automatic scaling. A similar PaaS service is Microsoft's Windows Azure Cloud Services.

Software as a service model is often referred to as "software-on-demand", and its typical representative is the Google Apps which offers a number of web applications that look like and act as desktop applications, but they aren't. For example, Gmail is a mail client which a user uses just like a desktop mail client, but the user doesn't know where his/her electronic mail is actually located nor does he/she have any actual control over it.

A publish-subscribe service in a cloud would have to use an IaaS cloud service for its functioning, or alternatively be developed in a cloud itself, because it needs "infinite" resources to scale to any amount of workload. It needs "infinite" resources because it will inherently have a centralised structure with only one broker, located in the cloud, that can grow how ever much it needs and wants to. Once developed, the publish-subscribe service itself could be viewed as either a SaaS or a PaaS cloud service.

It could be viewed a SaaS cloud service if it would be a predefined and started web

application to which a user, or a device (like a sensor), can connect to. The users could then define subscriptions or publish publications on it, and the service itself would just do the matching and deliver notifications to the appropriate subscribers.

It could also be viewed as a PaaS if it would be a library with which a user could setup his/her own publish-subscribe service in a cloud. In that case a user would be given an API with which he/she could define a publish-subscribe service as he/she wants, and then run it in the cloud as a web application.

4. State-of-the-art in cloud-based publish-subscribe systems

A big part of current publish-subscribe and stream processing research is still oriented towards distributed systems, but researchers have started to use the cloud approach as well. It still appears not to be the mainstream approach, but there seems to be a rise in papers that describe systems which use a cloud in various combinations with publish-subscribe systems and stream processing engines.

One very specific use of a publish-subscribe system in combination with a cloud is described in [10]. There a topic-based publish-subscribe system is used for updating shared data that is located in a specially designed layered cloud. Each layer of the cloud contains some shared data and corresponds to a topic. The users of the data in the cloud are both publishers and subscribers. They subscribe to the topics/layers that contain data that they want to receive updates on, and they publish the changes they made to the data they had locally.

Another aspect of using clouds was addressed in [4] where a stream processing engine is expanded to the cloud if necessary. Amazon's EC2 cloud service is used as the cloud infrastructure, and System-S from IBM is used as the basis for a parallel stream processing engine that can dynamically create additional processing modules to process streams in parallel. When local processing resources become low, virtual machines can be automatically started in Amazon's cloud and additional System-S processing modules can be started on them. The system then becomes distributed, and processes the same streams in parallel both locally and within the cloud. The results of stream processing are then collected from both the local machine and the cloud, and are merged before notifying subscribers. The problem that the paper was addressing was the optimization of the use of cloud resources with respect to the performance requested by an SLA[1] and the cost of the used cloud resources.

[1] SLA - Service Level Agreement

Very few general-purpose cloud-based publish-subscribe solutions have been published or architectures for such systems suggested. One example of such a system is the BlueDove service developed at IBM [7], and another is the cloud-based publish-subscribe service developed at the Faculty of Electrical Engineering and Computing of the University of Zagreb as part of a master thesis [9].

BlueDove is an elastic and scalable content-based publish-subscribe service running in the cloud. It is based on a two-tier architecture of dispatcher and matcher components, and uses a special subscription space partitioning technique that enables it to have smart redundancy for server fault tolerance, and to exploit the skewness in the data distribution. The dispatchers have public interfaces such that publishers and subscribers can connect to them, and send them publications and subscriptions. Every dispatcher is connected to all the matchers, and its only job is to forward received publication and subscription requests to the appropriate matchers. The appropriate matchers are found using a simple one-hop look-up, which enables dispatchers to be very lightweight and have very high throughput.

The subscription space partitioning technique is based on splitting the domains of each of the attributes, and assigning each matcher a part of the domain of each of the attributes. This demands that the service has predefined allowed attributes, and boundaries of their domains, which is the first constraint of the BlueDove system we encounter. A predicate of a subscription is a constraint over an attribute that spans a range of its domain. The corresponding matcher to that predicate is a matcher that was assigned a part of the domain, of that same attribute, that covers the range of the predicate. When a subscription arrives to a dispatcher, the dispatcher looks up a matcher for each of the k predicates of the subscription, and sends a copy of the subscription to all k of them. This way "smart" and moderate replication and server fault tolerance is achieved. In order to reduce the amount of subscriptions to be matched, each matcher has a separate subscription index and a publication incoming queue for the subscriptions it receives based on a particular attribute. This, however, cuts down the number of subscriptions to be matched by only a single dimension, which is not much if the subscription space is high-dimensional.

When a publication arrives to a dispatcher, it again looks up a matcher for each of the k attributes in the publication, but, unlike a subscription, it is not sent to all of them but only to the least loaded one. The policy for choosing the least loaded matchers can be simple, based only on the number of subscriptions on each of the matchers, but also an adaptive policy is implemented that is based on the average processing time of

publications calculated from both their queuing times and matching times.

The main problem and overhead source of the BlueDove's architecture is the global view of the system that all components have to maintain. Each matcher maintains a table with contact information and segment boundaries (for each attribute) of all the other matchers. Those tables are periodically exchanged with $\log(N)$ randomly chosen matchers, using a gossiping protocol. This way matcher tables are kept up-to-date, and each dispatcher pulls the table from a randomly chosen matcher once in a while to get an up-to-date view of the global state.

A new matcher is added to the system by contacting a dispatcher that, based on the workloads of existing matchers, chooses a heavily loaded existing matcher and splits each of its attribute segments into two parts. One part of each segment is assigned to the new matcher, and the subscriptions belonging to those segments are transferred to it. This enables elasticity and scalability of the system, but the overhead of introducing a new matcher seems substantial, especially because it completely changes the global picture of the system of which all components must have an up-to-date view.

University of Zagreb student Lucija Zadrija, in her master thesis [9], tried a different approach at developing a cloud-based publish-subscribe service. Ms. Zadrija based her publish-subscribe system on a "subscription covering forest" algorithm that organizes subscriptions into trees of mutually covering subscriptions. Since two subscriptions can be such that neither covers the other, naturally one tree will not be enough but there will be a forest of such trees where root subscriptions cover the subscriptions lower in the tree. The results of testing on centralised, non-cloud-based publish-subscribe systems, presented in Section 7.1, show that this algorithm has considerably lower performance than the BE-Tree algorithm used in the service developed within this thesis.

The cloud-based architecture consists of a central component called the "Proxy", of a component that delivers publications to subscribers, called the "Queue", and of a forest of matcher components. All publishers and subscribers connect to the "Proxy" component, and it receives all publication and subscription requests that it then forwards to other cloud components. The whole cloud broker arranges subscriptions in only one logical subscription forest, the roots of which are the roots of the trees of the root matchers in the matcher forest. That one subscription forest is distributed among matchers, that are also organized in a forest, so that each of the matchers internally keeps its own, smaller subscription forest. The subscription trees in each matcher are branches of the same subscription tree in its parent matcher.

The central, “Proxy” component contains a model of the matcher forest, which is basically consisted of the root subscriptions of all subscription trees of the root matchers. When a subscription or a publication arrives, the “Proxy” component matches it against the subscriptions in the model and sends it to the appropriate root matcher accordingly. The model of the matcher forest has to be updated as the top levels of the subscription and matcher forests change. This model somewhat burdens the “Proxy” component with matching and maintaining the model, which makes it a bottleneck due to the fact that all requests to the cloud have to pass through it.

The matcher forest is self maintaining, which means that the matchers themselves monitor their load and decide when to add more matchers to the forest, and when to remove matchers from the forest. The load parameter is based on the amount of matching requests, and their processing time, on a single matcher. The load parameter is forwarded up a matcher tree so that load balancing can be achieved by splitting matcher trees and forests resulting in new trees and forests with the load parameters as similar as possible. When a matcher decides that it is overloaded (using a threshold) it sends a request to the “Proxy” component to initialize a new matcher component, and then the load balancing algorithms are run. The load balancing algorithms are quite complicated, with many special cases, so they won’t be further explained here.

From the user perspective, the biggest difference between this publish-subscribe service and the BlueDove is that this system allows subscriptions and publications defined on any attributes and with any values, it is not necessary to define allowed attributes and their domains before the starting of the service. Although this seems like a good feature, it is worth thinking about what that actually means to a user and is it worth any trade-off in the sense of lower efficiency. A publish-subscribe service will probably always be made for a specific purpose, and for that purpose, whatever it is, it probably won’t be too hard to determine the allowed attributes and their allowed values. A system that needs to be able to process any given subscription or publication is very hard to imagine. Even more, there could be benefits to having predefined attributes, and their allowed values, because that could be used to warn users that they are doing something wrong, and also help avoid accidental misspellings of attribute names or similar human errors.

5. Description of the developed system

The developed system is a publish-subscribe service based on the BE-Tree algorithm and designed for the cloud environment. Strictly speaking, it is not a cloud-based system because it doesn't connect to a remote cloud nor does it use remote resources. Instead it emulates a kind of a cloud environment by using separate processes for each component. This is logically the same as a cloud only everything happens on a single machine. That machine acts as the whole cloud environment and its OS acts as the cloud's hypervisor. The role of a virtual machine (VM) is taken by a process, therefore a request from the service to the OS to start a new component as a new process is logically the same as the service requesting a cloud manager to start a new VM, and then running something on it. The developed system uses the fact that all components are processes on the same machine to its advantage, by using the local inter-process input and output streams for inter-component communication. The architecture of the system is affected by that fact to a certain degree, but it is not crucial. All communication could be implemented using TCP/IP instead, with minimal influence on the current system architecture and no influence on the algorithms and ideas that are in the focus of this master thesis.

There are two main aspects of the developed system: matching and subscription management as the logical part, and load and processing time management as the architectural part. The former is concerned with subscription organization and matching optimization, which are not affected greatly by the fact that the whole system should be an elastic and scalable cloud-based service. The latter part deals with the elasticity and scalability of the system.

5.1. Matching and subscription management

Subscription management and matching are handled by an implementation of a BE-Tree (Section 2.2). A single BE-Tree manages subscriptions and performes matching for the entire publish-subscribe broker, i.e. the entire developed system. It is a sin-

gle tree only logically, and not physically, because it is distributed over more than one system component, each of which manages their own branch of the tree mostly independent of other components. This will be explained in more detail in Section 5.2.

The implemented BE-Tree doesn't use the $Loss$ part of the ranking function (mentioned at the end of Section 2.2 and explained in detail in [8]), because the expected gain in performance is not significant with respect to the computational overhead it would cause. Everything else is implemented just as described in [8].

Since BE-Tree is a structure that requires discrete attributes, all allowed attributes along with their ranges have to be defined prior to starting the publish-subscribe broker. This is done through a configuration file. There are two types of allowed attributes: numeric and string. For numeric attributes, a lower bound, an upper bound and a step must be defined, while for string attributes all possible values have to be listed. Every subscription and publication entering the broker is checked against the defined attributes. All publications that contain an undefined attribute, or an attribute with a non-legal value, are dismissed and disregarded. Similarly, all subscriptions that contain a triplet with an undefined attribute, or contain a triplet that cannot be matched (for example a numeric operator on a string attribute, or a numeric triplet that matches only values outside of the defined bounds for that attribute) are also disregarded. Additionally, all numerical values in both subscriptions and publications are set to the nearest whole step defined for the corresponding attribute. This improves the performance of the BE-Tree algorithm, which is intended to work with exactly such data, but also allows the system to be used with naturally continuous data discretized to the wanted precision.

5.1.1. Subscription operators

As described in [8] a subscription is a set of triplets. A triplet consists of an attribute, a value, and an operator that puts the given attribute and value in a relation. Each triplet represents a logical predicate, therefore, a subscription is a conjunction of predicates that all have to be true in order for a publication to match a subscription.

A list of supported operators, and their transformations to interval logic, as specified by [8], are given in Table 5.1.

This system has all of the operators listed in Table 5.1 implemented internally, but offers to the user a different set of operators. The operators offered to the user for defining a subscription are:

– LESS_THAN - a numeric operator ($<$)

Table 5.1: Operator Transformations

Operator	Interval-based
$i < v_1$	$[v_{min}, v_1 - 1]$
$i \leq v_1$	$[v_{min}, v_1]$
$i = v_1$	$[v_1, v_1]$
$i > v_1$	$[v_1 + 1, v_{max}]$
$i \geq v_1$	$[v_1, v_{max}]$
$i \in \{v_1, \cdots, v_k\}$	$[v_1, v_k]$
i BETWEEN v_1, v_2	$[v_1, v_2]$

- LESS_OR_EQUAL - a numeric operator ($\leq$)
- EQUAL - both a numeric and a string operator
- GREATER_THAN - a numeric operator ($>$)
- GREATER_OR_EQUAL - a numeric operator ($\geq$)
- BETWEEN - a numeric operator that takes two values, a lower and an upper bound
- CONTAINS_STRING - a string operator that matches all strings that contain a given string
- STARTS_WITH_STRING - a string operator that matches all strings that start with a given string
- ENDS_WITH_STRING - a string operator that matches all strings that end with a given string

As can be seen, the only difference is in the string operators. All string operators (except the EQUALS operator) are internally translated to the $\in$ operator. The system does this upon all triplets defined over a string attribute of every subscription that enters the broker, by finding all possible values of that attribute that could satisfy the predicate defined by the triplet. Those make up the set of values of the $\in$ operator (if the set is empty the subscription is dismissed because it can never be satisfied).

This way of defining string operators is both practical and necessary. It is practical for users because those operators are natural for defining string queries, and it is necessary because the user cannot know the order and indices of the values of a string

attribute (unless the user has access to the broker configuration files, which it should not).

5.2. System architecture

The main part of all publish-subscribe systems are the brokers, to which publishers and subscribers connect to and communicate with. The primary idea of the developed publish-subscribe service is that it is cloud-based (in contrast to distributed, for example) which means that there is only a single broker for all publishers and subscribers. That broker has to be elastic and adaptive to the workload, meaning that it should be able to process requests in parallel, and additionally be able to grow or reduce itself in the number of components that process the requests in parallel.

An overview of the system architecture is given in Figure 5.1. There are logically two parts of the broker - the fixed part and the dynamic part. The fixed part consists of three components: the **MessageReceiver**, the **DeliveryService** and the **CloudBroker**. They are created on broker startup, and have to exist during the entire lifetime of the broker. Those three components manage the communication to and from external entities (publishers and subscribers), and the flow of client requests and internal messages between themselves and the dynamic part of the broker. The dynamic part of the broker consists of components called *matchers*, organized in a dynamic tree structure. The only constant of the dynamic part of the broker is the **RootMatcher** component, which is, like the fixed part of the broker, created on broker startup, and has to exist during its entire lifetime.

There are two kinds of matcher components: a RootMatcher and ChildMatchers. Each matcher component is responsible for a branch of the single logical BE-Tree that spans the entire broker, more precisely its dynamic part. The RootMatcher is the top matcher, its parent is the CloudBroker component. There are only two relevant differences between the two types of matchers. One is that the RootMatcher holds the root node of the whole BE-Tree, which is by its type a c-node, and a ChildMatcher holds the root of the branch of the BE-Tree dedicated to it, which is by its type a p-node. The other difference is that ChildMatchers propagate messages up the matcher tree, while the RootMatcher is usually their "last stop".

A parent matcher communicates asynchronously with its child using a special node of the BE-Tree called a *proxy* p-node. When a proxy p-node is created, it creates a new matcher component as a separate process, and is the only one that has the ability to communicate with it over the standard OS input/output streams. The newly created

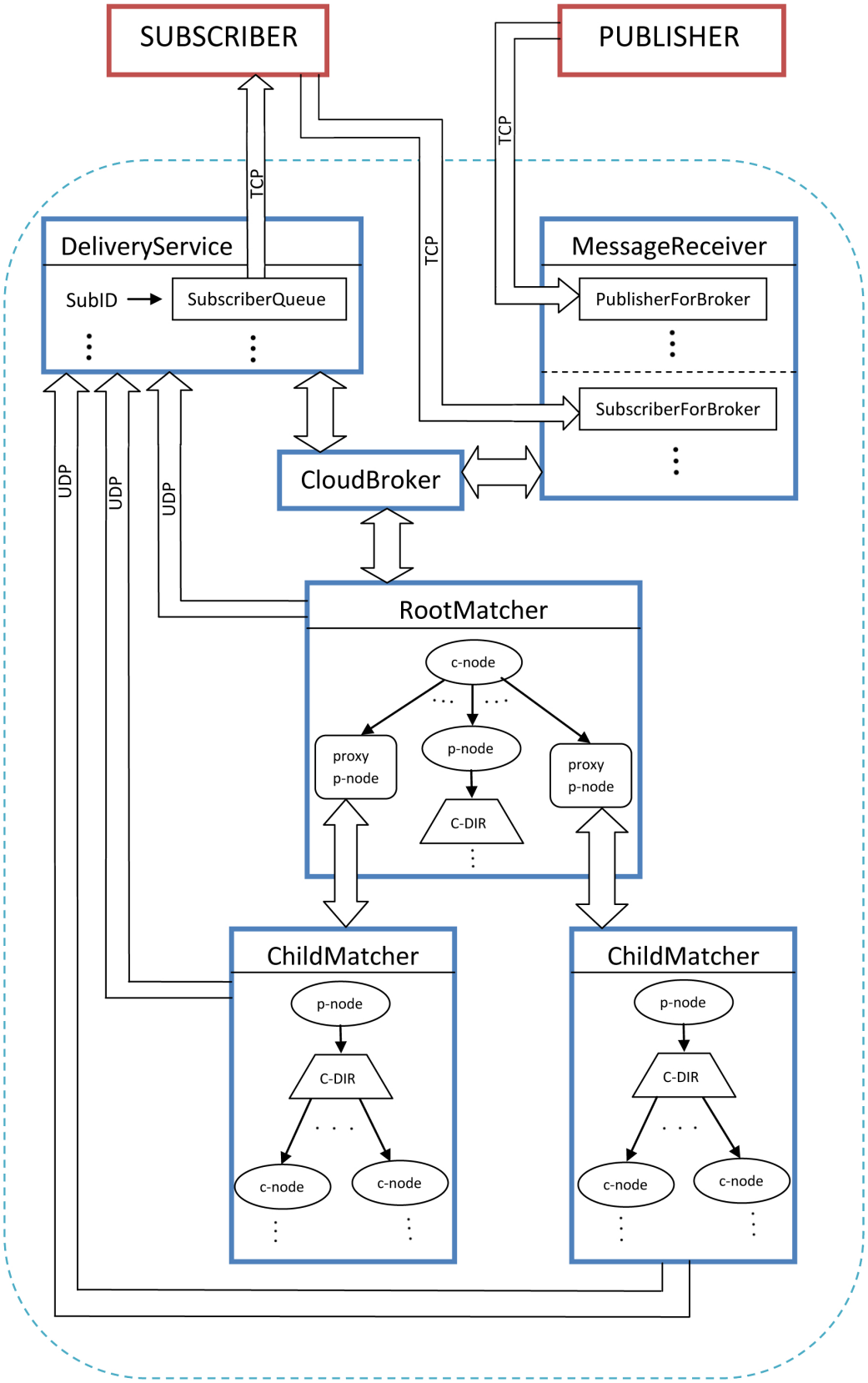

Figure 5.1: System components and their communication pathways

proxy p-node replaces a branch of the subtree of the BE-Tree located on the parent matcher, and sends it to the newly created ChildMatcher component.

The existence of child matchers is transparent even to the parent matcher itself, because they are hidden behind a seemingly regular node in the BE-Tree. Only the proxy p-nodes are aware of the ChildMatcher components, and each of them only of its own. This, coupled with the fact that each matcher can send UDP messages to the DeliveryService independently of any other component, enables the system to be loosely coupled, which means that only local information is required by each component. None of the components has an overview of the whole system, nor can it control it directly. This greatly reduces the architectural overhead in processing and communication.

The CloudBroker is the central component of the system in its current design, although its only function is to be a relay between the MessageReceiver, DeliveryService and RootMatcher components, and can basically be omitted in a slightly different design. The CloudBroker component starts the other three basic components as separate processes, and is the only component that has direct access to their input/output streams. Because of that, the only way those components can communicate with each other is through the CloudBroker component, which reads messages from their input streams and forwards them to the appropriate output streams.

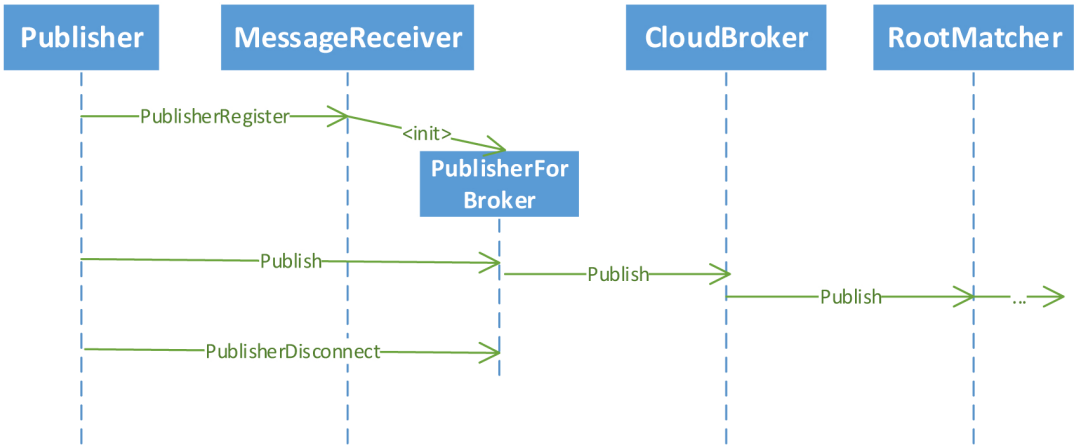

Figure 5.2: Communication of a publisher with the cloud broker

The purpose of the MessageReceiver component is to accept connections and process requests from publishers and subscribers. MessageReceiver accepts TCP connections from publishers and subscribers, and creates a handler object for each of them. A handler for a subscriber is called a *SubscriberForBroker*, and a handler for a publisher

is called a *PublisherForBroker*. A publisher can communicate with the broker only through the MessageReceiver component, and it can send three different messages, as depicted in the sequence diagram in Figure 5.2. When MessageReceiver receives a *PublisherRegister* message, it creates a PublisherForBroker handler object for that publisher, which processes all subsequent requests from that publisher. A *PublisherDisconnect* message triggers the destruction of the publisher's handle, its removal from the MessageReceiver and the teardown of the TCP connection. The only other message a publisher handler can process is the *Publish* message, which is used for both publishing and unpublishing a publication. Upon receiving a *Publish* message, the publication is saved on the MessageReceiver, and the *Publish* message forwarded to the CloudBroker component, which in turn just forwards it to the RootMatcher component, after which the matching process begins. A subscriber's communication with the broker is somewhat more complex, as can be seen in Figures 5.3 and 5.4. This is because the broker has to be able to notify a subscriber about a publication at any given time and from a different broker component, and also because a subscriber can remain registered while not connected (which is a state not necessary for a publisher).

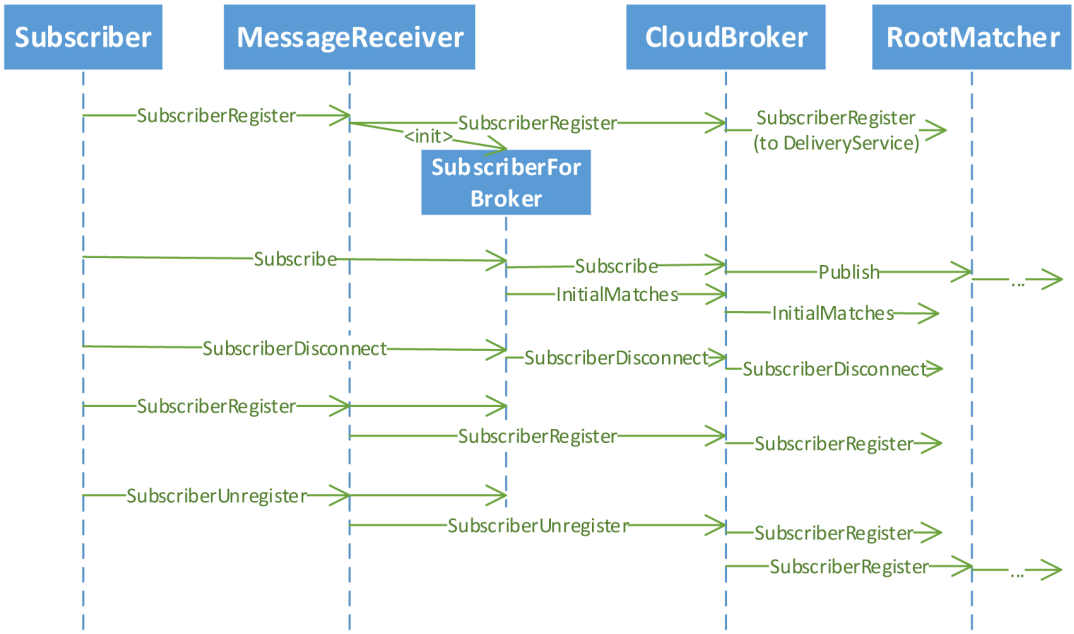

Figure 5.3: Communication of a subscriber with the cloud broker, the MessageReceiver part

Upon receiving a *SubscriberRegister* message, the MessageReceiver component creates a SubscriberForBroker handler object for that subscriber, which processes all subsequent requests from that subscriber. After having done that, the MessageReceiver

forwards the *SubscriberRegister* message to the CloudBroker, which in turn forwards it to the DeliveryService. A *SubscriberDisconnect* message simply puts the subscriber's handler object in an inactive state, not removing it from the MessageReceiver, and is also forwarded to the DeliveryService via the CloudBroker. A *SubscriberUnregister* message triggers the destruction of the subscriber's handler, its removal from the MessageReceiver, and the teardown of the TCP connection, but is also forwarded to the CloudBroker. The CloudBroker duplicates the message, sending one copy to the DeliveryService and the other to the RootMatcher, because all the subscriptions of the unregistering subscriber need to be removed from the BE-Tree upon its unregistration. A *Subscribe* message, used for both subscribing and unsubscribing a subscription, is processed by the subscriber's handler in two steps. The first step is to simply forward the message to the RootMatcher via the CloudBroker, and the second step is to match the new subscription against all active publications that have entered the broker, and send an *InitialMatches* message containing all the matched publications to the DeliveryService via the CloudBroker.

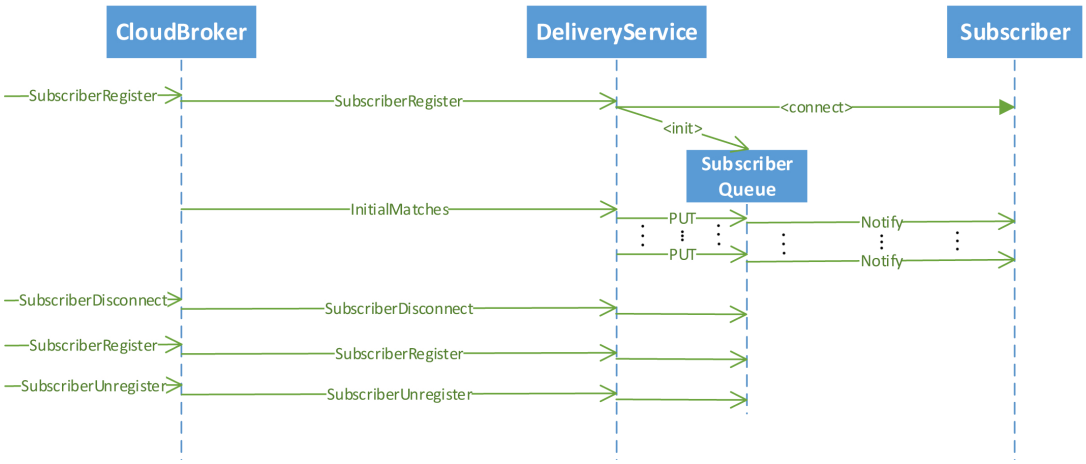

Figure 5.4: Communication of a subscriber with the cloud broker, the DeliveryService part

The purpose of the DeliveryService component is to receive matched publications for registered subscribers from other broker components (mostly matchers), and to send them to the appropriate subscribers if it is able to do so, or else to hold them in a queue until being able to send them. The DeliveryService has a TCP connection to every connected subscriber, and a queue object for every registered subscriber. The TCP connection to a subscriber is established, and a queue object created if the subscriber was not already registered, when a *SubscriberRegister* message is received

from the CloudBroker, as can be seen in Figure 5.4. In this TCP connection the role of the "server" is with the subscriber, which expects the connection request from the broker, and messages are only sent from the DeliveryService to the subscriber without any response. A *SubscriberDisconnect* message from the CloudBroker causes a teardown of the TCP connection to the disconnecting subscriber, but not the removal of the queue object, which afterward accumulates publications for the subscriber until the subscriber connects again. When the subcriber reconnects, the accumulated publications are sent to it by the DeliveryService from the matcher's corresponding queue. A *SubscriberUnregister* message destroys both the TCP connection and the queue object of the unregistering subscriber, removing it completely from the DeliveryService. The only remaining message the DeliveryService can receive from the CloudBroker component is the *InitialMatches* message, which carries a set of publications for sending to one subscriber and that is exactly what the DeliveryService does, it puts them all in that subscriber's queue object for sending. Additional to the OS level connection to the CloudBroker and the TCP connections to the subscribers, the DeliveryService is also a UDP server which receives UDP packets from matcher components. Each UDP packet from a matcher contains a publication and a set of subscriber ID's that belong to subscribers which have a subscription that the publication matched. That publication is sent to each of the subscribers, or put in their respective queues.

5.3. Subscription and publication processing

The reason why asynchronous communication between matchers, and consequently parallel processing, is possible is because changes in the BE-Tree propagate only downward. The only exception is the updating of node scores which propagates upward through the tree after insertion, deletion or matching. But since the node scores affect only the efficiency of the BE-Tree, and not its consistency, there is virtually no effect if the score updating information comes with a delay of a few operations. Additionally, in order to reduce the communication overhead, the $Loss$ part of the scoring function is ignored, which means that the matching operation has no effect on the node scores, and consequently there is no need to send any information up the tree as a response to a matching request.

Since matchers are organized in a tree structure, it is perhaps unclear how the requests are being processed in parallel. Since all the matchers hold parts of the same BE-Tree, a single request has to be processed sequentially by the matchers in the same branch of the matcher tree. Parallelism is enabled by the asynchronous communica-

tion, which allows a matcher that has processed a request and handed it down to its child(ren) matcher(s) to immediately be able to start processing a new request. This way multiple requests are being processed in parallel on different parts of the same logical BE-Tree.

There are five basic requests that a BE-Tree has to be able to process: publishing and unpublishing a publication, subscribing and unsubscribing a subscription, and removing all subscriptions of a single subscriber (removing a subscriber from the BE-Tree). These five requests are made using three different messages, each of which every matcher component can process. They are: the *Subscribe* message, the *Publish* message and the *SubscriberUnregister* message. The communication inside the matcher tree for each of the requests can be described by a sequence diagram between a single parent and one of its children matchers, because it is the same for every parent-child pair of matchers. The only exception is the RootMatcher, which differs only in the part of forwarding a score update response message to its parent, because it doesn't have a parent matcher.

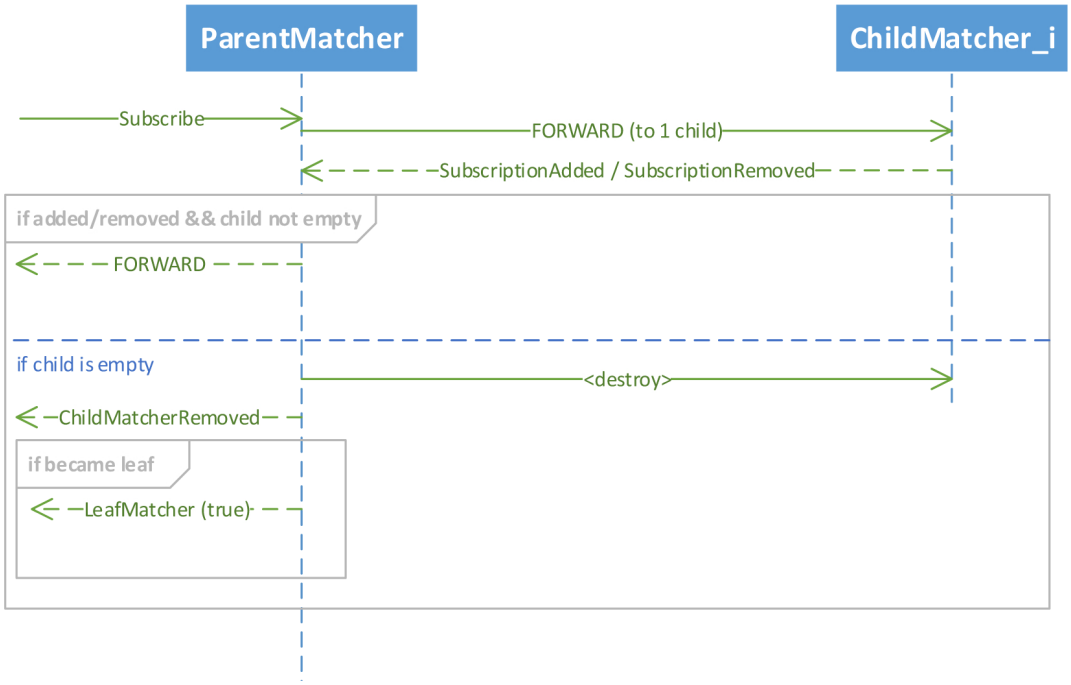

Figure 5.5: Processing of a "Subscribe" message, if the subscription was added on the ChildMatcher

When a ChildMatcher receives a *Subscribe* message from its parent, it tries to add the subscription into its BE-Tree branch. If it succeeds, it returns a *SubscriptionAdded* message to its parent with the *added* flag set to *true* and carrying the new score of

the branch-root p-node. The exact same thing happens if the subscription should have been added, but for some reason wasn't. The only difference is the *added* flag set to *false*. These scenarios are depicted in Figure 5.5. The parent matcher, upon receiving a *SubscriptionAdded* message, checks if the child matcher is now empty (this makes sense because a *Subscribe* message can carry an unsubscribe request). If not, it simply updates the node scores up its local BE-Tree branch and then sends the same *SubscriptionAdded* message to its parent, but with the new score of its branch root p-node. This process continues up the matcher tree until it reaches the RootMatcher. If, on the other hand, the child matcher is now empty, the parent matcher destroys it (the process) and updates its own scores using a *ChildMatcherRemoved* message instead of the *subscriptionAdded* message received from the former child matcher. That *ChildMatcherRemoved* message is then passed to the parent's parent matcher and upward to the top of the matcher tree, just like the *SubscriptionAdded* messages. Additionally, if after destroying its child matcher the parent matcher became a leaf matcher in the matcher tree, it sends a *LeafMatcher* message to its parent matcher to let it know about the change, because communication between matchers is a bit different if a child matcher is a leaf matcher.

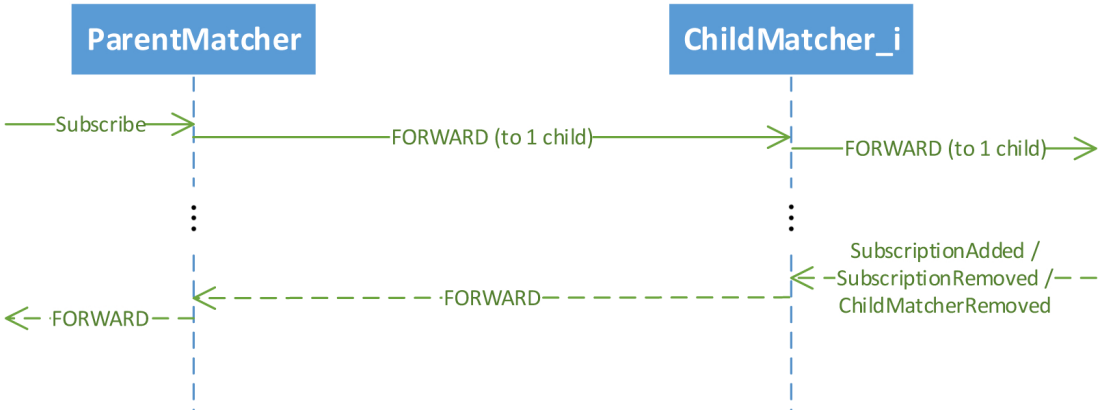

Figure 5.6: Processing of a "Subscribe" message, if the subscription was forwarded down the matcher tree

The second possible case is that the subscription is not supposed to be added in the child matcher's BE-Tree branch. In that case, the child matcher, or more precisely the appropriate *proxy* p-node, forwards the *Subscribe* message to another child matcher down the matcher tree and doesn't respond to the parent matcher. This scenario is depicted in Figure 5.6. Some time later the child matcher receives either a *SubscriptionAdded* or a *ChildMatcherRemoved* message from the matcher it forwarded the request

to, and does with them the same as already described earlier and depicted in Figure 5.5, as do all the other matchers up the matcher tree to the RootMatcher.

The purpose of a *SubscriberUnregister* message is to remove all subscriptions of a single subscriber from all the matcher components. Because of that, the *SubscriberUnregister* message is a flooding request which means that it is sent by every matcher to all of its children matchers, the consequence of which is that every matcher in the matcher tree will process the same *SubscriberUnregister* message once it enters the system. Since the updates of node scores have to come from the last changes in the BE-Tree, and update messages propagate up the matcher tree to the RootMatcher, only the leaf matchers should send a response to the *SubscriberUnregister* message after having removed the appropriate subscriptions from their BE-Tree branches. A *SubscriberDeleted* response message is processed in a similar way as a *SubscriptionAdded* message. All this is shown in Figure 5.7.

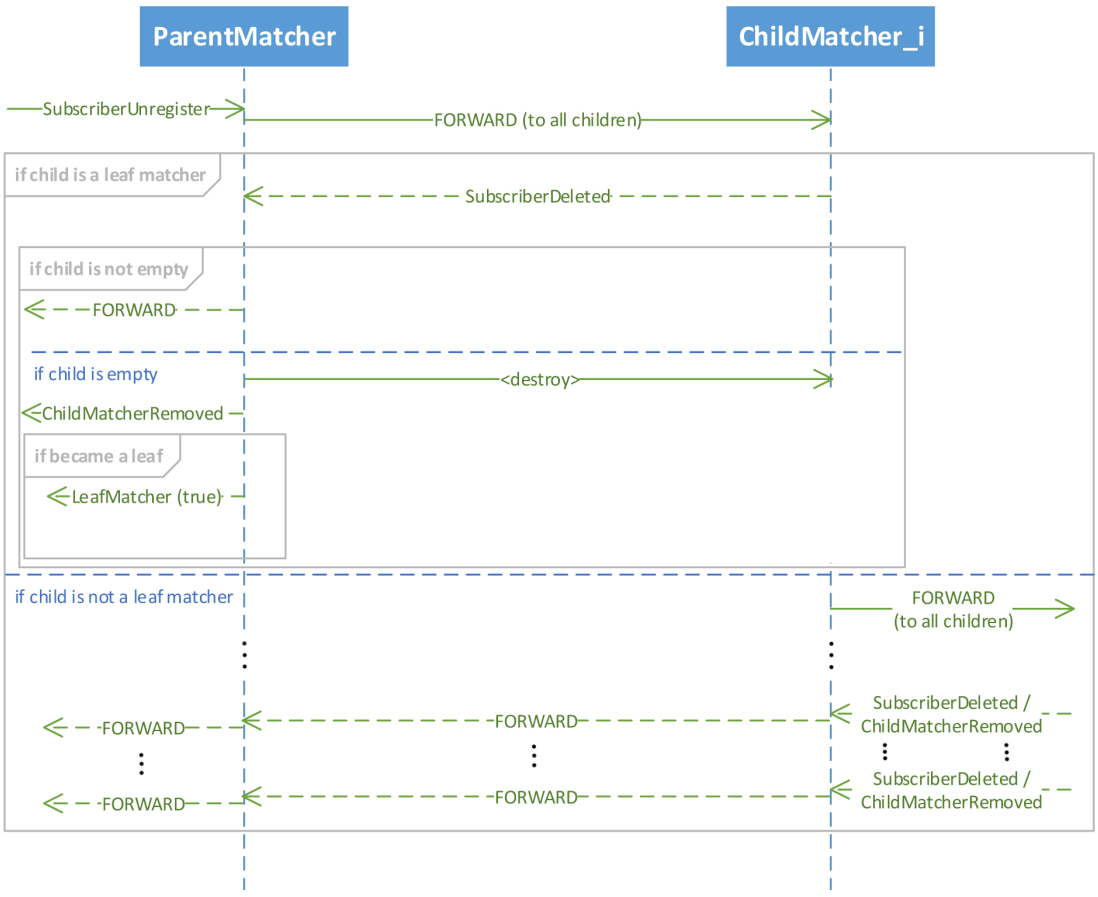

Figure 5.7: Processing of a "SubscriberUnregister" message

The only remaining message a matcher component can process is a *Publish* mes-

sage. The communication between a parent-child pair of matchers following a *Publish* request is shown on Figure 5.8. The *Publish* message is also a flooding request, but unlike the *SubscriberUnregister* message, it doesn't affect the node scores, so there is no need for a response to it. However, there exists a response message to it because of matcher merging needs (more about that in section 5.4), but only from the leaf matchers to their parent matchers, and the response is not further propagated up the matcher tree. When a matcher receives a *Publish* message it starts a matching process on its local branch of the BE-Tree, which is done recursively and returns a set of all subscribers that have a local subscription that matches the given publication. When the matching process gets to a *proxy* p-node, it forwards a copy of the *Publish* message to the corresponding ChildMatcher and returns to continue the matching process locally. The final result of the matching process on a matcher is, therefore, a copy of the *Publish* message sent to every child matcher of that matcher, and a set of subscriber IDs that is sent to the DeliveryService component in a UDP packet along with the original publication.

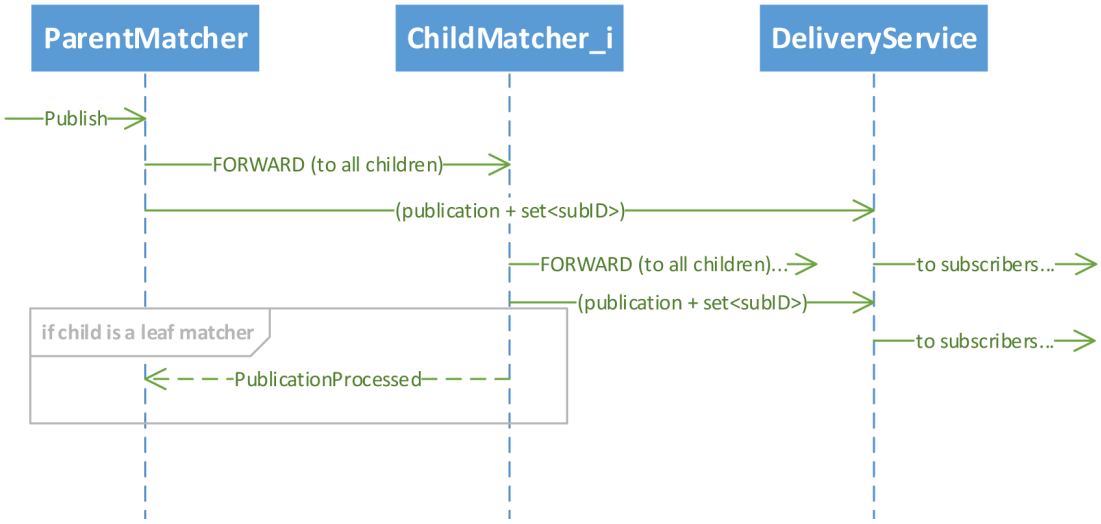

Figure 5.8: Processing of a "Publish" message

5.4. Load balancing

The elasticity and scalability of the system is achieved through the splitting and merging of matcher components. Both the splitting and the merging are triggered by the processing times of the last N publications, where N is a parameter of the broker.

The problem with matcher splitting is that it cannot be done at just any point in the BE-Tree without running into serious complications. For example, if a BE-Tree branch could be split on any node type then a lot of implementation problems would arise, not the least the need for different "proxy" nodes and communication protocols for each of them. The c-nodes are not a convenient place to split a BE-Tree because they are a part of an inner tree structure, the c-directory, which would also then have to be split between multiple processes. Therefore, the p-nodes are a logical selection as splitting points. However, not all p-nodes are an equally good place to split a BE-(sub)Tree either. For example, if a subtree of a BE-Tree would be split on the second level of p-nodes from the local subtree root, then a subsequent split at the level between would be impossible in the current implementation without destroying some existing matchers, because a reference to a created process cannot be transferred. Even if it could be done with a different system design it would still give rise to merging problems, problems with deciding on a split point, and other complications. Taking into account all of the stated, I have realised that the simplest and the most elegant way to split a BE-Tree branch located on a matcher is to do it on the first level p-nodes. That way entire c-directories stay on the same component, links to child matchers are located in hash tables (p-directories) which is also convenient, and the decision of the split point becomes simplified and easy to make.

5.4.1. Splitting of Matchers

The splitting of a matcher is initiated when the minimal processing time in the window of N last publication matchings is greater than a given threshold, i.e. when the processing time for all publications in the window is larger than a given threshold. The splitting criterion defined in this way is robust with regard to sudden spikes in processing times, which do not indicate a real overload of a processor, but is also a bit unfortunate due to the manual setting of the threshold on broker startup and the inability to change it during runtime.

After a *Publish* request has been processed and forwarded (Figure 5.9), a check is made whether the minimal processing time in the window (including the one just completed) is greater than the given splitting threshold. If it is, a search for a first level p-node with the biggest matching time among all first level p-nodes in the local subtree of the BE-Tree is started. When such a p-node is found, a *proxy* p-node is created that substitutes that p-node in its parent c-node's p-directory, and that p-node, along with the entire subtree to which it is the root, is sent to the newly created ChildMatcher. The

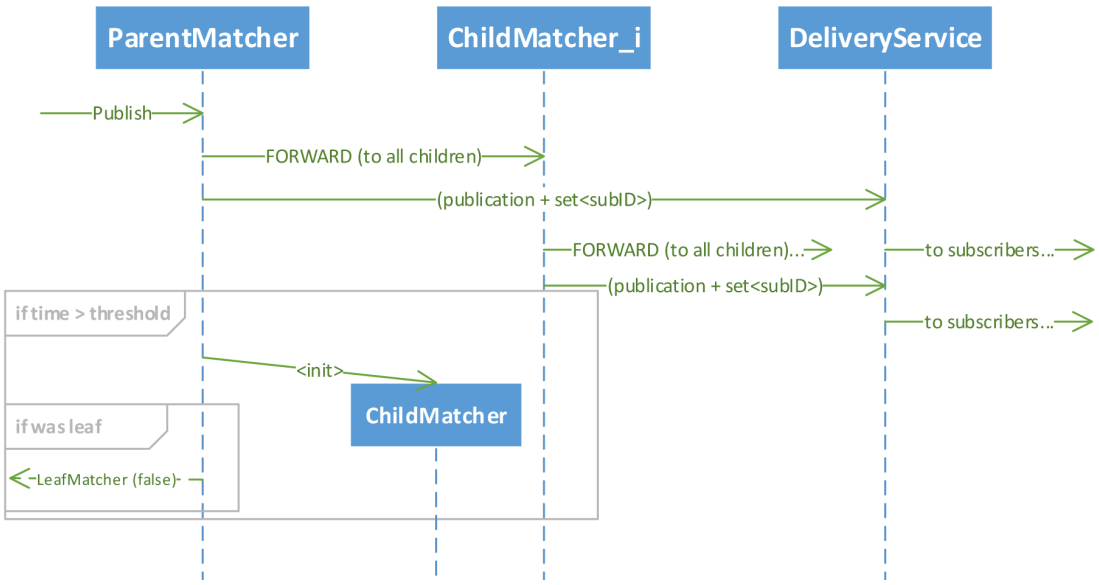

Figure 5.9: Processing of a "Publish" message followed by a matcher splitting

final step is for the matcher to notify its parent matcher if it had stopped being a leaf matcher after splitting. This is done by the matcher sending a *LeafMatcher* message.

5.4.2. Merging of Matchers

The merging of a matcher is initiated when the maximal processing time in the window of N last publication matchings is less than a given threshold, i.e. when all publications in the window where processed in lower processing time than the defined threshold. This criterion is, like the splitting criterion, chosen for its robustness with regard to sudden spikes in processing times.

The same as with the splitting of matchers, checking of the previously described condition is performed after a *Publish* request has been processed and forwarded. If the condition is met, the matcher that wants to be merged (the ChildMatcher in Figure 5.10) sends a *MatcherMerging* message to its parent. A *MatcherMerging* message contains the branch of the BE-Tree of the merging matcher along with other relevant publish-subscribe information. Upon receiving a *MatcherMerging* message, the *proxy* p-node on the receiving matcher:

1. destroys the corresponding ChildMatcher,

2. processes all requests (over the received subtree) that might have been sent to the child matcher before it was destroyed, but didn't have a chance to get processed,

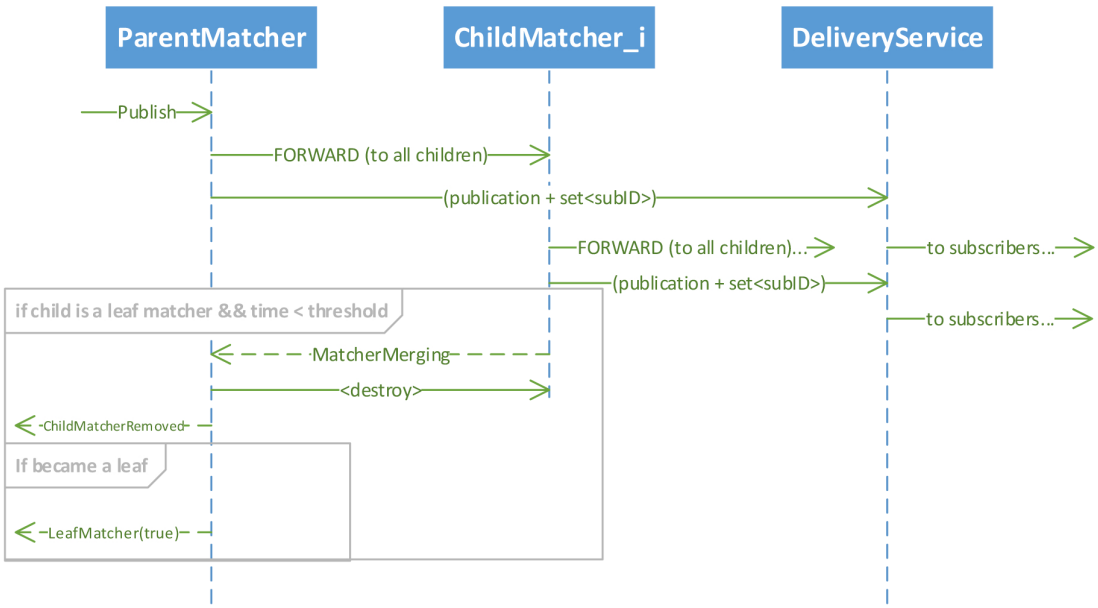

Figure 5.10: Processing of a "Publish" message followed by matchers merging

3. replaces itself in the p-directory of its parent c-node with the received subtree,
4. sends a *ChildMatcherRemoved* message up the local BE-Tree branch, and to its matcher's parent matcher,
5. sends a *LeafMatcher* message to its matcher's parent matcher if its matcher became a leaf matcher after the merging.

6. Implementation of the developed system

The entire system is implemented in the Java programming language using the Eclipse IDE[1] and the Java 1.7 JDK[2]. The basis for the system was an existing publish-subscribe system developed at the Department of Telecommunications of the Faculty of Electrical Engineering and Computing in Zagreb. The existing publish-subscribe system was a distributed system with a covering forest matching algorithm. As a first step I stripped that publish-subscribe system of all code that was redundant to me and made it into a centralised publish-subscribe system with that same covering forest algorithm, removing bugs and inefficient code in the process and redesigning it. Little of the original code was left untouched, mostly interfaces and abstract classes. That centralised system served to me as a skeleton for a centralised BE-Tree version which I used to implement and test the BE-Tree algorithm. The centralised BE-Tree version in turn I used as ground work for the cloud version, which is the final product, and a "primitive" cloud version which just uses n centralised BE-Tree versions as separate matchers in parallel. The "primitive" cloud version is used in testing for comparing results and will be mentioned later in Chapter 7.

6.1. Common classes

The "common classes" refers to all the classes that are common to all the publish-subscribe systems, both the centralised and the cloud versions. They are located in a separate project (named `PubSubCommon`) and are pretty much a collection of interfaces, abstract classes and general publish-subscribe classes like subscription and publication implementations.

The two main packages are the `hr.fer.tel.pubsub.artefact` package

[1] Integrated Development Environment
[2] Java Development Toolkit

and the `hr.fer.tel.pubsub.common` package. The contents of the `common` package are:

- `Attribute` interface - an interface defining an attribute supported by the broker
- `NumericAttribute` class - an implementation of the Attribute interface for numeric attributes (has a lower and upper bound and a discretization step)
- `StringAttribute` class - an implementation of the Attribute interface for string attributes (has a list of strings and a default step of 1)
- `Attributes` class - a class similar to the Java `Properties` class that reads a configuration file containing specification of the allowed attributes and their allowed values and holds them in a `HashMap` with their names as keys and `Attribute` objects as values. It also contains methods that check if a subscription or a publication is correctly defined with regard to the allowed attributes and prepares them for the broker to work with. This class implements the singleton pattern and is sent to each new component by the component that starts it.
- `MinimalistLinkedHashQueue` class - this is a simplistic implementation of a doubly linked list backed up by a `HashMap`. It is templated and implements only the `Iterable` interface while offering all the standard queue operations: offer, poll, peek and remove. In addition it offers the `contains` method which is $O(1)$ because of the backing `HashMap` and it demands a capacity at construction. These two extra features are necessary for the `SubscriberQueue` class that the DeliveryService uses as queues for the subscribers because the matching results come to the DeliveryService from unrelated matchers and there is a high chance of duplicate publications for subscribers which this collection solves.
- `enums.Operator` enum - an enumeration listing all the possible numeric and string operators available to a user for defining Boolean predicates. They are listed and described in Subsection 5.1.1.
- `Triplet` class - this class is one of the most important basic classes. An instance of this class represents a single Boolean predicate. It consists of a `String` key, an `Object` value (which can be a `String`, a `Double` or a `Double[]`) and a previously mentioned `Operator`. It's key method is the `covers(Triplet)` method which returns *true* if the `Triplet` object cov-

ers the given `Triplet` object or *false* otherwise. There are also convenience methods `covers(String)` and `covers(double)` that are the same as calling the `covers(Triplet)` method where the keys of triplets are the same and the operator is `EQUALS`.

- `UniqueObject` abstract class - a class that generates a random `UUID` object at construction and associates it with `this` object.

The `artefact` package contains the definitions of the primary objects that circulate the publish-subscribe system, namely publications and subscriptions. The contents of the package are:

- `Publication` abstract class - the base class for any publication implementation. It extends only the `UniqueObject` class and makes the start time and the validity of a publication its obligatory information.
- `Subscription` abstract class - the base class for any subscription implementation. Like the `Publication` abstract class extends the `UniqueObject` class and makes the start time and validity of a subscription its obligatory information. More importantly it defines two abstract methods: the `coversPublication(Publication)` method and the `coversSubscription(Subscription)` method.
- `ActivePublication` class - represents an active publication in the sense of a publication that came from a publisher and is now on the broker. The class itself extends the `Publication` abstract class and an instance of the class contains another `Publication` object (the original) and a `UUID` object that represents the ID of the publisher that published the publication.
- `ActiveSubscription` class - represents an active subscription in the sense of a subscription that came from a subscriber and is now on the broker. The class itself extends the `Subscription` abstract class and an instance of the class contains another `Subscription` object (the original) and a `UUID` object that represents the ID of the subscriber that made the subscription.
- `HashtablePublication` class - this is the basic (and only) implementation of a publication. The publication as a Boolean expression is in the form of a `HashMap` where keys are the names of the attributes and the values are the corresponding values of the attributes. Each key-value pair is a Boolean predicate with an implicit EQUALS operator.

– `TripletSubscription` class - this is the basic (and only) implementation of a subscription. The subscription is defined in terms of `Triplet` objects. The `Triplet` objects are kept grouped by the attribute on which they are defined since there can be multiple predicates defined over a single attribute. They are kept in a `HashMap` where the keys are the attribute names and the values are sets of `Triplet` objects which makes working with `TripletSubscription` objects very easy and practical and checking of coverage very efficient. There is also an additional `HashMap` named *stringAttributesBorders* that should be `null` outside of the broker and set by the `Attributes` class upon checking the subscription when it enters the broker. The keys of that map are the names of only the string attributes that have predicates defined over them in that subscription and the values are numerical `Triplet` objects with a BETWEEN operator and the indices of the lowest and highest value of the attribute that satisfy the constraints on that attribute in that subscription.

Besides the listed and explained classes and interfaces there are also the `PublisherInterface`, the `SubscriberInterface` and the `NotificationListener` interfaces in the `entity` package, along with the `NetworkEntity` abstract class that holds the IP address, port and name of an entity and which all entities that communicate over the TCP/IP extend. The two interfaces in the `message` package that define the two basic types of messages: the `Message` interface (for inter-entity communication) and the `InternalMessage` interface (for intra-broker communication). And various utility classes in the `util` package like the `LogWriter` class.

6.2. BE-Tree implementation

The classes that implement the BE-Tree are located in the `hr.fer.tel.pubsub.BETree` package of the `CloudBeTreePubSub` project which is the main project of the developed cloud-based BE-Tree publish-subscribe system. The BE-Tree classes are practically the same as the ones used in the centralised BE-Tree publish-subscribe version. The differences between the identically named classes are mostly in some thread synchronization code and a few extra methods and there are only two extra interfaces and an implementation of the *proxy* p-node which is essential for the cloud-based broker. An important difference is also that the cloud-based BE-Tree nodes are all (except the `ProxyPnode`) `Serializable` which enables them (and whole

branches of the BE-Tree) to be easily transferred between processes.

The contents of the `hr.fer.tel.pubsub.BETree` package are:

- `BETreeParams` class
- `BETreeActiveNode` interface
- `CdirBucket` class
- `Cnode` class
- `Pnode` interface
- `RealPnode` class
- `ProxyPnode` class

The `BETreeParams` class extends the Java `Properties` class and is used to load BE-Tree parameters from a file and keep them saved. This class uses the *singleton* pattern and is transferred to each component started in a new process by the component starting it (the same as the `Attributes` class described in the previous section).

The `BETreeActiveNode` interface defines the methods that a BE-Tree node has to implement. Those are:

- `findMatchingSubscribers` - does the matching and returns the time that was required to complete it (in nanoseconds). The return time is necessary for the splitting and merging criteria.
- `insert` - tries to inserts the given subscription in the BE-Tree. It can return SUB_ADDED, SUB_NOT_ADDED (duplicate for example) and SUB_FORWARDED which are integer constants defined in the `BeTreeParams` class.
- `remove` - tries to remove the given subscription from the BE-Tree. It can return SUB_REMOVED, SUB_NOT_REMOVED (non existent for example) and SUB_FORWARDED which are also integer constants defined in the `BeTreeParams` class.
- `deleteSubscriber` - removes all subscriptions of the given subscriber from the BE-Tree.
- `isEmpty` - returns true if the node has no descendants and holds not subscriptions.

– `scoreUpdateToTop` - updates the score of the node and then calls the same method on its parent node.

– `highestMatchingTimePNode` - its function is to find a first level p-node (from the level of the first caller) that has the greatest last matching time.

The `Pnode` interface extends the `BETreeActiveNode` interface with a few more methods that are specific for p-nodes. The interface is necessary because two different p-node implementations were necessary, namely the `RealPnode` and the `ProxyPnode`. The two relevant methods defined by this interface are the `getScore` method and the `getMatchingTime` method. The first method returns the score of the p-node while the second returns the time it took to match a publication against the subtree below that p-node the last time it was done (in nanoseconds).

The `RealPnode` class is, as its name states, the implementation of a p-node as described in the BE-Tree article [8] and it, of course, implements the `Pnode` interface. It is a fairly simple class that in its constructor creates a new `CdirBucket` that becomes the root of the c-directory. Almost all it's methods just delegate the work to the same-named methods of the c-directory's root `CdirBucket`. Some of them in addition call the `updateScore` synchronized method which is practically the only method in the `RealPnode` that doesn't act like the others. This method calls the `retrieveAndSumScores` method of the `CdirBucket`, which sums the scores of all the c-nodes in the c-directory, and then calculates the score using the formula from the BE-Tree article.

The `CdirBucket` class represents a "bucket" of the c-directory. This class implements the `BETreeActiveNode` and the `Serializable` interfaces. The serialization is straightforward and requires no special care since all the class fields are also of types that implement the `Serializable` interface. All the `BETreeActiveNode` methods are implemented simply by calling the method with the same name on the bucket's `Cnode` object and then on its left and right child `CdirBucket` objects (if they exist) and combining the three results in a logical way. The only exception is the `scoreUpdateToTop` method which just calls the equally named method on the bucket's parent. In addition to the methods defined by interfaces there are a few more methods, the most important of which is the `spaceClustering` method implementing the second phase of the space-cutting technique described in [8]. The

implementation of the space clustering is fairly straightforward. If the child c-node is overflowing and the bucket is a non-atomic, non-leaf bucket then it creates two `CdirBucket` objects, each with half of its range, and transfers subscriptions that can fit in the new buckets from its c-nodes l-node to theirs. The process is finished by calling the `spacePartitioning` method of its `Cnode` and the `spaceClustering` methods of its new children `CdirBucket`s. If the bucket is atomic or is already not a leaf node then just the `spacePartitioning` method of the child `Cnode` is called.

The information that defines a c-directory bucket is an attribute and a range of the attribute's domain. From this information a `Triplet` object with a BETWEEN operator is created in the `CdirBucket`'s constructor. The value of that triplet is a pair of lower and upper bound values of the attribute range, where the values are numbers rounded to the nearest step for a numeric attribute or indices of string values for a string attribute. This `Triplet` object is then saved in the class field named *attrRange* and is added to the *key* `HashMap` (received by the constructor) under the attribute's name, which is then passed to the constructor of the child `Cnode`. The main purpose of the *attrRange* `Triplet` object is to simplify and optimise the `encloses` methods that check if a publication or a subscription are enclosed by the bucket's range on the same attribute. For a publication this is done by simply calling the `Triplet.covers` method of the *attrRange* with the value of the corresponding attribute of the publication sent as an argument of the method. For a subscription check its *stringAttributeBorders* map (described earlier under `TripletSubscription` in Section 6.1) is used if the attribute is a string attribute. In that case simply the `Triplet.covers` method of the *attrRange* is called on the value assigned to the attribute in question in the *stringAttributeBorders* map. If the attribute is numeric then the same method is simply called on all predicates defined over the attribute in question in the subscription that is being checked.

6.2.1. The `Cnode` class

The `Cnode` class is an implementation of the c-node as described in [8] together with the l-node which is implemented as a list of `ActivePublication` objects in a `Cnode` object. It implements all methods specified by the `BETreeActiveNode` interface as well as other methods required by the algorithm description in [8] (for both c-nodes and l-nodes) like the `spacePartitioning` method that does the first phase of the BE-Tree space-cutting technique, the `highestScoreUnusedAttribute` method that finds the best attribute to partition on, the `updateLnodeCapacity`

method that updates the maximum capacity of an l-node if partitioning is impossible or infeasible, and the methods for updating the score of a c-node. A `Cnode` object also contains a `HashMap` of `Triplet` objects mapped to attribute names called the *key*. This map is filled by `CdirBucket` objects up the tree, one in each c-directory, and constitutes not only a unique key of the `Cnode` but also a set of predicates that each of the Boolean expressions (subscriptions) in this node and every node down the tree from this node has to satisfy. This key is used to very efficiently check for subsumed Boolean expressions which influence the score of a c-node in a different way then all the other Boolean expressions, as described in [8].

Since the `BETreeActiveNode` methods of the `Cnode` class do most of the work in a BE-Tree and the class functionality differs from the c-node functionality described in the BE-Tree article, a list of the most important methods of the class is given along with short descriptions of each of them:

- `findMatchingSubscribers` - first the subscriptions in the list representing the l-node are matched and appropriate subscriber IDs added to the result set. The l-node matching time is measured using the `System.nanoTime` method. After the "local" matching the `findMatchingSubscribers` method of each of the p-nodes from the p-directory is called and their matching time added to the "local" matching time before returning the overall matching time.
- `insert` - finds the p-node with the highest score and tries to add the subscription down its branch. If the adding fails for some reason or there is no such p-node the subscription is added to the list representing the local l-node. If the l-node overflows after the adding the space clustering is called on the c-nodes parent `CdirBucket` object else just the node score is updated.
- `remove` - tries to remove the subscription from the local l-node list. If the subscription isn't found locally the `remove` method of the `Pnode` objects in the p-directory is called in some order until one of them returns the SUB_REMOVED constant. If the node has been removed from the local BE-Tree subtree the node score is updated (otherwise it will be updated upon receiving a message from a child matcher).
- `deleteSubscriber` - removes all subscriptions with given subscriber `UUID` from the local l-node and then calls the `deleteSubscriber` method of each of the `Pnode` objects in the p-directory. When all removing is done the node score is updated.

– `spacePartitioning` - after insertion into the local l-node this method checks if the l-node has overflown. As long as it is overflown the partitioning is done such that the best attribute to partition by is found using the `highestScoreUnusedAttribute` method after which a new `RealPnode` is constructed with that attribute and placed in the p-directory. The new p-node is filled with subscriptions from the local l-node that have a predicate defined over the new p-node's attribute which are at the same time removed from the local l-node. After the partitioning is finished the score of the node is updated.

– `highestScoreUnusedAttribute` - this method simply counts the number of subscriptions that have a predicate defined over an attribute and then returns the attribute with the biggest such count. The attributes contained by the *key* of `Cnode` and the ones that were already used for partitioning are ofcourse excluded.

– `updateLnodeGain` - this method calculates the $Gain$ part of the score of the l-node using the following formula:

$$Gain(l_j) = (1 - \beta)(\#\text{subsumed}) + \beta(\#\text{covered}) \tag{6.1}$$

where "subsumed" stands for predicates of the Boolean expressions in the l-node that are exactly matched by the predicates in the *key* of the c-node and "covered" stands for all the other predicates covered by the key. The β factor is a BE-Tree parameter between 0 and 1.

– `updateLnodeScore` - the score of an l-node is given by the following formula:

$$Score(l_j) = (1 - \alpha)Gain(l_j) - \alpha Loss(l_j) \tag{6.2}$$

but since the $Loss$ part of the score is always 0 in this implementation then the score of an l-node is practically just its $Gain$ part.

– `updatePnodeScores` - this method just sums up the scores of all `Pnode` objects in the p-directory. It is synchronized on the `Cnode` object because it can be called by different threads, the thread of the matcher and a `ProxyPnode`'s thread.

– `updateCnodeScore` - this method calculates the overall score of a c-node which is reduced to just the sum of the results of the `updatePnodeScores` and `updateLnodeScore` methods because of the lack of the $Loss$ part of

the scoring formula. Like the `updatePnodeScores` method this method is also synchronized for the same reason.

- `scoreUpdateToTop` - this method is called by a `ProxyPnode` located in the p-directory of the `Cnode` object and is the only method that goes "up the tree". It causes this c-node to update its score and then calls the same method of this c-nodes parent node and continues like that until it reaches the root node of the BE-Tree (sub)tree. In the root node instead of calling the same method on the node's parent node (which is a `null` reference) the method calls the `notifyParentMatcher` method of the matcher the node is located on.
- `highestMatchingTimePNode` - since this method has to find a first-level p-node (counting from the first caller) with the highest last matching time it only traverses the collection of `Pnode` objects of the p-directory and calls their `getMatchingTime` methods to get the required information.

6.2.2. The `ProxyPnode` class

The `ProxyPnode` is a very specific class which links the two otherwise independent main aspects of the project - the logical BE-Tree part and the architectural matcher tree part. It is formally a part of the BE-Tree but does none of the BE-Tree's functions. Instead it creates a new matcher component that only it can directly communicate with and then delegates all the BE-Tree requests it receives from its parent c-node to its `ChildMatcher` by sending it messages via the standard OS process input stream.

The class implements the `Pnode` interface to be able to be put in a p-directory of a `Cnode` object but has a number of methods that return always the same result or even throw an `UnsupportedOperationException`. For example, the `isEmpty` method always returns `false` and the `getMatchingTime` method always returns 0 while the `scoreUpdateToTop` and `highestMatchingTimePNode` methods throw an exception because it makes no sense to call those methods on a `ProxyPnode` and that should never happen.

There is also a subclass similar to the `InternalCommunicationsThread` of the component implementations (explained later in Section 6.3) called the `ChildCommunicationsThread`. An instance of this subclass is created when the child matcher is created and is started in a new `Thread` object that keeps running as long as the child matcher process is alive. Its purpose is to constantly read the input stream from the child matcher and process any messages that the child matcher sends up the matcher tree.

Being a transparent interface between the BE-Tree inside of a matcher and its child matcher located in a different process the `ProxyPnode` class is the most sensitive and complicated part of the system. It has to deal with complicated issues such as multiple threads making changes to the same BE-Tree nodes and the child matcher being merged while some requests have already been sent. All of this required extreme care and complicated synchronization solutions.

The first thing that needed to be observed is that communication with a child matcher is different depending on whether the child matcher is a leaf matcher in the matcher tree. The first reason for that are messages like the `SubscriberUnregisterMessage` that flood the matcher tree but only a leaf matcher should send a reply to them. But the far more important reason is the merging of matchers which can happen only if the child matcher is a leaf matcher.

If the child matcher is a leaf matcher the matcher merging process can begin at any given time as a response to a message from the child matcher. The important thing to note here is that the methods that send messages to the child matcher are called from inside the current matcher's BE-Tree, which means by the main matcher thread, while the methods that run the merging process are called by the thread that receives the messages from the child matcher. This means that when the merging process begins sending messages to the child matcher has to somehow be blocked. Since the only common point of those actions is the `ProxyPnode` itself it makes sense to make all the methods that send messages to the child matcher and the method that does the merging of matchers synchronized on a `ProxyPnode` object.

This however doesn't solve another important issue. Since the communication between matchers is asynchronized the child matcher's responses to a request come with a delay in which the current matcher is free to send more messages to the child matcher. Since the child matcher is located in a completely different process those messages are sent and buffered in the process's input stream regardless of whether the child matcher processes them or not. Therefore all the messages sent to a child matcher in the time between the message that causes it to merge and the initiation of the merging process by the `ProxyPnode` would be lost which is unacceptable. The solution to this problem is a message queue in the `ProxyPnode` class that will save all messages sent to the child matcher and remove them from the queue upon receiving a response. This however requires a child matcher to respond to every request it receives, which is not generally necessary, and is no good if the child matcher is not a leaf matcher because most responses are forwarded all the way to the RootMatcher component and so responses from different matchers can be mixed in a single `ProxyPnode`. Luckily

there is no need for a message queue in a `ProxyPnode` if its child matcher is not a leaf matcher for the reasons already explained. So the compromise is made that only leaf matchers must respond to every request they receive and those additional responses are not forwarded up the matcher tree. The only complication now is the turning of the message queue on and off as the child matcher can stop being a leaf matcher and also become a leaf matcher again if it wasn't for a certain period of time. That problem is fixed with some thread synchronization and a variable for message synchronization which will be explained later in some more detail.

In the following text the constructor and some of the more important methods of the class will be described in order for the reader to better understand the solutions to the just mentioned problems. After that an overview of the actions triggered by messages from the child matcher will be given which will complete the description of the `ProxyPnode` class.

The constructor of the class receives a `RealPnode` as an argument. That object represents not only a single node but the whole BE-Tree branch "below" it by reference. The constructor then creates a new process on the operating system by doing an equivalent of calling "`java -cp <classpath> hr.fer.tel.pubsub.entity.broker.ChildMatcher <args>`" on the command line where the "`<classpath>`" part is taken from the current matcher (which it received from the CloudBroker component) and the "`<args>`" part are a bunch of different arguments like the UDP port of the DeliveryService component. After the process is created the BE-Tree branch given as the argument of the constructor is serialized and sent to the new process after which a `ChildCommunicationsThread` object is created and started in a separate thread. If all of that went without exceptions being thrown the `RealPnode` is replaced by the new `ProxyPnode` in the p-directory of the `RealPnode`'s parent `Cnode` and the *numChildren* variable of the current matcher is incremented by 1. The incrementation of the *numChildren* counter is also synchronized (on a mutex object in the `AbstractMatcher` class) and if incremented from 0 to 1 causes the sending of a message to the current matcher's parent matcher informing it that its child matcher (the current matcher of the `ProxyPnode`) has just stopped being a leaf matcher.

In order to better understand the BE-Tree methods the reader should first understand the merging process. The merging process is done by the `mergeWithChild` private method which is synchronized on the `ProxyPnode` object. This synchro-

nization makes sure that the BE-Tree methods are blocked from execution while the merging process is not finished. The argument of the method is a `RealPnode` object that is the root of the BE-Tree branch that the `ProxyPnode` received from the child matcher. A `null` argument means that the child matcher is empty and in that case the method constructs a new `RealPnode` to work with.

The first thing this method does is destroy the child matcher process and sets the `stopped` Boolean class-level flag to `true`. After that it takes all the messages from the queue of sent but unresponded messages and processes them in a standard way over the `RealPnode` received from the child matcher. That way no messages get lost in the merging process. When the updating of the former child's BE-Tree branch is finished it is put in the `ProxyPnode`'s parent `Cnode` p-directory instead of the `ProxyPnode` and the `scoreUpdateToTop` method of the parent `Cnode` is called with a `ChildMatcherRemovedMessage` object as its argument. The last thing the `mergeWithChild` method does is the opposite of the last thing the constructor does. It decreases the *numChildren* class-level variable of the current matcher by 1. The decrementation of the *numChildren* counter is also synchronized (on a mutex object in the `AbstractMatcher` class) and if decremented from 1 to 0 causes the sending of a message to the current matcher's parent matcher informing it that its child matcher (the current matcher of the `ProxyPnode`) again became a leaf matcher.

The methods `findMatchingSubscribers`, `insert`, `remove` and `deleteSubscriber` (which I will grouply refer to as "BE-Tree methods") all follow the same idea so I'm not going to explain them separately. As mentioned before they are all synchronized on the `ProxyPnode` object which makes them wait while the merging process is being done. However, once the method is called it cannot be uncalled which leaves the case of what happens after the merging has been done and the `ProxyPnode` is no longer a part of the BE-Tree. Since the `mergeWithChild` method sets the `stopped` flag a BE-Tree method can check if a merging has occurred while it waited. If that happened the method just gets the new `RealPnode` from the parent `Cnode`'s p-directory and calls its same-named method and returns whatever it returns. That way a "bypass" around the old `ProxyPnode` is done for all that call its methods before they register the change in the `Cnode`'s p-directory.

The second thing a BE-Tree method does is that it adds the message received as the method argument into the `ProxyPnode`'s message queue, unless the queue is a `null` reference which indicates that the child matcher is not a leaf matcher. Also if the method finds the queue empty at this point it also saves the ID of the message into the

firstMsgID class variable. This information is necessary to synchronize the messages once the child matcher becomes a leaf matcher again because responses to messages before it became a leaf matcher could still arrive and they shouldn't result in message polling from the queue.

The final part of a BE-Tree method is simply forwarding to the child matcher the message received as the method argument and returning a fixed value such as 0 for the `findMatchingSubscriber` method, the SUB_FORWARDED constant for the `insert` method or the UNSUB_FORWARDED constant for the `remove` method.

The last method left to describe is the `requestProcessed` method. This method is also synchronized on the `ProxyPnode` object which means that it also blocks the execution of the BE-Tree methods. That is necessary because this method polls messages from the message queue and has to do all of it as an atomic operation. The method is triggered by a message received from the child matcher. That message is the argument of the method. The method first checks that the queue is not a `null` reference, then it checks if the *firstMsgID* variable is set and if its value is equal to the ID of the argument message (which should be the same for a request-response message pair). If the IDs match or the *firstMsgID* variable was not set a message is polled and the *firstMsgID* set to `null`. This effecively synchronizes the message queue by letting all the responses to messages sent before the message queue was "turned on" go by without polling from the queue. The first response message that manages to trigger a poll from the queue is a response to the first message put into the queue after it was "turned on".

The `ChildCommunicationsThread` subclass of the `ProxyPnode` is a processor as well as a receiver of messages from the child matcher. It can process seven different messages the child matcher can send and they are very different in the way they have to be handled.

Perhaps the most standard ones are the `SubscriberDeletedMessage`, the `SubscriptionRemovedMessage` and the `SubscriptionAddedMessage`. They are all handled in a similar fashion. First the `requestProcessed` method is called and the score of the `ProxyPnode` set to the value carried by the message object. Then the `scoreUpdateToTop` method of the parent `Cnode` is called but only if the subscription was successfully added or removed and if the score is not negative infinity. A negative infinity score of the child matcher indicates that the child matcher's BE-Tree is empty and that triggers the merging, or rather deleting, of the child matcher.

This is done by calling the `mergeWithChild` method with a `null` argument followed by a `return` statement that will exit the `run` method and consequently stop the thread. Besides those three messages only a `ChildMatcherRemovedMessage` has to reach the RootMatcher component. It's processing consists of simply setting the new `ProxyPnode` score and calling the `scoreUpdateToTop` method of the parent `Cnode`.

The remaining three messages are only processed locally and don't propagate up the matcher tree. The `MatcherMergingMessage` triggers a call to the `mergeWithChild` method after which it simply returns from the `run` method which will cause the thread to stop and make the `ProxyPnode` garbage-collectable. The `PublicationProcessedMessage` is the simplest to process. It is only sent by leaf matchers and its only purpose is to poll a publish message from the message queue. The `LeafMatcherMessage` is specific for the fact that it is the only message sent from the child matcher self-initiatively. Its processing depends on a Boolean flag it's carrying. If `true` a new `ProxyPnode` message queue is constructed and the *firstMsgID* variable set to `null`. If `false` both the message queue and the variable are set to `false`. In any case the operations are done synchronized on the `ProxyPnode` so they are atomic in respect to the BE-Tree methods.

6.3. Broker implementation

The other important package in the `CloudBeTreePubSub` project is the `hr.fer.tel.pubsub.entity` package where implementations of all the entities, i.e. broker, subscriber and publisher, are located. The `...entity.broker` package contains all the classes that implement the system's main component - the centralised cloud-based broker. There are three classes that are not implementations of broker components, i.e. are not run in separate processes. They are:

- `PublisherForBroker` class - this class is the implementation of the aforementioned publisher handler (Section 5.2). It is simply a representation of a publisher on the broker. It hold the basic information about the publisher like its `UUID` identifier, its IP address and its local port. It also implements the `Runnable` interface which makes it runnable in a thread and that is its main feature. The `run` method continually reads objects from the input stream of the TCP connection to the publisher and processes the received messages. There are only two types of messages intended for it to receive:

the PublisherDisconnectMessage and the PublishMessage. The reception of a PublishMessage object triggers a call to the publish method of the MessageReceiver object that created the PublisherForBroker. The reception of a PublisherDisconnectMessage simply causes the while loop to break, which makes the run method return after it terminates the TCP connection with the publisher and calls the removePublisher method of the MessageReceiver.

– SubscriberForBroker class - this class is the implementation of the aforementioned subscriber handler (Section 5.2). Its purpose is the same as that of the PublisherForBroker class, as is the information it holds about the subscriber and the implementation idea. The only difference is in the messages it can receive and the state that it can be in. What I mean by the "state that it can be in" is that since a subscriber can be registered while being disconnected a SubscriberForBroker object can exist without running in a thread. Upon receiveing a SubscriberDisconnectMessage object the SubscriberForBroker simply breaks the TCP connection and forwards the message to the CloudBroker but doesn't remove itself from any structure that may hold it inside the MessageReceiver object. That is done when it receives a SubscriberUnregisterMessage. Then after breaking the TCP connection it calls the removeSubscriber method of the MessageReceiver.

– SubscriberQueue class - this class contains a queue structure for holding NotifyMessage objects to be sent to a subscriber, a TCP connection to a subscriber and implements a mechanism for automatic, continual sending of queue elements to the subscriber whenever a TCP connection is available. The queue structure used is an instance of the MinimalistLinkedHashQueue class described in Section 6.1 and made specifically for this use. The SubscriberQueue implements the Runnable interface and is run inside a separate thread whenever a TCP connection with the subscriber is established. The run method is simply a loop that in each iteration takes the first message from the queue and tries to send it to the subscriber. It does this synchronized on a mutex object so messages can be sent atomically. This is important because messages have to also be put in the queue atomically using the put method and removed from it atomically using the remove method since different threads execute those methods.

The implementations of the three static broker components are: the `MessageReceiver` class, the `DeliveryService` class and the `CloudBroker` class. The base for a matcher component is the `AbstractMatcher` abstract class which implements most of the matcher functionality. It is extended by the `RootMatcher` and `ChildMatcher` classes which implement the subtle differences between the two kinds of matchers. Each component implementation, except the `CloudBroker`, contains a subclass called `InternalCommunicationsThread` that implements the `Runnable` interface and is used as a thread that constantly waits for incoming messages from other components and starts their processing. This thread is started in the constructor of every component implementation and is what keeps alive the process that the component was started in.

6.3.1. The "CloudBroker" component

Instead of having a single `InternalCommunicationsThread` subclass like the other component implementations, the `CloudBroker` class has three different subclasses with similar tasks, one for each component the CloudBroker is connected to. Their names are: the `MessageReceiverRelay`, the `DeliveryServiceRelay` and the `RootMatcherRelay`, and they are called "relays" because all they do is read messages from the input stream of a component and then send them to other components via their respective output streams, perhaps writing a message to the log in the process.

Beside the relaying function the CloudBroker's only use is to setup the broker parameters by reading various properties files, create all the other components and to start and shutdown the broker on command.

In its constructor the `CloudBroker` class loads the broker properties file that contains all the broker parameters like its name and port, the matching and merging thresholds, the logging and testing flags and so on. It also loads two other properties files the paths to which should be in the broker properties file. One should hold the attributes for the BE-Tree and be loaded into the `BETreeParams` singleton class, and the other should hold all the allowed attributes and their allowed values and be loaded into the `Attributes` singleton class. After loading all the parameters the constructor creates the DeliveryService, MessageReceiver and RootMatcher components and starts their respective relay threads. It creates all of them by an equivalent of writing "`java -cp <classpath> <class package and name> <args>`" in the command line where "`classpath`" is a string parameter in the prop-

erties file and "args" are various arguments for the components' constructors. After starting a component the Attributes and/or BETreeParams singleton classes are serialized and sent to it so they are the same on all components. They are received and set in the constructors of the components' respective InternalCommunicationsThread subclasses.

When the start public method is called a ComponentStartMessage object is sent to both the MessageReceiver and the DeliveryService components which triggers a call of the start methods of their respective implementation classes. The shutdown method simply kills the CloudBroker's process which triggers an automatic chain reaction of component shutdowns because of the closing of the processes input/output streams.

6.3.2. The "MessageReceiver" component

The MessageReceiver class contains two subclasses in addition to its InternalCommunicationsThread: the BrokerListenerThread and the BrokerServingThread classes, both of which implement the Runnable interface. The BrokerListenerThread subclass is constructed and started inside its own Thread object in the start method. The start method is called when the InternalCommunicationsThread receives a StartComponentMessage object from the CloudBroker. The only other object the InternalCommunicationsThread of the MessageReceiver can process is an instance of the SubscriberDisconnectMessage which triggers the teardown of the TCP connection to a subscriber.

The class also contains three collections of objects: a list of all publications that entered the broker, a list of active publishers as a map of PublisherForBroker objects mapped to publisher IDs, and a list of active subscribers as a map of SubscriberForBroker objects mapped to subscriber IDs.

The BrokerListenerThread contains a ServerSocket object which means that it listens for incoming TCP connection requests. In fact that is all the BrokerListenerThread does, accept TCP connections in a loop as long as the MessageReceiver process is alive. When it accepts a TCP connection a Socket object is created which is then passed to a newly created BrokerServingThread object which is in turn started in its own new thread so the thread running the BrokerListenerThread object can go back to accepting new connections.

The purpose of the BrokerServingThread is to perform the initial steps in

connecting a publisher or a subscriber to the broker. Accordingly it accepts only two kinds of messages: the `PublisherRegisterMessage` and the `SubscriberRegisterMessage`, and only one copy of it before its `run` method ends. This is necessary because at the initial TCP connection establishment the broker cannot know if a publisher or a subscriber is connecting to it. If a `PublisherRegisterMessage` is received a new `PublisherForBroker` object is created and placed in the *activePublishers* map. The `Socket` object representing the TCP connection is given to the publisher handler and it is run in a separate thread the communicates with the publisher from that point on. If a `SubscriberRegisterMessage` is received it is first checked if the *activeSubscribers* map contains an entry under the subscriber ID in the message. If it does it means that the subscriber was just disconnected and the old `SubscriberForBroker` object is given the new `Socket` object and ran in a new thread. If there was no old entry in the map a new `SubscriberForBroker` object is created and everything else is done the same. At the end the `RegisterSubscriberMessage` is forwarded to the CloudBroker component for relaying to the DeliveryService component.

6.3.3. The "DeliveryService" component

In the construction of a `DeliveryService` object a `HashMap` called a *queueDirectory* is made and an `InternalCommunicationsThread` object is instantiated and started in a new `Thread` object. The *queueDirectory*'s keys are `UUID` objects representing subscriber ID's and its values are `SubscriberQueue` objects. The `InternalCommunicationsThread` of the DeliveryService can process the following five messages:

- `StartComponentMessage` - when this message is received the `start` method is called which simply constructs a new `NotifyReceiverThread` and runs it in a new `Thread` object.
- `InitialMatchesMessage` - this message contains a list of `Publication` objects that all have to be put in a queue of the subscriber whose ID is also carried by the message. This is done in the `bulkNotify` method by simply getting the subscriber's `SubscriberQueue` object from the *queueDirectory* map by its ID and then constructing a `NotifyMessage` for each `Publication` and putting in the queue.
- `SubscriberRegisterMessage` - upon receiving this message the *queueDirectory* is searched for an existing queue in the case the subscriber

was just disconnected. If not found a new `SubscriberQueue` object is constructed and placed in the *queueDirectory*. Then a TCP connection to the subscriber is established using information from the message sent by the subscriber and forwarded to the DeliveryService through the broker. The resulting `Socket` object is then given to the `SubscriberQueue` object which is then ran within a new thread so the contents of the queue can be sent to the subscriber.

- `SubscriberDisconnectMessage` - this message causes only the TCP connection to the subscriber to be terminated which will automatically stop the thread running the subscriber's `SubscriberQueue` object, but the queue itself will not be removed from the *queueDirectory* map.
- `subscriberUnregisterMessage` - this message causes both the termination of the TCP connection as well as the removal of the `SubscriberQueue` object from the *queueDirectory* map of the DeliveryService.

The purpose of the `NotifyReceiverThread` object ran in a thread started in the `start` method is to continually accept UDP packets from matcher components. Those UDP packets should contain a `PublishMessage` object containing an `ActivePublication` and a set of subscriber ID's. Upon receiving a UDP packet the `NotifyReceiverThread` will unpack the information and call the `notifySubscribers` method of the `DeliveryService`. The `notifySubscribers` method will get a `SubscriberQueue` object for each of the subscribers, by using the *queueDirectory* and the subscriber ID's, and then put a `NotifyMessage` object containing the original publication in the queues. In case the request was actually an *unpublish* (indicated by a flag in the `PublishMessage`) the publication will try to be removed from the queues, and if not found a `PublishMessage` with a flag indicating unpublication will be put in the queue to be sent to the subscribers instead.

6.3.4. Matchers

As already mentioned in the beginning of Section 6.3, the base of all matcher components is the `AbstractMatcher` abstract class. This class implements the singleton pattern which is necessary in order for the `ProxyPnode` class to have access to its "home" matcher. This singleton instance is set in the `main` method at the starting of the matcher component.

The two matcher implementation classes, `RootMatcher` and `ChildMatcher`, extend the `AbstractMatcher` class and implement only two things by which they fundamentally differ. Those are the `notifyParentMatcher` method and the constructor of the `InternalCommunicationsThread`, both of which are defined as abstract in the `AbstractMatcher` class.

Beside the `InternalCommunicationsThread` there is another subclass of the `AbstractMatcher` - the `UDPMatchingResultsManager`. Unlike every other subclass in the project it isn't started in a separate thread. It's instantiated only once, in the constructor, and its purpose is to construct a UDP socket and wrap the UDP communication with the DeliveryService in a simple method.

The most important information held by a matcher is located in the *branchRoot* `BeTreeActiveNode` and the *numchildren* integer counter. Both of them are fields of the `AbstractMatcher` class and the later also has a related mutex object for thread synchronization when making changes to its value.

The `InternalCommunicationsThread` of a matcher knows how to process only the three messages that can reach the RootMatcher from the CloudBroker, i.e. the ones that have an effect on the BE-Tree or initiate the matching process.

A `SubscriberUnregisterMessage` is processed simply by calling the `deleteSubscriber` method over the *branchRoot*. If the matcher is a leaf matcher (*numChildren* is 0) then the `notifyParentMatcher` method is called with a `new SubscriberDeletedMessage` object.

The processing of a `SubscribeMessage` object depends on its *unsubscribe* flag but is logically identical. Either the `removeSubscription` or the `addSubscription` method is called upon the *branchRoot*. Depending on the return value of the called method the `notifyParentMatcher` method is called with the appropriate message object to send to the parent matcher (described in Section 5.3, Figures 5.5 and 5.6).

The matching process on a matcher begins with it receiving a `PublishMessage` object. The matching part of the process consists of three simple steps: 1) call the `findMatchingSubscribers` method of the *branchRoot*, 2) use the returned value to update the minimum and maximum matching time in the matching window, 3) call the `send` method of the `UDPMatchingResultsManager` that sends the results of the local matching to the DeliveryService. After the matching part is done a decision has to be made whether to split or merge the matcher. The condition for splitting is checked first and if it's satisfied the `highestMatchingTimePNode` method of

the *branchRoot* is called to find the splitting node after which a new `ProxyPnode` is constructed with that node as the argument. All necessary actions for splitting are done in the `ProxyPnode` and are described in Section 6.2.2. If the splitting condition is not satisfied the merging condition is checked. The action for merging is just sending a `MatcherMergingMessage` object via the `notifyParentMatcher` method which the corresponding `ProxyPnode` will receive and process accordingly, as is described in Section 6.2.2. If neither of the conditions were satisfied so far then only if the matcher is a leaf matcher a `PublicationProcessedMessage` object is sent to the parent matcher, otherwise the matching process is over.

The only thing left to describe are the matcher implementation classes. Both of them contain very little code although the `ChildMatcher`'s is a lot more functional. The `ChildMatcher`'s `InternalCommunicationsThread` implementation has a constructor that reads the BE-Tree branch from the input stream from the parent matcher and saves a reference to it in the *branchRoot* class-level field. The `notifyParentMatcher` method contains code that inserts the current score of the root node into `SubscriberDeletedMessage`, `SubscriptionRemovedMessage`, `SubscriptionAddedMessage` and `ChildMatcherRemovedMessage` objects before it sends them to the parent matcher through the process's output stream. Also it checks if the root node of the BE-Tree branch is empty in which case it puts negative infinity as the score to indicate emptiness to the parent matcher and sets the *matcherRunning* field of the `AbstractMatcher` class to `false` to stop further processing of messages by the `InternalCommunicationsThread`. It also fills the `MatcherMergingMessage` with the BE-Tree branch before sending it to the parent matcher. All other message types are just forwarded to the parent matcher without change.

The `RootMatcher` class contains almost no functional code. The `BrokerComm` subclass that extends the `InternalCommunicationsThread` adds no code to it at all. The `notifyParentMatcher` method, defined as abstract in the `AbstractMatcher` class, also has no functional code, merely logging. And the last part (or the first) is the constructor that simply starts the `InternalCommunicationsThread` and creates a new, empty `Cnode` object as the root of the BE-Tree.

7. Experimental evaluation

To test the functionality of the developed system four experiments have been performed, one for each developed publish-subscribe system version. The centralised versions were tested to compare the covering forest and the BE-Tree matching algorithms (Section 7.1), and the cloud versions were tested to investigate the general behaviour of the developed system, and to compare its performance with a different, more simple cloud architecture (Section 7.2).

All four experiments have been performed on two desktop computers with characteristics given in Table 7.1, and connected directly with an Ethernet cable. In all experiments one broker was running on one of the computers, while the second tested one subscriber and one publisher.

Table 7.1: Computer characteristics

Processor	Intel Core i3-2120	
	# of cores	2
	clock speed	3.3 GHz
Working memory	4 GB	
Operating system	Windows 7 Professional	

The data used in the experiments are real world sensor data collected during the scope of the OpenSense project[1]. In particular, I have used five files, located in the "testdata" folder, which contain various sensor measurements and their timestamps:

- the "*carbon-monokside*" file contains 76 rows, each of which contains measurements from 5 sensors of the carbon-monoxide gas in the air (ppm[2]).
- the "*ozone*" file contains 909 rows, each of which contains measurements from

[1] http://opensense.epfl.ch

[2] ppm = parts per million

10 sensors of the ozone gas in the air (ppb[3]).

- the "*schimmelstrasse*" file contains 919 rows, each of which contains measurements from 5 sensors of the O_3 (ppb), CO (ppm), NO (ppb), NO_x (ppb) and NO_2 (ppb) gasses in the air.
- the "*stampfenbachstrasse*" file contains 919 rows, each of which contains the same type of information as the rows in the "*schimmelstrasse*" file
- the "*ultrafine-particles*" file contains 808 rows, each of which contains measurements from 10 sensors of ultra-fine particles in the air, although many rows miss some of the measurements.

The classes used for testing are located in the "*test*" folder of each project, and are named the same and work identically in all the projects. The testing class that starts the publisher is called `TestdataPublisher`, and is located in the `hr.fer.tel.pubsub.entity.publisher` package. Its main method expects three command-line arguments: the location of the configuration file for the publisher, the location of the testing data files and the number of publications it has to generate. When started, it creates a new `Publisher` class, and reads the data from all five data files into lists of strings, where each string is a row of a file. Before publishing the sensor data, a `Random` object is created, but with a fixed "*seed*" number. Because of this, exactly the same "random" data will be used at each testing execution, which is important for comparison of experimental results.

The generation of publications is done in a loop by randomly choosing a row in each of the files, and publishing all sensor measurements from those rows as separate publications until the number of published publications exceeds the number given as the command-line argument. At the end of each loop iteration, a publication with a "test" attribute and the number of published publications is created and published. This is done so the subscriber doesn't write to the screen all the publications it receives (which slows it down), but only writes these "test" ones instead.

The testing class that starts the subscriber is called `TestdataSubscriber`, and is located in the `hr.fer.tel.pubsub.entity.subscriber` package. Until it comes to the generation of subscriptions, it is in every way the same as the `TestdataPublisher` class. It accepts the same three command-line arguments, creates a `Subscriber` class, reads the data from the same files and creates a `Random` object with a fixed "*seed*" number.

[3] ppb = parts per billion

The generation of subscriptions is split in five "for" loops, one for each of the data files. One fifth of the specified number of subscriptions is generated from each of the files, and out of that fifth three quarters are subscriptions on sensor measurements, and the rest are subscriptions on timestamps. The subscriptions on timestamps are made to give "width" to the generated set of subscriptions, "width" in the sense of less overlapping and mutual covering. Each of the subscriptions on sensor measurements are made by randomly choosing a row of a data file, and then choosing a random measurement in that row and a random numerical operator (BETWEEN excluded) to create a `Triplet` object. All generated subscriptions contain only a single `Triplet` object.

7.1. Performance evaluation of the centralised versions

The performance evaluation of the centralised publish-subscribe version was done primarily to test the BE-Tree algorithm implementation, and to compare its results to the results of the covering forest algorithm. All measurements were done by first starting a broker on the first computer followed by starting the subscriber and then the publisher on the second computer. As soon as the subscriber is started, all subscriptions are generated and sent to the broker, and also as soon as the publisher is started all publications are generated and sent sequentially to the broker as fast as possible. Three measurements were done for each combination of the number of subscriptions and publications, and the components were shut down and restarted in the same order for each measurement.

The measured time is the sum of the execution times of the `publish` method of the broker, which was done by adding timing code inside the `publish` method. The `publish` method includes the following operations:

1. finding the publisher handler by a hash table look-up,
2. checking if the publication satisfies the attribute constraints (`checkPublication` method of the `Attributes` class),
3. adding the publication to the `HashSet` of active publications,
4. doing the matching process, i.e. finding all subscribers that need to be notified,
5. finding the subscriber queue for each of the subscribers (a hash table look-up),

6. creating a `NotifyMessage` object for each of the subscribers and putting it into the subscriber's `SubscriberQueue`,

which approximately corresponds to the pure processing time of a single publication, without its system entry and exit times that are influenced by the network.

The experimental results of the system using the covering forest algorithm are given in Figure 7.1, and the results of the system with the BE-Tree algorithm are given in Figure 7.2.

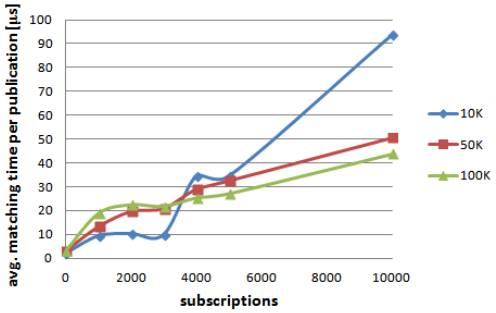

Figure 7.1: Testing results for the covering forest centralised publish-subscribe system

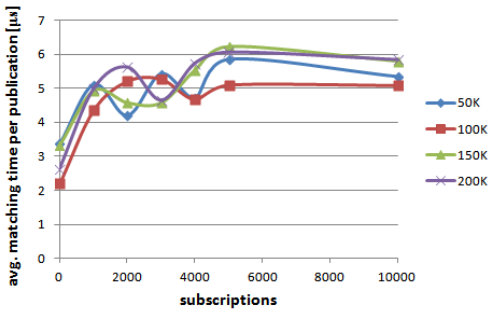

Figure 7.2: Testing results for the BE-Tree centralised publish-subscribe system

Each of the graph lines in figures is a plot of the average processing time for a single publication, given in microseconds, against the number of subscriptions. As it can be seen, the measurements were made for 0 subscriptions up to 5000 subscriptions with a step of 1000 subscriptions, and an extra measurement for 10000 subscriptions to see if the behaviour is consistent. The measurements were made when 10000, 50000 and 100000 publications were generated, in the case of the covering forest algorithm, and for 50000, 100000, 150000 and 200000 publications in the case of the BE-Tree algorithm. The number of publications for the BE-Tree algorithm is greater because its performance was surprisingly good, but the results can still be compared on the two overlapping amounts of publications. For 5000 subscriptions the performance of the BE-Tree algorithm is approximately 5 times faster while for 10000 subscriptions it is approximately 10 times faster. The amount of time needed by the BE-Tree centralised version to process a hundred thousand publications on ten thousand subscriptions is half a second on the average of three measurements, which is a great result. But the most important information from the plots is the general tendency of the processing time per publication of the system as it contains more subscriptions. For the covering

forest algorithm that tendency looks linear, but for the BE-Tree algorithm it is clearly logarithmic which is its biggest advantage in comparison.

7.2. Performance evaluation of the cloud versions

The experiments of the cloud-based systems were conducted based on the same principles as the experiments of the centralised systems, meaning that there were three runs for each combination of the number of subscriptions and publications, and that the system entities were run on the same computers and in the same order. The difference is only in the main system (the BE-Tree based cloud broker) that has to have a training period to build and stabilize the matcher tree before the time-measured testing can begin. This was done manually by publishing smaller numbers of publications (from few tens to few thousands) until the broker started to process them smoothly.

The difficulty with testing the developed cloud-based BE-Tree publish-subscribe system arises from its biggest advantages - its locality and weak connectedness. The whole system is designed in such a way that no single component has the knowledge about the state of the whole system and cannot control it directly. This causes a problem for testing because it is very difficult to monitor the current structure of the matcher tree, or even only the total number of matchers and the tree depth. The same is with any other information, including the most important one for testing - the matching times for each component. Therefore, in order to be able to collect that information, a lot of additions to the code had to be made, including two completely new message classes and code to process them on every component. With such code modifications, the matching times of all components can be reset (necessary after the training period) and collected on user input to the broker. When a user enters a sequence of keys, a message object is created that floods the matcher tree, and the other (static) components, and all components respond to the message and forward the others (responses of the components lower in the matcher tree) to the CloudBroker, which then prints the information on the screen.

Note that all the operations that are performed within a single method on the centralised systems are now performed across not only different methods or objects, but also different processes. And the matching operation is, therefore, burdened and constrained by the inter-process communication, which takes a lot of time, especially on a machine with only two cores and one processor to service a lot of processes and even more threads. All the operations that were timed, and the components that they were timed on, are as follows:

– **MessageReceiver**

1. finding the publisher handler with a simple hash table look-up
2. checking the publication with the `Attributes` singleton class
3. forwarding the `PublishMessage` to the CloudBroker (!)
4. adding the publication in the *activePublications* `HashSet`

– **CloudBroker**

5. forwarding the `PublishMessage` to the RootMatcher (!)

– **every matcher**

6. doing the matching process on the local BE-Tree
7. making a UDP packet and sending it to the DeliveryService (!)
8. responding to the parent matcher (only leaf matchers)

The operations marked with "(!)" are the ones that take a lot of time and are especially affected by the fact that the testing was done on a computer with little parallel processing capabilities. On a more suitable computer those times should be considerably lower.

Since a lot of processing is done in parallel on multiple matchers, the question is what exactly is the total matching time for all the published publications. In case of the main cloud system, the total matching time on the matchers is the total matching time of the RootMatcher, through which all publications pass, plus the difference between the end time of processing of the last publication on the RootMatcher and the matcher that last finished processing the last publication. After the experiment has ended, it was determined that the total matching time on the matcher tree could be approximated with just the matching time of the RootMatcher, because the described difference was almost always 0 ms. This leads to the conclusion that the matching time of the whole BE-Tree is equal to the matching time of the subtree left on the RootMatcher plus some inter-process communication overhead. The times that need to be added to the RootMatcher's processing time are the processing times of the MessageReceiver and the CloudBroker components. Figure 7.3 shows the results for the average matching time per publication of all components combined, while Figure 7.4 shows the results

for just the RootMatcher component, and Figure 7.5 depicts the results for the MessageReceiver and CloudBroker components combined, i.e. the matching overhead.

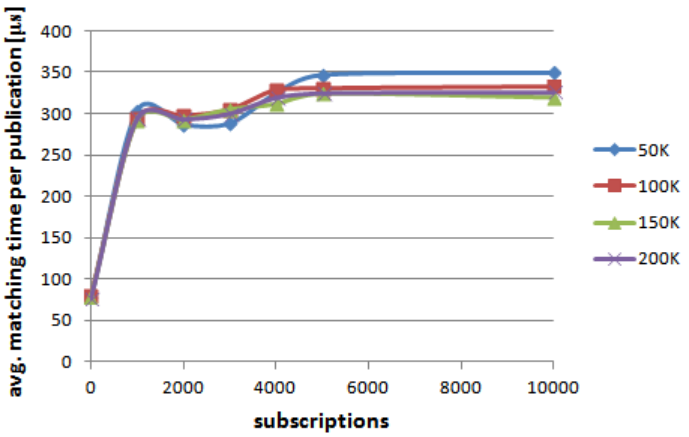

Figure 7.3: Overall average processing time for the cloud broker

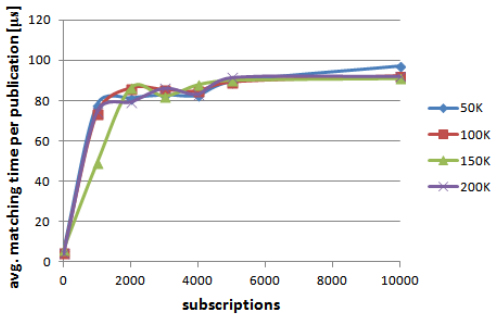

Figure 7.4: Average matching time for the cloud broker

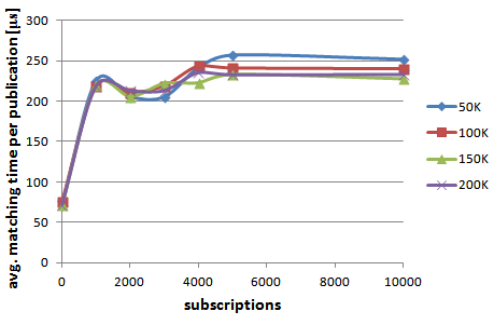

Figure 7.5: Average processing overhead time for the cloud broker

The graphs on Figure 7.3 show that the average processing time per publication is between 300 and 350 microseconds, which is around 50 times slower than the centralised BE-Tree system. Probably the biggest reason for such a huge difference is the lack of parallel processing power of the computer, which considerably slows down the inter-process communication of the many processes. But more important is the shape of the graph, which seem logarithmic and similar to the centralised BE-Tree version's results. Moreover, the graph in Figure 7.4, which shows the average matching times of just the matcher tree, is clearly similar to Figure 7.2 where a slow, but steady incline can be seen. In the other two figures the graph lines look more like constant functions, which is also an important result. The reason why the graph lines in Figure 7.3

look constant is because their shapes are dominated by the graph lines on Figure 7.5, which is due to the matching times on the matcher tree being almost three times lower than the matching times on the static components. The result, that the overhead processing times act as constant functions of the number of subscriptions, is both logical and good. This all means that the average matching time rises logarithmically to the number of subscribers and has a huge constant factor, which can probably be lowered significantly with proper hardware.

The other system that was tested has a different, more simple architecture, which is why it is referred to as the "primitive" cloud version. The idea was to measure the performance of a system that has a less complicated, non-dynamic architecture, but uses the same matching algorithm, in order to be able to compare the results of the main developed system and to see how much it loses or gains from the complicated architecture that gives it the important elasticity feature.

The "primitive" cloud version uses a fixed number of matcher components in parallel. Those matchers are the same as the RootMatcher component of the elastic system, but without the splitting possibility. The CloudBroker component of the "primitive" system starts all the matcher components at startup, and has a direct connection with all of them. The rest of the "primitive" system architecture is exactly the same as that of the main, elastic system.

This system has a simple way of processing subscriptions and publications. The CloudBroker distributes received `SubscribeMessage` objects in a round-robin fashion to the matcher components. That results in all of the N matchers having a BE-Tree containing one N-th of the total subscriptions on the broker. The `PublishMessage` objects are replicated on the CloudBroker and sent to all the matchers. This slows the broker down because of inappropriate hardware even more than in the elastic system's case, because the CloudBroker has to make N costly communication operations in a row, and the other components have to wait for it to finish, which makes the CloudBroker become the bottleneck of the system.

As with the elastic system there is a question of what is the real total matching time of all the publications. In the case of the elastic system, that was the matching time of the RootMatcher plus the difference between the matcher that last finished, which in the case of the "primitive" system clearly isn't applicable. Since all the matchers run in complete parallel, and get all the publications at the same time, the total matching time of the "primitive" system should be the maximum matching time among the matching times of all the matchers. The average overall matching times of the "primitive" system

are shown in Figure 7.6, and are calculated by adding the "overhead" matching times shown in Figure 7.8 to the maximum matcher matching times show in Figure 7.7. The number of matchers that were used is $N = 15$, which roughly equals the sizes of the matcher trees that were created in the testing of the elastic cloud system.

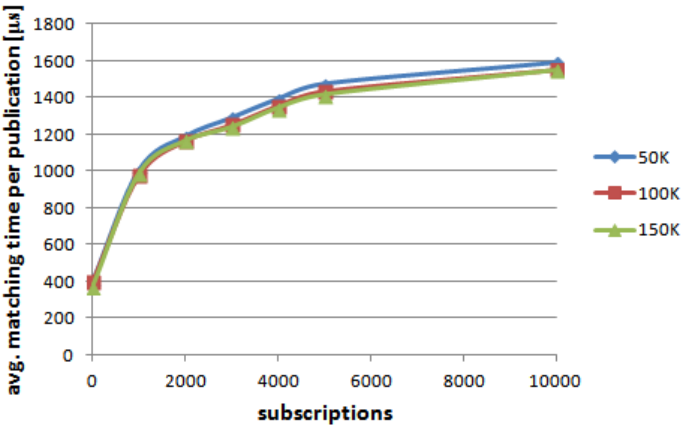

Figure 7.6: Overall average processing time for the "primitive" cloud broker

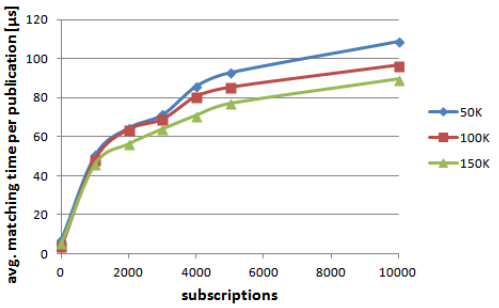

Figure 7.7: Average matching time for the "primitive" cloud broker

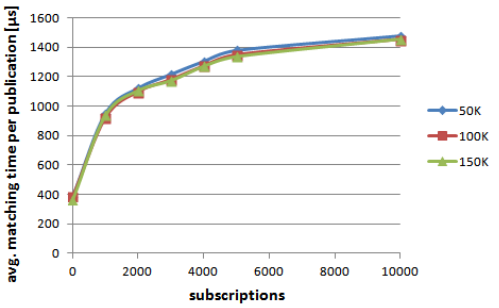

Figure 7.8: Average processing overhead time for the "primitive" cloud broker

The first obvious thing to see in Figure 7.6 is that the average matching times per publication are three to four times greater than in the elastic system case, and also that the graph lines have a more linear look to them. If we look at the matching times of just the matchers (Figure 7.7), we can see that at first the processing times are smaller than in the elastic cloud case, but later become equal and even greater, and more importantly, that after that point they have a linear-looking tendency to rise, as opposed to logarithmic. The reason that at first we detect better results is probably due to a less complicated architecture and the simplicity of each matcher on its own, while the cause of the linear-looking graph lines could be due to the fact that the BE-

Trees of each of the matchers are built on a random subset of all the subscriptions, which disables the BE-Tree structure to use the complete information of the whole subscription set and results in N smaller, but less efficient BE-Trees.

The overhead times of the "primitive" system are evidently huge. Even at 0 subscriptions they are almost twice the overhead times of the elastic system, and later on after 5000 subscriptions seem to grow linearly, ending in five to six times the overhead times compared to the elastic system. This is probably due to the fact that the CloudBroker becomes the communication bottleneck of the system, and the weak parallel processing power of the hardware. But even with better hardware the shapes of the lines would probably persist, and the elastic cloud system would offer better performance, especially for a large number of subscriptions, while retaining its biggest advantage - its elasticity.

8. Conclusion

The purpose of this thesis was to develop a model for cloud-based publish-subscribe services that could be used in services that have to process big amounts of streaming data in real-time, and deliver them in near real-time to a large number of users with dynamic interests.

The thesis presented the motivation for building such services, and also the benefits of the cloud approach against the commonly used, and much researched, distributed approach. The thesis provided an overview of the current work on cloud-based publish-subscribe models, and pointed out some of their shortcomings. The conclusion is that a publish-subscribe service in a cloud should be as loosely connected as possible, with as little information as possible being communicated internally, because that generates overhead that slows down the processing of publications. It is also concluded that interface components, to which publishers and subscribers connect, have to do as little work as possible in order to avoid becoming bottlenecks. A single interface component could be enough if its only job is to accept, check and forward subscriptions and publications to internal components of the broker. The general feeling is that simplicity should be strived for, because in the cloud environment things seem to get very complicated very easily when a lot of processes or VM's are used and share common information. Also, an argument is made about how there is no good reason why definition of allowed attributes, and their allowed values, at startup would be a bad thing, especially if it can make the service more efficient.

A recently developed algorithm for indexing and matching Boolean expressions was presented in the thesis - the BE-Tree algorithm [8]. The algorithm was apparently very good and showed excellent performance results, that were also confirmed by an experiment made as a part of this thesis. The idea of the thesis was to design a model based on the BE-Tree algorithm that satisfies all of the conclusions made by studying the existing cloud models. Such a model was designed and a cloud implementation of it was built and tested to evaluate its performance.

The developed model presents a way to distribute a single logical BE-Tree across

an unbounded number of matchers in the cloud, arranged in a self-organizing tree structure that can freely contract and expand to meet the requirements of the current workload. The connections between the matchers are transparent even to the matchers themselves, by being hidden in "proxy" BE-Tree nodes, making the whole broker structure as loosely coupled as possible. All broker components function completely asynchronous and in parallel.

The implementation has some limitations which are discussed later on, while the experiments were run on hardware that was not appropriate for the developed software, but still the performed experiments show promising results. The publications are delivered on an "at most once" basis, but are all processed on the broker BE-Tree. The ones that are not delivered are almost certainly not delivered due to UDP buffer overflowing on the DeliveryService component. The results show that the matching time per single publication over the whole BE-Tree is practically reduced to the matching time of a publication on just the top matcher component, effectively reducing the matching time by a factor of the number of matchers. This behaviour is consistent with full parallelization, however this model shows such behaviour while having components organized in a tree structure, which is naturally dynamic and has less communication overhead, instead of a common flat structure. The processing times do not allow a high throughput (about 3500 publications per second) but it is shown that the average processing time grows very slowly (logarithmically) with the number of subscriptions, and on proper hardware can probably be reduced to the level more close to that of the centralised BE-Tree broker, which is around 50 times smaller.

The BE-Tree algorithm implementation can be improved by adding various adaptation policies and optimizations suggested by the creators of the BE-Tree algorithm in their article [8]. The developed system also has room for improvement. For example, UDP communication with the DeliveryService could be replaced by TCP, and the CloudBroker component could be removed completely, making communication more direct and saving time on relaying through an otherwise functionless component. A better splitting and merging strategy could be developed as well, one based not only on the matching times on a single matcher, which depend mostly on the size of the matcher-local BE-Tree structure, but also on the intensity of incoming publications to a matcher and their queuing time.

Author signature:

BIBLIOGRAPHY

[1] Gianpaolo Cugola and Alessandro Margara. High-Performance Location-Aware Publish-Subscribe on GPUs. *Proceedings of the 13th International Middleware Conference*, 2012.

[2] Michael Freeston. A General Solution of the n-dimensional B-tree Problem. *Proceedings of the 1995 ACM SIGMOD international conference on Management of Data*, 1995.

[3] Vincenzo Gulisano, Ricardo Jiménez-Peris, Marta Patino-Martínez, Claudio Soriente, and Patrick Vladuriez. StreamCloud: An Elastic and Scalable Data Streaming System. *IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS*, 2012.

[4] Atsushi Ishii and Toyotaro Suzumura. Elastic Stream Computing with Clouds. *IEEE 4th International Conference on Cloud Computing*, 2011.

[5] Yannis Manolopoulos, Alexandros Nanopoulos, Apostolos N. Papadodpoulous, and Yannis Theodoridis. *R-Trees: Theory and Applications*. Springer-Verlag, 2006.

[6] Alessandro Margara and Gianpaolo Cugola. High Performance Publish-Subscribe Matching Using Parallel Hardware. *IEEE Transactions on Parallel and Distributed Systems*, 2013.

[7] Li Ming, Fan Ye, Kim Minkyong, Chen Han, and Hui Lei. BlueDove: A Scalable and Elastic Publish/Subscribe Service. *Proceedings of the 2011 IEEE International Parallel and Distributed Processing Symposium*, 2011.

[8] Mohammad Sadoghi and Hans-Arno Jacobsen. Analysis and Optimization for Boolean Expression Indexing. *ACM Transactions on Database Systems*, 2013.

[9] Lucija Zadrija. Usluga objavi-pretplati u računalnom oblaku. Master thesis, University of Zagreb, Faculty of Electrical Engineering and Computing, 2013.

[10] Yuquin Zhu, Jianmin Wang, and Chaokun Wang. Ripple: A pub/sub service for multidata item updates propagation in the cloud. *Journal of Network and Computer Applications*, 2011.

Appendix A
The format of the configuration files

There are three different configuration files needed to start an instance of the developed cloud broker. They can be distinguished by their standard suffixes, although there are no actual formal constraints on their names or suffixes.

The configuration file ending in ".config" is the main broker configuration file. It is written in standard Java `Properties` class format, which is a key and a value in the same row with an equals (=) sign between them. The properties that are defined in the main broker configuration file are properties of the broker as an entity, like its name, port numbers and merging and splitting thresholds. Two of the properties that also have to be defined are paths to the other two configuration files.

The configuration file ending in ".betree" is of the same format as the main configuration file, and holds the specific properties required by the BE-Tree algorithm, like the minimal and maximal size of an l-node. The contents of this configuration file are loaded into the `BETreeParams` singleton class on broker startup.

The last of the configuration files is the one ending in ".attributes", and in it are defined all the attributes, and their allowed values, that can be used in subscriptions and publications. The contents of this configuration file are loaded into the `Attributes` singleton class on broker startup, and are given in a special, custom format.

Each attribute has to be defined in a separate line. Empty lines are allowed, as are comment lines which have to start with the "%" sign. Two types of attributes can be defined: numeric attributes and string attributes. A numeric attribute is defined by stating its name, its lower bound, its upper bound and its discretization step, in the following format:

<attribute name>#<lower bound>:<upper bound>:<discretization step>

for example, `ozone4#0:200:0.001`. A string attribute is defined by stating its name, followed by all the possible values it can achieve, in the following format:

<attribute name>$<first value>:<second value>: ...:<nth value>

for example, `testText$cat:mouse:dog:bird:dolphin:whatever`.

A Dynamic and Elastic Publish-Subscribe Service for the Cloud Environment

Abstract

This thesis explores the publish-subscribe communication model and the cloud computing paradigm as two currently very popular technologies that are still not often used together. An argument is made about the potential of the combination of the two technologies in solving modern information flow problems, especially in mobile environments with big numbers of users that both consume and generate data. Some existing cloud-based publish-subscribe solutions are presented, and their weaknesses described. A description of the loosely-coupled, elastic cloud-based publish-subscribe service, developed for this thesis, is then given. The developed service is based on the idea of distributing a BE-Tree structure over multiple processes in a cloud and parallelizing the operations over it. In the end, results of experiments are presented that show relatively good performance and active potential of the BE-Tree algorithm and of the developed cloud-based publish-subscribe model.

Keywords: Publish-subscribe, cloud services, BE-Tree, data stream processing.

Dinamična i elastična usluga objavi-pretplati u računalnom oblaku

Sažetak

Ovaj rad proučava komunikacijski model objavi-pretplati te računarstvo u oblaku kao dvije vrlo popularne tehnologije koje se još uvijke ne koriste često zajedno. Argument je dan o potencijalu kombinacije ove dvije tehonologije u rješavanju problema modernih informacijskih tokova, posebno u slučaju mobilnih okruženja s velikom količinom korisnika koji istovremeno konzumiraju i proizvode informacije. Prezentirani su neki od postojećih objavi-pretplati sustava zasnovanih na oblaku te su opisani njihovi nedostaci i slabosti. Zatim je dan opis slabo-povezanog, elastičnog objavi-pretplati sustava zasnovanog na oblaku koji je razvijen u sklopu ovog diplomskog rada. Razvijeni sustav baziran je na ideji distribuiranja BE-Tree strukture na više procesa u oblaku te paralelizacije izvođenja radnji na njom. Na kraju su predstavljeni rezultati eksperimenata koji pokazuju relativno dobre rezultate i aktivni potencijal BE-Tree algoritma i razvijenog publish-subscribe modela za oblak.

Ključne riječi: Objavi-pretplati, usluge u oblaku, BE-Tree, obrada toka podataka.