Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Master's Thesis Nr. 41

Systems Group, Department of Computer Science, ETH Zurich

Privacy in Pub/Sub Systems

by

Lucas Victor Braun

Supervised by

Prof. Donald Kossmann
Tahmineh Sanamrad

October 2011 – April 2012

inf | Informatik
Computer Science

Contents

1 Introduction

Publisher/subscriber systems (pub/sub systems for short) have become very popular today and are used in a variety of applications. The principle of loosely coupled components states that publishers do not know about subscribers and vice-versa. Nevertheless, publishers may want to state to whom their data should be made available and to whom not. This is why there is a need for privacy.

One concrete scenario is the following: a certain car manufacturer (let's call it SUPER-CARS) assembles electronic cars with the capability of reporting their position, trajectory, batter level, etc. to a pub/sub system. This system is very useful for SUPER-CARS as they get feedback about the quality of their cars but may also be used by car drivers.

For example they may want their cars to automatically call an ambulance in case of an accident. On the other hand, they may want to restrict who gets what information about them and under what circumstances. In this thesis we will show how they can make even sophisticated statements like "If I have an accident, report my position to an emergency car and at regular time intervals report my battery level and speed to SUPER-CARS unless I'm driving too fast." Furthermore, we will also investigate how the published information can be encrypted in order to enforce such statements.

1.1 Pub/Sub Systems

One of the first examples of a pub/sub system was SIFT [17]. SIFT is an information dissemination system in which information comes from several different sources and is routed to subscribers which can state their interest in certain topics. One important property of a pub/sub system is the fact that publishers and subscribers are only loosely coupled, which means information flows from the former to the latter without them knowing about each other's existence. A further important property is that a pub/sub system allows asynchronous communication, which means that the information flows from a publisher to a subscriber even though they may not be on-line at the same time.

Today's pub/sub systems can be divided into three main categories as illustrated by Eugster et al. in [4]:

- **Topic-Based Pub/Sub:** To this category belong SIFT and similar systems that are based on the model of channels each of them associated with a keyword. A publisher can publish on several channels and a subscriber can subscribe to a set of channels, which allows him to get all information that is published on these channels.
- **Content-Based Pub/Sub:** Subscribers do not subscribe to static channels but give constraints on the content of messages. This is typically done

by means of a description language. In many cases boolean predicates are used. If, for example, the published messages consist of name-value pairs (e.g. message m = [stock='S-C', price='200', currency='CHF']), then the subscriptions are boolean predicates on these pairs. A subscriber could define the subscription s = [stock='S-C' AND (price < 190 OR price > 210)] in order to receive notifications if the S-C (SUPER-CARS) stock falls outside the defined range.

- **Type-Based Pub/Sub:** Instead of keyword-based topics, we have a class hierarchy which allows for sub-typing and inheritance. This type of pub/sub is suitable for applications that deal with messages that are represented by (Java -) business objects. Assume that the information system of an insurance company uses a pub/sub system in order to route customer requests to employees who need to know about them. There could be a general type *CustomerRequest* with subtypes *ChangePolicyRequest* and *ChangeAddressRequest*. There may be employees that need to be informed about all customer requests (be it a change of policy or a change of address), so they would subscribe to *CustomerRequest*. On the other hand, the secretary who manages all customer addresses would subscribe to *ChangeAddressRequest* as she is not interested in other customer requests.

1.2 Contributions

We have seen in the introductory example why it is crucial to make pub/sub systems private and some work has already been done in this area. What we try do in this thesis is to develop more general concepts for how privacy can be brought into the picture. We will give a general definition of private pub/sub systems and try to paint a picture of the design space of such systems. Another contribution is a concrete implementation for which we do a simple performance analysis.

1.3 Overview

The remainder of the thesis is organized as follows: in section 2 we give a definition of private pub/sub systems and try to describe their design space. In section 3 we show one concrete solution and discuss implementation ideas and alternatives. The benchmark results of the implementation are available in section 4 and section 5 contains a short conclusion. The complete results and a list of figures and tables can be found in the appendix.

2 Privacy in Pub/Sub Systems

In this section we present our concept for defining private pub/sub systems. As we assume such systems to be large-scale distributed, we will sometimes use the term "cloud" instead of "system". A user of a cloud can be a publisher, a subscriber or even both. The terms "user" and "agent" are used interchangeably.

2.1 Requirements

We require from our private pub/sub system to satisfy the following properties:

- **Asynchronous communication:** Information should flow from publisher to subscriber even if they are not simultaneously on-line. We will use persistent messaging in order to guarantee this.
- **Privacy:** Publishers have to be given as much flexibility as possible for defining to whom, under which circumstances as well as which data should be available. We will formulate the concept of a policy in order to address this.
- **Untrusted Cloud:** The data released by a publisher should not leak to any unauthorized users. This also includes the cloud operators. Therefore we have to think carefully about the encryption scheme we want to use.
- **Data Availability:** All data ever published should be stored in some form in the cloud. This means that the cloud also serves as a historical data storage which we may want to use at a certain point of time to do off-line analysis. We could require our encryption scheme to allow publishers to define not only read access on their data but also access for certain aggregate functions. They could formulate access statements like "SUPER-CARS should not be able to see the speed of my car, but can compute the average speed of all cars in the system". Of course, an access rule like this only makes sense if other car owners also agree to let SUPER-CARS compute the average speed. The implementation of such access rules requires a very sophisticated encryption scheme and is out of scope of this thesis.
- **Scalability:** Our system should scale to a large number (i.e. millions) of agents. This is why we have to make it distributed.

2.2 Agents

We model agents as objects that have two kinds of attributes:

- **Static attributes:** These are the attributes that characterize an agent. In the car example name, description, brand and model are static attributes. Static attributes do not change over time. Depending on the application we might want them to be certified.

- **Dynamic attributes:** These attributes change their values. Dynamic attributes contain the information to be published. Due to this fact, they are also called post or status attributes. In the car example position, trajectory and battery level are dynamic attributes.

2.3 Subscriptions

We want to formulate subscriptions in such a way that they are topic-based and content-based. This is why we allow a subscriber to put constraints on the static attributes of a publisher (which is in fact a topic-based subscription) as well as on the dynamic attributes (which corresponds to content-based subscription). Moreover a subscriber must state in which dynamic attributes he is interested in. Therefore a subscription consists of the following three parts:

- **Profile:** constraints on the static attributes of a publisher,
  e.g. *description = 'car' AND brand = 'SUPER-CARS'*
- **Condition:** constraints on the dynamic attributes of a message,
  e.g. *battery-status < 10%*
- **Visibility:** post attributes that a subscriber is interested in,
  e.g. {*x-position, y-position, battery-status*}

If we combine the examples from above we get a subscription in which a subscriber states that he is interested in x-position, y-position and battery-status of SUPER-CARS cars whenever they have a battery-status below 10%. This subscription would make sense for somebody who wants to offer his help to people whose car battery will be empty soon.

2.4 Policies

While every pub/sub system has of course the concept of a subscription, we introduce policies as a new concept that helps us defining privacy for publishers. A policy is somehow the symmetric counterpart of a subscription, it makes constraints on possible subscribers and states under which circumstances which information is released to whom. It again consists of three parts which have the same names as the parts of a subscription but with a slightly different meaning:

- **Profile:** constraints on the static attributes of a subscriber,
  e.g. *description = 'company' AND brand = 'SUPER-CARS'*
- **Condition:** constraints on the dynamic state of a publisher or, in other words, the dynamic attributes of the message the publisher wants to publish, e.g. *speed < 60 mph*
- **Visibility:** post attributes that a publisher wants to release to subscribers that match the profile,
  e.g. {*battery-status, x-acceleration, y-acceleration*}

An interpretation of the above examples would mean that a publisher wants to release his battery-status, x- and y-acceleration to all companies of brand SUPER-CARS but only if his speed is below 60 miles per hour.

In order to ensure real publisher privacy we need that the attributes of subscribers be certified. If a publisher is willing to release data to SUPER-CAR companies, somebody has to check and verify that a subscriber who claims to be such a company really is what he claims to be.

2.5 Design Space for Private Pub/Sub Systems

Before going into the details of our concrete design of a private pub/sub system, we want to paint a picture of the private pub/sub landscape. We describe the design space as a multi-dimensional hyper-space by describing its axes. This space is by no mean extensive and to further explore it is considered interesting future work. Let us do an example how we can use the design space to classify existing solutions: the system proposed by Shifka et al. [14] supports content-based pub/sub (section 2.5.1), type P messages (section 2.5.2), does not need key exchange (section 2.5.4) and ensures best privacy for publishers (section 2.5.7) and subscribers (section 2.5.8) in a way that their interests stay private.

2.5.1 Types of Pub/Sub System

One thing we have to do when defining a private pub/sub system is to state which subset of the three pub/sub types (topic-based, content-based, type-based) we support. If we use subscriptions as proposed in section 2.3 we support topic-based as well as content-based pub/sub. In our concrete solution in section 3 we focus on topic-based pub/sub.

2.5.2 Message Types

A further dimension of our design space is the type of messages that can be sent by using the system. We can think of the following message types:

- **Type M messages:** These are **m**ail-like messages, which consist of a sender, one or several specified receivers and a message body which has no specified structure.
- **Type P messages:** These are **p**ost messages, i. e. messages that are published by a publisher and whose receivers (namely the subscribers) are not known a-priori. Type P messages are structured, they consist of a number of attribute-value pairs.
- **Type S messages:** **S**emi-structed messages, e.g. XML messages. If one uses XML messages then he may use XQuery to process (and maybe even decrypt) them.

Of course a system can support several message types or introduce new messages. Moreover, type P messages can be built on top of Type M messages and vice-versa. For example, the cloud could take a post, find out to whom it has to be delivered and then create a new type M message for every subscriber. The body of the message would contain all the relevant attribute-value pairs. The opposite can be done by interpreting a type M message as a type P message with attributes sender and body. In order to send the messages to the desired receivers a publisher would have to create a special policy whose profile exactly matches the set of receivers.

2.5.3 Policy System

In section 2.4 we gave a definition for policies. There are existing policy systems that we may use to incorporate our own policies:

- **Ciphertext policies:** In CP systems every published message is encrypted and comes along with a policy that states who can read it. One application of ciphertext policies is CP-ABE, which stands for Ciphertext Attribute-Based Encryption and is shown in [2]. In CP-ABE an agent is modelled as a set of attributes that defines him, e.g. the SUPER-CARS company could be modelled as $\{$*SUPER-CARS, company*$\}$. A publisher who wants to make sure that only this company can read his message would hence encrypt the message with the policy $p =$ *SUPER-CARS* $\wedge$ *company*.

- **Key policies**: Key policies somehow go the opposite way than ciphertext policies: Every possible message receiver gets a key along with a policy that defines what a message should look like if he should be able to decrypt it. There are KP-ABE systems (as shown in [6]) where every message is annotated with a set of attributes. The key authority distributes keys which are annotated with policies on these attributes. For example, a SUPER-CARS car would send a message and annotate it with $\{$*confidential, for-companies-only*$\}$ and if we assume that the SUPER-CARS company has a key k for policy $p_1 =$ *confidential* $\wedge$ *for-companies*, it can read the message. On the other hand, an agent with policy $p_2 =$ *confidential* $\wedge$ *for-government* would not be able to read it.

2.5.4 Key Authority

If we use an encrypted system we need to define who creates and distributes keys. There are several possibilities for generating keys in pub/sub systems:

- **Central Authority:** There is a central authority that generates and distributes keys. The advantage of this approach is that we can share common keys (e.g. every attribute has one global key). The obvious drawback is that we must trust this authority and hence it cannot be part of the cloud, which we consider untrusted (compare with section 2.1).

- **Publisher:** Every publisher generates and distributes all keys necessary for decrypting messages that he sends. Like this, he has best control on his published data, which comes along with the drawback of some additional work that has to be done at the publisher's side.
- **No Keys:** There are encryption schemes that work without the need of key exchanges between publishers and subscribers. An example for this is commutative multiple encryption, which is shown in [14].

2.5.5 Certification Authority

It seems a reasonable privacy requirement that someone has to check whether a subscriber that claims to be an emergency service really is what he claims. Otherwise anyone can get information exclusively available to emergency services by simply claiming to be one, which we want to prevent. This is why there is a strong need for a certification authority. This authority can be part of the cloud or an external entity. However, as we do not trust the cloud it should be external. We could use a central certification authority (CA) or, as our system grows, add additional CAs and use hierarchical certification. Both approaches are well shown in [8].

2.5.6 Data Availability

As we have seen in section 2.1, data availability is a requirement for private pub/sub systems. As our system is encrypted, it makes sense that for every entry in the historical database part there is a non-empty set of agents who can (jointly) decrypt it. There are several options how we can do that. The most simple way is that for each entry there is exactly one agent that can read it, namely the author. But we can also think about other schemes where a subset of at least k agents can decrypt it or where there are certain agents that have the capacity of computing and decrypting aggregates (as sum or average) on certain entries without decrypting them individually.

2.5.7 Publisher Privacy

One of the requirements for a private pub/sub system is publisher privacy (compare 2.1). However, there are several variants for this. One option is just to enforce the publisher's policies (as stated in 2.4) without hiding them. In this case a subscriber would see on behalf of which publisher policy he received the message. There are applications where we want to prevent that. One example for this is PEAPOD [9].

2.5.8 Subscriber Privacy

Besides publisher privacy, our system could also give some privacy to the subscribers. The options for this are the following:

- **no subscriber privacy:** If you assume that a subscriber does not care about anybody knowing what he subscribed for, we do not need subscriber privacy.
- **hiding message reason:** Intruders (especially the cloud operators) can see from whom a subscriber got messages but they do not know on behalf of which subscription. Still, they know the content of each subscription.
- **hiding subscription content:** Intruders can see who has how many subscriptions but the subscription content is hidden. This option is also known as hidden-credentials and well shown in [9] and [14].
- **hiding subscriptions:** Intruders have no clue if there are subscriptions at all.

For the car-example we could assume that SUPER-CARS does not care about anybody knowing their subscriptions. However, there are more delicate use cases where subscriber privacy becomes an issue. Imagine for example that there is a library which uses a pub/sub system to inform clients about new books. It may be that certain clients do not want that employees of the library know in which books they are interested in.

2.5.9 Anonymity of sender and receiver

We want to ensure publisher privacy in a sense that we want that no one except the authorized receivers of a message can decrypt it. But we can also have a stronger requirement for senders and receivers, which we will call anonymity. Not only the content of the message is secret but also the fact that they communicate at all. This leads to the following options:

- Sender S and Receiver R are publicly known. Everybody is aware of their communication.
- Both, S and R stay secret. A possible intruder of the system can only see that there is communication but is not able to see who sends messages to whom.
- S is known while R stays secret.
- S is secret and R is known.

Most of these options can be implemented by using mixnets [3]. However, this comes at a significantly increased cost.

2.5.10 Adversary Model

When designing a private pub/sub system we can also think about different adversary models. In order to analyse the differences between the models, we give a generic model and the different options would then only be instances of

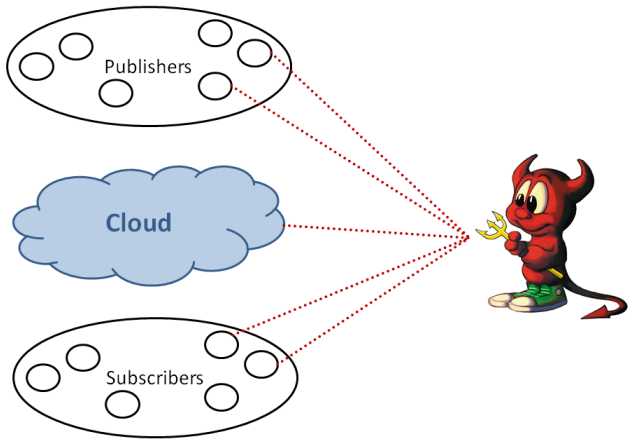


Figure 1: Adversary controls cloud and subsets of publishers & subscribers

this model.

What we expect from our system is that it is private with respect to the chosen adversary instance (for a notion of privacy refer to section 2.6). The generic model consists of an adversary who controls the cloud and any subset of publishers and subscribers (see figure 1). Every controlled entity can either be actively or passively corrupted. In standard security models passive corruption means that an entity tries to eavesdrop on channels and tries to decrypt messages it is not intended to. Actively-corrupted parties also try to send wrong messages, delete messages or re-send ancient messages again (which is called replay-attack). Examples for harmful adversary actions are given in table 1.

	passive	**active**
cloud	open whole database to adversary	fill tables with invalid data, ignore messages from publishers, send wrong post values to subscribers
publishers	open all subscription keys to adversary, open all posted (cleartext) values to adversary	send wrong postings (invalid values) to cloud, post all the time (denial of service attack)
subscribers	open all subscription keys to adversary, open all received postings (cleartext) to adversary	send subscriptions all the time (DoS attack)

Table 1: Examples for harmful adversary actions in private pub/sub systems

2.6 Notion of Privacy

After having seen a lot of dimensions of the design space and concluding with the possibilities for an adversary, we want to state precisely what achieving privacy in a pub/sub system means. Ideally (from a security point of view), a post directly goes from a publisher to a subscriber without the cloud being involved. In our concrete world this post is routed through the cloud. (compare figure 2).

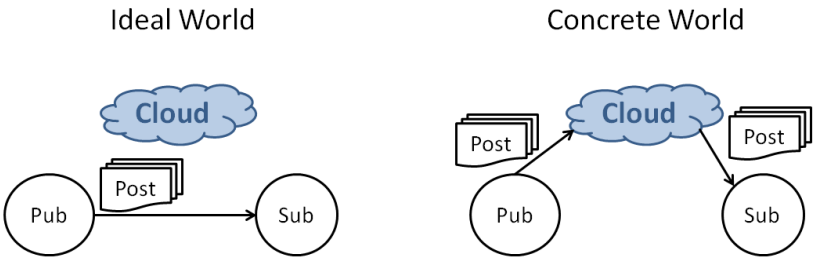


Figure 2: Ideal and Concrete World from a Security Perspective

We define a private pub/sub system to fulfil the privacy requirement if and only if an adversary that controls a certain set of parties does not get more information in the concrete world as he would get in the ideal world.

For example, assume that the adversary only controls subscriber S and publisher P posts a message m which is intended for S. The cloud will route m to S who will open it to the adversary. Although this is not very nice, it is not a privacy problem as the adversary had also gotten m in the ideal world where m would be transmitted directly from P to S. On the other hand, our system would not achieve privacy in the following scenario: We have again a message m going from P to S but the adversary only controls the cloud this time. If the adversary is able to decrypt m then there is something wrong with our system because he gets more information than in the ideal world (where he would not get m as it is sent directly).

3 Concrete Design

After having shown several different design options, we will now present one concrete design that we implemented and benchmarked. For one thing, we want to show that implementation of a private pub/sub system is possible and for another, we try to answer the question what privacy costs. Therefore we have built three variants of our system. First, we have a completely open system in which every agent can potentially read every message and agents can impersonate one another (*V1*). Second, we have a variant (*V2*) in which we add authentication, which means that messages are still open but an agent has to sign every message and the cloud verifies the signature prior to publishing it, thus making it impossible for agents to impersonate others. The third (and most interesting) variant (*V3*) of our system uses encryption in order to ensure privacy of the messages.

3.1 Assumptions and Limitations

In order to have a simple system, we make some basic assumptions which impose some limitations:

- **Subscriptions without condition clause:** We allow only subscriptions with a visibility and a profile clause, but without the possibility to constrain the post attributes of a subscriber. This restricts our system to support topic-based pub/sub only.
- **Dynamic attributes private:** We choose the form of publisher privacy where we make sure that only authorized subscribers can read the message. Still everybody can see which publishers are publishing at which time.
- **No central key authority:** Every publisher is his own key authority. We do not have to trust any central authority.
- **No certification authority:** Certification is a well-understood problem and orthogonal to the other problems we try to solve. This is why we just assume that attributes are certified and leave it open to plug-in any certification system later.
- **Passivly corrupted cloud:** We want to make our system private with respect to an adversary that controls the cloud passively. This means, the adversary can read all database tables in the cloud and see all communication going on between cloud and agents. Still the corruption is not active, which means the cloud executes the processes correctly. This is especially important to ensure that messages are routed and verified correctly.
- **Support for type M and type P messages:** We will allow both message types. Type P messages are used for the posts and type M messages for all other messages in the system (e.g. key request messages and post notifications).

- **Ciphertext policies:** We incorporate policies by using a ciphertext policy system. However, we do not use CP-ABE, but our own encryption technique based on AES [11] which is presented in section 3.9.
- **Data Availability:** As we want that all data ever published should stay in the cloud and that there has to be at least one party who can decrypt it, every publisher has a default policy. The key of this policy remains with the publisher and is not shared. Every post is done at least under the default policy. The only one able to decrypt messages encrypted under the default policy is the publisher himself.

3.2 Chosen Scenario: Electronic Cars

Instead of implementing an abstract private pub/sub system, we implemented the electronic cars scenario already mentioned in the previous sections. We used the static and dynamic attributes shown in table 2. We have chosen the dynamic attributes in a way that we cover both, integer and double data types.

(a) static attributes

name	**data type**
name	string
description	string
brand	string
model	string

(b) dynamic attributes

name	**data type**
x-position	integer
y-position	integer
x-acceleration	double
y-acceleration	double
battery-status	double

Table 2: static and dynamic attributes for agents

3.3 Data Model

We used the data model show in figure 3. We can see the SQL tables used in our cloud's database. Note that not all the tables are used for all variants of our system. In the following we give a short description what the tables are used for.

3.3.1 Agents

Let us have a closer look at the agents table. Beside the static attributes (*name, description, brand, model*) it also contains a unique id. Moreover there is host and port information to indicate where we can reach an agent while he is online (every time an agent logs in, he sends this information to the cloud). Every agent has a public key (for encryption) and a verification key (for verification of signatures). Besides the verification key we use a sequence number (which is automatically increased after each lookup) in order to ensure freshness of messages and prevent attackers from replaying messages. Of course, verification

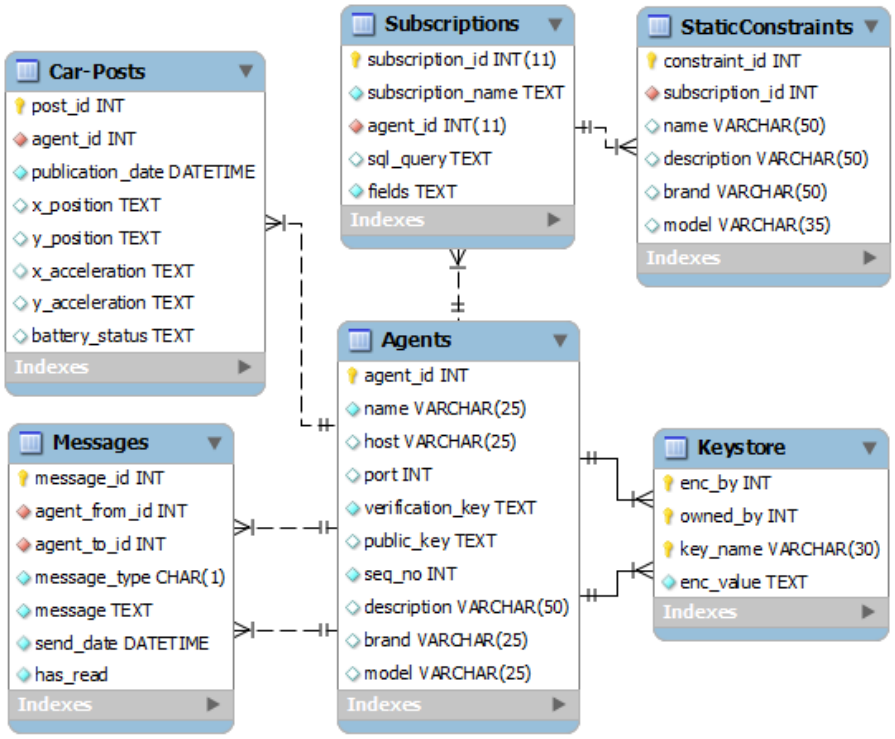


Figure 3: Data Model for electronic car Scenario

key and sequence number are not used when we use the unauthenticated version of our system (*V1*). Public key is only used in the encrypted systems (*V3*). For more information about the different versions of our system consult section 3.5.

3.3.2 Messages

The messages table contains the obvious fields to indicate the sender (*agent-from-id*), recipient (*agent-to-id*), *message* and *send-date*. Every message has a unique identifier (*message-id*) and a bit (*has-read*) that indicates whether the message has already been read by the agent. A message has a type (which is encoded as a character). It is either of type 'k' (key request), 'p' (post notification) or 'm' for a normal message (at any time, any user can send an email-like message to any other user).

3.3.3 Subscriptions

The subscriptions table contains all subscriptions of all agents. Every subscription has a unique identifier (*subscription-id*) and a name (which is chosen by the user in order to remember what he subscribed to). Of course there is a

(a) Subsriptions table

id	name	a_id	sql_query	fields
3	cars	7	(description = 'car' AND brand = 'S-C') OR (name LIKE '%car%')	x-acceleration, y-acceleration, battery-status
4	companies	2	(description = 'company') OR (name LIKE '%company%') OR (model = 'ISO')	x-position, y-position

(b) StaticConstraints table

id	sub_id	name	description	brand	model
56	3		= 'car'	= 'S-C'	
57	3	LIKE '%car%'			
58	4		= 'company'		
59	4	LIKE '%company%'			
60	4				'ISO'

Table 3: Examples for subscriptions and corresponding static constraints

reference to the owner of the subscription (*agent-id*) and a field that states the visibility of a subscription in the form *att1, att2, ...* A special field is *sql-query*. It contains a string which is the where clause of a SQL query that returns the set of publishers who match the profile of the subscription. This is useful at subscription time to find out which publishers match a subscription. Note that *sql-query* is always in disjunctive normal form (DNF) and contains boolean operators out of the set $\{=, <, \leq, \geq, >, LIKE\}$ where *LIKE* can only be used in string comparisons.

3.3.4 Static Constraints

The static constraints table stores the profile of subscriptions in another way. Every entry in the subscriptions table corresponds to one or several entries in the static constraints table. These entries represent predicate conjunctions and form the profile as a disjunction of these conjunctions. This way of storing profiles comes in handy when we have a new agent that registers to the system with a set of static attributes and we want to find out which subscribers are interested in him (or in other words: which subscriptions he matches). Besides the fields *name, description, brand* and *model* (which are used for the conjunctions), there are also the fields *constraint-id* (which is a unique identifier) and *subscription-id* (which serves as a reference to the subscriptions table). Examples for subscriptions and corresponding static constraints can be seen in table 3. Note that % is used as the wild-card symbol.

3.3.5 Posts

For every type of publisher we have a separate posts table with columns corresponding to the publisher type. As in our implementation all publishers are cars, we have only one posts table, namely the car-posts table whose columns correspond to the post attributes of cars (*x-position, y-position, x-acceleration, y-acceleration* and *battery-status*). What is more, we have a unique identifier for every post (*post-id*), a reference to the publisher (*agent-id*) and the point in time when the post was created (*publication-date*).

3.3.6 Key Store

We only need the keystore table when we use the encrypted system (V3). While a publisher uses only a small set of keys to encrypt (and can hence store them locally), a subscriber uses possibly many keys (one or several keys per publisher) to decrypt and therefore is given the possibility to store them encrypted in the cloud. He uses his (RSA) public key to encrypt and his private key to decrypt the publisher keys. An advantage of using asymmetric encryption is that a publisher who gets a key request from a subscriber can directly add the encrypted keys to the key store by using the subscriber's public key and thus without having to communicate to him. In order to make decryption fast, every subscriber has a key cache for heavily-used keys. An entry in the key store consists of the id of the subscriber for which the key is encrypted (*encrypted-by*), the id of the publisher who uses the key for encrypting his posts (*owned-by*), the name of the key (in our system this is the number of the policy for which the key is used) and the encrypted key itself (*enc-value*). This table does not need an additional identifier as the combination {*encrypted-by, owned-by, key-name*} is already unique.

3.4 Communication Model

As we implemented our system in Java, we can use Java RMI [12]. RMI (Remote Method Invocation) works on top of RPC [1] and lets a user make function calls on remote objects in the same way as for local objects. Therefore RMI calls also have a return value and are synchronous (the caller of an RMI function has to wait for the result of the call before he can proceed). In order to cope with that, RMI is inherently multi-threaded, which means that each call on a remote object is executed in a separate thread.

We use RMI for the communication between agents and the cloud. For one, agents can call the remote methods of the cloud interface (see 3.6) and on the other hand the cloud can also call the *notifyNewMsg(.)* method on the agent interface in order to notify him about new messages. There is one RMI connection per agent.

The cloud has several SQL connections to the mysql database (DB). We use three different connection types:

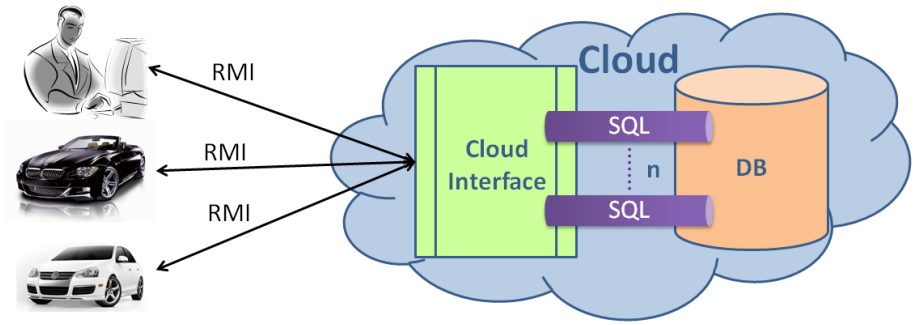


Figure 4: Communication Model with RMI connections and SQL bundles

- **batch connection**: does not commit after every update statement but spans a transaction with several updates. We use this connection for inserting post notifications into the messages table as they come always as batches (several subscribers have to be informed about a new post).
- **update connection**: is used for all other update operations.
- **read-only connection**: is used for read operations on the database. As no updates are done on this connection, we can let the Java connection interface take care of doing all these reads in parallel.

We use connection bundles which contain three connections, one of each type. The number of connection bundles is an input parameter of our system. The more bundles we use, the more can be done in parallel and the faster the system should get (at least up to a certain point). We will go into the performance details in section 4. A graphical representation of the used connections (RMI and SQL) is shown in figure 4.

3.5 Implementation of different variants

Our base system is the authenticated open system *V2* (fig. 5). We use a SHA1/DSA digital signature scheme [13] for every safety-critical call, i.e. in every call that should only be called by a certain agent. We concatenate the string representation of all arguments of the function call and add a fresh sequence number at the end. This string then serves as an input for the signing function, together with the secret key of the calling agent. The cloud (that has the verification key of each agent) verifies the signature before executing the call. In order to ensure integrity on the result returned by the cloud, we run RMI on SSL channels.

In order to get a non-authenticated system *V1* (fig. 6(a)), we can disable the authentication layer in the client application and turn off verification in the cloud. We do not require SSL channels any more. For getting the authenticated

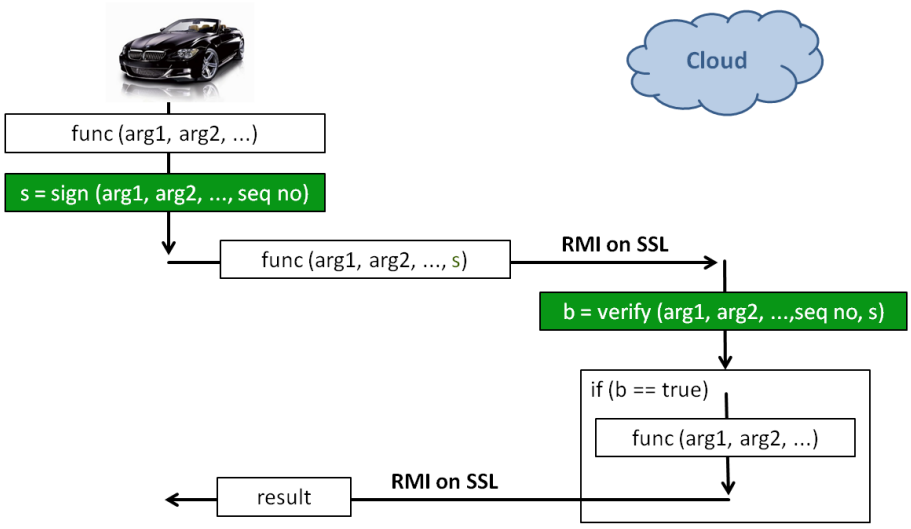
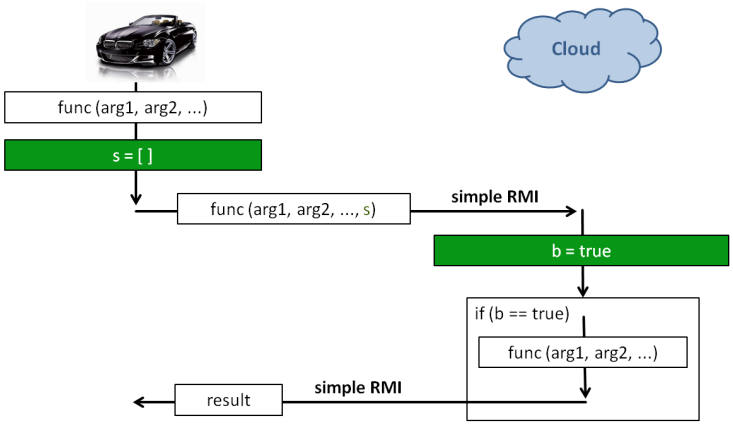


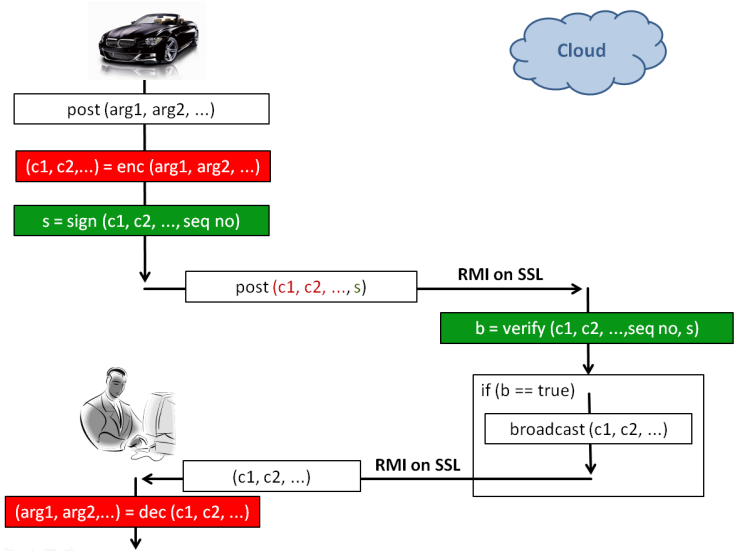
Figure 5: Function calls in the Open Authenticated System (V2)

private system *V3* (fig. 6(b)), we add an additional encryption layer at the agents' side. As the cloud only routes messages without paying attention to encryption, we do not have to change much there. The encryption scheme we use is explained in section 3.9.

Note that only *V3* is a private pub/sub system. *V1* and *V2* only serve as baselines in order to measure the cost of privacy for *V3*.



(a) Calls in the Unauthenticated System (V1)



(b) Posts in the Authenticated Encrypted System (V3)

Figure 6: Derived Systems V1 and V3

3.6 Cloud Interface

In the following, we try to give an overview on the most important functions of the cloud interface. We mention the function names, arguments and return values and give a short textual description of what the functions are supposed to do (see table 4). Note that all safety-critical function calls come along with a signature which signs all other function arguments as well as the newest sequence number of the agent involved. This signature is verified by the cloud using the

verification key of the involved agent. If we use the non-authenticated version of the cloud (*V1*) the signature is ignored. The two core processes (subscription and publication process) are explained in more detail in section 3.8.

add_key_to_store(encrypted_key, owner_id, holder_id, key_name)*   Inserts an encrypted key into the key store. The key has a name (to state what it is used for) and is encrypted with the holder's public key. The key is used to decrypt messages sent by the agent denoted by owner_id. Returns true if operation was successful.
create_agent(name, signing_key, signature)   Creates a new agent defined by his name and signing key. If the non-authenticated version is used, signing key is ignored. The function returns the agent_id if successful and -1 otherwise.
delete_agent(agent_id, signature)   Deletes the agent denoted by agent_id. Of course only the agent himself can call this function. The function returns true if deletion was successful.
delete_message(agent_id, message_id, signature)   Deletes the message denoted by message_id from the system but only if the agent denoted by agent_id is the recipient of the message. The function returns true if deletion was successful.
delete_subscription(subscription_id)   Deletes the subscription identified by subscription_id if called by the owner of the subscription. As soon as this is done, an agent doesn't receive post notifications for this subscription any more. Returns true if deletion was successful.
get_all_dynamic_attributes()   Returns a list of all dynamic attributes for agents. This is used to formulate subscriptions.
get_all_messages(agent_id, signature)   Returns a list of all messages for the agent denoted by agent_id. This allows an agent to display his "mailbox".
get_all_static_attributes()   Returns a list of all static attributes for agents. This is used to formulate subscriptions.
get_certified_agent(agent_id)*   Returns an agent object with the certified static attribute values of the agent denoted by agent_id. This is used by publishers' client application in order to decide which policy keys have to be given to a certain subscriber. Note that in our simple system we do not use any certification mechanism but just assume that the attributes are indeed certified.
get_keys_encrypted_by(agent_id, owner_id)*   Returns a map of key names to encrypted key values. Is used by the subscriber denoted by agent_id to get the necessary keys of the publisher denoted by owner_id in order to decrypt his post message. In order to prevent extensive function calls, every subscriber has a key cache.

get_public_key_of(agent_id)*   Returns the public key of the agent denoted by agent_id. Is used by publishers to encrypt keys intended for a certain subscriber.
get_next_sequence_number(agent_id)   Returns the next sequence number of the agent denoted by agent_id. This is used for signing function calls.
get_all_unread_messages(agent_id, signature)   Works the same way as get_all_messages(.) but returns only the messages not marked as read yet. Is normally called when an agent logs in.
login_agent(agent_id, host, port, signature)   Is used to log-in the agent denoted by agent_id and tell the cloud under which host and port the agent can be reached. Returns true if login was successful.
publish_post(agent_id, attributes, signature)   Posts the new dynamic state of the agent denoted by agent_id. attributes is a map from attribute names to values. The values can either be cleartext or encrypted, depending on the variant we use. Returns true if operation was successful.
send_message(from_agent_id, to_agent_id, message, signature)   Is used by the agent denoted by from_agent_id to send a normal (type M) message to a recipient denoted by to_agent_id. Returns the message_id of the new message if the operation was successful and -1 otherwise.
set_attributes(agent_id, attributes, signature)   Is used by an agent denoted by agent_id to set some of his static attributes. attributes is a map from attribute names to values. In a more elaborate system, this function would include certification mechanisms in order to ensure that the claimed attribute values are correct.
set_public_key(agent_id, key, signature)*   Is used by the agent denoted by agent_id to set his public key. Returns true if operation was successful.
subscribe(agent_id, static_constraints,   **fields[ ], subscription_name, signature)**   Subscribes an agent for certain publishers by giving constraints on their static attributes. static_constraints is a boolean formula (in DNF). fields[ ] is an array used to specify which post attributes should be delivered to the subscriber. The id of the publisher is always delivered by default. The function returns a subscription id, which can later be used to delete the subscription. Giving the subscription a name helps to remember what it does. In a more elaborate system a subscriber could also formulate constraints on dynamic attributes.

Table 4: Cloud Interface *functions only used in *V3*

3.7 Cloud Index

As we want to make the routing of messages fast, we can build an index inside the cloud. This index consists of an array of agent-ids. Each entry links to a list of subscriptions whose profile is matched by the corresponding agent. Like this, every time agent X makes a new post, we can lookup X in the index and find all subscriptions and along with them all subscribers that have to be notified about the post. An example of this index can be found in figure 7. In our implementation the index resides in memory. If we would like to deal with a really large number of agents, the index may not fit into cache and hence we would have to move it to the database. The index is easy to maintain. It has to be adjusted only at subscription time and when a subscription is deleted. At subscription time we make one call to the database in order to get the agent-ids of the publishers that match the profile of the encryption and then add the subscription into the index once for every publisher. If a subscription is deleted, we just mark the subscription object as deleted and then let a separate thread go over the index and clear every reference to it.

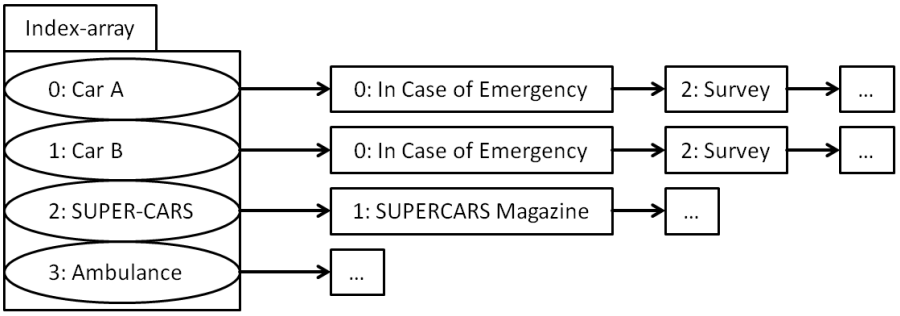


Figure 7: Cloud Index example

3.8 Processes

In the following we give a step by step description of what happens if an agent does a subscription and how the publication of a post works. These are the two core processes of our system. We do this by using concrete examples in the authenticated encrypted system (*V3*). Steps marked with * would be left out in *V1*, steps marked with ** would be left out in *V1* and *V2*.

3.8.1 Subscription Process

Assume that a subscriber (agent 2) wants to subscribe to all agents which match the static constraint "description = 'car' AND brand = 'SUPER-CARS'". Moreover, he states that he is interested in x-acceleration, y-acceleration and battery status.

1. Agent 2 prepares arguments:

```
static_constraints = "description = 'car'
                      AND brand = 'SUPER-CARS'"
fields = [x-acceleration, y-acceleration, battery-status]
sig = []
```

2. Agent 2 gets the next sequence number from the cloud:*

```
seq_no = cloud.get_next_sequence_number(2)
```

3. Agent 2 prepares signature:*

```
sig = sign(static_constraints + string(fields)
    + seq_no, signing_key_2)
```

4. Agent 2 subscribes at the cloud:

```
cloud.subscribe(2, static_constrains, fields,
    'test subscription', sig)
```

5. Cloud checks signature and proceeds only if it is correct:*

```
verify(sig, verification_key_2)
```

6. Cloud determines set of matching publishers and adds the subscription into the index once for every publisher.

7. Cloud sends a key request to every matching publisher:**

```
message = "ask key:" + string(fields)
for all publishers p in matching_publishers {
  send_message(2, p.agent_id, message)
}
```

   Note that the cloud can send messages without providing a signature.

When a publisher gets a key request (which can be at a later point in time when he logs-in), he asks the cloud for the certified attributes of the subscriber, checks them with respect to his policies and adds the necessary keys to the key store.

3.8.2 Publication Process

Assume that a publisher (agent 5) wants to publish the values (x-position = 87, y-position = 12, x-acceleration = 2.37, y-acceleration = 1.52, battery-status = 0.1). Further assume that agent 5 has the following policies:

1. **SUPER-CARS Survey**:
   - **Profile:** description = 'company' AND brand = 'SUPER-CARS'
   - **Condition:** battery-status ≥ 0.1
   - **Visibility:** x-acceleration, y-acceleration, battery-status
2. **Police Surveillance**:
   - **Profile:** description = 'service' AND name = 'police'
   - **Condition:** x-acceleration < 1.5 AND y-acceleration < 1.5
   - **Visibility:** x-acceleration, y-acceleration
3. **In Case of Emergency**:
   - **Profile:** description = 'car' AND name LIKE ' %emergency%'
   - **Condition:** battery-status ≤ 0.1
   - **Visibility:** x-position, y-position, battery-status

The following steps are executed:

1. Agent 5 prepares attribute map for posting:

```
attributes = new Map()
attributes[x-position] = 87
attributes[y-position] = 12
attributes[x-acceleration] = 2.37
attributes[y-acceleration] = 1.52
attributes[battery-status] = 0.1
```

2. Agent 5 determines the set of matching policies (policies whose condition match the post values). In this case this is Policy 1 and 3.**
3. Agent 5 replaces the values in the attributes map by their encrypted versions by encrypting them with the matching policies (for details on this refer to section 3.9):**

```
attributes[x-position] = enc(attributes[x-position], {3})
attributes[y-position] = enc(attributes[y-position], {3})
attributes[x-acceleration] =
   enc(attributes[x-acceleration], {1})
attributes[y-acceleration] =
   enc(attributes[y-acceleration], {1})
attributes[battery-status] =
   enc(attributes[battery-status], {1,3})
```

4. Agent 5 prepares signature as shown for the subscription process.*
5. Agent 5 calls cloud:

```
cloud.publish_post(5, attributes, sig)
```

6. Cloud verifies signature as shown for the subscription process and only proceeds if the signature is correct.*
7. Cloud inserts the post into the post table.
8. Cloud notifies all interested subscribers by using the index:

```
for all s in interested_subscribers: {
  generate interesting_attributes which contains only
       attribute values s is interested in
  insert a post notification for s into the Messages table:
       "s posted " + string(interesting_attributes)
  if s is on-line, notify s about the new post
}
```

3.9 Encryption and Key Management

For *V3* we use a simple encryption based on AES [11]. Prior to the encryption, a publisher has to compute the set of matching policies. This is the set of policies whose condition is satisfied by the values the subscriber wants to post. Every publisher has one (symmetric) AES key per policy (this key is generated by himself whenever he creates a new policy) and shares this policy key with every subscriber S who meets the following requirements:

- S matches the profile of the policy.
- S has issued a key request for a subscription *sub*.
- There is at least one attribute that appears in the visibility of the policy and as well in the visibility of subscription *sub*.

If we assume a publisher P has policies $p_1, p_2, ..., p_n$ with corresponding policy keys $k_1, k_2, ..., k_n$ and a matching policy set M, the encryption of an attribute A works as follows:

1. for every policy $p_n \in M$: if $A \in$ visibility(p_n) then $A_n =$ AES (A, k_n)
2. the encryption of A is the string concatenation of policy numbers and encrypted attribute values: enc$(A, M) = n_1 : A_{n_1} \# n_2 : A_{n_2} \# ...$

Note that by AES (X, k) we mean the AES encryption of string X with key k.

Let us illustrate this by a small example. Assume that we have the same policies as in the example in section 3.8.2 and want to post the same values. As we have seen, the set of active policies is $M = \{1, 3\}$. Let us see the encryption of the attribute *battery-status*:

- *battery-status* (BS) appears in the visibility of both policies (1 and 3) and hence has to be encrypted for both. Additionally we always encrypt for the default policy (which is policy 0).
- BS_0 = AES ("0.1", k_0)
- BS_1 = AES ("0.1", k_1)
- BS_3 = AES ("0.1", k_3)
- the final encryption is:
  enc (BS, {1, 3}) = "0:" $||BS_0||$ "1:" $||BS_1||$ "3:" $||BS_3$
  where || means string concatenation.

The decryption of such a string is straight forward: a subscriber scans the string and decrypts with the first known policy key. If none of the used policy keys is known to him, he is not able to decrypt.

3.9.1 Our Encryption vs CP-ABE Encryption

The encryption scheme that we use is straight-forward, easy to implement (as there exist libraries for doing AES encryption in Java) and works fast. Still it comes with some significant drawbacks: for one, every time that a subscriber issues a new subscription, new key requests are made and publishers have to send keys to the key store. This is not so much of a problem as the subscriber is guaranteed to get the necessary key from a publisher before having to decrypt one of his new posts. Either the publisher is on-line at subscription time, this means he is notified about the key request and immediately adds the necessary policy keys to the key store (encrypted with the subscriber's public key). Or, if a publisher is off-line he cannot post anything at all. Once he logs in, the first thing he will do is to ask for new messages and to process them. One of these new messages will be the key request of the subscriber.

A more serious drawback is the fact that every time a publisher creates a new policy, he has to distribute the new policy key to all interested subscribers who match the profile. Luckily, we have the index in the cloud, which could help to find these subscribers. Still, for every subscriber we have to query for the certified attributes in order to decide whether to give him the key or not. Another solution (and this is how it is done in our system) is that a publisher does nothing at all after the creation of a new policy. Subscribers get all messages they are interested in, but may not be able to decrypt them. From time to time, they re-send key requests for all subscriptions for which they get messages they cannot read entirely. Unfortunately, this solution is not really satisfactory neither.

This is why it would be a nice idea to bring into the picture CP-ABE Encryption [2]. In this encryption scheme, a publisher would create key shares for

all possible attributes and attribute values that he needs for describing policies. Once he has done this, he does not have to create any additional keys any more. When a subscriber sends him a key request, he answers this by adding certain key shares to the key store but then he can forget again about the subscriber, which fits nicely into the pub/sub paradigm of publishers and subscribers not knowing about each other. He can create new policies as he wishes, without the need of re-sending keys to anyone as there are no additional key shares. A comparison between our system and CP-ABE is given in table 5.

Of course, also CP-ABE has its drawbacks. For one thing, we need a key share for every possible attribute value ("brand = 'SUPER-CARS'" and "brand = 'LEMONS'" would be considered as two different "attributes" in a CP-ABE system), which is very limiting as it is hard to enumerate all possible values of brand or name that we would like to use for our policies in the future. For another, this encryption takes much more time and the messages get very long (as you can see in table 5). Nevertheless, CP-ABE has high potential and this is why we consider future exploration into that direction as highly interesting.

	Message Complexity	**Number of Keys**
Our Encryption	grows linearly with respect to the number of policies	grows linearly with respect to the number of policies
CP-ABE	grows linearly with respect to policy complexity (number of predicates in a policy) and therefore potentially exponentially with respect to the number of attribute values	grows linearly with respect to number of attribute values

Table 5: Comparing Message Complexity and Number of Keys

3.9.2 Attack on our Encryption

There is one attack on our encryption where a publisher may get information that he is not allowed to read: assume that we have policy 1 with visibility {x-position} and policy 2 with visibility {x-position, y-position}. Further assume that a publisher P wants to publish the values {x-position=1, y-position=1} with policy 1 and 2. Assuming that AES("1", k_1) = "Ua76*" and AES("1", k_2) = "&2Tq!" the encryption would be:

```
x-position = 1:Ua76*2:&2Tq!
y-position = 2:&2Tq!
```

Imagine that subscriber S matches only policy 1, but not policy 2. With policy key k_1 he can decrypt the x-position and gets "1". Therefore he knows that

"&2Tq!" corresponds to AES("1", k_2). As he reads the same value again for the y-position, he can conclude that y-position must be "1" as well. This violates privacy as P stated by his policies that S is authorized to read x-position, but not y-position. Referring to our formal notion of privacy (from section 2.6), we can also argue that in the ideal world, P would directly send the post [x-position = 1] to S without using the cloud and hence S would not get the value of the y-position. However, in the concrete world P sends an encrypted post to the cloud which forwards it to S who can read as well the y-position as shown before. This means that S gets more information in the concrete world than he would get in the ideal world and therefore privacy is not achieved.

Luckily we can prevent this attack by creating one more AES key per attribute. Before encrypting with the policy key, each attribute is encrypted with its attribute key. Like this, the encryption of "1" does not look the same for x-position and y-position and in the above example we would not get twice the same encrypted value. The cost of this more secure encryption scheme is that the number of keys grows linearly with the sum of number of attributes and number of policies. Another method to prevent the above attack would be to add a specified number of random bits to the attribute value before encrypting it, thus making sure that the encryption of the same value does look different every time.

3.10 Extension Points

In the following we try to give an overview of possible features that our system does not have, but which could be explored and may even be added in the future:

- **Distribution:** Our system consists of a cloud that runs on a single machine. This is because the thesis does not focus on the implementation of a highly-scalable distributed system but on general concepts. Nevertheless, it is an obvious contradiction to what *cloud* actually means. Therefore the first step in future work would be to make the cloud distributed over several machines. The distribution scheme is straight forward: as we can see in figure 3, agent-id is a (foreign) key in all the tables, which means we can horizontally partition the agents table and then use derived horizontal partitioning for all the other tables. Even the cloud index uses agent-id as a key and can therefore be partitioned as well. Besides partitioning we could also use replication in order improve availability of the system.

- **Certification:** One important step towards the development of a commercial product would be to plug in a certification mechanism into our system. There is a lot of existing services that we could use. Moreover, many existing cloud solutions (as e.g. Windows Azure [10]) offer certification services as part of their cloud infrastructure.

- **Deletion of policies:** Our system does allow deletion of policies. Still

the question is what the semantics of a policy deletion are. If by deletion we mean that from now on a publisher does not publish with this policy any more, we can implement this easily (and this is how we have done it). But it could also be that we want to revoke the permission to read certain data from a specific subscriber. This is hard as once we have given a key to a subscriber, we cannot just revoke it from him. Another problem is that if a subscriber changes his static attributes, he may not match certain policies any more but he still has the corresponding keys and there is no way to revoke them. One way to address both problems is that publishers use keys which are only valid for a certain amount of time (let's say for one month) and that the certification of static attributes of subscribers is also valid for just a period of time (let's say a year). What is more, a publisher only gives policy keys to a subscriber who can prove that his static attributes hold for at least the same time as the key. The overhead of generating new keys from time to time seems reasonably small.

- **Dynamic constraints in subscriptions**: there is a reason why we did not allow constraints on the publisher's dynamic attributes for subscriptions: they are hard to enforce when we are dealing with encrypted data. Of course, there exists a naive approach where we just add a filtering step at the subscriber's side where all post notifications of posts that do not match the dynamic constraints are filtered out. But the more elegant way to do this is that the cloud does not at all send notifications to subscribers that are not interested. Moreover, this looks like a typical routing task and therefore should be done in the cloud. The question to address is how can the cloud decide to which subscribers to send the data while being unable to read it? One possible solution is to use some sort of partially order-preserving encryption (POP) scheme [15]. However, it is not obvious how we can integrate such a scheme.

- **Anonymous Posting:** In our data model the field *agent_id* of the posts tables is always in cleartext. Of course we could be interested in adding sender anonymity to our system. We would address this by using some sort of mix-net [3] or any other anonymous communication model. Also receiver anonymity would be interesting to explore.

- **Statistics:** What we mean by this is that a dedicated agent can be given the capacity to compute statistics on the historical database. This includes simple aggregate functions (as sum or average), but also any other function that uses addition and multiplication. One way to approach this is to use homomorphic encryption [5]. The big challenge is that there exist homomorphic encryption schemes for addition and others for multiplication, but it seems very hard to find a scheme that supports both.

4 Performance Experiments and Results

4.1 Our benchmark

As there is no standard benchmark for testing pub/sub systems, we designed our own benchmarking system. It consists of a data generator (similar to dbgen for the TCP-H benchmark [16]) and an experiment runner (who takes the data generated, lets the system run and logs certain things). Our data generator takes as input a whole bunch of parameters:

- **number of publishers and number of subscribers**: in our benchmark every agent is either a publisher or a subscriber, not both
- **parameters for subscriptions**: number of subscriptions per subscriber and number of disjunctions in the profile clause
- **parameters for policies**: number of policies per publisher, number of disjunctions in the profile clause, number of disjunctions in the condition clause and number of matching policies per post

Every subscriber has the same subscriptions and the subscription profiles partition the publishers into equally-sized disjoint groups. Every publisher has the same policies and the policy profiles partition the subscribers into equally-sized disjoint groups. The posts are designed in such a way, that always the same number of policies is matching and that every policy is matching with the same frequency. All this imposes of course some restrictions on the parameters, e.g. that they have to be multiples of others. Note that our benchmark is dense, which means that every post is eventually sent to every subscriber. Moreover, the visibility clauses of subscriptions and policies contain the entire set of post attributes. For more details, please refer to the documentation of the data generator.

The experiment runner takes the files generated by the data generator, asks the operator some additional questions (which system to test, with how many database bundles and how many minutes to run the experiment) and executes the following steps:

1. **Setup Phase:** The databased is initialized and the cloud process is started. The agent binaries and configuration files are copied to different directories, one directory per agent. Every agent is launched in a separate process, the experiment runner communicates to it by RMI. During the creation procedure, the agents register themselves at the cloud and do the log-in.
2. **Set policies:*** The publishers are told to state their policies. This step does not involve any cloud interaction as the policies are stored only locally at the publishers' side.

3. **Set subscriptions:** The subscribers are told to make their subscriptions. During this step key requests are sent and policy keys are immediately added to the key store (as everybody is on-line).

4. **Posting:** The publishers start posting, which means they have a list of postings which they repeat again and again. This step is where we actually measure performance. For every posting we log at which point in time it entered the cloud and how long it took to be processed. The processing includes the insertion into the posts table as well as the notification of all subscribers about the new post. If we test *V3* we measure encryption time at every publisher's side and decryption time at every subscriber's side.

5. **Shutdown:** All agents are shut down, then the cloud is shut down. The logs are collected from all sites and copied into a log directory.

6. **Analysis:** The logs are processed and the following numbers are produced:

   - throughput, average service time and standard deviation of service time for every minute of the experiment
   - overall average throughput with standard deviation
   - overall service time with standard deviation
   - optional additional numbers (as for e.g. average encryption and decryption time)

* in step 2 nothing happens if we test *V1* or *V2*.

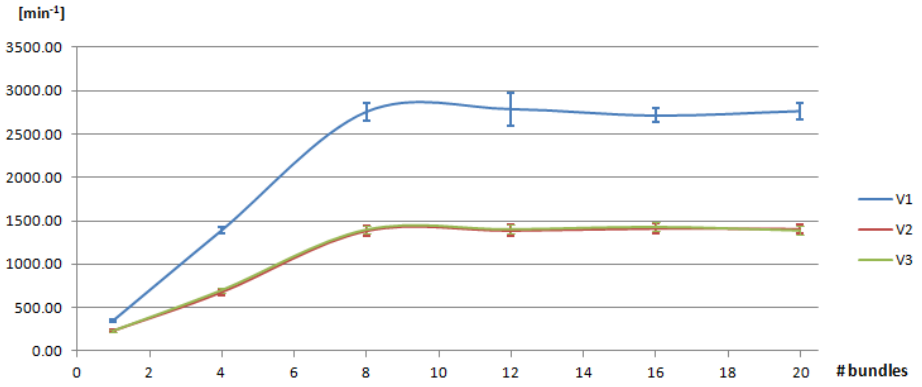
4.2 Experimental Setup

All the experiments were conducted on a single machine, namely a linux machine with Ubuntu 11.10 (Oneiric Ocelot), 4 Intel Core i5 CPUs with a clock speed of 2.67 GHz, 8 MB L2-cache and 8 GB memory.

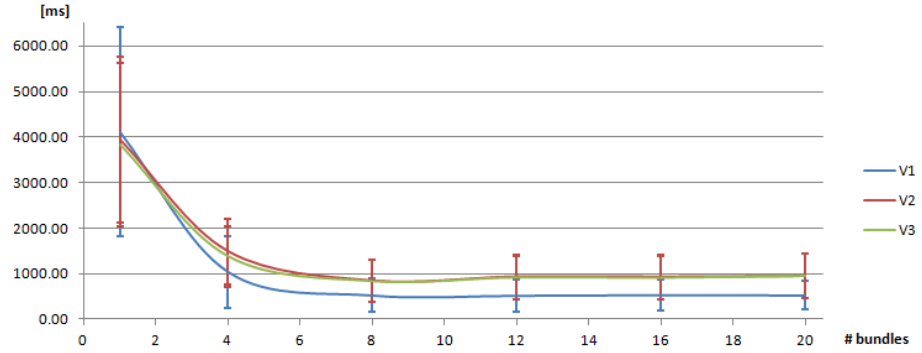
Our setup consists of 24 publishers and 6 subscribers, each with three different subscriptions (which partition the 24 publishers into 3 disjoint groups of 8 agents). For the testing of V3, we added three policies per publisher that were designed in such a way that for each post two of them are matching and that the 3 policies partition the 6 subscribers into disjoint groups with 2 agents each. We tested *V1, V2* and *V3* for 1,4,8,12,16 and 20 bundles. Every experiment was conducted twice and lasted 20 minutes. Note that the key cache size was chosen in such a way that all keys fit into cache.

4.3 Results

The detailed results can be found in section 7.1. In the following we try to summarize them and give answers to these questions: what is the optimal number



(a) Average Throughput



(b) Average Service Time

Figure 8: Cloud performance for different systems

of database bundles? What is the cost of privacy in the system? And does our system scale well?

4.3.1 Optimal Number of Database bundles

We can see that adding more bundles leads to an increased throughput and decreased service time in the first place. However, it seems that after a certain time they stay at a more or less stable level. For *V1* the optimal number of bundles seems to be around 12 as there we have the highest throughput and also the lowest service time. *V2* and *V3* behave very similar: we have the shortest service time for 8 bundles and highest throughput for 16 bundles. Depending on what we want to achieve with our system, we will chose some number in between. Probably, when we have to deal with many users, we would give throughput a higher priority as long as service time stays at a reasonable level.

4.3.2 Cost of Privacy

As we can see in figure 8(a), the average throughput in the cloud is very similar for *V2* and *V3*. Taking also into account the standard deviation, we can say that they are identical. However, this is not very surprising. Knowing that the cloud only routes posts without looking at the content, it makes sense that there are no additional costs there. What makes a huge difference is to use the un-authenticated system *V1*. In that system we do not need SSL channels and can leave out verification, which makes it very fast.

At the agents' side, encryption of messages takes between less than 1 and 20 milliseconds, but in average it is 2.5 milliseconds. The decryption depends on whether the necessary keys are cached. If they are, decryption takes about 1 millisecond, but if they are not cached it varies from about 80 milliseconds to 3.5 seconds (with an average of 1.25 seconds). The reason why this varies so much is that we have to wait until a database bundle gets idle, which again depends on how many other agents are trying to use it at the same time. Obviously, this number is hard to predict. All in all, we can say that if we assume that decryption keys are cached, the *average cost of privacy* is:

average cost of encryption + *average cost of decryption*
= 2.5 ms + 1 ms = 3.5 ms

If we compare these additional 3.5 milliseconds per post to the average service time at the cloud for the best case (829 ms with 8 bundles for *V3*) this is about 0.42% and seems to be an overhead which makes it worth to make the system private.

By comparing throughput of the three systems we can see that adding authentication (which corresponds to adding SSL channels) slows the system down by about 50%, which is a high price. However, this price has to be paid as an un-authenticated system where each agent can take the role of another is completely insecure and useless.

4.3.3 Scalability

As we have shown in section 3.10, our cloud can be distributed very well in a sense that the database tables can be partitioned well and also the index (which is the main data structure of the cloud) can be partitioned. As no other dynamic data structures have to be maintained in the cloud, we have a strong believe that our system scales well if we distribute it over several machines and if we assume that the number of machines grows linearly with the number of agents in the system.

5 Conclusion and Future Work

In this thesis we tried to give a definition of private pub/sub systems and have drawn a picture of their design space, showing different options for supported pub/sub systems, message types, policy system, key authority, publisher and sender privacy, publisher and sender anonymity and adversary model. This design space is by no way complete and any contributions to enrich it are welcome. Moreover, we implemented a simple system and benchmarked it, from which we could conclude that the cost of privacy in our pub/sub system is considerably low and that our system would also scale well. What we would like to do in the future is to make the system distributed and conduct a number of additional experiments on it. To explore other design points, e.g by. adding dynamic constraints to the subscriptions or coming up with new encryption schemes that allow certain agents to compute aggregates on certain values without decrypting them individually, is also considered interesting future work.

5.1 Acknowledgements

I would like to thank my supervisor Prof. Donald Kossmann from the ETH Systems Group who coached and motivated me and always found a meeting slot in his overloaded time schedules. After all, he encouraged me to come up with my own ideas and get a glimpse of what research is really meant to be. I also want to thank my tutor, Tahmineh Sanamrad, also from Systems Groups, for her great contributions in the meetings but also for her help concerning technical details of my implementation. A special thank goes to Venkatesan Ramarathnam from Microsoft Redmond who gave a lot of valuable input to the cryptographic part of the implementation.

6 References

[1] Bershad, B., Ching, D., Lazowska, E., Sanislo, J., and Schwartz, M. A remote procedure call facility for interconnecting heterogeneous computer systems. *Software Engineering, IEEE Transactions on SE-13*, 8 (aug. 1987), 880 – 894.

[2] Bethencourt, J., Sahai, A., and Waters, B. Ciphertext-policy attribute-based encryption. In *Security and Privacy, 2007. SP '07. IEEE Symposium on* (may 2007), pp. 321 –334.

[3] Chaum, D. L. Untraceable electronic mail, return addresses, and digital pseudonyms. *Commun. ACM 24*, 2 (Feb. 1981), 84–90.

[4] Eugster, P. T., Felber, P. A., Guerraoui, R., and Kermarrec, A.-M. The many faces of publish/subscribe. *ACM Comput. Surv. 35*, 2 (June 2003), 114–131.

[5] Fontaine, C., and Galand, F. A survey of homomorphic encryption for nonspecialists. *EURASIP J. Inf. Secur. 2007* (Jan. 2007), 15:1–15:15.

[6] Goyal, V., Pandey, O., Sahai, A., and Waters, B. Attribute-based encryption for fine-grained access control of encrypted data. In *Proceedings of the 13th ACM conference on Computer and communications security* (New York, NY, USA, 2006), CCS '06, ACM, pp. 89–98.

[7] Hardy, M. Law of total variance. `http://en.wikipedia.org/wiki/Law_of_total_variance`.

[8] Hunt, R. Pki and digital certification infrastructure. In *Networks, 2001. Proceedings. Ninth IEEE International Conference on* (oct. 2001), pp. 234 – 239.

[9] Kapadia, A., Tsang, P. P., and Smith, S. W. Attribute-based publishing with hidden credentials and hidden policies. In *In The 14th Annual Network and Distributed System Security Symposium (NDSS 07) (To Appear* (2007), pp. 179–192.

[10] Microsoft. Windows azure. `http://www.windowsazure.com`.

[11] NISC. Announcing the advanced encryption standard (aes). `http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf`.

[12] Oracle. Remote method invocation. `http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136424.html`.

[13] Oracle, S. Java 2 platform - class signature. `http://docs.oracle.com/javase/1.4.2/docs/api/java/security/Signature.html`.

[14] Shikfa, A., nen, M., and Molva, R. Privacy-preserving content-based publish/subscribe networks. In *Emerging Challenges for Security, Privacy and Trust*, D. Gritzalis and J. Lopez, Eds., vol. 297 of *IFIP Advances in Information and Communication Technology*. Springer Boston, 2009, pp. 270–282. 10.1007/978-3-642-01244-0_24.

[15] Stefan Hildenbrand, Donald Kossmann, T. S. C. B. F. F., and Woehler, J. Partially order-preserving encryption. Tech. rep., ETH Zurich, 2011.

[16] (TPC), T. P. P. C. Tpc-h. `http://www.tpc.org/tpch/`.

[17] Yan, T. W., and Garcia-Molina, H. The sift information dissemination system. *ACM Trans. Database Syst. 24*, 4 (Dec. 1999), 529–565.

7 Appendix

7.1 Result Details

In section 4 we explained the benchmark we used as well as the experimental setup. This section presents the full details about average throughput and service times with their corresponding standard deviations as well as time for encryption and decryption. We conducted every experiment twice. For computing the total average for throughput (X) and service time (T) of two experiments, we just took the average of averages. For computing the total standard deviation we used the law of total variance [7] and derived the following formulas:

total std of throughput: $s_X = \sqrt{\frac{({s_{X1}}^2+(X1-X)^2)+({s_{X2}}^2+(X2-X)^2)}{2}}$
Note that the samples are equally-sized (n=20).

total std of service-time: $s_T = \sqrt{\frac{X1\cdot({s_{T1}}^2+(T1-T)^2)+X2\cdot({s_{T2}}^2+(T2-T)^2)}{X1+X2}}$
Note that we take average throughput as relative sample sizes.

The numbers for throughput and service time are shown in table 6. The time for encryption and decryption is shown in table 7.

Exp no	System	# bundl.	X avg	X +/-	T avg	T +/-
1a	V1	1	351.90	11.23	4089.90	2290.73
1b	V1	1	348.11	13.69	4133.82	2295.27
avg	**V1**	**1**	**350.01**	**12.66**	**4111.86**	**2293.09**
2a	V1	4	1382.46	46.07	1040.89	799.29
2b	V1	4	1397.16	34.41	1030.22	777.18
avg	**V1**	**4**	**1389.81**	**41.32**	**1035.56**	**788.27**
10a	V1	8	2802.45	79.96	513.25	362.02
10b	V1	8	2707.20	89.49	530.71	374.82
avg	**V1**	**8**	**2754.83**	**97.31**	**521.98**	**368.47**
13a	V1	12	2757.10	254.04	521.68	347.57
13b	V1	12	2818.70	76.72	510.40	336.12
avg	**V1**	**12**	**2787.90**	**190.16**	**516.04**	**341.88**
3a	V1	16	2724.00	90.32	528.84	325.60
3b	V1	16	2701.35	68.45	532.39	333.96
avg	**V1**	**16**	**2712.68**	**80.93**	**530.62**	**329.79**
16a	V1	20	2731.41	96.10	527.16	321.30
16b	V1	20	2797.45	84.71	514.46	305.02
avg	**V1**	**20**	**2764.43**	**96.41**	**520.81**	**313.23**
4a	V2	1	224.70	22.54	3910.89	1860.18
4b	V2	1	231.35	9.36	3977.68	1781.86
avg	**V2**	**1**	**228.03**	**17.58**	**3944.29**	**1821.18**
5a	V2	4	684.75	22.83	1461.95	707.56
5b	V2	4	665.61	40.10	1503.66	729.05

avg	**V2**	**4**	**675.18**	**34.00**	**1482.81**	**718.54**
11a	V2	8	1374.91	49.97	846.78	466.30
11b	V2	8	1392.85	51.75	833.86	456.41
avg	**V2**	**8**	**1383.88**	**51.65**	**840.32**	**461.39**
14a	V2	12	1391.21	65.40	926.25	499.16
14b	V2	12	1383.96	59.30	929.58	493.42
avg	**V2**	**12**	**1387.59**	**62.53**	**927.92**	**496.31**
6a	V2	16	1418.71	62.56	920.03	480.96
6b	V2	16	1405.80	56.15	929.00	494.69
avg	**V2**	**16**	**1412.26**	**59.79**	**924.52**	**487.86**
17a	V2	20	1414.30	49.47	937.26	483.35
17b	V2	20	1399.55	59.48	946.50	492.05
avg	**V2**	**20**	**1406.93**	**55.20**	**941.88**	**487.72**
7a	V3	1	230.95	8.30	3437.33	1820.74
7b	V3	1	229.56	8.73	4222.80	1683.28
avg	**V3**	**1**	**230.26**	**8.55**	**3830.07**	**1797.01**
8a	V3	4	701.90	27.99	1373.06	659.78
8b	V3	4	701.40	28.75	1375.45	678.56
avg	**V3**	**4**	**701.65**	**28.37**	**1374.26**	**669.23**
12a	V3	8	1394.10	40.72	833.23	470.32
12b	V3	8	1409.60	43.71	825.40	457.30
avg	**V3**	**8**	**1401.85**	**42.95**	**829.32**	**463.84**
15a	V3	12	1409.16	61.70	912.06	480.47
15b	V3	12	1403.25	52.40	915.02	483.59
avg	**V3**	**12**	**1406.21**	**57.32**	**913.54**	**482.03**
9a	V3	16	1446.00	41.98	897.65	462.63
9b	V3	16	1420.91	45.99	915.53	477.74
avg	**V3**	**16**	**1433.46**	**45.78**	**906.59**	**470.26**
18a	V3	20	1385.21	48.51	950.99	502.95
18b	V3	20	1400.55	51.20	939.59	479.81
avg	**V3**	**20**	**1392.88**	**50.46**	**945.29**	**491.49**

Table 6: Conducted experiments with experiment number, used system, used number of database bundles, average throughput and its standard deviation, average service time and its standard deviation

Table 7: Min, Max and Avg Time for Encryption and Decryption in ms

	Min	Max	**Avg**
Encryption	1	20	**2.5**
Decryption (cached keys)	< 1	15	**1**
Decryption (with key lookup)	80	3500	**1250**

7.2 List of Tables

7.3 List of Figures

* produced with Microsoft Powerpoint
** produced with Microsoft Excel
*** produced with MySQL Workbench
° pictures taken from:
`http://www.art-scene.org/wp-content/uploads/2008/06/low-car-insurance.jpg`
`http://media.treehugger.com/assets/images/2011/10/vw-jetta-tdi-2009.jpg`
`http://blogs.mcall.com/.a/6a00d8341c4fe353ef016301f1d2d4970d`