Working Paper Series
ISSN 1177-777X

Implementing an Event-driven Service-oriented Architecture in TIP

Michael Rinck & Annika Hinze

Working Paper: 02/2010
June 17, 2010

Department of Computer Science
The University of Waikato
Private Bag 3105
Hamilton, New Zealand

Implementing an Event-driven Service-oriented Architecture in TIP

Michael Rinck[1], Annika Hinze[2]
[1] Humboldt University Berlin, Germany,
rinck@informatik.hu-berlin.de
[2] University of Waikato, New Zealand,
hinze@cs.waikato.ac.nz

Abstract

Many mobile devices have a density of services, many of which are context or location-aware. To function, many of these services have to collaborate with other services, which may be located in many different places and networks. There is often more then on service suitable for the task at hand. To decide which service to use, quality of service measurements like the accuracy or reliability of a service need to be known. Users do not want third parties to have statistics on how and where they used services. Therefore the collaboration needs to be anonymous. This project implements a model of event-based context-aware service collaboration on a publish/subscribe basis. We compare different implementation designs, with focus on anonymity and quality of service of the services.

1 Introduction

New applications for mobile devices are released every day, with a growing need for communication between the services. Often services are able to share information, but only want to do so anonymously. Quality of service is for some services important, e.g., the reliability, distance or accuracy of a service. In addition, many services are only available locally and their quality of service values depend largely on the position and context of their user. So what is needed is a way for services to collaborate anonymously with constantly changing partners, without losing focus of quality of service issues. Hinze and Michel suggested an architecture meeting these requirements and implemented a first prototype [4]. Eschner created a formal model of an anonymous event-driven publish/subscribe middleware solution to this problem [3]. It proposes a system where the different services connect to a trusted broker, which will then handle their requests. Publishing services do deliver their data to this broker, which routes it to subscribers in a way where the subscribers cannot get any restricted information on the publisher of the data. The publishers also notify the broker about the general type of their data, as well as quality of service values of it, if available. Subscribers, on the other hand, only notify the broker of the type of event they want to subscribe. They also deliver some rules and conditions under which they are willing to accept events from the broker. The conditions define, under which constraints events will be accepted. The rules serve as a guideline on what events are send, in case there is more then one matching the type of the subscription. Brokers may also act as publishers if

connecting to other brokers, as well as subscribers. Requirements of our solution are (following Eschner [1]):

- anonymity of services (R1)
- suitability of architecture for mobile devices (R2)
- openness of architecture – services may be offered by anyone (R3)
- support of quality of service driven subscriptions (R4)
- services may be location bound (R5)
- peer to peer communication should be possible between brokers (R6)
- proof of the anonymity of services is possible (R7)

This project focusses on creating an architecture and software prototype that both implement the formal models and meet the seven requirements as suggested in [1]. We now briefly discuss some scenarios and some background for the project. In Section 2 we evaluate related work and select an architecture for implementation. Our implementation is documented in Section 3. The paper ends with a summary and a discussion of future work.

1.1 Scenarios

In all scenarios, a user's device will first start the local broker and observer, and then the services. It then allows the services to connect to the local brokers and then, if needed, to global brokers.

Scenario 1 In the first example, shown in Figure 1, the user uses a map service, which needs location events. While using it he moves into a museum, in which the GPS location service does not work very well. The museum offers an alternate location service, whose quality of service is growing the closer the user is to the museum. The user's map service is subscribed to the broker on location events. The broker receives location events from the GPS location service located on the user's device. As the user comes within the range of the museum, the local broker also receives location events offered by the location-restricted museum RFID location service. As long as the user does not enter the museum, these events are of inferior quality to the events from the GPS location service. On entering the museum, the GPS location services loses quality and the broker starts to deliver the location events from the RFID service instead, since those increased in quality.

Scenario 2 The second scenario, shown in Figure 2, has two users with mobile devices. User 1 uses a map service. User 2, on the other hand, has a peer to peer service that shares map data. User 1 starts his device, which has the GPS location service and the map service installed. The user 1 GPS service uses the local broker to advertise map events and the map service uses the local broker to subscribe to map events. The broker also receives map events from user 2, who is in the vicinity of user 1. On user 2's device the same procedure as on user 1's device is run, except that the local broker on user 2's device also starts a global broker to advertise the events produced by the p2p location service. The user 1 local broker subscribes to these events, but does not deliver them to the user 1 map service, because they are of lower quality. User 1's GPS service fails at some time (maybe because he entered a museum). The user 1 broker will now deliver the p2p map events received through the user 2 global broker. After a time user 2 shuts down his device, which leaves the local broker of user 1 without events to

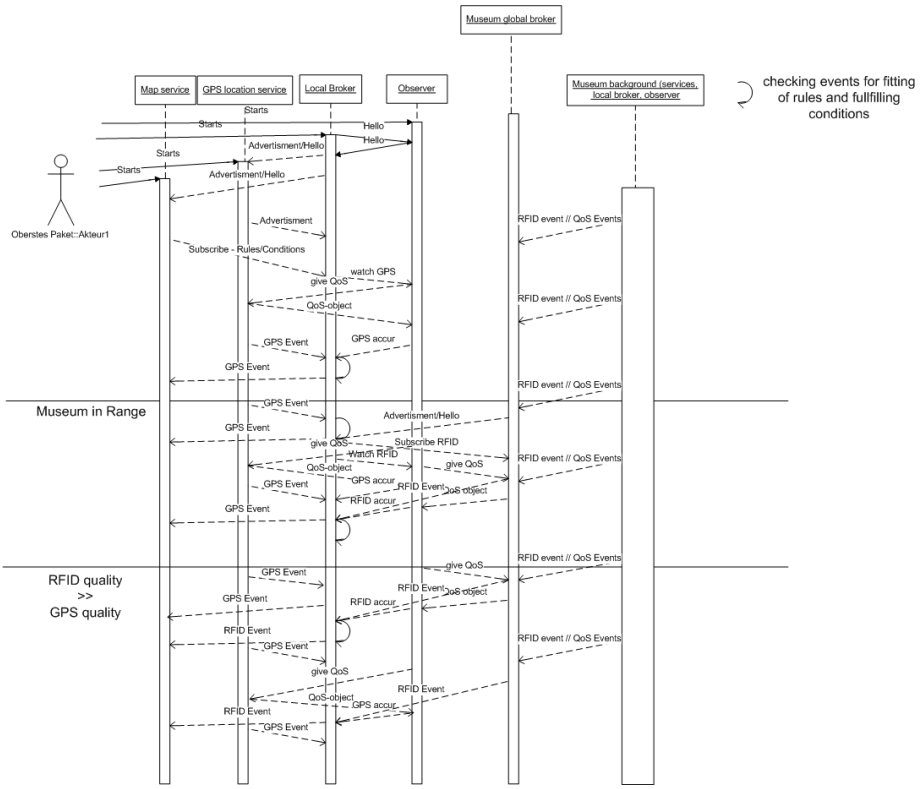

Figure 1: Example: GPS RFID Switch

deliver and the map service ceases to work. This example would need the architecture to be capable of p2p communication.

Scenario 3 This last example works similar to the second one, however, this time, the GPS location service fails and there is no alternative service from which to receive location events. The local broker is informed about the missing location data by the observer and notifies the subscribing map service, that its condition is broken. The example is shown in Figure 3

1.2 Background

Event-based systems often focus on quality of service or reliability issues. These systems focus less on the problem of an ever changing portfolio of known and accessible services, whose suitability depends on the changing context of the user. The following two approaches try to cope with changes on services and service properties in the runtime.

Hinze and Eschner [1] propose a middleware approach. In this approach, a broker handles all events which are transmitted from service to service. As a result, services do not connect directly. Instead they connect to the broker and subscribe to an event type, rather then a service. Services may also connect to a broker as publishers for an event type. Subscribing services may also define conditions under which the subscription

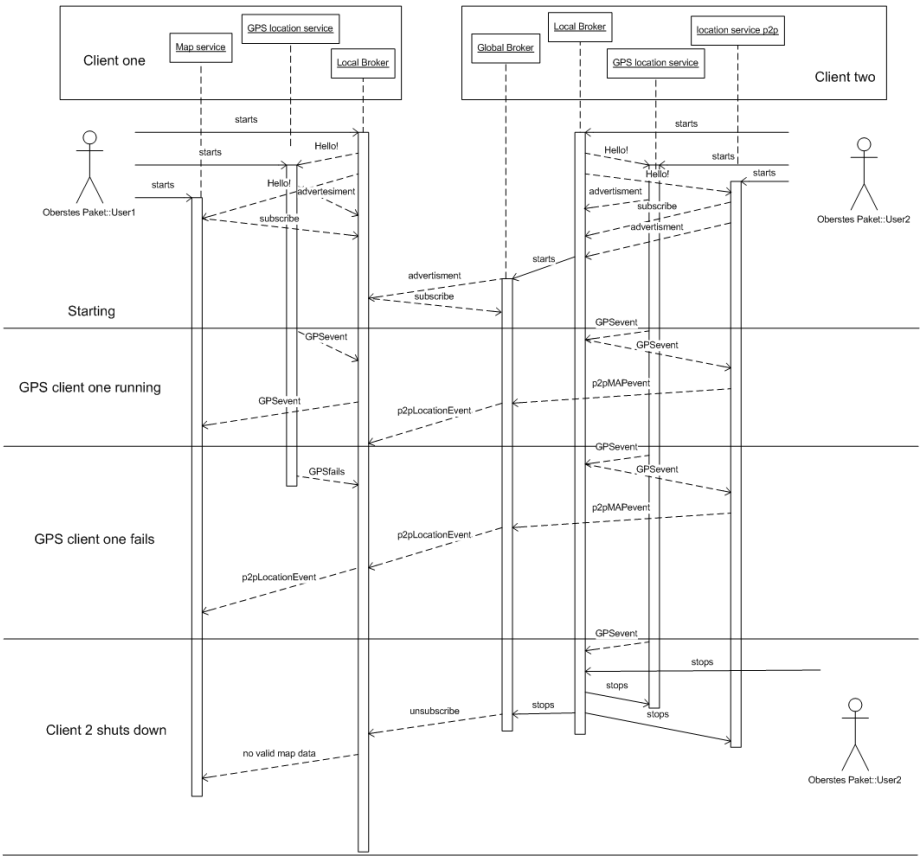

Figure 2: Example: Sequence of Two Users

is valid, like the existence of service type. They may also define rules, which clarify which service's events are to be delivered, in the case that there are two or more publishers of the same event. These rules mostly contain quality of service constrains. The broker as the trusted middle man secures privacy. Through the rules and conditions, events are filtered on context, for example location and quality. Hinze and Eschner also allow broker to broker communication, therefore allowing p2p communication. Services may be provided by everyone, as long as they follow the rule and condition standard. Proof for privacy and anonymity was not given.

Serugendo et al. [6] discuss a problem very similar the one addressed here. The paper focusses on quality of service issues for SOA. The authors want the single services to directly communicate with another, which makes anonymity impossible. The proposed concept of DRMs (Dynamic Resilience Mechanisms) and resilience policies, allows a *Metadata Registry* to maintain and evaluate service links. This is similar to our approach on rules and conditions. There is no language given for these mechanisms, and also no information on where the *Metadata Registry* would be located. This hampers the proposed usage on p2p approaches as well as in mobile environments.

Michlmayer et al. [5] propose an environment in which services communicate which a centralized engine through the EQL (esper query language), which is similar to SQL. They also assume that the services know which services they may expect,

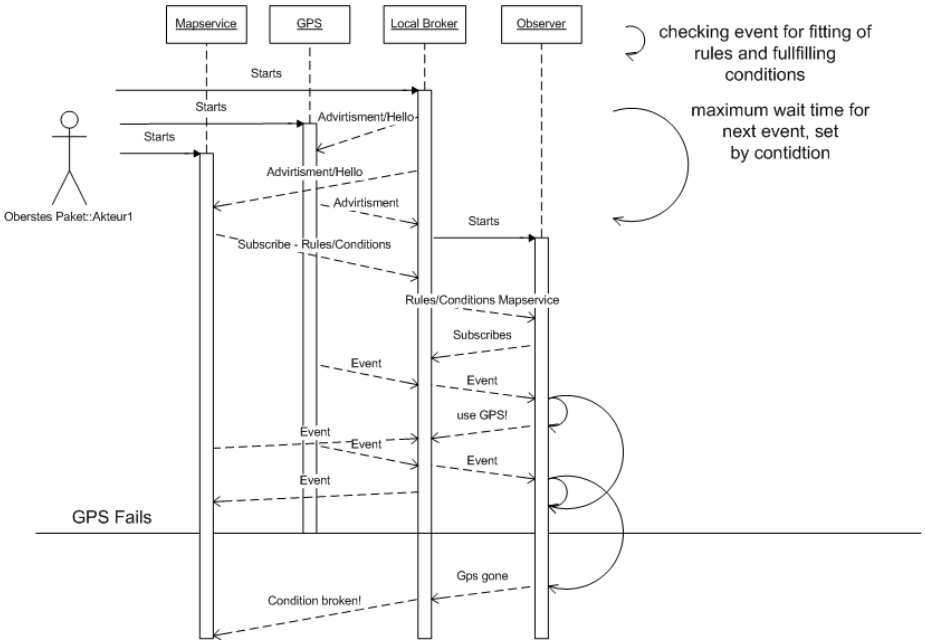

Figure 3: Example: GPS Fails

which breaches our anonymity rules. The EQL could be used for anonymous event subscription. The underlying database would have to be free of private data. Proof of anonymity may be possible, since the EQL is an SQL like language, where proof may be given through relational algebra means. Their architecture already includes means to constrict the usage of published events, but only in knowing the exact groups of participants before runtime. The centralized approach of the Esper engine hinders any p2p or mobile usage of it.

Table 1 evaluates the related work on the requirements named in Section 1. This table shows that even though openness and quality of service orientation are met in the evaluated architectures, the aspects of anonymity, mobility or p2p capability did not receive sufficient attention.

	R1	R2	R3	R4	R5	R6	R7
Hinze and Eschner [1, 3]	++	++	++	++	++	+	o
Di Marzo et al. [6]	–	o	++	++	o	o	–
Michlmayer et al. [5]	+	–	++	++	o	–	+

Table 1: Evaluation of Related Work
Legend: ++ fully supported, + supported, o partly supported, – somewhat supported, – not supported

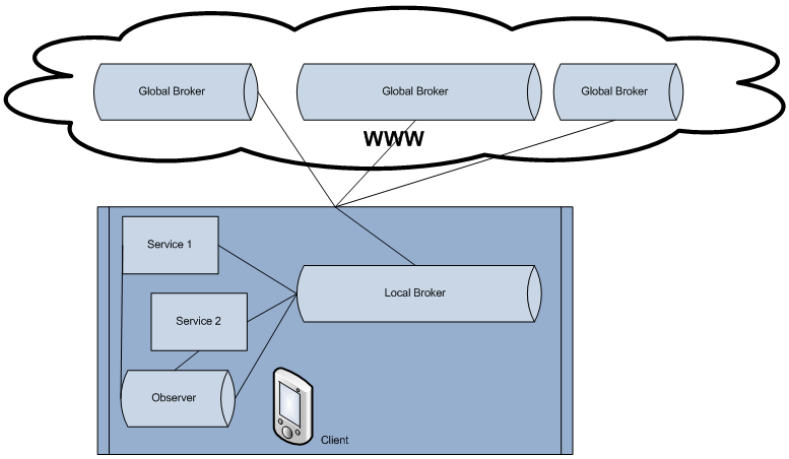

Figure 4: Included Broker

2 Architecture

This section gives detailed information about the conceptual architecture selected to address the issues introduced in Section 1. We also discuss the possible use of existing software frameworks. The section fincishes with the selection of an architectural approach for our implementation.

In the following subsections, we explain how the components (brokers and services) of the model [1] interact with each other. We compare and contrast three approaches and select one for implementation.

2.1 Included Broker

The included broker, shown in Figure 4, locates the broker on the mobile device. Services outside of the device are only accessible through global brokers, to which the local broker connects. This model has several advantages, it ensures fast responses from the local broker to the local services and maximized security, as the messages from the local services to the local broker do not leave the device (R1). Events sent to global brokers could be restricted to-non restricted ones. This would alleviate the proof of the anonymity (R7). Peer to peer approaches could be implemented easily, as local brokers could always start a global one to cooperate with other local brokers near them (R6). This also ensures good quality of service measurements (R4). Subscription of local bound services is easy, since the local broker is always near that location (R5) and since the local broker can connect to any kind of global broker, openness is attained (R3). The disadvantages, however, are significant. All events, used or not used, would be delivered to the broker, which would cause more traffic on the client side (to receive only events needed would breach anonymity rules) and the processing of the rules and conditions on the client device would consume much computation power (R2).

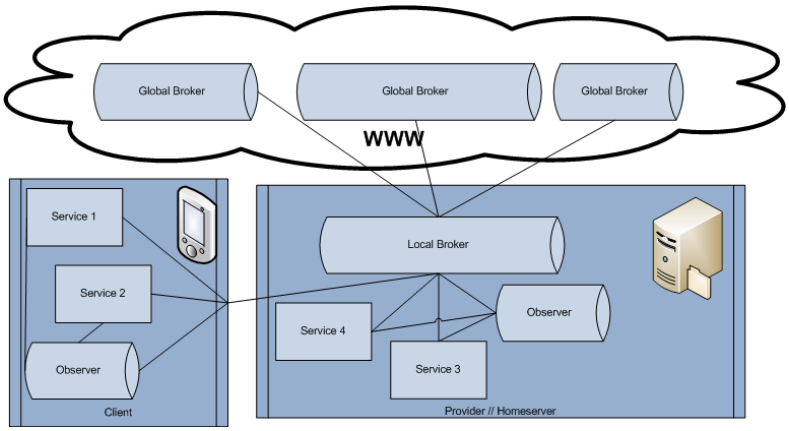

Figure 5: Remote Broker

2.2 Remote Broker

The remote broker concept works like the included broker, except that the local broker is not located on the client machine. It is located on a private server accessible to the user. All connections and events would be routed through this local broker, which would only send events needed to the client device. This concept would bring heavy advantages on saving computing power and traffic on the mobile device (R2). There could be also home supported services, that support the client device, these would be located on the remote server, saving even more capacity of the client. For services needing heavy database support, computation power and traffic capacity could be moved to the remote server. It would hinder connections to locally restricted services and p2p approaches. (R5,R6). As this would mean sending messages concerning subscriptions and advertisements through the internet, security concerns are to be considered (R1,R7). This could be countered with encrypting, which would again consume computation power of the client device. Another problem with this concept is the continuous connection between the client device and the remote local broker. Without this connection, services on the client device would not be able to cooperate. This concept could be better suited for business models, as there could be local broker providers supporting different service types (R3). The remoteness of the broker would bring constraints to quality of service transmission, since these values often depend on the distance between broker and service (R4). The concept of this architecture is shown in Figure 5.

2.3 Intelligent Gateway – Distributed Broker

This architecture is shown in Figure 6. It aims to keep the advantages of the included broker concept without having the restrictions a mobile device would bring with it (R2). It locates only a part of the local broker on the client machine, which only consists of basic capabilities, like delivering the events from one local service to another. It has limited to no reasoning capabilities, that means to process rules or conditions. For this tasks it has to connect to its other part, a heavy logic node on a remote server. It would bring most of the advantages of the included broker (R2,R3,R4,R5), with less disadvantages (R1,R6,R7). One disadvantage is, that the connection between these two parts

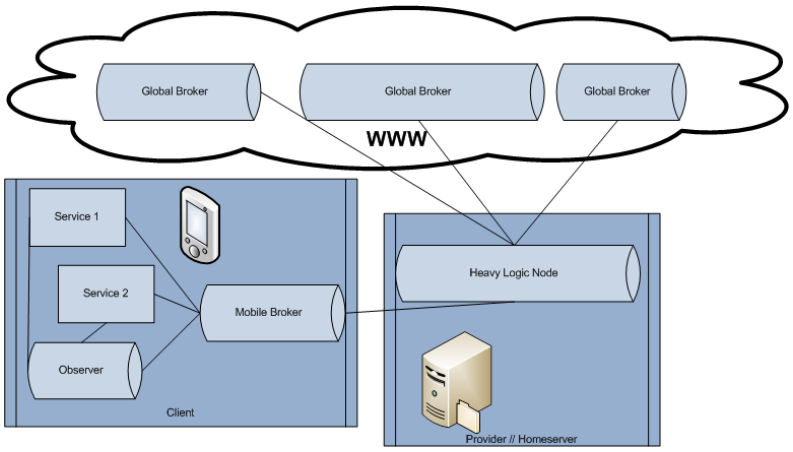

Figure 6: Intelligent Gateway

must be heavily secured. Pear to pear capabilities depend on the exact implementation of the local mobile broker and would be rather complicated. That leads to the main disadvantage of this concept, it would be rather hard to implement. This is because of the communication between separated parts of the broker would need secure ways. The distribution of the work load would also be a problem. This has to be adjusted for every possible client device.

2.4 Related Software

In this subsection we evaluate existing software solutions that may suit our needs (as described in the previous subsections).

JMS The Java Message Service (JMS), is an API used for message based communication between clients. The broker from our model would be represented by the JMS server. It could be used as included broker (JMS server on the mobile device) or remote broker model (JMS server not on the mobile device.) This solution has the following disadvantages:

- No broker to broker communication.
- No changing rules/conditions in runtime.
- Services may only subscribe to one broker type

But has the following advantages:

- Processing of rules/conditions possible.
- It is Open source.
- Different provider (broker) implementations available.

Since this API is open source, it could be adapted to our needs, but this is out of the scope of this work.

Smart Gateways Guerrero et al. [2] aim to work with small computing power in RFID chips in distributed networks. Their goal is to push some of the logic usually

	R1	R2	R3	R4	R5	R6	R7
Included Broker	++	o	++	++	++	++	+
Remote Broker	+	++	++	o	o	o	o
Intelligent Gateway	+	++	++	++	++	o	–

Table 2: Architecture Comparison
Legend: anonymity (R1), mobility (R2), openness of architecture (R3), qos support (R4), services may be location bound (R5), p2p communication (R6), proof of anonymity possible (R7);
++ fully supported, + supported, o partly supported, – somewhat supported, – not supported

processed in a central node closer to the single RFID chip. To reach that goal, they placed smart gateways between the RFID chip and the central node, which preprocess the data following simple event based rules. This architecture is not build to adapt rules while running and connect different services, but the smart gateway concept is very similar to our intelligent gateway concept.

2.5 Conclusion

In comparison (see Table 2), the included broker model seems to be the best in terms of anonymity and usability in different scenarios. It is not really suited for very small mobile devices, but device characteristics are developing fast. The remote broker model does not have this particular weakness, but lacks applicability for the proposed scenarios. Anonymity would still be possible to maintain. The intelligent gateway or distributed broker approach would cover most of the weak spots the other models have. Proof of anonymity of services would be hardest on this model. This is due to the distributed logic, which would have to be well defined and reasoned. It is also the most complex solution to implement.

Although the intelligent gateway may be the best solution for the problem, we decided not to use it, because of the mentioned problems. We therefore decided to work with the included broker solution. The related software solutions we found may be adaptable to this goal, but this would probably be more complex then implementing our own prototype.

3 Implementation

This section focusses on the implementation details of the prototype. The overview of the general architecture is followed by detailed descriptions of the database used. The last part is a detailed documentation of the programmed code of the prototype.

3.1 General Architecture

In this section the general architecture used for the prototype and its components is shown and described in detail. The relationships between the different parts are shown in Figure 7. This figure is a more detailed view of the broker shown in Figure 4 and is showing the exact position of observer and connector. The reasoner and the database tables will be introduced in detail later. At the end of the section the data flow will

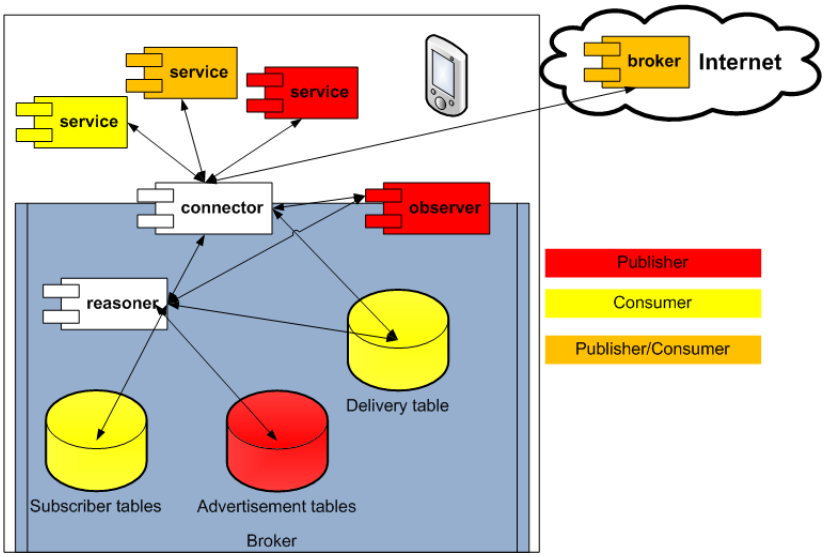

Figure 7: General Architecture

be explained. The blue area represents the broker and its parts and the outside objects represent services. The tables shown in the figure belong to one database, later on referred to as the database. The arrows show which parts are communicating with another.

The Broker The Broker consists of four parts, the connector, the observer, the reasoner and the database. It is the starting environment of these objects. Furthermore it contains configuration information about the broker, for example where to set up the database file (see Section 3.2).

The Reasoner The reasoner is the main component of the broker. It is responsible for setting up the right mappings (that means, which subscriber should get what events) between subscribers and publishers in the database (delivery table), as well as maintaining the database. For example one of the publishers could time out. The corresponding entries in the Database are to be deleted then, as well as possible mappings to subscribers. A reasoner is always connected to one and only one database and used by only one broker.

The Connector The connector is the interface for the broker to subscribers, publishers and other brokers. It maintains connections to all participants and routes incoming events to subscribers, following the mappings in the database (see Subsection 3.2). It will route all incoming subscriptions and advertisements to the reasoner, as well as termination events from the observer or services. It also notifies the observer to start surveillance on advertising publishers.

The Observer The observer checks on preset intervals, if publishing services are still available. It also requests QoS information on these intervals, if the service is able to deliver those.

Tables (Database) The information the broker gets from subscribers and publishers is stored in the database tables. There are different tables for advertisement information and subscriber information, as well as information about which services get which events. For detailed information see Section 3.2. All the tables are stored in one database used by the broker, which is referred to as the database later on.

Subscribers (Consumers) Consumers are services, that want to subscribe on events of a certain type other services publish. In example map events. They do only subscribe under certain conditions In the case more then one publisher produces events that meet the type the subscriber wants, there are rules to sort out which event is to take. A Subscriber service sends a subscription message to the broker (connector), which contains the type and the conditions and rules defined by the subscriber.

Rules Rules are always connected to a subscription and are needed to distinguish which event should be delivered to what subscribers, in case there is more then one that fits the description. They are connected to a subscription via the subscriptions `id`. Rules focus on quality of service attributes of the publishers. Rules therefore always have a quality of service type they correspond to. There are three types of rules: smaller (<), greater (>) and equal (=). These rule types refer to the values the quality of service types may have. If the rule type is =, the value the quality of service type should have is needed as well.

Example 3.1 *Example Rule 1*

- *owner: 1*
- *rule type: >*
- *qos type: accuracy*

This rule means that the accuracy of the delivered event should be the highest.

Example 3.2 *Example Rule 2*

- *owner: 1*
- *rule type: =*
- *qos type: distance*
- *value: 500*

This rule means, that the distance of the service delivering the the events should be exactly 500 (for example: meters).

Conditions Subscribers may define conditions. These conditions specify the terms under which events are accepted from a broker. In this prototype there is only one thing conditions can refer on: the existence of services. Through this, a subscriber is able to demand absence of publishers or presence of a exact number of publishers of a type.

Example 3.3 *Example Condition 1:*

- `ID: 5`
- `object: map`
- `state: exists`
- `cardinality: 2`

This conditions means, that the subscriber only wants to get events from the broker under the term that two publishers for map events exist.

Example 3.4 *Example Condition 2:*

- `ID: 4`
- `object: map`
- `state: exists`
- `cardinality: -1`

This condition means, that there should be at least one publisher existing which is generating map events.

Publishers (Providers) Publishers publish events. To specify the type of the event, publishers send advertisements to brokers. After that, the publisher starts to send events. The publisher also listens to the broker, in case the broker requests quality of service data. If the publisher is able to provide this data, it will send it to the broker in form of a quality of service event (see Subsection 3.3).

Data Flow This section gives a quick overview on how the information is processed when the application runs. Figure 8 is showing an example in a sequence diagram. The difference to the scenarios shown in the introduction is, that the current architecture does not support broker to broker communication. Therefore only local services are accepted.

Outside The Broker On startup, the broker notifies subscribers and publishers about its existence. The services then send subscriptions and advertisements to the broker. The broker notifies subscribers about availability of the requested event type. The broker also notifies publishers if there events are wanted by the broker. Usually the broker accepts all kind of events in case some subscriber may want them. Denegation of events is a option for publishers which are found malicious.

The broker checks publishers occasionally with quality of service requests, or just check if they are still there. In case conditions of subscribers are broken through cease of existence or arriving of publishers, the broker notifies this subscribers.

Inside The Broker All messages (events, subscriptions, advertisements) are handled by the connector, which hands subscriptions and advertisements to the reasoner and also opens channels to the services. Events are routed directly according to the delivery table. The connector also starts observer threads for advertisements.

3.2 Database Layout

This section gives details of the database used by the broker, including detailed description of the used database software and tables.

The database is set up in SQLITE3. SQLITE3 needs no database server running, which suits our needs for mobile devices. It is file based and easy to use on small machines. If for any reason multiple brokers would run on the same machine, differentiation of the different databases can be made through the file names. For this prototype the filename is set to `test.db`.

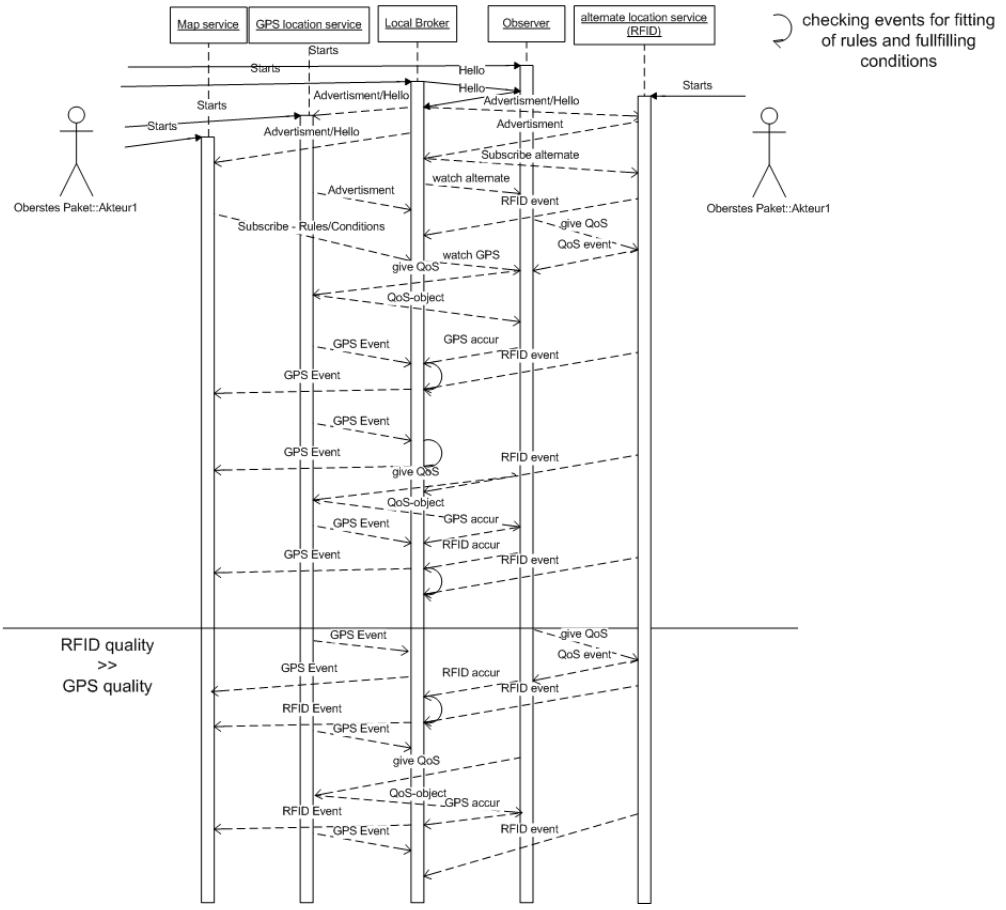

Figure 8: Prototype Sequence

The database shown in Figure 9 consists of six tables. Use and meaning of the tables is explained in the corresponding sections.

Restrictions like `PRIMARY KEY` or `NOT NULL` are secured through the application, not the database. This allows very fast responses from the database.

Table Subscribers The subscribers table is used by the reasoner (see Subsection 3.1) to store incoming `subscription` objects (see Subsection 3.3). It has the three attributes:

- `ID INTEGER -- PRIMARY KEY`
- `OWNER INTEGER -- NOT NULL`
- `type TEXT -- NOT NULL`

The attributes are filled with the corresponding values from the delivered subscriber object. This table identifies the owner of subscriptions and the type of the subscription. The owner is important to later on identify the address the subscription comes from, which would be stored in another table. However, this part is left out in this work, because the network communication is not the focus of the work.

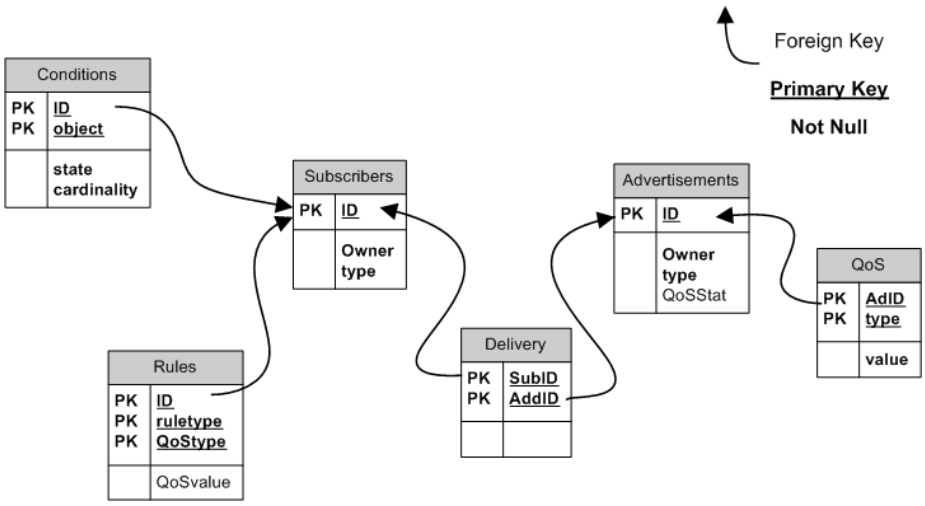

Figure 9: Database Layout

Example 3.5 *Example subscription for map events:*

- *ID: 5*
- *OWNER: 3*
- *type: map*

Table Rules This table exists to store the `rule` objects belonging to a subscription. The table contains the following attributes:

- `ID INTEGER -- PRIMARY KEY (FOREIGN KEY Subscriber ID)`
- `ruletype TEXT`
- `qostype TEXT -- PRIMARY KEY`
- `qosvalue TEXT`

Via this table the application can look up the rules belonging to each subscription with the id `ID`, where the attributes `ruletype` and `qostype` correspond to the similar named variables in `rule` objects (see Subsection 3.3). The value `qosvalue` is only used for rules that are of the type =.

Example 3.6 *Example rule for accuracy of events being high:*

- *ID: 5*
- *ruletype: >*
- *qostype: accuracy*
- *qosvalue: NULL*

Example 3.7 *Example rule for distance of events being 50:*

- *ID: 5*
- *ruletype: =*
- *qostype: distance*
- *qosvalue: 50*

Table Conditions `Condition` objects (see Subsection 3.3) are stored similar to rules in a condition table, which has the following attributes:

- `ID INTEGER -- PRIMARY KEY (FOREIGN KEY Subscriber ID)`
- `object TEXT -- PRIMARY KEY`
- `state TEXT -- NOT NULL`
- `cardinality INTEGER -- NOT NULL`

The `ID` identifies the `subscription` the condition belongs to and the three following attributes correspond to the values given in `condition` objects.

Example 3.8 *Example condition saying there has to be exact two mapservices:*

- *ID: 5*
- *object: map*
- *state: exists*
- *cardinality: 2*

Table Advertisements This table is used to store incoming advertisement objects (see Subsection 3.3) received by the reasoner (see Subsection (3.1)). The following attributes are given:

- `ID INTEGER -- PRIMARY KEY`
- `OwnID INTEGER -- PRIMARY KEY`
- `type TEXT -- NOT NULL`
- `QoSStat TEXT`

The use of this table is to get the id of events of certain types and the id of their provider. The key phrase needed for getting QoS events for certain advertisements is also found here.

Example 3.9 *Example of an advertisement for map events:*

- *ID: 10*
- *OwnID:8*
- *type: map*
- *QoSStat: getMapQoS*

Table QoS QoS values of advertisements are stored in the QoS table, which has the following attributes:

- `AdID INTEGER -- PRIMARY KEY (FOREIGN KEY Advertisement ID)`
- `type TEXT -- PRIMARY KEY`
- `value INTEGER -- NOT NULL`

This table is used when QoS values need to be checked. The id `AdID` marks the advertisement these QoS values belong to with pairs of `type` and `value`. These pairs correspond to the value pairs in the QoSlist (see 3.3).

Example 3.10 *Example of QoSlist entry saying that the distance of advertisement 8 is 50:*

- *AdID: 1*
- *type: distance*
- *value: 50*

Table Delivery This table is used to store the mappings calculated by the reasoner for subscribers and advertisements (which subscriber gets which publishers events). Therefore it has the following two attributes:

- `SubID INTEGER -- PRIMARY KEY (FOREIGN KEY Subscriber ID)`
- `AddID INTEGER -- PRIMARY KEY (FOREIGN KEY Advertisement ID)`

All events marked with the id `AddID` should be delivered to the owner of the subscription marked with the id `SubID`.

Example 3.11 *Example of subscriber 5 subscribing to advertisement 8:*

- *`SubID: 5`*
- *`AddID: 8`*

3.3 Classes and Methods

This section gives details about the prototype implementation on software level. A detailed description of the different usages of the parts in the conceptual architecture see Section 3.1.

Objects The `Objects` project is used for having an object library for the different participants of the architecture. It contains all objects used by more then one participant. Therefore it is included in the other projects.

Class Subscription The `subscription` objects purpose is transmitting the subscriptions of services. It therefore contains the `id`, the `rules`, the `owner` and the `conditions` of the subscriber. Most important, it contains the `type` of events the subscriber wants.

```
public class Subscription implements Serializable{
    private int ID;
    private int owner;
    private String type;
    private Vector<Rule> rule;
    private Vector<Condition> condition;
    public Subscription(int ID, int owner, String type,
                        Vector<Rule> rule,
                        Vector<Condition> condition){
        this.ID=ID;
        this.owner=owner;
        this.type=new String(type);
        this.rule=(Vector<Rule>) rule.clone();
        this.condition=(Vector<Condition>) condition.clone();
    }
```

The owner represents the sending service of the `subscription`, that means a service may send more than one `subscription` objects. In that case the `ID` value would differ, but the `owner` value would stay the same. It is serializable. This reason for this is, that the subscriptions are send via object writers and readers. There are getters for every value the `subscription` object contains, but they can only be set at creation, to prevent manipulation.

```
public String getType(){
        String rettype=new String(this.type);
        return rettype;
    }
```

Returned objects are always new ones, to prevent corruption of the reference of the stored values.

Class Advertisement `Advertisement` objects function similar to the `subscription` objects concerning the `ID`, `owner` and `type` values.

```
public class Advertisement implements Serializable{

    private int ID;
    private int owner;
    private String type;
    private String getQoS;

    public Advertisement(int ID, int owner,
                         String type, String getQoS) {
        this.ID = ID;
        this.owner = owner;
        this.type = new String(type);
        this.getQoS = new String(getQoS);
}
```

Of course the `type` value here is to determine the events offered by the publisher sending this advertisement. The `getQoS` value contains the quality of service phrase which is to be send to the publisher to get quality of service values corresponding to the advertisement. This is useful in case a publisher offers more then one eventtype. Similar to the `subscription` objects, there are getters for all values, but no setters to prevent accidental change of the values. Like the `subscriber` objects, the `advertisement` objects are serializable so that they can be send through object writers.

```
public String getQoS() {
        String retQoS = new String(this.getQoS);
        return retQoS;
}
```

Class Rule The `rule` objects store rules belonging to subscriptions (see Subsection 3.1). A `rule` always belongs to only one `subscription`.

```
public class Rule implements Serializable{

    private int owner = 0;
    private char ruletype = '=';
    private String qostype;
    private int qosval = 0;

    public Rule(int owner, char type,
                String value, int qosval) {
        this.owner = owner;
        this.ruletype = type;
        this.qostype = new String(value);
        this.qosval=qosval;
    }
    public Rule(int owner, char type, String value) {
        this.owner = owner;
        this.ruletype = type;
        this.qostype = new String(value);
}
```

The `subscription` owning the `rule` is identified via the `owner` value, the other values correspond to the rule definition made in 3.1. The `qosval` is only needed when `ruletype` equals =. There are getter methods for the attributes which look like:

```
public String getQoStype(){
        String retval=new String(this.qostype);
        return retval;
}
```

There are no setter methods except the constructor, for the same reason as with the `subscription` or `advertisement` objects.

Class Condition The `condition` objects store conditions belonging to subscriptions (see Subsection 3.1).

```
public class Condition implements Serializable{

    private int owner = 0;
    private String object;
    private String state;
    private int cardinality = 0;


    public Condition(int owner, String object,
                String state, int cardinality){
        this.owner=owner;
        this.object=new String(object);
        this.state=new String(state);
        this.cardinality=cardinality;
    }

    public int getOwner() {
        return this.owner;
    }
```

The use of the single values is given in Subsection 3.1. Same as `rule` objects, `condition` objects belong to exactly on `subscription`. There are getter methods for all values as shown in the code block above.

Class QoSlist These objects are used to transport quality of service information.

```
public class QoSlist implements Serializable{
        private int ID;
        private Vector<String> types = new Vector<String>();
        private Vector<Integer> values =new Vector<Integer>();

        public QoSlist(int ID, Vector<String> types, Vector<Integer> values) {
                this.ID=ID;
                this.types=(Vector<String>) types.clone();
                this.values=(Vector<Integer>) values.clone();
}
```

Since an `event` could have more than one quality of service criteria, the `QoSlist` has vectors of `types` and `values`. Values on the same index in the vectors are a pair. The `ID` of a `QoSlist` links it to the corresponding `advertisement`. There are getter methods for `ID` and the other values (returns the whole vector) and a setter method for the values.

```
public synchronized void setValues(Vector<Integer> values){
        this.values=(Vector<Integer>) values.clone();
}
```

The new `values` vector should have the same size as the former one.

Class Terminated These objects are used to notify an application about the termination of the connection of a participant.

```
public class Terminated implements Serializable{

    private int ID;
    private String type;
    private String reason;

    public void Terminated(int ID, String type, String reason) {
        this.ID = ID;
        this.type = new String(type);
        this.reason = new String(reason);
    }
```

The use of the `ID` value depends on where the object is used. While used inside the `broker` (that means in between classes of the broker application) the `ID` represents the service whichs connection has been terminated. The `ID` of `Terminated` objects send to other services represents the broker sending it. The `reason` represents a short message explaining why termination occurred (i.e.: Condition broken). The `type` is used to notfiy subscribers on which type of event cannot be delivered at the moment. The values of this object are set at construction, getter methods for all of them exist:

```
public int getID() {
        return this.ID;
}
```

A `Terminated` object may be send to:

- the broker notifies a publisher that its events are not needed
- the broker notifies a subscriber that event of a given type cannot be delivered at the moment
- an observer thread notifies the connector that a certain service is not available anymore
- a publisher notifies the broker that its going to end the participation
- a subscriber notifies the broker that its going to end the participation

Class Event The `event` objects are used to carry the actual data send by the publishers. They have to be serializable to be send through object writers and read by object readers.

```
public class Event<T extends Serializable> implements Serializable{

        private int ID;
        private T todeliver=null;

        public Event(T obj, int id){
                this.todeliver=obj;
                this.ID=id;
        }
        public int getID(){
                return this.ID;
        }
        public T getContent(){
                return this.todeliver;
        }
}
```

Since we do not know which kind of data the publishers may provide, the data object `T` is initialized on construction of the object. Since `event` objects are generic, they have to be called like this:

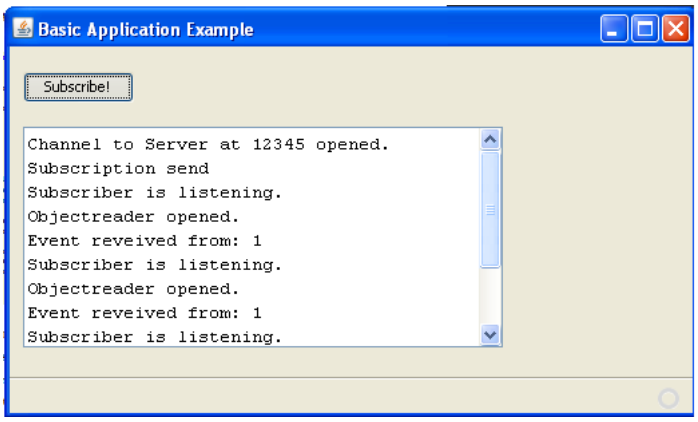

Figure 10: Subscriber GUI

```
Event<String> newevent = new Event<String>("Hello World", 1);
```

To properly read the event, the type of the data should be known. The `ID` refers to the sender of the `event`.

3.4 Subscriber Application

This example service was created as a netbeans desktop application project. The GUI is shown in Figure 10. The `subscribe!` button starts the method

```
public void connect()
```

This method firstly initializes a new subscription object

```
Vector<Rule> rules = new Vector<Rule>();
Vector<Condition> conds = new Vector<Condition>();
rules.add(new Rule(10, '>', "accuracy"));
rules.add(new Rule(10, '<', "latency"));
rules.add(new Rule(10, '<', "distance"));
conds.add(new Condition(10, "info", "exists", -1));
Subscription sub = new Subscription(10, 10, "info", rules, conds);
```

and sends this via the socket 12345 to the broker. It then starts a `listener` object in a new thread to accept events delivered from the broker.

```
Listener listen = new Listener(1210,outputfield);
Thread listenerThread = new Thread(listen);
listenerThread.start();
```

The text area is accessible through the the variable `outputfield`, via the command:

```
outputfield.append(String text);
```

Class Listener The listener class

```
public class Listener implements Runnable
```

has two global variables, which are set through the constructor.

```
private int port;
private javax.swing.JTextArea outputField;
Listener(int port, javax.swing.JTextArea outputField){
        this.port=port;
        this.outputField=outputField;
    }
```

The integer port is the socket address used on listening to the broker for events. In this prototype implementation it is usually 1200 + service id. On call the class starts the `listen` method to wait for event objects:

```
public void listen()
```

This method starts a `server socket` to listen to the broker and uses an object reader to get the serialized objects send from the broker.

```
Event ev=null;
ServerSocket srvr=null;
        try {
                while (true) {
                        srvr = new ServerSocket(this.port);
      outputField.append("Subscriber is listening.\n");
      Socket skt = srvr.accept();
      ObjectInputStream inObjs = new ObjectInputStream(skt.getInputStream());
      outputField.append("Objectreader opened.\n");
      Object obj = null;
      while (true) {
        try {
                obj = inObjs.readObject();
          if (obj instanceof Event) {
                ev = (Event) obj;
            outputField.append("Event reveived from: " + ev.getID() + "\n");
          }
        } catch (Exception e) {
                        break;
                }
        }

       inObjs.close();
       skt.close();
       srvr.close();
      }
   } catch (Exception e) {
 outputField.append("subscriber listener error: " + e+"\n");
 try {
        srvr.close();
 } catch (IOException ex) {
        System.out.println(ex);
  }
}
```

In case events are received, the information is displayed in the text area. The application runs till stopped by the user.

3.5 Publisher Application

The example publishing applications gui is shown in Figure 11. It hast the global variables:

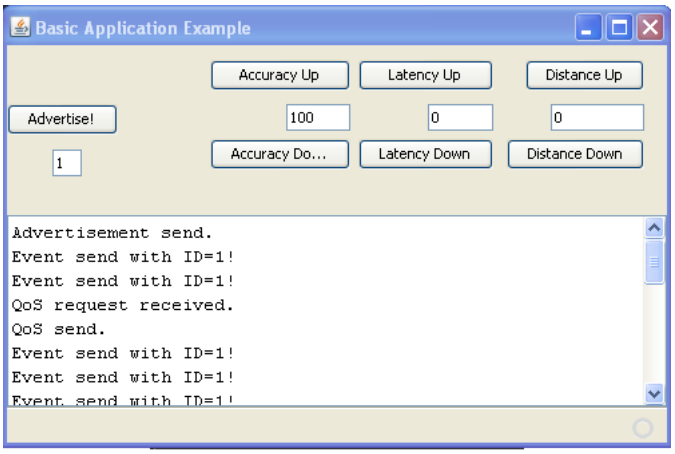

Figure 11: Publisher GUI

```
private Vector<String> types=new Vector<String>();
private Vector<Integer> values=new Vector<Integer>();
private QoSlist qoslist;
```

These variables are needed to store the quality of service values of this application. There are three types

```
types.add("accuracy");
types.add("latency");
types.add("distance");
```

for this example application. Their values are changeable through the “up” and “down” buttons. These buttons start simple methods to access the quality of service object `qoslist` like in the following example for “accuracy up”:

```
@Action
public void accuracyUp() {
        int accuracy=qoslist.getValues().get(0);
        accuracy++;
        Vector<Integer> temp = qoslist.getValues();
        temp.setElementAt(accuracy, 0);
        qoslist.setValues(temp);
        accuracyField.setText(Integer.toString(accuracy));
}
```

Before starting to advertise, an id has to be chosen via the text field below the “Advertise!” button. Upon pushing the “Advertise!” button, the method

```
public void connect()
```

is called and starts to send a preset advertisement to the broker as well as setting up the `qoslist` with the entered id.

```
Integer ID = new Integer(IDfield.getText());
qoslist.setID(ID);
Advertisement ad = new Advertisement(ID, ID, "info", "QoS");
Eventstack events = new Eventstack(ID);
```

The Eventstack `events` is used for communication between the `Sender` and the `Listener` object (see Subsections below this one), its a synchronized class that allows different threads to share a source of events.

```
class Eventstack {

    private Vector<Event> events;
    private int ID;

    Eventstack(int ID) {
        this.events = new Vector<Event>();
        this.ID=ID;
    }
    public synchronized void add(Event ev){
        this.events.add(ev);
    }
     public synchronized int getID(){
         return this.ID;
     }
    public synchronized Event getEvent() {
        try {
            Event el=this.events.firstElement();
            this.events.remove(el);
            return el;
        } catch (Exception e) {
            return null;
        }
    }
}
```

After sending the advertisement the sender and listener threads are started to send the events to the broker and listen for quality of service requests or decline of events.

```
Listener listener = new Listener(listenport, events, outputField);
Thread listenerThread = new Thread(listener);
listenerThread.start();
Sender sender = new Sender(12345, events, qoslist,outputField, ID);
Thread senderThread = new Thread(sender);
senderThread.start();
```

The application runs until stopped by the user.

Class Listener The `Listener` class is a separate thread of the publishing application

```
public class Listener implements Runnable{
    private int port;
    private Eventstack evstack;
    private javax.swing.JTextArea outputField;
    Listener(int port, Eventstack evstack, javax.swing.JTextArea outputField){
        this.port=port;
        this.evstack=evstack;
        this.outputField=outputField;

}
```

and works like the listener of the subscriber application (see Subsubsection 3.4). It has one difference. A `Eventstack` is handed down from the main application. The listener will put a simple `event` containing the string “qos” on the stack, in case it receives request for quality of service values from the broker.

Class Sender The `sender` class is a separate thread

```
public class Sender implements Runnable{
    int port;
    Eventstack events;
    private QoSlist qoslist;
    private javax.swing.JTextArea outputField;
    int ID=0;

    public Sender(int port, Eventstack events, QoSlist qoslist,javax.swing.
         JTextArea out, int ID) {
        this.port = port;
        this.events = events;
        this.qoslist=qoslist;
        this.outputField=out;
        this.ID=ID;

}
```

and sends the `events` produced by the application (which it would store in the `eventstack`). Since this a example application does not produce real events, it sends dummy events with no content. In case a `event` is on the `eventstack`) it sends the current quality of service object `qoslist`.

```
private void sending() {
        Event<String> event = new Event<String>("infoevent", ID);
        while (true) {
            try {
                Socket skt = new Socket(InetAddress.getLocalHost(), port);
                ObjectOutputStream outObjs = new ObjectOutputStream(skt.
                     getOutputStream());
                Thread.sleep(2000);
                skt = new Socket(InetAddress.getLocalHost(), port);
                outObjs = new ObjectOutputStream(skt.getOutputStream());
                while (events.getEvent() != null) {
                    outObjs.writeObject(this.qoslist);
                    outputField.append("QoS send.\n");
                }
                outObjs.writeObject(event);
                outputField.append("Event send with ID="+ID+"!\n");
                outObjs.close();
                skt.close();
            } catch (Exception ex) {
                System.out.println(ex);
            }
        }
    }
```

The command `outObjs.writeObject(event);` serializes an event and writes it in the object stream, connected with the broker. Therefore events have to be serializable.

3.6 Broker Application

The prototypic broker application focusses on the reasoner details, as this was the main point of this work.

```
    public static void main(String[] args) {
        Dbcon con = null;
        try {
            con = new Dbcon("c:\\test.db"); //path to the database file
            con.create();
        } catch (Exception ex) {
            System.out.println(ex);
        }
        Connector connect = new Connector(12345, con);
        Thread connectThread = new Thread(connect);
        connectThread.start();
```

```
    }
```

Firstly a new database is created, which is then along with the port 12345 used for starting up a connector. The connector itself starts observer threads and the reasoner. The path to the database file may be altered by just changing the string used to create the `Dbcon` object. It is does not suffice to just create a new `Dbcon` object, the `Dbcon.create()` method has to be called to. Any exceptions thrown by accessing the database file are thrown to the calling class.

Class DBCon This class is used for creating, accessing and maintaining the database used by the application.

```
public class Dbcon{
    private Connection conn;

    public Dbcon(String loc) throws Exception {
        //Connecting
        Class.forName("org.sqlite.JDBC");
        this.conn = DriverManager.getConnection("jdbc:sqlite:"+loc);
    }
```

The sqlite java library is used for creating a Database, in case the file does not exist yet, it is created. The next step is to create the tables needed.

```
public void create() throws Exception{
        Statement stat = this.conn.createStatement();
        stat.executeUpdate("drop table if exists advertisements;");
        stat.executeUpdate("drop table if exists delivery;");
        stat.executeUpdate("drop table if exists QoS;");
        stat.executeUpdate("drop table if exists conditions;");
        stat.executeUpdate("drop table if exists rules;");
        stat.executeUpdate("drop table if exists subscribers;");
        stat.executeUpdate("create table advertisements" +
                "(ID INTEGER PRIMARY KEY, OwnID INTEGER, type Text, QoSStat Text
                    );");
        stat.executeUpdate("create table delivery (SubID INTEGER, AddID INTEGER)
            ;");
        stat.executeUpdate("create table QoS (AdID INTEGER, type Text, value
            INTEGER);");
        stat.executeUpdate("create table rules (ID INTEGER, ruletype Text,
            qostype Text, qosvalue INTEGER);");
        stat.executeUpdate("create table conditions (ID INTEGER, object Text,
            state Text, cardinality INTEGER);");
        stat.executeUpdate("create table subscribers (OWNER INTEGER, type Text,
            ID INTEGER PRIMARY KEY);");
    }
```

The first step is to delete any old tables left in the database, then the new ones are created following the definition in Subsection 3.2. A method for closing the connection is available:

```
    public void close()throws Exception{
        this.conn.close();
    }
```

To check if a subscribers conditions are met at the current time, the method `condmet(Vector<Condition>)` can be called.

```
public boolean condmet(Vector<Condition> conds) throws Exception {
        boolean retval = true;
        Iterator<Condition> conditions = conds.iterator();
        Integer card = 0;
```

```
        Statement stat = this.conn.createStatement();
        ResultSet rs;
        Condition temp;
        while (conditions.hasNext()) {
            temp = conditions.next();
            if (temp.getState().equals("exists")) {
                rs = stat.executeQuery("SELECT count(DISTINCT ID) FROM
                     advertisements WHERE type ='" + temp.getObject() + "';");
                card = rs.getInt(1);
                if (temp.getCardinality() != card && temp.getCardinality() > -1)
                     {
                   retval = false;
                }
                if (card!=0 && temp.getCardinality()==0) {
                    retval = false;
                }
            }
        }
        stat.close();
        return retval;
    }
```

At the moment, only conditions with the state `existing` (see Subsection 3.1)can be checked. The method returns a boolean. The boolean is true if the conditions are met and false if not. To add a condition to the database, the method

```
 public void add(Condition cond) throws Exception {
        Statement stat = this.conn.createStatement();
        stat.executeUpdate("INSERT INTO conditions VALUES(" + cond.getOwner() +
             ", '" + cond.getObject() + "', " +
                "'" + cond.getState() + "', " + cond.getCardinality() + ");");
        stat.close();
    }
```

is called. This method is overloaded several times, to add all objects needed to the database:

```
public void add(Subscription sub) throws Exception {
        Statement stat = this.conn.createStatement();
        stat.executeUpdate("INSERT INTO subscribers VALUES ("+ sub.getOwner() +"
             , '"+ sub.getType() +"', "+ sub.getID() +");");
        stat.close();
}

public void add(Rule rule) throws Exception {
        Statement stat = this.conn.createStatement();
        stat.executeUpdate("INSERT INTO rules VALUES (" + rule.getOwner() + ", '
             " + rule.getType() + "', '" + rule.getQoStype() +"', " + rule.
            getQosval() + ");");
        stat.close();
        }

public void add(Advertisement ad) throws Exception {
        Statement stat = this.conn.createStatement();
        if (ad.getQoS().isEmpty()){
                stat.executeUpdate(
                "insert into advertisements values ("+ ad.getID() + ", "+ ad.
                     getOwner()+", '" + ad.getType() +"', '');");
        }
        else{
                stat.executeUpdate(
                "insert into advertisements values ("+ ad.getID() + ", "+ ad.
                     getOwner()+", '" + ad.getType() +"', '"+ ad.getQoS() +"');"
                     );
        }
        stat.close();
}

public void add(int ID, String type, int Value) throws Exception {
        Statement stat = this.conn.createStatement();
```

```
        stat.executeUpdate("insert into QoS values (" + ID + ", '" + type + "',
             " + Value + ");");
        stat.close();
}

public void add(int eventid, int sendto) throws Exception {
        Statement stat = this.conn.createStatement();
        stat.executeUpdate("insert into delivery values (" + sendto + ", " +
             eventid + ");");
        stat.close();
}
```

Calling this method with

```
public void add(int ID, String type, int Value)
```

creates new quality of service entries in the `qos` table. Calling it this way:

```
public void add(int eventid, int sendto)
```

creates new entries in the `delivery` table. To remove a `condition`, `subscription`, `rule`, `advertisement` or `qos` object from the database, the following methods are used.

```
public void remCondition(int id) throws Exception {
        Statement stat = this.conn.createStatement();
        stat.executeUpdate("delete from conditions where ID=" + id + ";");
        stat.close();
}

public void remRule(int id) throws Exception {
        Statement stat = this.conn.createStatement();
        stat.executeUpdate("delete from rules where ID=" + id + ";");
        stat.close();
}

public void remSubscription(int id) throws Exception {
        Statement stat = this.conn.createStatement();
        stat.executeUpdate("delete from subscribers where ID=" + id + ";");
        stat.close();
        this.remDelSub(id);
}

public String remAd(int adid) throws Exception {
        String ret=null;
        Statement stat = this.conn.createStatement();
        ResultSet rs = stat.executeQuery("SELECT type FROM advertisements WHERE
             ID=" + adid + ";");
        if(rs.next()){
                ret = rs.getString(1);
        }
        stat.executeUpdate("DELETE FROM advertisements WHERE ID=" + adid + ";");
        stat.close();
        this.remDelAdd(adid);
        this.remQoS(adid);
        return ret;
}
public void remQoS(int adid) throws Exception {
        Statement stat = this.conn.createStatement();
        stat.executeUpdate(
        "delete from QoS where AdID=" + adid + ";");
        stat.close();
}
```

They return nothing except for the `remAd` method, which returns the `type` of the removed `advertisement` in case it is needed. In the case a `qos` value needs to be updated in the database, this method is to be called:

```
public void upQoS(int adid, String type, int value) throws Exception {
        Statement stat = this.conn.createStatement();
        int exe = stat.executeUpdate("UPDATE QoS SET value= "+ value +" WHERE
             AdID=" + adid + " AND type='"+type+"';");
        if(exe==0){
                stat.executeUpdate("insert into QoS values (" + adid + ", '" +
                     type + "', " + value + ");");
        }
        stat.close();
}
```

To remove entries from the `delivery` table, the following two methods are needed:

```
public void remDelAdd(int evid) throws Exception {
        Statement stat = this.conn.createStatement();
        stat.executeUpdate("delete from delivery where addid=" + evid + ";");
        stat.close();
}
public void remDelSub(int subid) throws Exception {
        Statement stat = this.conn.createStatement();
        stat.executeUpdate("delete from delivery where subid=" + subid + ";");
  stat.close();
}
```

The method `remDelAdd` removes all entries belonging to the given `advertisement id`. The method `remDelSub` does this for a given `subscriber id`.

There are several methods for extracting data from the database, they usally return the object mentioned in the name.

```
public Vector<Condition> getConditions(int owner) throws Exception {
        Vector<Condition> conds = new Vector<Condition>();
        Statement stat = this.conn.createStatement();
        ResultSet rs = stat.executeQuery("SELECT * FROM conditions WHERE ID=" +
             owner + " ;");
        while (rs.next()) {
                Condition cond = new Condition(owner, rs.getString(2), rs.
                     getString(3), rs.getInt(4));
                conds.add(cond);
        }
        stat.close();
        return conds;
}

public Vector<Rule> getRules(int owner) throws Exception {
        Vector<Rule> rules = new Vector<Rule>();
        Statement stat = this.conn.createStatement();
        ResultSet rs = stat.executeQuery("SELECT * FROM rules WHERE ID=" + owner
              + " ;");
        while (rs.next()) {
                Rule rule = new Rule(owner, rs.getString(2).charAt(0), rs.
                     getString(3), rs.getInt(4));
                rules.add(rule);
        }
        stat.close();
        return rules;
}

public Vector<Subscription> getSubscriptions(int id, String qostype) throws
     Exception{
        String adtype=null;
        Vector<Subscription> subs = new Vector<Subscription>();
        Statement stat = this.conn.createStatement();
        ResultSet rs =stat.executeQuery("SELECT type FROM advertisements WHERE
             id="+id+";");
        if(rs.next()){
                adtype=rs.getString(1);
        }
        rs = stat.executeQuery("SELECT DISTINCT * FROM subscribers s JOIN rules
             r ON s.id=r.id WHERE s.type='"+ adtype +"' AND r.qostype='"+qostype
             +"';");
```

```
        while(rs.next()){
                subs.add(new Subscription(rs.getInt(3), rs.getInt(1),rs.
                    getString(2),this.getRules(rs.getInt(3)),this.getConditions
                    (rs.getInt(3))));
        }
        stat.close();
        return subs;
}

public Vector<Subscription> getSubscriptions(String type) throws Exception{
        Vector<Subscription> subs = new Vector<Subscription>();
        Statement stat = this.conn.createStatement();
        ResultSet rs = stat.executeQuery("SELECT * FROM subscribers s LEFT OUTER
             JOIN conditions c ON s.id=c.id WHERE s.type='"+ type+"' OR c.
            object='"+type+"';");
        while(rs.next()){
                subs.add(new Subscription(rs.getInt(3), rs.getInt(1),rs.
                    getString(2),this.getRules(rs.getInt(3)),this.getConditions
                    (rs.getInt(3))));
        }
        stat.close();
        return subs;
}

public int getProvider(String type) throws Exception {
        Statement stat = this.conn.createStatement();
        ResultSet rs = stat.executeQuery("SELECT id FROM advertisements WHERE
             type ='" + type + "';");
        if (rs.next()) {
                int ret = rs.getInt(1);
                stat.close();
                return ret;
        }
        stat.close();
        return -1;
}
public Vector<Integer> getUnservedSubs(String type) throws Exception{
        Vector<Integer> subs = new Vector<Integer>();
        Statement stat = this.conn.createStatement();
        ResultSet rs = stat.executeQuery("SELECT id FROM subscribers WHERE type
             ='"+ type +"' AND id NOT IN (Select subid FROM delivery);");
        while(rs.next()){
                subs.add(rs.getInt(1));
        }
        stat.close();
        return subs;
}
public Vector<Integer> getDel(int evid) throws Exception {
        Statement stat = this.conn.createStatement();
        Vector<Integer> subs = new Vector<Integer>();
        ResultSet rs = stat.executeQuery("SELECT subID FROM delivery WHERE addid
             =" + evid + ";");
        while(rs.next()){
                subs.add(rs.getInt(1));
        }
        stat.close();
        return subs;
}
```

Usually the objects are filtered through there ID, accept for the `getProvider`, `getUnservedSubs` and `getSubscriptions` methods. The `getProvider` returns a vector of integers representing the `ids` of advertisements offering a event type, defined by the string used when calling the method. The `getUnservedSubs` method returns a vector of integers representing the `ids` of subscribers of a specified type (by the string used to call the method) that currently do not get any events delivered. The two `getSubscriptions` methods return a vector of `subscription` objects. The call `getSubscriptions(String type)` returns all `subscription` objects with the type used in the call. The call `getSubscriptions(int id, String qostype)` is called with an `advertisement id` and a quality of service `type`.

It returns all `subscription` that are subscribed on the `type` of the `advertisement` belonging to the given `id` and have rules regarding the quality of service `type` used in the method call. The reason for this method is to get any subscribers that could be affected by a quality of service change of a `advertisement`. Finally there are three methods for getting the `advertisement ids` for the highest, the lowest or the exact quality of service value of a given quality of service and event type:

```
public Vector<Integer> getMaxQoS(String QoStype, String servicetype) throws
     Exception {
        Vector<Integer> ids = new Vector<Integer>();
        Statement stat = this.conn.createStatement();
        ResultSet rs = stat.executeQuery("SELECT max(q.value) FROM
             advertisements a, QoS q WHERE" +
                " a.ID=q.AdID AND a.type='" + servicetype + "' AND q.type='" +
                    QoStype + "' ;");
        ResultSet rs1 = stat.executeQuery("SELECT a.ID FROM advertisements a,
             QoS q WHERE" +
                " a.ID=q.AdID AND a.type='" + servicetype + "' AND q.type='" +
                    QoStype + "' AND q.value=" + rs.getInt(1) + ";");
        while(rs1.next()){
                ids.add(rs1.getInt(1));
        }
        stat.close();
        return ids;
}

public Vector<Integer> getMinQoS(String QoStype, String servicetype) throws
     Exception {
        Vector<Integer> ids = new Vector<Integer>();
        Statement stat = this.conn.createStatement();
        ResultSet rs = stat.executeQuery("SELECT min(q.value) FROM
             advertisements a, QoS q WHERE" +
                " a.ID=q.AdID AND a.type='" + servicetype + "' AND q.type='" +
                    QoStype + "' ;");
        ResultSet rs1 = stat.executeQuery("SELECT a.ID FROM advertisements a,
             QoS q WHERE" +
                " a.ID=q.AdID AND a.type='" + servicetype + "' AND q.type='" +
                    QoStype + "' AND q.value=" + rs.getInt(1) + ";");
        while(rs1.next()){
                ids.add(rs1.getInt(1));
        }
        stat.close();
        return ids;
}

public Vector<Integer> getExactQoS(String QoStype, String servicetype, int qos)
     throws Exception {
        Vector<Integer> ids = new Vector<Integer>();
        Statement stat = this.conn.createStatement();
        ResultSet rs = stat.executeQuery("SELECT a.ID FROM advertisements a, QoS
              q WHERE" +
                " a.ID=q.AdID AND a.type='" + servicetype + "' AND q.type='" +
                    QoStype + "' AND q.value=" + qos + ";");
        while(rs.next()){
                ids.add(rs.getInt(1));
        }
        stat.close();
        return ids;
}
```

Class Reasoner The reasoner class is constructed with a `Dbcon` object.

```
public class Reasoner {

        private Dbcon conn;

        public Reasoner(Dbcon conn) {
        this.conn = conn;
```

```
        }
```

This `Dbcon` object is used for all data stored and processed by the reasoner. The `process` method is overloaded several times to process all possible incoming objects.

```
public boolean process(Subscription sub) {
        Iterator<Rule> rules = sub.getRule().iterator();
        Iterator<Condition> conds = sub.getCondition().iterator();
        boolean conditionmet = true;
        try {
                conn.add(sub);
                while (rules.hasNext()) {
                        conn.add(rules.next());
                }
                while (conds.hasNext()) {
                        conn.add(conds.next());
                }
                conditionmet = conn.condmet(sub.getCondition());
                if (conditionmet) {
                        this.map(sub);
                }
        } catch (Exception ex) {
                System.out.println("processerror (subscription):" + ex);
        }
        return conditionmet;
}
```

The return value of a processed `subscription` is a boolean stating wether it receives events at this time (true) or not (false).

The processing of `advertisement` objects returns a list of subscribers which are still or again unserved after processing the object.

```
public Vector<Integer> process(Advertisement add) {
        Vector<Integer> lostsubs = new Vector<Integer>();
        Vector<Integer> subs = new Vector<Integer>();
        int temp;
        try {
                conn.add(add);
                subs = conn.getUnservedSubs(add.getType());
                Iterator<Integer> sub = subs.iterator();
                while (sub.hasNext()) {
                        temp = sub.next();
                        if (conn.condmet(conn.getConditions(temp))) {
                                conn.add(add.getID(), temp);
                        }
                }
        }
        catch(Exception ex){
                System.out.println("processerror (adverdisement): "+ex);
        }
        return lostsubs;
}
```

The last option to call the `process` method is this one:

```
public Vector<Integer> process(int qosowner, String qostype, int value) {
        Vector<Integer> lostsubs = new Vector<Integer>();
        Vector<Subscription> subs = new Vector<Subscription>();
        Subscription temp;
        try {
                conn.upQoS(qosowner, qostype, value);
                String addtype = conn.getAddtype(qosowner);
                lostsubs = conn.getUnservedSubs(addtype);
                subs = conn.getSubscriptions(qosowner, qostype);
                Iterator<Subscription> sub = subs.iterator();
                while (sub.hasNext()) {
                        temp = sub.next();
                        conn.remDelSub(temp.getID());
                        this.map(temp);
```

```
                }
        } catch (Exception ex) {
                System.out.println("process error (QoS)" + ex);
        }
        return lostsubs;
}
```

It is designed to update or create new quality of service entries in the database and returns a list of currently unserved subscribers.

The next two methods process the termination of `subscription` or `advertisement` objects.

```
public Vector<Integer> terminateAdd(int id){
        Vector<Integer> lostsubs = new Vector<Integer>();
        Vector<Subscription> subs = new Vector<Subscription>();
        Subscription temp;
        try{
                String type = conn.remAd(id);
                subs = conn.getSubscriptions(type);
                Iterator<Subscription> sub = subs.iterator();
                while(sub.hasNext()){
                        temp=sub.next();
                        if (conn.condmet(temp.getCondition())){
                                this.map(temp);
                        }
                }
                lostsubs=conn.getUnservedSubs(type);
        }
        catch(Exception ex){
                System.out.println("termination error (advertisement)" + ex);
        }
        return lostsubs;
}

public void terminateSub(int id){
        try{
                conn.remSubscription(id);
        }
        catch(Exception ex){
                System.out.println("termination error (subscription)" + ex);
        }
}
```

While the `terminateSub` method just removes the `subscription` from the Database, the `terminateAdd` removes the `advertisement` given by the id used to call the method and returns a list of now or still unserverd subscriptions.

The last two methods in this class are designed to process the allocation of the published events to subscribers.

```
private void map(Subscription sub) {
        int prov;
        try {
                if (sub.getRule() != null) {
                        prov = this.rulecheck(sub.getRule(), sub.getType());
                        if (prov != -1) {
                                conn.add(prov, sub.getID());
                        }
                        else {
                                prov = conn.getProvider(sub.getType());
                                conn.add(prov, sub.getID());
                        }
                }
                else {
                        prov = conn.getProvider(sub.getType());
                        conn.add(prov, sub.getID());
                }
        } catch (Exception ex) {
                System.out.println("mapping error: " + ex);
```

```
        }
}
```

This method checks if the `subscription` handed down has any rules and if so, uses the `rulecheck` method to evaluate these. Else the method just gets the first hit on the wanted event type and links the `subscription` to it.

```
private int rulecheck(Vector<Rule> ruleset, String type) {
        Iterator<Rule> rules = ruleset.iterator();
        Vector<Integer> v1 = new Vector<Integer>();
        Vector<Integer> v2 = new Vector<Integer>();
        Rule temprule;
        try {
                switch (ruleset.firstElement().getType()) {
                        case '>':
                                v1 = conn.getMaxQoS(ruleset.firstElement().
                                     getQoStype(), type);
        break;
                        case '<':
                                v1 = conn.getMinQoS(ruleset.firstElement().
                                     getQoStype(), type);
                                break;
                        case '=':
                                v1 = conn.getExactQoS(ruleset.firstElement().
                                     getQoStype(), type, ruleset.firstElement().
                                     getQosval());
                }
                while (rules.hasNext()) {
                        temprule = rules.next();
                        if (temprule.getType() == '>') {
                        v2 = conn.getMaxQoS(temprule.getQoStype(), type);
                        }
                        if (temprule.getType() == '<') {
                                v2 = conn.getMinQoS(temprule.getQoStype(), type)
                                     ;
                        }
                        if (temprule.getType() == '=') {
                                v2 = conn.getExactQoS(temprule.getQoStype(),
                                     type, temprule.getQosval());
                        }
                        if (v1.isEmpty()) {
                                v1 = v2;
                        }
                        v2.retainAll(v1);
                        if (!v2.isEmpty()) {
                                v1.retainAll(v2);
                        }
                }
        } catch (Exception ex) {
                System.out.println("rulecheckerror: " + ex);
        }
        if(v1.isEmpty()){
        return -1;
        }
        return v1.firstElement();
}
```

This methods returns for a given ruleset the `advertisement id` which fits most. Firstly the first rule is checked and if some advertisements pass the check, then the these are checked for the second rule and so on. In case no `advertisement` passes the check, all advertisements are considered to be valid.

Class Connector The connector class is the communication center of the broker, it starts listener, reader, sender and watcher threads to keep in touch with all participants.

```
public class Connector implements Runnable{
        private int port;
```

```
        private Dbcon con;
        private HashMap<Integer, Eventstack> eventstacks=new HashMap<Integer,
             Eventstack>();
        private HashMap<Integer, Thread> threads = new HashMap<Integer, Thread
             >();

        public Connector(int port, Dbcon con){
                this.port=port;
                this.con=con;
        }
```

The constructor sets the port on which it is accessible and the Database which the reasoner it will call use on the handed down values. The class will be started in a new thread, which will invoke this method:

```
public void listen() {
        Reasoner reasoner = new Reasoner(con);
        try {
                ServerSocket srvr = new ServerSocket(this.port);
                System.out.println("Connector is listening.");
                while (true) {
                        Socket skt = srvr.accept();
                        Reader reader = new Reader(skt, reasoner, this.con, this
                             .eventstacks, this.threads);
                        Thread readerThread = new Thread(reader);
                        readerThread.start();
                }
        } catch (Exception e) {
                System.out.println("connector Error:" + e);
        }
}
```

The method basically opens a port and waits for services to connect to it, if a service does so, a new reader thread is started. It also hands down the hash map of threads(where each new started thread is noted) and the hash map of event stacks (where the different event stacks are stored). The methods then waits for the next service to connect.

Class Reader A reader threads firstly initializes the handed down reasoner, event stacks and thread hash map, the socket to listen to and the database connection.

```
class Reader implements Runnable {
    private Socket skt;
    private Reasoner reasoner;
    private Dbcon con;
    private HashMap<Integer, Eventstack> eventstacks=new HashMap<Integer,
         Eventstack>();
    private HashMap<Integer, Thread> threads = new HashMap<Integer, Thread>();

    Reader(Socket skt, Reasoner reasoner, Dbcon con ,HashMap<Integer, Eventstack
         > stacks, HashMap<Integer, Thread> threads){
        this.con=con;
        this.eventstacks=stacks;
        this.threads=threads;
        this.skt=skt;
        this.reasoner = reasoner;
    }
```

The reasoner thread now invokes the `read()` method, which tries to read the incoming object:

```
public void read(){
        try{
        ObjectInputStream inObjs = new ObjectInputStream(skt.getInputStream());
        Object obj = inObjs.readObject();
```

Depending on the object read, the next actions are selected:

```
if (obj instanceof QoSlist) {
        System.out.println("QoSlist received!");
        QoSlist qos = (QoSlist) obj;
        Iterator<String> types = qos.getTypes().iterator();
        Iterator<Integer> values = qos.getValues().iterator();
        while (types.hasNext()) {
                reasoner.process(qos.getID(), types.next(), values.next());
        }
}
if (obj instanceof Subscription) {
        Subscription sub = (Subscription) obj;
        reasoner.process(sub);
        Eventstack events = new Eventstack(sub.getID());
        this.eventstacks.put(events.getID(), events);
        Sender sender = new Sender(sub.getID(), events, this.reasoner);
        Thread senderThread = new Thread(sender);
        this.threads.put(sub.getID(), senderThread);
        senderThread.start();
}
if (obj instanceof Advertisement) {
        Advertisement add = (Advertisement) obj;
        Vector<Integer> lostsubs = reasoner.process(add);
        System.out.println("Advertisement received from: "+ add.getID());
        Watcher watcher = new Watcher(add.getID(), add.getQoS(), this.reasoner);
        Thread watcherThread = new Thread(watcher);
        this.threads.put(add.getID(), watcherThread);
        watcherThread.start();
        System.out.println("");
}
if (obj instanceof Event) {
        Event ev = (Event) obj;
        int s;
        Vector<Integer> subs = con.getDel(ev.getID());
        Iterator<Integer> sub = subs.iterator();
        System.out.println("Event received from: !" + ev.getID());
        while (sub.hasNext()) {
                s = sub.next();
                this.eventstacks.get(s).add(ev);
                this.threads.get(s).interrupt();
        }
}
if (obj instanceof Terminated) {
        Terminated term = (Terminated) obj;
        if (term.getType().equals("adverdisement")) {
                int s;
                this.threads.get(term.getID()).stop();
                this.threads.remove(term.getID());
                Vector<Integer> lostsubs = reasoner.terminateAdd(term.getID());
                Iterator<Integer> sub = lostsubs.iterator();
                Event<Terminated> evterm = new Event<Terminated>(term, -1);
                while (sub.hasNext()) {
                        s = sub.next();
                        this.threads.get(s).interrupt();
                        this.eventstacks.get(sub.next()).add(evterm);
                }
        }
        if (term.getType().equals("subscription")) {
                reasoner.terminateSub(term.getID());
                threads.get(term.getID()).stop();
                threads.remove(term.getID());
        }
}
inObjs.close();
skt.close();
```

A `QoSlist` object will be processed with the reasoner. A `subscription` will be processed as well. There is also a new `Eventstack` be generated and a sender thread started (thread will be added to the thread hash map). The reasoner, the `subscriber id` and `Eventstack` will be given to the sender thread.

In case the read object is a `advertisement`, it will be processed with the reasoner. After this, a watcher thread for observation is started.

After the detection of a `Event` object, the receivers stored in the `delivery table` (see Subsection 3.2) will have that `Event` put on there `Eventstack`. Also there sender thread will be interrupted to get it. A `Terminated` object will be checked if its type equals `subscription` or `advertisement` and then process the termination of the found type through the reasoner. Also all threads related to terminated objects will be stopped.

Class Sender Sender threads are used for delivering `Events` to `subscribers`.

```
class Sender implements Runnable {
        private int port;
        private Eventstack evstack;
        private Reasoner reas;

        Sender(int port, Eventstack evstack, Reasoner reas) {
                this.port = port;
                this.evstack = evstack;
                this.reas=reas;
    }
```

The sender thread checks on the `Eventstack` if interrupted and will remove found `Events` from the stack and send them to the subscriber.

```
public void run() {
 Event event = null;
 int i = 0;
 while (true) {
   try {
     int client = this.port + 1200;
     while (true) {
       try {
         Thread.sleep(5000);
       }
       catch (InterruptedException e) {
         Socket skt = new Socket(InetAddress.getLocalHost(), client);
         System.out.println("Channel to Client " + client + " opened.");
         ObjectOutputStream outObjs = new ObjectOutputStream(skt.getOutputStream
              ());
         event = this.evstack.getEvent();
         while (event != null) {
           System.out.println("Event send: "+i);
           i++;
           outObjs.writeObject(event);
           event = this.evstack.getEvent();
          }
          outObjs.close();
          skt.close();
        }
      }
```

Class Watcher The watcher threads represent the observer (see Subsection 3.1).

```
class Watcher implements Runnable {
        private int port;
        String QoS;
        private Reasoner reas;

        Watcher(int ID, String Qos, Reasoner reas) {
                this.port = ID+1200;
                this.QoS=new String(Qos);
                this.reas=reas;
    }
```

It is generated with the `advertisement id`, the quality of service pass phrase and the reasoner to process termination issues. On invocation the thread simply queries the publisher of the `advertisement` on set times about his quality of service values. In case the pubslisher is not available, the thread will terminate itself and notify the reasoner about the termination:

```
public void run() {
  while (true) {
    try {
      while (true) {
        Socket skt = new Socket(InetAddress.getLocalHost(), port);
        ObjectOutputStream outObjs = new ObjectOutputStream(skt.getOutputStream
            ());
          try {
            Thread.sleep(5000);
            Thread.yield();
            outObjs.writeObject(this.QoS);
            System.out.println("QoSrequest send!");
          }
          catch (InterruptedException e) {
          }
        outObjs.close();
        skt.close();
       }
     }
     catch (Exception e) {
       System.out.println("Watcher error: " + e);
       reas.terminateAdd(this.port-1200);
     }
```

Class Eventstack The `Eventstack` class is a synchronized source for `Events` for different threads.

```
class Eventstack {
  private Vector<Event> events;
  private int ID;

  Eventstack(int ID) {
    this.events = new Vector<Event>();
    this.ID=ID;
  }
```

There is a method to add `Event` objects to the vector and a method to get the general `ID` of the events.

```
public synchronized void add(Event ev){
  this.events.add(ev);
}
public synchronized int getID(){
  return this.ID;
}
```

The method `getEvent()` returns the last element on the vector and removes it. If the vector was empty, the method returns null.

```
public synchronized Event getEvent() {
  try {
    Event el=this.events.lastElement();
    this.events.remove(el);
    return el;
  }
  catch (Exception e) {
    return null;
  }
}
```

4 Summary and Future Work

This last section summarizes the work and gives a brief overview on future work.

4.1 Summary

In this project, we implemented a broker software prototype and two services. We evaluated three different approaches to implement the event-based service collaboration. We implemented the included broker architecture in our prototype. This architecture meets the requirements of anonymity, openness, use of location-bound services and quality of service driven service subscription. The mobility requirement is currently, since this prototype was developed on a standard desktop computer. The p2p requirement is also not implemented, but is part of the broker to broker communication. Formal proof of anonymity and privacy are still outstanding.

4.2 Future Work

Architecture The implementation covers the scenarios shown in Figures 3 and 8. However, certain aspects had to be left open. For example, aspects of distribution were not explored. We also need a way for brokers to find other brokers on *ad hoc* basis, or via the web. The next step would be to develop more types of rules and conditions, and to support more brokers. Some brokers could advertise their currently available event types and others may not.

Currently there does not exist any method to restrict the subscribers of events. We suggest a role based model like RBAC, it could be featured in the handshake procedure of services with the broker.

We did not discuss in detail how services would know about event type identifiers. Alternatives may need to be explored here.

Implementation The implementation is a prototypical one. Some of the tasks still to be implemented are:

- Handshake procedure between services and broker
- Support of more condition/rule types
- Support of broker to broker communication
- Support of p2p communication between brokers

The broker to broker communication is the most important task to realize, since it is needed for the architecture to be able to connect services on different devices or locations. The p2p communication aspect could be included in this step.

References

[1] L. Eschner. Design and formal model of an event-driven and service-oriented architecture for a mobile tourist information system. Master's thesis, Freie Universität Berlin, July 2008.

[2] P. Guerrero, K. Sachs, M. Cilia, C. Bornhövd, and A. Buchmann. Pushing business data processing towards the periphery. In *23rd ICDE*, 2007.

[3] A. Hinze, Y. Michel, and L. Eschner. Event-based communication for location-based service collaboration. In *Twentieth Australasian Database Conference (ADC 2009)*, 2009.

[4] Y. Michel. Location-aware caching in mobile environments. Master's thesis, Freie Universität Berlin, June 2006.

[5] A. Michlmayr, P. Leitner, F. Rosenberg, and S. Dustdar. *Principles and Applications of Distributed Event-based Systems*, chapter Event processing in web service runtime environments. IGI Global, 2010.

[6] G. D. M. Serugendo, J. Fitzgerald, A. Romanovsky, and N. Guelfi. A metadata-based architectural model for dynamically resilient systems. *Proceedings of the 2007 ACM symposium on Applied computing*, ACM:566–572, 2007.