Event-based communication for location-based service collaboration

Annika Hinze[1] **Yann Michel**[2] **Lisa Eschner**[2]

[1] University of Waikato, New Zealand
Email: `hinze@cs.waikato.ac.nz`

[2] Freie Universitaet Berlin, Germany
Email: {`ymichel,eschner`}`@mi.fu-berlin.de`

Abstract

Location-based context-aware services for mobile users need to collaborate in disparate networks. Services come and go as the user moves and no central repository is available. The user's personal information and service usage statistics need to be protected. To support service collaboration we propose a service infrastructure that relies on an event-based service-oriented architecture. We implemented a basic version of the architecture and used it for a tourism information system. An advanced version has been modelled using formal methods to evaluate privacy aspects. This paper reports about both architectures and our experiences of their application to tourism-related services.

1 Introduction

Imagine a museum guide that provides rich information about its exhibits on your mobile phone, or a car navigation system that shows the way to the nearest specialist depending on your health status. Mobile users of such services receive information tailored to their situation and preferences on their own portable devices. Context-aware mobile services use information about a user's location to deliver customized information [3]. Many of these services will only be offered in a restricted territory, e.g., close to a hospital or in a museum. For example, Figure 1 shows the areas of availability of two services A and B. As the user moves, services appear and disappear (at the four points marked) and no common network may be available.

To reach their full potential, different services need to collaborate, such as the tour guides, route planners, and health monitors in our example. For a seamless user experience, services must collaborate automatically and share user-related data as necessary. Because services are not known in advance, their collaboration has to be initiated and sustained in an ad hoc manner. At the same time, services and their communication infrastructure should protect a user's personal information, thus, collaboration must be anonymous. It must also be immediate [12].

Service collaboration raises several issues: for a seamless experience, configuration should not be required and services should be able to deal with collaboration partners that appear and vanish as the user moves around. We address these issues by developing a method for independent services to automatically collaborate in a mobile environment without knowing about each other.

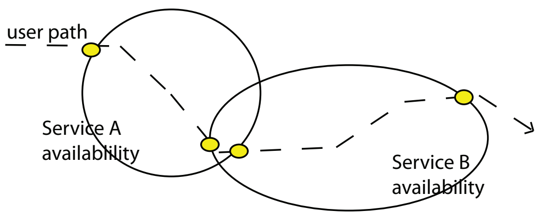

Figure 1: Mobile user and service availability.

We developed this research in the context of the tourism application TIP. The Tourist Information Provider (TIP) is a mobile tourist information system that provides information about sights to travellers based on context, i.e., their personal preferences and their location. The personal preferences are defined in profiles and describe a user's likes and dislikes. Different from other context-aware systems, the TIP system supports several services, e.g., information service [14], recommendations [13] and a map service [16]. In [12] we highlighted the need for redesigning the mobile context-aware TIP system into a service-oriented architecture.

The remainder of this paper is structured as follows: Section 2 discusses related approaches to service collaboration. Section 3 introduces our event-based SOA, Section 4 gives details of service management and collaboration. Section 5 discusses our basic architecture. Section 6 introduces advanced concepts, which are then discussed in Section 7. Section 8 briefly describes the two prototypes; the paper closes with a summary.

2 Related service collaboration approaches

Currently, location-based systems are available only separately. They do not collaborate and the user often has to obtain specialised hardware and software [1][2][3]. No infrastructure exists to support such collaboration. Interesting related issues in location-based systems have been addressed but so far do not consider service collaboration. The solution suggested in [22] provides a subscriber-based means to access location-restricted services in a mobile environment (by using a UDDI channel instead of a repository). Similarly, cloaking has been used to retain privacy

[1]Zingo Taxi, location aware taxi hailing in London, online information at `http://www.zingotaxi.co.uk/` , see also online article at `http://www.jacobsen.no/anders/blog/archives/2003/03/19/locationaware_mobile_services_zingo_taxis.html`

[2]TomTom, personal navigation systems, online information at `http://www.tomtom.com/`

[3]Geominder, location-based personal reminder, online information at `http://www.ludimate.com/products/geominder/`

when accessing location-based services (using a centralised third party[10] or a P2P approach [4]).

Previously introduced techniques for service collaboration use a service repository to search for collaboration partners [6]. This traditional Service-Oriented Architecture (SOA) is unfit for collaboration of location-based services in mobile environments because it assumes that services are openly known and that they directly enter bilateral contracts for longer business relations. These assumptions do not hold in a changing mobile environment with disconnected network patches and locally offered services. The original Web Services Architecture from the W3C [1] is even more restrictive and not applicable for mobile environments.

SOA as a more general design principle has been applied to mobile services. However, current solutions do not cater for the complex situation of collaborating location-aware services: architectures for mobile telephones [21] connect static services on mobile devices and require services to be explicitly known. Hodes et al. suggest an architecture for composable and location-based services in an ad hoc network environment [15], in which they use a meta-service index which shares information about the available services in a given network cell. The index service is an equivalent to the SOA service repository and the same limitations apply. Similarly, Le Sommer suggests a service register for remote services using a publish/subscribe mechanism [17]. The ensuing service collaboration is not designed to be anonymous. We observe that current architectures for service collaboration are either not suitable for changing mobile environments or do not support collaboration between anonymous services. Middle-ware solutions for ad hoc environments [23, 18, 20, 11] do not support anonymous service collaboration either.

On the other hand, anonymous and immediate communication is supported in publish/subscribe systems [8]. The aim of these systems is to disseminate published information to a set of subscribers. Each subscriber receives only the information that matches their subscription; communication via the network is uni-directional and anonymous. Publish/subscribe systems use event-based communication that is initiated by the publisher; the event information is filtered and routed through the network. The focus of publish/subscribe developments for mobile and ad hoc environments [9, 5] is on efficient routing strategies in a changing and possibly disconnecting network. However, efficient routing is not required for localised services because the user is in close proximity to the service provider (e.g. in the museum) and thus the service network generally requires only single hops. The publish/subscribe paradigm does not include a service concept nor does it support ongoing collaboration between publishers or subscribers. We explore event-based communication as used in publish/subscribe systems as a means for anonymous collaboration in a service-oriented architecture for mobile location-based services.

3 Basic Architecture

The new TIP 3 architecture introduces an event-based middle-ware, with which every local service interacts. The component representing this middleware layer is called the broker. For client/server interaction, co-operating services exchange their information via their local brokers, i.e., every communication from a client to a server service or vice versa is only handled by the brokers and is transparent to the services. On the server-side, the TIP server sends and receives all information via the broker. The server-sided services, e.g., the information service and recommendation service, can also use a database as their information back-end. The services residing on the client-side may interact directly with the user, e.g., a user clicks on the map of the map service to submit a position to the server. In addition, the client system can submit information autonomously, e.g., the location service automatically submits a user's changed position by using the data of a locally attached GPS receiver. For technical reasons, both sides use a wrapper process that is used to effectively start and stop all running processes, i.e., the broker and services. For the server-side, this results in a TIP server process. On the client-side, this process is called the TIP client.

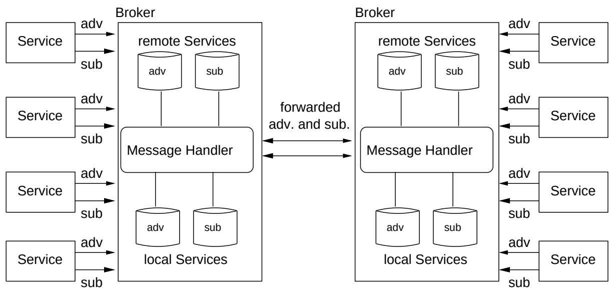

Figure 2: Local and forwarded service advertisement- (adv) and subscription- (sub) handling by the message handlers TIP 3.

The original TIP 1 [14] and TIP 2 [12] systems are based on client/server interaction. The TIP 3 system is planned to also support peer-to-peer and client-to-multi-server connections. In the peer-to-peer approach, servers can be seen as special peers with more available data and unlimited bandwidth or power.

As each service directly communicates with a broker only, the broker is responsible for routing the incoming information to the appropriate services or other brokers. When a service is started, it connects to a broker, advertises the provided information and subscribes for other information. The broker maintains this information in a registry. When a broker connects to another broker, it exchanges its registries' information with the remote broker. This principle is illustrated in Figure 2.

3.1 Event System

A generic and platform independent format for information exchange is required. Therefore, we use the XML-based *simple object access protocol* (SOAP). Slominski et al. [19] propose an event system architecture for their grid project. They use an event-based communication for submitting status information and jobs between the participating grid nodes. Some of their requirements also apply to the TIP system:

- Language and platform independence: The components used for the TIP system should neither depend on a special programming language nor on a special platform.
- Extensibility: New services should be easily added. In addition, existing services should be easily extended.
- Lightweight Publishers: Standard libraries should provide basic functionality. This is a very important fact especially on small devices such as the supposed mobile clients of the TIP system.

Figure 3 shows the basic SOAP message format for the TIP system. Any SOAP message consists of a

```
<?xml version='1.0' ?>
<e:Envelope xmlns:e="http://www.w3.org/2003/05/soap-envelope">
 <e:Header>
  <m:event xmlns:m="http://isdb.cs.waikato.ac.nz/tip/event"
          env:mustUnderstand="true">
   <m:id>{id}</m:id>
   <m:type cacheable="{boolean}" forwarded="{boolean}"
          prefetched="{boolean}">
   {type}
   </m:type>
   <m:dateAndTime>{dateAndTime}</m:dateAndTime>
  </m:event>
 </e:Header>
 <e:Body>
        [...]
</e:Body>
</e:Envelope>
```

Figure 3: Basic TIP SOAP message format for the TIP system.

root element which is the `Envelope`. The `Envelope` contains two elements: a `Header` and a `Body` element. The `Header` is comparable to a common letter head, i.e., it describes general parameters of the submitted information. For our purposes, we use the following parameters:

- `id`: a unique identifier (event number),
- `type`: the type of this event (descriptor for the contained body information),
- `dateAndTime`: the date and time this event was created.

The `type` element is also used for routing the incoming messages properly to the subscribed services without being forced to parse the whole message but just the `header` element, i.e., the `body` element is skipped for this purpose. In addition, several attributes are available for internal handling of the message:

- `cacheable`: This attribute indicates if a message is cacheable (*true*) or not (*false*).
- `forwarded`: This attribute indicates if a message was forwarded (*true*), i.e., it was received from a broker, or locally (*false*), i.e., by a service.
- `prefetched`: This attribute indicates if the message was triggered by a pre-fetching-service (*true*) or if it is an ordinary message (*false*).

The first flag (`cacheable`) is used for deciding if this message can be stored inside a cache or not. The other two attributes, `forwarded` and `pre-fetched`, affect the message routing. If a message was forwarded from a foreign broker, any reply should be sent to this broker only and not to any other services or brokers. If a message has the `pre-fetched` flag set to *true*, a reply should be sent but the reply is then not forwarded to the client's display. The reply is used only for storing the information locally prior to any user request. The `Body` element is the main container for the type specific information that is transported in any message. Therefore its structure is not defined in general but can be freely designed, depending on the type's information.

3.2 Caching

Figure 4 shows the broker of a TIP 3 client. The cache resides within the broker or can be accessed by the broker only. The use of a cache is therefore fully transparent to any service. Inside the broker, the *message handler* is responsible for efficient message

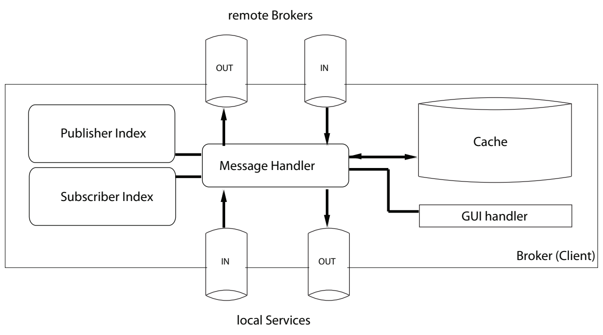

Figure 4: Internal message handling of a client-sided event-broker (basic architecture).

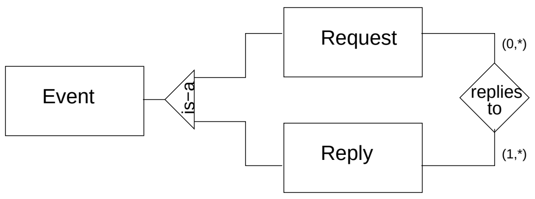

Figure 5: The entity relationship model (ERM) of the cached objects, i.e., events.

handling. The aim of the message handler is to reduce the external network traffic, i.e., the traffic to remote brokers. The local traffic, i.e., the traffic of services with the broker is not considered here. Messages are handled differently depending on their direction.

The algorithm for handling the outgoing messages to remote brokers is shown in Algorithm 1. When a message handler decides to send a message to a remote broker, it first checks if a similar message was sent before. If that is the case, it sends the still cached replies back to the local services. If no similar message could be found in the cache or cached replies are missing, the message is forwarded.

Any message that is retrieved from a remote broker is handled according to the algorithm shown in Algorithm 2. If the incoming message was a reply to a previous request (message) and is cacheable, it is stored in the cache and then forwarded to the local services.

The relationship of the cached messages, i.e., the cached events, is shown in Figure 5. An event can be either a request or a reply. A request can have zero or many replies and a reply has at least one parent it relates to but can also have more. This allows for effective re-use of cached replies that refer to more than one request. Assuming that a reference to the request object uses less memory then a reply object, this results in less space required for cache memory.

Algorithm 1 Message handling of outgoing messages.

```
1: if message is cached then
2:    if replies available then
3:       return replies;
4:    end if
5: else
6:    send message;
7: end if
```

Algorithm 2 Message handling of incoming messages.

```
1: if message is a reply then
2:    if cacheable then
3:       store message in cache;
4:    end if
5: else
6:    forward message;
7: end if
```

4 Basic Architecture – Services

In [12] we suggested the use of a *service-oriented architecture* (SOA) for TIP 3. All components, i.e., the brokers and services, are loosely coupled only. That means that every component acts autonomously and communicates with any other by using a standard protocol, preferably SOAP. As we have already seen, services interact with brokers only. The brokers forward information to other brokers or services that subscribed for a certain event type. This section describes how a service is designed, how the registration process works and how existing services were migrated.

4.1 Service Architecture

A service is a modular component that is loosely coupled to a broker. It runs as a separate process and communicates with a broker via a network connection. After a service is started, it listens on a specified port for incoming events. All outgoing events are sent to a broker whose hostname and port number have to be known by the service. A service can register and un-register itself from a broker. That means it advertises events and subscribes for events that it is interested in. Which events a certain service subscribes to or which it is going to provide depends on the type of the service, i.e, the service's purposes. Figure 6 shows this principle: A service receives an event that it has previously subscribed to. This event is then processed and the service then provides an advertised event to the broker. Services can also work differently, i.e., a service may only provide events but consume none. An example for such a service is a location service that submits a user's location at regular intervals. Therefore, it needs no events to subscribe to but only produces them. A service can also be a sink, i.e., the service only consumes events but does not produce any. An example of such a service may be a history service that only tracks the user's locations to log them into a database. Another type of service can be a split service, e.g., one part of the service is running on the client and the other part on the server-side. An example of such a service is the map service. The map is shown on the client-side but must be retrieved from the server prior to that. A running TIP system has a hybrid mixture of all possible combinations of service types introduced.

All services have the same external interfaces and therefore look identical from the exterior, i.e., broker's view. Services are also capable of receiving any event but will only process some of them, i.e., the ones they subscribed to. To reduce unnecessary communication and workload, the broker should take care that only events that it subscribed to are forwarded to each service.

4.2 Service Management

All information is sent through event messages. The subscription and advertisement of events is wrapped into events as well. We distinguish between system

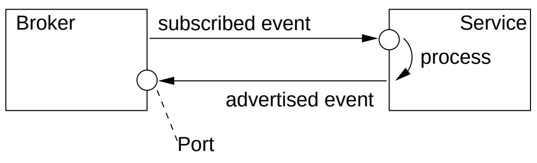

Figure 6: Event exchange between the broker and a service processing them.

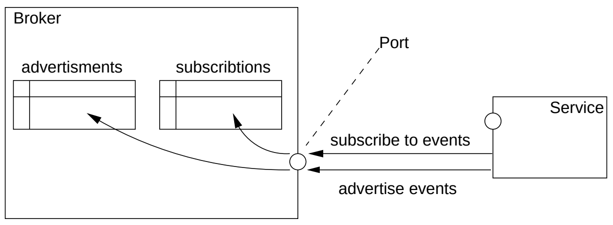

Figure 7: Event subscription and advertisement of a service to a broker.

events, i.e., all events that have an administrative purpose, and service events, i.e., events that are processed by the services depending on their purpose. As this section is describing the service management, we are referring to system events throughout the rest of this section. The service events are described in the next section.

Figure 7 shows a service that subscribes to and advertises events. Referring to Figure 3, the corresponding event would have the `type` element set to *Advertisement*. The `Body` would contain the description of the advertised event. If our service was, for example, a location service, it would provide *Location* events. The corresponding Body element provides the name of the event enclosed by the <m:name/>-tags. The subscription events would look similar. The `type` element would simply be set to *Subscription*.

When the broker receives advertisements and subscriptions, the information about the event type and the service that generated these events is stored in an internal registry. This registry is later used for efficient event routing as proposed above, i.e., events are directly forwarded to services that requested them, and not broadcasted.

To draw a bigger picture of the service registration process, we illustrate the sequence by using the UML sequence diagram shown in Figure 8. Assuming that the TIP server is already running and has registered its services, we start the description on the mobile user's site. A TIP user starts its TIP client residing on their mobile device. At first the client software starts the local broker as a core component that is run for every TIP client. Then the locally avail-

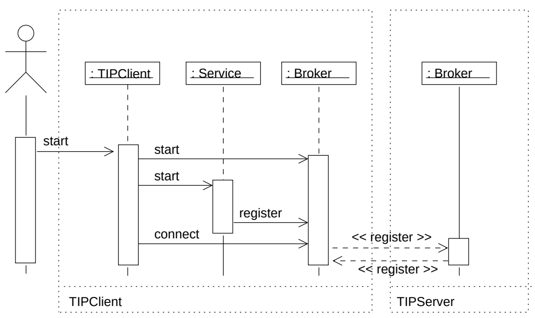

Figure 8: The sequence of the service and broker registration-procedure of the TIP 3 system.

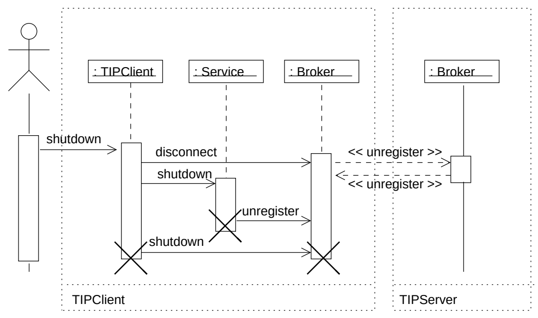

Figure 9: The sequence of the service and broker un-registration–procedure of the TIP 3 system.

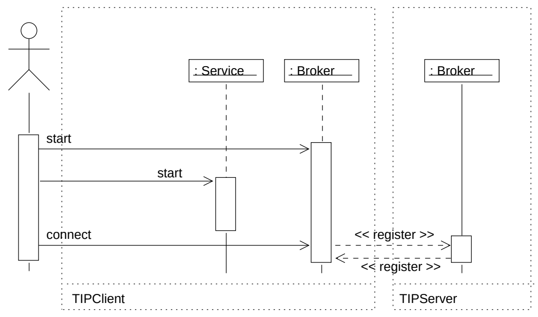

Figure 10: The idealized re-start sequence of the TIP 3 client.

able services are started and told to register with the previously started local broker. Once every service is started, the broker is forced to register itself to a remote server-broker. By registering to it, all local subscriptions and advertisements are forwarded. In addition, the remote server-broker forwards the subscriptions and advertisements of its services. Once this initialization procedure is finished, the user can start working with the TIP system. If the user wants to shut down the TIP client, an un-registration procedure is started. This procedure is shown in Figure 9 and symbolizes the reverse registration procedure: First the client-broker un-registers from the remote server-broker, i.e., the locally advertised events are revoked and the local subscriptions are cancelled. The remote server-broker does this as well for its local services. Then every local client-service is shut down by un-registering from the local broker first and then terminating. When all local services are shut down, the broker itself is shut down and terminated. Then the TIP client terminates.

An idealized procedure for a client startup is shown in Figure 10. Here the user has previously started all services plus the local broker but decided to shut down the client software. Now the client should be re-started. Again, the broker is started first. After that, all services are started, but this time, the local registration procedure is skipped because the service setup has not changed, i.e., there are neither new services nor are pre-known services missing. Therefore the known registry information is re-used. Only the broker is re-registering its local services to the remote site. Similarly, the shutdown procedure is only un-registering the broker from the remote site. Then, the local processes are simply shut down, i.e., the broker and every service is stopped. This means that the registry's entries are kept until the client services are restarted.

In addition, Figure 10 shows a sequence where only the services and the brokers are involved but not the TIP client process that was used above. This is based on the assumption that in this idealized scenario, all services are started independently, whereas in the previous example all services were started together with the broker by using a wrapper process.

4.3 Integration of Legacy Services

In TIP 2, a service is a monolithic software component that combines the data, the logic and the presentation of the provided data. For TIP 3, this single component has to become uncoupled. We use the model-view-controller (MVC) design pattern. The *model*, i.e., the state of the service, might be, for example, represented by the database back-end. Therefore it resides on the server-side in the TIP 3 system. The *view* visualizes the data available to the user. It is the representation layer and therefore part of the TIP client. The *controller*, i.e., the linkage between the two other parts, is represented by the program logic and the event layer. As both the client and the server use this event-layer for their information exchange, this component is found on the TIP server as well as the TIP client. We illustrate how selected services are migrated:

Location Service: This service is a *provide-only* service, i.e., it provides *Location* events but does not consume any other events. Therefore, this service resides on the client-side only.

Information Service: This service cannot be directly converted into a TIP 3 service. The service is split into several smaller ones. A first service uses the location-information, the user's profile and history and the available events to suggest the touristic items. Instead of sending all item information to the client, it would send the item type and identifier to the client. The client then requests the items if the complete item information is not yet locally available. This would ease the reuse of fine grained information, i.e., if an item was part of two consecutive *Information* events, it could be re-used without being requested again. In addition, this division keeps the messages small. In addition, a new service has to be run for every item type that is suggested within its provided events. For the client-side, a *Display* service displays the provided information to the user and requests the item information prior to displaying it.

Recommendation Service: Similar to the Information Service, this service has to be split into several smaller services. The service also subscribes to location information. The provided *Recommendation* event again is only a description of the item types and their recommended identifiers. The item information can be obtained by the same services that have to be introduced for the Information service and provide the item information only. This has to be handled by the introduced *Display*-service.

Map Service: This service consists of a client and a server part in the TIP 3 system. The server part provides maps and subscribes for *MapRequest* events. Based on these events, it provides maps, i.e., *Map* events. The client part displays the map on the mobile device. In addition, it displays the items obtained by the Information and Recommendation Service on these maps. When a user leaves the scope of the locally available map, a new one for the current location is requested. Therefore the client-sided map service submits a *MapRequest* event.

5 Discussion of the Basic Architecture

The basic architecture and the event-based infrastructure well support ad-hoc service collaboration. One example is the typical scenario of the GPS (location service) publishing the user's location to the broker, the information service subscribes to the location events to deliver touristic information. The map service also subscribes to location events to show the user's location on the map. It also offers locations as the user may select points on the map that are then reported as new location events.

This basic scenario can be extended to deal with a more complex situation: The user enters a museum where location information becomes available by reading RFID tags. At the same time, the GPS service is only available with reduced quality. A new information service is available within the museum and a new map (for the museum) needs to be added to the existing map service.

This situation introduces a number of challenges: alternative services with differing quality, parallel services and exclusive services; and the general question of quality of service. We will focus on two questions in particular: (1) How to ensure quality of the map service when the user location data is no longer available? (2) How does the system decide which location service to use?

For the first question: in a traditional request/response interaction the map would be aware of missing location data. This is not the case in event-based interaction. The assumption in publish/subscribe systems is that if no event data is available, no event happened. This assumption does not hold in a mobile context.

For the second question: The concept of parallel services and exclusive services has not been used in this architecture so far. In general, the concept of service types has not been described.

Both questions were approached using formal modelling, as the mere implementation of more services and coding of a particular solution was not deemed to be sufficient for a general collaboration infrastructure that aims to be open to third party services.

6 Advanced Concepts

This section introduces advanced concepts for our event-based SOA that have been verified and evaluated using formal modelling. TIP 3 services are now classified into *service categories*. Service categories are a new concept to TIP. A service category describes the functionality a service provides. A service category groups services, so that services with similar functionality belong to the same service category. Different information and recommendation services belong to the informative service category whereas map services belong to the map service category. A good example is the map service category. Two map services A and B both offer a user interface where the user sees their current location on a map. Map service A offers basic features: the map displays sights; the user can zoom in and out from the map, i.e., change the map's scale. Map service B offers the same features as map service A and some additional features: the user can also select a new location, e.g., by dragging the map; the user can select a start and an end point for a route planner that map service B co-operates with. Both map services belong to the same service category.

Similarly, TIP events are classified into *event types* as described above. An information service subscribes to location events. Whenever it has processed a location event it publishes an information event. Event types are classified into *event categories*. When the information service publishes a location event, this location event and the information events belong to the event category information events, enabling subscribers to differentiate between location events from

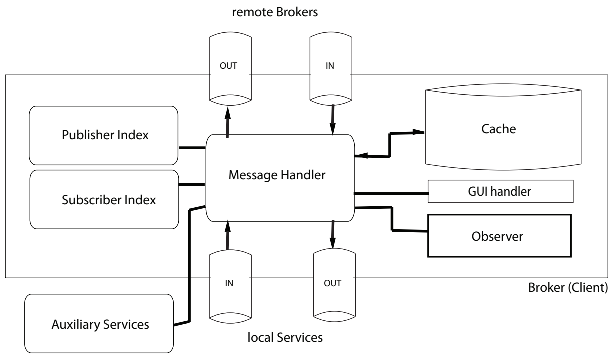

Figure 11: Internal message handling of an extended event-broker.

different publishers and to treat them accordingly. A location event belonging to the information event category is treated differently by the map service from a location event belonging to the location event category, for example.

Figure 11 shows the advanced TIP 3 client architecture. We start with a short discussion of a TIP 3 peer. We then discuss TIP 3 services and their attributes before we discuss the observer and the broker.

A TIP 3 service offers functionalities, e.g., sights on locations and information about the sights. A service may publish events and subscribe to events. Event publishers provide an advertisement. Event subscribers provide a set of functional conditions and subscription rules. Every service provides a service description.

A service does not locate co-operation partners. It simply is subscribed to the events needed to provide its functionality. Services are grouped into service categories. For example, services providing information on sights belong to the informative service category while services offering map tiles and services displaying the map tiles belong to the map service category.

A TIP service provides a service description, an advertisement, functional conditions and subscription rules.

The service description provides information about the service. It specifies what service category the service belongs to. It defines the event categories used by the service, as well as the event types it publishes. Furthermore it informs about the quality of the published data, the service's maximum latency and the service's maximum failure rate. The service description also informs about the owner of the service and provides other administrative information.

The service advertisement tells about the data published by the service. It defines the event and service category, the event type and the quality of data. A service may provide several advertisements, one for each event type that it publishes.

The service conditions specify a subscriber's functional pre-conditions. The service defines the event it requires to be able to supply its functionality, e.g., location events. It may further specify what kind of service should publish the data, i.e., the publisher's service category. The subscriber can also decide if the publisher should be a local service, i.e., a service that is located on the same device, or an external service. The information service, for example, would require location data from a location service. If the service's pre-conditions are not satisfied, the service cannot provide its functionality. A service condition is a tuple <data category, service category, local publisher, remote publisher>. The subscription rules de-fine these conditions in more detail – the required data format, quality of service and, if necessary, the publishing service, amongst others.

Rules enable the service to prioritise certain event types over others or to choose between several publishers. In a group, the rules are prioritised: a priority can be set on high, medium or low. Rules with high priorities are evaluated before those with lower priorities. If the rule with a higher priority has been evaluated successfully, i.e., the evaluation resulted in a subscription, rules that have lower priorities and belong to the same group are not evaluated. This gives the service the opportunity to choose between different event types from the same event category, or between different data qualities. Subscription rules are a tuple <priority, event type, event category, quality, exclusive subscription, service category of publisher, publisher, local, external, allowable latency, maximum failure rate>.

The event category is needed to select the subscribed event type if the same event type is offered by several services: location data may, for example, be published by the location service, and by the information service. A service that is only interested in the user's current location will subscribe to location data from the location category. Another service interested in location data from sights, would subscribe to location data from the information category.

When a service subscribes to events from only one publisher we call this an exclusive subscription. For example, the map service should only subscribe to location data from one location service, and not from several location services at the same time. The map service would then have to name its favourite publisher.

The service category of the publishing service can be named as well. The map service subscribes to location data both from the location service and the information service. The information service subscription is not exclusive, however. The map service then defines in a rule that it wants to subscribe to events that are published by services belonging to the category of information services.

Services can also specify if they want to subscribe to data generated locally, or if the data should be computed remote, e.g., on a client. This is needed for location subscriptions, amongst others. The allowable latency and the maximum failure rate specify features of the publishers, and prevent that a service subscribing to publishers that provide poor quality.

The observer evaluates the service conditions and rules. It monitors the connection to services or brokers, i.e., it monitors if services or brokers have been disconnected. In case of disconnection the observer removes the advertisements and subscriptions from the disconnected service or broker.

Although the observer may behave like an independent actor, it is located at the broker. The observer is called during the service startup routine. A service delivers its advertisements and subscription rules to the observer. The observer then selects some event types from the available types at the broker, i.e., from the event types other services have advertised, using those rules. When a newly registered publisher advertises its data, the subscriptions may be changed if needed or possible. When a publisher disconnects, the subscriptions are re-evaluated as well.

When a service wants to subscribe to data not available in the requested data format, the observer requests that the broker starts an auxiliary service that can convert the available data format into the requested.

The broker or event-manager provides a communication interface for local services and for other brokers. It connects local services with one another and connects to external brokers. The broker receives the events from publishers, filters them and forwards them to the respective subscribers. The broker starts auxiliary services if needed. The broker keeps track of which service publishes what data type, and which service subscribes to what data type. The auxiliary services convert from one data format to another. They are managed by the broker, i.e., if an auxiliary service is requested, the broker starts it.

The publisher and subscriber index are used by the broker to keep track of what service publishes which data, and who subscribes to which data. The subscriber index is accessed during the filtering process. The publisher index is accessed during the evaluation of rules and conditions. The TIP databases are typically located at the tip 3 server peer. They store geo-spatial data, information on sights and user data. Other services access the databases through a database service.

We refrain from showing all model parts here as the necessary level of detail for an in-depth discussion cannot be obtained. Figure 12 shows one example of the model for the (server) broker filtering incoming events. The broker is responsible for service registration, service deregistration, publishing events to the broker, and filtering events. Here we briefly sketch each step and give some details of the model for the filtering.

1. **Service registration** When a service registers with the broker, it first publishes its service description to the broker. A subscribing service provides a set of functional conditions and subscription rules. The observer evaluates the conditions. If they are satisfied, the observer evaluates the services subscription rules and subscribed the service to the events needed. Otherwise, the registration process is stalled until the conditions are satisfied, i.e., until the events needed have been advertised to the broker. Publishing services announce their advertisement.

2. **Service deregistration** When a service deregisters, its advertisement is removed. The subscriptions are removed as well. The broker does not try to filter messages to a subscriber that does not exist any more. After a time interval, services are deregistered from the broker by the observer if the service for some reason has been disconnected. This avoids deregistering of services and re-evaluation of subscription rules when a service disconnects for a short time and then re-connects.

3. **Publishing events to the broker** Services publish events to the broker. We first verify that services may publish events to the broker. If this is the case then the broker enqueues the message in its in-queue.

4. Filtering events When the broker receives events from local services or from other brokers, it filters the events and forwards them to the respective subscribers. In the first step, a service sends its message to the broker. The broker receives the event and enqueues it in its in-queue. Then the broker dequeues the message (Step 2) to first check if there are any subscribers (Step 3) or if the message was published by the database service. If the message was published by the database service, the event is forwarded directly to its recipient and the broker returns to the idle location. Otherwise the broker identifies the subscribers in Step 4. In our example, another service has subscribed to the event. In Step 5,

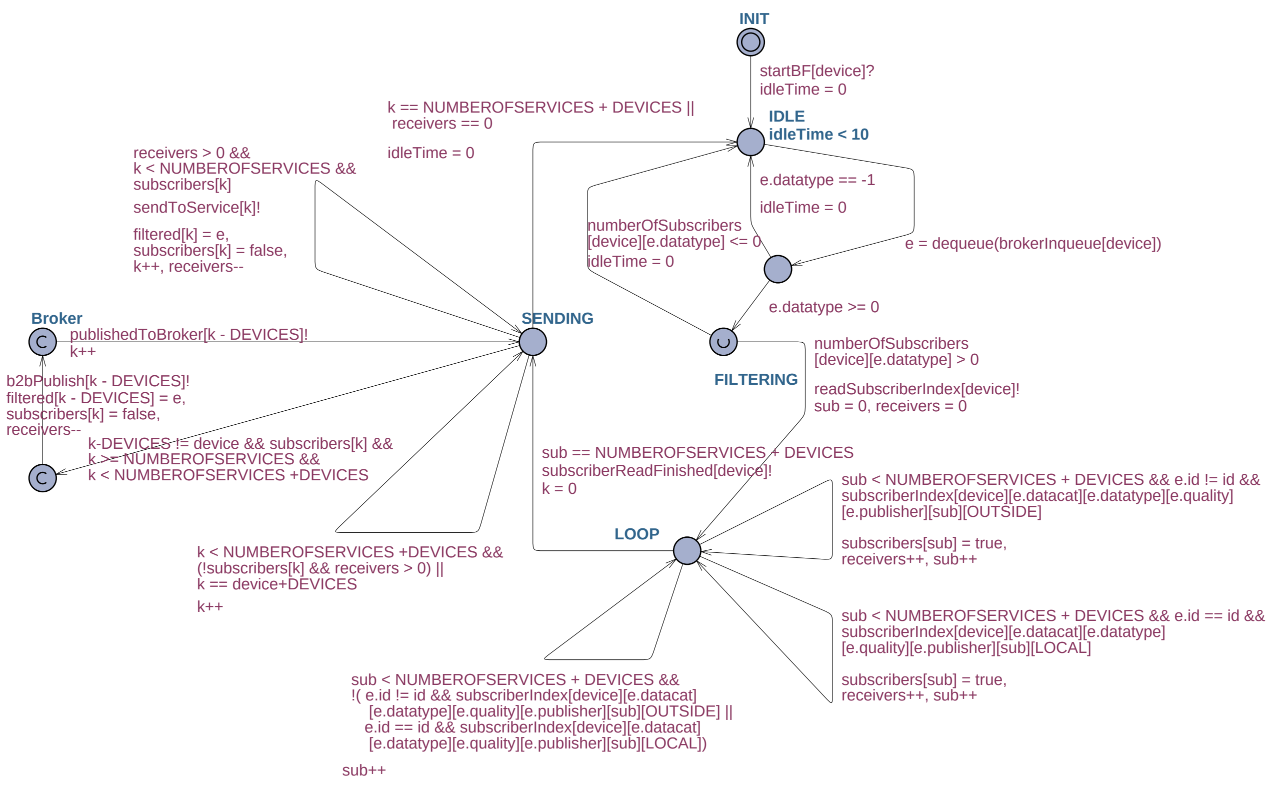
INIT
startBF[device]?
idleTime = 0
IDLE
idleTime < 10
k == NUMBEROFSERVICES + DEVICES ||
receivers == 0
idleTime = 0
receivers > 0 &&
k < NUMBEROFSERVICES &&
subscribers[k]
sendToService[k]!
filtered[k] = e,
subscribers[k] = false,
k++, receivers--
e.datatype == -1
idleTime = 0
numberOfSubscribers
[device][e.datatype] <= 0
idleTime = 0
e = dequeue(brokerInqueue[device])
e.datatype >= 0
Broker
publishedToBroker[k - DEVICES]!
k++
SENDING
FILTERING
numberOfSubscribers
[device][e.datatype] > 0
readSubscriberIndex[device]!
sub = 0, receivers = 0
b2bPublish[k - DEVICES]!
filtered[k - DEVICES] = e,
subscribers[k] = false,
receivers--
k-DEVICES != device && subscribers[k] &&
k >= NUMBEROFSERVICES &&
k < NUMBEROFSERVICES +DEVICES
sub == NUMBEROFSERVICES + DEVICES
subscriberReadFinished[device]!
k = 0
sub < NUMBEROFSERVICES + DEVICES && e.id != id &&
subscriberIndex[device][e.datacat][e.datatype][e.quality]
[e.publisher][sub][OUTSIDE]
subscribers[sub] = true,
receivers++, sub++
LOOP
k < NUMBEROFSERVICES +DEVICES &&
(!subscribers[k] && receivers > 0) ||
k == device+DEVICES
k++
sub < NUMBEROFSERVICES + DEVICES && e.id == id &&
subscriberIndex[device][e.datacat][e.datatype]
[e.quality][e.publisher][sub][LOCAL]
subscribers[sub] = true,
receivers++, sub++
sub < NUMBEROFSERVICES + DEVICES &&
!(e.id != id && subscriberIndex[device][e.datacat]
[e.datatype][e.quality][e.publisher][sub][OUTSIDE] ||
e.id == id && subscriberIndex[device][e.datacat]
[e.datatype][e.quality][e.publisher][sub][LOCAL])
sub++

Figure 12: Model for Broker message filtering.

the broker synchronises with the subscribing services through `sendToService[k]!`. The index k identifies the subscribing service and selects a communication channel. The service receives the message. For a detailed discussion of the modelling we refer to our Technical Report [7].

7 Discussion of the Advanced Architecture

The introduction of quality of service concepts, observers, rules and auxiliary services remedies the shortcomings identified in the discussion of the basic architecture (Section 5). Our interaction famework now covers alternative services, service selection, quality of services and exception handling. Revisiting our previous scenario of a user entering a museum, the following interactions will be triggered: (1) The observer is initialised with quality-of-service rules about location services, (2) The observer detects the low quality of the GPS service when entering the museum, (3) An auxiliary service translates RFID information into GPS data, (4) Preference is given to the higher quality data.

The next challenge facing our event-based SOA is that of sufficient privacy and protection of user-related information. How much information needs to be exchanged between services and how much information can be hidden from the service vendors? These questions will be addressed in future research as more complex modelling and validation are necessary that are, unfortunately, beyond the capabilities of most existing model checking tools.

8 Prototypes

Two prototypes were implemented, one uses Java programming and the other formal modelling in Uppaal.

Basic architecture The basic TIP 3 architecture was implemented using Java. The implemented network is based on a TCP/IP stack. Every component that submits or receives information over the network uses a Connector. To simplify the network access for services and brokers, this connector hides the send and receive implementations. The transported information is an event. This event is wrapped into a message that additionally contains the receiver details, i.e. hostname and host port. A connector provides send and receive methods. The sender and receiver objects are hidden inside and work autonomously. That means, whenever a new message is enqueued for sending, the sender automatically forks a worker process for submitting the contained event to the addressed remote party. Whenever the component, i.e., the service or broker, that owns the connector calls the receive method, a received message is taken from this queue. If the queue is empty the receive method blocks and waits until a new message is enqueued.

This prototype was used to implement and verify the concept of an event-based SOA. It was also used to implement and test caching and pre-fetching in TIP 3.

Advanced architecture We re-implemented the basic architecture using real-time discrete event modelling in Uppaal [2]. Uppaal is a tool-box for the verification of real-time systems. Uppaal uses timed automata to model processes. We extended this basic architecture with the advanced concepts described in the previous section: observers, rules, and auxiliary services. The model was divided into three parts for reasons of clarity and verification. We decided to model the client peer and its services in one model. The server peer and its services are modelled in a second model. The communication between several peers is modelled in a third model. The design and modelling process comprises three interleaving steps. In the first step, the usability requirements should be recognised, as they are needed later during the design and modelling process. During the modelling process, the properties are identified and formulated as verification queries in the second step. The requirements are used to formulate the properties. In the third step, the model is verified using the properties. The verification of the properties often leads to a revision of the model. The different steps often interleave: the analysis of the verification results can lead to a re-engineering of the model or the property, so that the model has to be verified again or the new property has to be verified. Finally the results of the last verification are analysed. The result analysis differentiates two cases: either the property holds under the assumptions upon which the model is based, or it does not hold. If the property does not hold, it is examined and analysed. A property that does not hold highlights weaknesses in the model and should result in a better model or property.

The prototype was used to verify basic properties of advanced architecture and infrastructure. The verification was largely performed using simulation. Our simulations showed that the model functions adequately and without deadlock: services are able to register, i.e., publishers can advertise, subscribers are subscribed to events if their functional conditions are satisfied; publishers can publish events to the broker; the broker filters events to the subscribers; services can deregister, or are deregistered by the observer in case of disconnection. The observer evaluates the services' functional conditions and subscription rules. When a new publisher has advertised to the broker, every subscribers' subscription rules are evaluated and subscriptions accordingly updated. If a registering service's functional conditions are not satisfied, the registration process is stalled until an appropriate advertisement has been published to the broker. Our simulations have shown that only the currently visible service reacts to user input.

9 Summary

In this paper, we proposed an event-based service-oriented architecture for collaboration of mobile context-aware services. All services are loosely coupled and interact via an event-based middle-ware. The information exchange is event-based, i.e., events trigger the submission of information. We illustrated the principle of events message exchange in two versions: a basic architecture that was implemented and practically evaluated, and a formal modelling of an extended architecture to allow for verification and generalizability of observations. We outlined how existing legacy services of a tourist information system can be migrated to the new TIP 3 architecture.

References

[1] Web services architecture. Technical report, World Wide Web Consortium, February 2004.

[2] G. Behrmann, A. David, and K. G. Larsen. A tutorial on uppaal. In *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, page 200236, 2004.

[3] G. Chen and D. Kotz. A survey of context-aware mobile computing research. Technical report, Hanover, NH, USA, 2000.

[4] C.-Y. Chow, M. F. Mokbel, and X. Liu. A peer-to-peer spatial cloaking algorithm for anonymous location-based service. In *GIS '06: Proceedings of the 14th annual ACM international symposium on Advances in geographic information systems*, pages 171–178, New York, NY, USA, 2006. ACM.

[5] G. Cugola and H.-A. Jacobsen. Using publish/subscribe middleware for mobile systems. *SIGMOBILE Mob. Comput. Commun. Rev.*, 6(4):25–33, 2002.

[6] T. Erl. *Service-Oriented Architecture: Concepts, Technology, and Design.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005.

[7] L. Eschner and A. Hinze. Design and formal model of an event-driven and service-oriented architecture for the mobile tourist information system tip. Technical report, University of Waikato, November 2008.

[8] P. T. Eugster, P. A. Felber, and A. marie Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35:114–131, 2003.

[9] D. Frey and G.-C. Roman. Context-aware publish subscribe in mobile ad hoc networks. In A. L. Murphy and J. Vitek, editors, *COORDINATION*, volume 4467 of *Lecture Notes in Computer Science*, pages 37–55. Springer, 2007.

[10] M. Gruteser and D. Grunwald. Anonymous usage of location-based services through spatial and temporal cloaking. In *MobiSys '03: Proceedings of the 1st international conference on Mobile systems, applications and services*, pages 31–42, New York, NY, USA, 2003. ACM.

[11] K. Henricksen, J. Indulska, and T. Mcfadden. Middleware for distributed context-aware systems. In *International Symposium on Distributed Objects and Applications (DOA*, pages 846–863. Springer, 2005.

[12] A. Hinze and G. Buchanan. The Challenge of Creating Cooperating Mobile Services: Experiences and Lessons Learned. In *Twenty-Ninth Australasian Computer Science Conference (ACSC 2006)*, Hobart, Australia, Jan. 2006.

[13] A. Hinze and S. Junmanee. Advanced recommendation models for mobile tourist information. In *Federated Int. Conference On The Move to Meaningful Internet: CoopIS*, pages 643–660, 2006.

[14] A. Hinze and A. Voisard. Location- and time-based information delivery in tourism. In *Conference in Advances in Spatial and Temporal Databases (SSTD 2003)*, volume 2750 of *LNCS*, Santorini Island, Greece, July 2003.

[15] T. D. Hodes and Y. H. Katz. Composable ad hoc location-based services for heterogeneous mobile clients. *ACM Wireless Networks*, 5:411–427, 1999.

[16] X. Huang. Travel Planning Map Service – Development of a Java-based Application for Travel Planning. Master's thesis, Dep. of Comp. Sc., University of Waikato, Sept. 2006.

[17] N. Le Sommer. Service Provision in Disconnected Mobile Ad Hoc Networks. In *International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies (UBICOMM 2007)*, pages 125–130, Papeete, French Polynesia (Tahiti), November 2007. IEEE Computer Society Press.

[18] R. Meier and V. Cahill. Steam: Event-based middleware for wireless ad hoc networks. In *Proceedings of The ICDCSW*, pages 639–644, 2002.

[19] A. Slominski, M. Govindaraju, D. Gannon, and R. Bramley. An Extensible and Interoperable Event System Architecture Using SOAP. Technical Report TR549, Department of Computer Science Indiana University Bloomington, IN, U.S.A., 2002.

[20] C.-F. Sorensen, M. Wu, T. Sivaharan, G. S. Blair, P. Okanda, A. Friday, and H. Duran-Limon. A context-aware middleware for applications in mobile ad hoc environments. In *MPAC '04: Proceedings of the 2nd workshop on Middleware for pervasive and ad-hoc computing*, pages 107–110, New York, NY, USA, 2004. ACM.

[21] J. van Gurp, A. Karhinen, and J. Bosch. Mobile service oriented architectures (mosoa). In F. Eliassen and A. Montresor, editors, *Proceedings of DAIS*, volume 4025 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2006.

[22] X. Yang, A. Bouguettaya, B. Medjahed, H. Long, and W. He. Organizing and accessing web services on air. *IEEE Transactions on Systems, Man, and Cybernetics, Part A*, 33(6):742–757, 2003.

[23] S. S. Yau, F. Karim, Y. Wang, B. Wang, and S. K. S. Gupta. Reconfigurable context-sensitive middleware for pervasive computing. *IEEE Pervasive Computing*, 1(3):33–40, 2002.