Working Paper Series
ISSN 1177-777X

DESIGN AND FORMAL MODEL OF AN EVENT-DRIVEN AND SERVICE-ORIENTED ARCHITECTURE FOR THE MOBILE TOURIST INFORMATION SYSTEM TIP

Lisa Eschner and Annika Hinze

Working Paper: 14/2008
December 2008

Design and Formal Model of an Event-driven and Service-oriented Architecture for the Mobile Tourist Information System TIP

Lisa Eschner, Annika Hinze
Freie Universitaet Berlin, University of Waikato
eschner@mi.fu-berlin.de, hinze@cs.waikato.ac.nz

Abstract

This thesis introduces a new collaboration framework for context-aware services in a mobile environment enabling services to co-operate with several anonymous co-operation partners. We extend the current TIP design and architecture so that new services may easily be added to and co-operate with existing ones. Obsolete services may be replaced by new ones providing the same functionality. Services are de-coupled. Service co-operation is completely changed. This means that services react to the events they receive, irrespective of the events publishers. We also show how service-oriented and event-driven architectures may be combined maintaining their respective advantages. We introduce features of service-oriented architectures to services co-operating via an event-based middleware. We describe the formal model of a new system for mobile tourist information and the newly introduced features of the collaboration framework. Those features fundamentally change the way services communicate and cooperate.

1 Introduction

The subject of this paper is the design of an architecture for a service oriented and event-driven tourist information system. The paper reports on a part of a greater ongoing project, the complete re-design and re-implementation of a mobile tourist information system. In the first step, a small prototype was implemented [26]. In the second step (reported here), we created a formal model of the design and examine the model. The design needs to support extremely loose coupling of services and service co-operation in a highly dynamic environment. The final step will be the implementation of the design.

The service architecture was designed for the Tourist Information Provider (TIP). TIP is a mobile tourist information system. Tourist information systems offer many services to tourist, e.g., hotel booking services and information on sights. Mobile tourist information systems offer similar features, but with the advantage that they run on portable computers, such as PDAs or smartphones. The portable computers must be able to ascertain the current location, e.g., through a GPS device. Thus, a mobile tourist information systems enables tourists to use the services en route.

Mobile tourist information systems enhance tourists' experiences. Tourists are ready to use mobile guides [27], even if they have little or no previous experience with computers or personal digital assistants. Tourists discover more sights and spend more time at sights when they use a digital guide that provides the tourists with a map displaying sights and information on the sights. Tourists without such a guide stayed at less attractions and walked more. The guided tourists saw more attractions in less time, and walked less than those without a digital guide. Mobile tourist information systems may provide information about sights the tourists would not have known about or merely walked by otherwise. Personalised mobile tourist information systems enhance tourists' independence [22], compared to tourists discovering a town with a guided group, a guide book or an audio guide.

In this section, we first present the existing TIP. We then explain the aim of the paper in more detail, and lastly present the outline of the paper.

1.1 TIP – the Tourist Information Provider: Main Characteristics

TIP is a tourist information system targeted at tourists using mobile computers [18]. TIP supplies tourists with information upon sights and other points of interest, based on the user's location and user profile. We introduce TIP's functionalities and main characteristics with a short usage scenario.

TIP 1 Usage Scenario Katherine, a TIP user creates a personal profile, where she defines that she is interested in churches, cultural heritage sites, museums and parks, and that she likes history, architecture and literature. She thus creates her sight profile and her topic profile. Katherine's sight profile specifies the kind of sights she is interested in, e.g., history, architecture and literature, while the topic profile defines her main thematic interests. The TIP system uses the topic profile to select the points of interest (POI) for a user, and selects the information that are displayed on the POIs with the aid of the interest profile. The user context comprises the sight and topics profile, the user's current location and time. Katherine visits Hamilton, New Zealand. She brings her own, TIP-enabled PDA with her.

As she walks through Hamilton, Katherine comes by the Waikato Museum. Her TIP display shows general information on the museum like the main exhibits and opening hours. The map shows several cones. One of them, the one on the museum's location, indicates this sight. The cones have two different colours, one for sights provided by the information service, the other for sights suggested by the recommendation service. Katherine can browse for information on distant locations. TIP shows this information in a brighter colour scheme. When Katherine revisits the museum, TIP displays the latest information shown to her on her last visit.

Location-based Services Location-based services (LBS) offer services that depend on the user's current location. [32] define LBS as "services (that are) accessible with mobile devices through the mobile network and utilising the ability to make use of the location of the terminals." However, this definition only applies to computerised LBS. A poster that announces a concert near the concert venue is an LBS, as well – LBS are not a new occurrence.

The main characteristics of computerised LBS are that the user can access them through an available communication infrastructure, such as a mobile network, a MANET or the internet. Moreover, LBS posses some means to determine the user's current location. The location is necessary so that the service can be performed. In the short usage scenario aove, the map is an example for a location-based service. The map has to know about Katherine's current location, otherwise it cannot display the appropriate map tile.

Context-awareness TIP is a context-aware mobile tourist information system. The context consists of the user's interest as defined in the sight and topic profile, the current location, the current time and the user's travel history. The context-awareness enables the TIP services to select information or other data based on the user's interest and topic profile. The current location is considered in the sense that information on sights in the user's current vicinity are delivered to the user. The current time may be considered as to how information is displayed, e.g., at night a cone on the map that indicates a closed museum would not be shown as brightly as a cone that indicates a club, or a restaurant.

Event-based Systems In TIP, services react to incoming location events, e.g., the information service receives a new location event and filters the TIP database for nearby sights.

The term event can be ambiguous. It can denote the fact that something has happened in the real world that is both observable and distinguishable [28]. On the other hand, the term event can also refer to the computer representation that describes the event. Here we use it in both senses, as the intended meaning will be clear from the context. The term event and event-driven architectures are discussed in more detail in Section 5.

Service-orientation Service-oriented computing aims to divide business processes into separate independent processes, and to package them as services. Services co-operate with other services. The main principles with service-oriented computing and service-oriented architectures (SOA) are that services should be re-usable, use service contracts, they should be independent from other services and loosely coupled [7]. SOA are discussed further in Section 5.

History of TIP Implementations The TIP 1 architecture was conceptualised by Hinze and Voisard [19]. TIP 1 combines location-based services in tourism with the concept of event-notification. Löf er [25] implemented a subset of the proposed functionality. The prototype uses a client-server architecture. The TIP data is stored in a PostgreSQL[1] database. Ottlinger [30] aimed at adding peer-to-peer-communication to TIP. Due to technical problems, this could not be achieved, however, he re-implemented TIP and introduced J2EE-based web applications, the result being TIP 2. TIP 2 soon experienced challenges as the software was extended and new services were introduced. The services were rather tightly coupled so that it was difficult to introduce new services without changing the existing co-operating ones. For example, if a service provider wanted to introduce a new service that would co-operate with the map service, she would have to expand the map service with a new communication interface. In addition, although the main concept for TIP 2 clearly was event-driven, the implemented software architecture did not support the concept directly. TIP 2 is discussed in detail in Section 2. Michel [26] created a simple implementation for TIP 3*. It served as a proof of concept for an event-based middleware. Services in TIP 3* are loosely coupled. However, as we will discuss in detail in Section 3, they have to be extremely loose coupled. TIP 3* also lacks some main characteristics of a service-oriented architecture, such as the service contract. We will show later how we introduce more service-oriented attributes to TIP. The main characteristics and features of TIP 3* are discussed in 4.1.5.

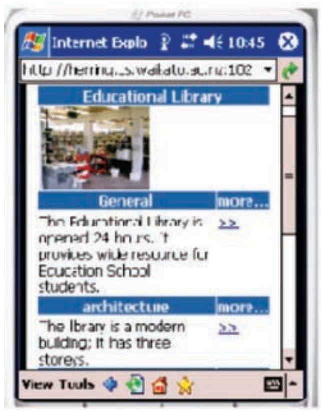
1.2 Aim and Structure of the paper

The previously discussed usage scenario shows how several services co-operate and interact with each other. At the moment, the services communicate through the exchange of events. However, once a service has subscribed to an event type, it will not change its subscription, even if the publisher unregisters with the broker – the subscriber does not change its co-operation partner. Similarly, the subscriber is not notified if another, possibly better publisher registers with the broker. TIP aims to be an easily extendable information system. We recognise the need for an event-based infrastructure that supports a dynamically changing set of services, that do not depend on each other, i.e., services have to be extremely loosely coupled. Nevertheless, some characteristics of a service-oriented architecture should be retained, that are lacking in the current prototype. In a dynamically changing environment, where the user, i.e., the mobile client, moves in and out of a location-based services' area, the set of available event types is changing dynamically, as well. This presents a challenge for service developers, as it is difficult to assess how a service reacts to the changing range of event types [1]. We recognise the necessity of a framework that helps service

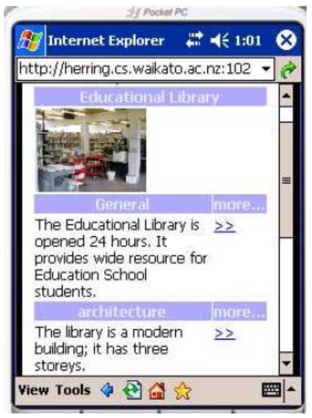
[1]http://www.postgresql.org/

designers to thoroughly analyse and evaluate their design. We have developed a service-oriented and event-driven architecture for TIP. We have implemented a prototype as a formal model with the help of UPPAAL[2] as a proof of concept.

In Section 2 we present the current TIP system, TIP 2. In Section 3, we formulate our requirements on TIP 3 in detail, and show why the existing architecture does not meet the requirements. We then analyse related work in Section 4. Section 5 discusses service-oriented and event-driven architectures. It compares them to the needs of a TIP 3 design. The new design and architecture of TIP 3 are presented in Section 6. Section 7 gives an introduction to formal modelling. We argue, why we decided to model the architecture instead of implementing it. We present the modelling tool UPPAAL, its modelling language and query language. In Section 8, we present the model of the new architecture as a proof of concept. Section 9 summarises the paper and discusses future work.


Internet Explo 10:45
http://herring...s.waikato.ac.nz:102
Educational Library
General more...
The Educational Library is opened 24 hours. It provides wide resource for Education School students. >>
architecture more...
The library is a modern building; it has three storeys. >>
View Tools

(a) TIP showing nearby sights.


Internet Explorer 1:01
http://herring.cs.waikato.ac.nz:102
Educational Library
General more...
The Educational Library is opened 24 hours. It provides wide resource for Education School students. >>
architecture more...
The library is a modern building; it has three storeys. >>
View Tools

(b) TIP showing remote sights.

Figure 1: TIP screenshots. The TIP screenshots show that the information service displays information on nearby sights in a darker shade (left). Remote sights (right) are shown in a brighter shade.

2 Background on TIP

In this section we present the Tourist Information Provider TIP. First we present the current version TIP 2's functionality. We introduce the main services, the location, information, recommendation and map service. These four services provide the basic functionalities of a tourist information system: a map that shows the user's current location and nearby sights, and that offers information on the sights. The services show how services interact in TIP 2. They exemplify the main characteristics of TIP 2. We then discuss the concepts and implementations of TIP 2, followed by a demonstration how services in TIP 2 communicate.

2.1 Important Services in TIP 2

TIP 2 comprehends several services [14], so that the user can choose from a variety of services. The most important ones from a functional point of view are the

The location service The location service [15] provides the TIP system with the user's current location. It runs locally on the PDA. The location service uses a GPS receiver. Whenever the location changes because the user moves, or because a certain time interval has passed, the location service communicates the new location to the TIP 2 server. The TIP 2 server forwards the new communication to interested services that run on the same client.

The information service The information service [15] provides information on sights and topics. It runs on the TIP 2 server. The information service receives the new location from the TIP 2 server. It then filters sights and topics from the TIP 2 database, based on the user's location and the corresponding user profile. The information service sends this information to the server. The server forwards it to all interested services, regardless if they run on the mobile client or on the server. Information on nearby sights (cf. figure 1(a)) is shown in a slightly darker shade than information on remote sights (cf. figure 1(b)).

The recommendation service The recommendation service [16] suggests new sights to the user, depending on the user's location and preferences as well as the user's travel history and feedback. Like the information service, the recommendation service runs on the TIP 2 server. It takes into account the feedback that other users with similar preferences have given.

The functionalities of the recommendation service and the information service are similar, but not the same. The information service selects information (i.e., topics and sights) based on the user's profile and history, while the recommendation service tries to predict sights and topics based on the user profile.

The map service The map service [15] displays a map showing the user's current surroundings. It runs on the mobile client. On the map, cones indicate sights, recommendations and other points of interest, for example, restaurants. The map service connects to the TIP server when it needs more map tiles, or when it does not know the current location. The TIP server supplies it with the sights, information and recommendation data needed for the map, and with the necessary map tiles.

The map service uses differently-coloured cones for information and recommendation data. The user can thus differentiate between recommendations and informations.

The information service and the recommendation service are located on the server: the computation, database access and filtering operation happens on the server. Such a service, where no computation is done on the client and only the

[2]www.uppaal.com

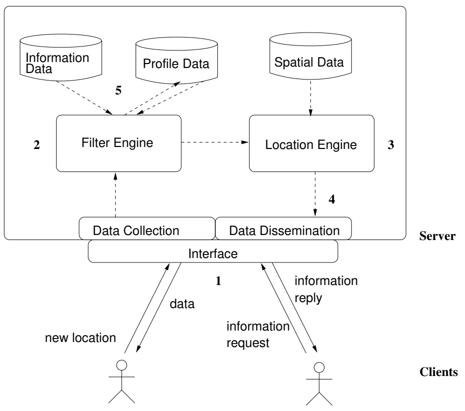


Figure 2: Simple client server interaction and the TIP 2 core system

results of the server-side computations are displayed on the client, is also called *thin client*. In TIP 2, thin clients typically share a common user interface and a common communication interface, e.g., the web browser for displaying and for interacting with the user.

The map service and the location service are services where every computation is done on the client. The map service only communicates with the TIP 2 server to request new map tiles, or to get the current location. The location service communicates the current location to the TIP 2 server. Both services are examples for *thick clients*. Thick clients may have their own user and communication interface.

2.2 TIP 2 – Concept and Implementation

Figure 2 illustrates the TIP 2 core system and its interaction with mobile devices, following [15] and [26].

The TIP 2 core system has several tasks. We describe them with the help of Figure 2: In Step 1, the TIP 2 core communicates with several TIP 2 clients at once. The TIP 2 core system and TIP 2 clients communicate via http or SOAP, using the TCP/IP-stack provided by the operating system on the mobile device. Services communicate with the TIP 2 core system via TIP 2's communication interface. In Step 2, the TIP core system filters sight information, taking the user context into consideration. In Step 3, the TIP core system provides spatial data, i.e., it selects the sights in a user's vicinity. This is done by the location engine that is a part of the TIP 2 core system. In Step 4, it offers information to other services and then stores and provides the user profiles and the system and services profiles in Step 5. The TIP 2 core system stores the users' sight and topic profiles, their travel history and user reactions to recommendations. The service profiles define which events are generated or needed by a service.

The TIP 2 core system provides an information database, a geo-spatial database, a sights database and a profile database. It further comprises a filter engine and a location engine, see Figure 2. The TIP 2 core system runs on a web server. The current TIP 2 implementation stores user and system profiles, information on sights and spatial data, used for mapping sights to the coordinate system used by TIP2, in a common database.

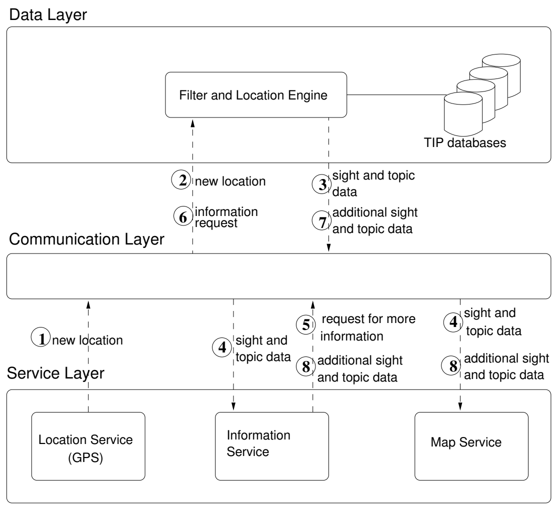


Figure 3: Service interaction in TIP 2

Although TIP 2 was conceptualised as an event-driven system, the implemented architecture is not event-driven. It lacks important features of event-driven systems, such as an event-notification middleware.

2.3 Service Interaction in TIP 2

In TIP 2, services cooperate to provide a functionality. Figure 3 illustrates how the information service, location service and map service interact.

In Step 1, the location service on the mobile client transmits a new location to the TIP 2 server. The server forwards the new location to the TIP 2 filter and location engine running on the TIP 2 server (Step 2). The filtering engines select the appropriate data with one SQL-query. It is possible to access all stored data at once as it is stored in a single database.

In Step 3, the selected data is returned to the client. The information data is received by the information service's thin client, e.g., a browser, and by the map service on the mobile client in Step 4. When the user asks for more information (Step 5), the request is forwarded to and processed by the filter engine (Step 6). The resulting data is transferred to the information service and possibly the map service (Step 7 and 8). The communication layer is conceptualised in the TIP 2 architecture, though it is not implemented in detail.

3 Requirements on TIP 3

In the previous section we discussed the TIP 2 architecture and implementation. In this section we develop an extended usage scenario in Section 3.1. We develop our requirements on the TIP architecture from that scenario in Section 3.2 and discuss the requirements in detail.

3.1 An Extended Usage Scenario

Let us again join Katherine, the tourist arriving by car in Hamilton, New Zealand. She visits some friends. Katherine brings her own TIP enabled PDA with her.

The route planner service On arriving in Hamilton, Katherine starts TIP's Route Planner as she does not know the way to her friend Anne. The Route Planner shows her a list of possible starting points: her current location, and the main attractions in Hamilton. After she selected a starting point, the Route Planner lists some possible destinations, based on Katherine's user profile. Katherine may of course enter Anne's address manually, if none of the possible destinations suits her.

The Route Planner guides Katherine to Anne's home. En route, Katherine hears TIP's alert for cultural heritage sites. She stops at the roadside and looks what kind of site this is. First, she clicks on the alert symbol displayed on the map, so that TIP displays general information about the site. As it is interesting, she scrolls down for more information, and clicks on the "more"-button. The new information is displayed on the screen.

The sightseeing tour author Later, Anne and Katherine decide to take a stroll through the city. Anne creates a sightseeing tour for Katherine, using TIP. She lists the sights she wants to show Katherine. TIP creates a route, based on the sights list and Katherine's and Anne's profiles. TIP provides several different travel modes, walking, going by bike, by car or using public transport. The created routes differ, depending on the selected travel mode.

Several user pro les Anne's profiles are activated, as well as Katherine's, so TIP alerts them whenever they pass a sight that matches one of their profiles.

TIP s user interface TIP's user interface consists of a web browser that displays the delivered information. A change of service may be displayed as the opening of a new browser window, or a reload in the current window, where the new content is displayed. The change of service is transparent to the user. The user only notices it in terms of changed or extended functionality.

The close-by friends service and the chat service As they walk, TIP suddenly alerts them that Katherine's friend Steve is nearby. Katherine clicks on the "more information"-button. The map shows a new icon, representing Steve. Some contact details hover above the icon. As long as Steve and Katherine do not move out of their mutual maps, their icons are shown on the other's map. Katherine chats with Steve, using TIP's chat service, and they decide to meet in a café nearby, shown on Steve's map. Katherine decides she wants to see the same information as Steve, thus seeing the same map tiles as Steve does. Now she sees where the café is, and she can navigate through the map to her own location.

Several information providers On their way to the café, Katherine and Anne pass by Hamilton Gardens. Anne wants to show them to Katherine. On entering the gardens, the map service shows a new map, provided by the Hamilton Garden's TIP server. This new map shows the gardens in more detail. If Katherine does not want to see the Hamilton Garden's map, she can easily switch back to the previous map. In the same way, she can select a new information source, provided by the garden's local TIP server.

The event service While they are viewing Hamilton Gardens, they are informed by the event service about a concert taking place there the same evening.

3.2 Requirements on the new Architecture

The extended usage scenario presented above shows several new features and requirements a mobile tourist information system could offer.

We introduced new features: a route planner, a sightseeing tour author, the possibility to use several profiles simultaneously, a close-by friends service, the possibility to choose an alternative service or information provider and an event service. Several other services are feasible, of course.

These services co-operate with each other without any difficulty. The map service displays the route to Katherine's destination. The sightseeing tour author requests the route planner to calculate a route for Anne and Katherine. The route is shown on the map, and information upon the sights is displayed by the display service. The close-by friends service makes the map display Steve's location. It uses both TIP's information and map service to show Katherine the information and the detail of the map that Steve is seeing.

Our usage scenario displays a mixture of both location-based and only location-aware services. The Hamilton Garden's map service is location-based, while the information service, close-by friends service and map service are location-aware. Services need to react to this dynamic environment of changing services: when Katherine entered the Hamilton Gardens, her TIP client connected to the Hamilton Garden's TIP server. The services on the mobile client and the services on the Hamilton Garden's server co-operated. When Katherine left the garden and the server services became unavailable, the services on her PDA continued to function without any problems. Likewise but at the same time differently, the close-by friends service suddenly showed that a friend of Katherine's was nearby. It showed where that friend was and some contact information. This close-by friends service would be composed of several services: one service that discovers nearby friends, one that offers a chat functionality, and another that provides information on the friends, for example. The close-by friends service co-operates with other services, such as the information service. However, if its own chat service is not available, the close-by friends service would simply display the friend's location and possibly some other information. If another chat service was available, the close-by friends service could use it.

We identify some issues that a new architecture should consider. We explore the different questions in depth later. The main requirements concern (1) communication between services, (2) service management (3) server management, (4) rule-based subscriptions, (5) data handling and (6) privacy and confidentiality.

1. Communication between services:

1a. Prompt[3] communication. Some services need recent information. The close-by friends service, as depicted in 3.1, needs to be notified whenever a friend's location changes, so that it is able to display the new location on the map accordingly. Published information should be delivered to subscribing services swiftly. We call this *prompt communication*. An event should be transmitted to all subscribers within a fixed time after the event was generated.

1b. Local communication Services that run on the same node should communicate without a detour to the server. We think this could reduce expensive radio communication.

2. Service management Services offer functionalities. When a service that offers a certain functionality, e.g., location data, disconnects, its functionality becomes unavailable to the subscribers. Whenever other services that offer the same functionality are available, the disconnected service should somehow be replaced by one of the other services. However, when a functionality becomes unavailable, the subscribers cannot fulfil their tasks.

   Several services can offer the same or similar functionality. An interested service ought to be able to choose the service whose functionality best matches the requester's needs. For this reason, functionalities offered by services may need to be categorised hierarchically. Let us look at an example: Location data can be provided through different means, such as GPS coordinates, an address, or the coordinates used by the map service. The GPS-service could categorise its data as `location/gps`, while the address-service could categorise its data as `location/address`. When the GPS-service is unavailable, the service that requests `location/gps`-data would change its request into a location-request. The request would be answered by the map service and by the address service. The first requirement that we presented, connect to another service with the same functionality, may easily be tested when several services provide the same functionality at the same time. The requester selects one of the services. When that service disconnects, the requester should connect to another service provider.

   The second requirement may be tested with a similar scenario: Several services that offer similar services are available. The service requester connects to one of them, that becomes unavailable. The requester should connect to the service provider that offers the best similar service.

3. Rule-based subscriptions This requirement does not emerge out of the usage scenario, as some of the requirements above did. However, it arises partly out of requirement "Service management", which asks for loosely coupled and possibly transparent services. When the subscribing service decides what kind of data it needs instead of specifying a certain publisher, the disconnection of the publisher may be transparent to the subscriber. When the subscribers specify the data needed, publisher and subscribers are further decoupled. Services should always subscribe to the best data available.

4. Server management When more than one server is available, the client may need to choose between them, or to connect to more than one server at the same time. The user should be able to define in their profile whether to connect to a new server, at least when it offers services or data the user has to pay for. We think it is desirable that the client is able to connect to more than one information server simultaneously. Obviously, the user experience is enhanced when more information is available.

5. Data handling TIP stores and distributes data towards the users. The same datum, e.g., basic information on a museum, may be shown as an information to one user, and as a recommendation to another user. The map, or a display service like the browser, shows information data and recommendation data in different shades. These services have to be able to distinguish between informations and recommendations although the data themselves might be the same. The datum needs to be tagged by its provider before it is delivered to the client. However, the tags cannot be static, as they depend on the producing service. On the other hand, TIP differentiates sight types (topics), e.g., architecture sights and historical sights. Items that are included in a TIP database are associated with one or several topics.

   Data that are up to date and accurate increase the attractiveness of a tourist information system. We are convinced that user input – e.g., restaurant reviews, or hints about events – contributes to up-to-date data. However, data that were supplied by TIP users could be faulty. We propose that users ought to be able to rate data. Suitable rating can help users to decide whether they trust an information datum. The rating of information enables the TIP system to rank information data according to their trustworthiness and quality, as well.

   The rating of data through the users could be a sensible approach to handle spam. Of course, other measures will be needed as well, but spam protection is not the scope of our project [8]. We identify two requirements: (5 a) Data classification and (5 b) mechanisms that help to ensure data trustworthiness and quality. We acknowledge this issue but will not attempt to resolve it in this paper.

[3] We use "prompt" in the meaning that "something will happen within the next seconds, or within a minute", i.e., near in time. We do not use the term real-time that typically connotes a communication that takes place within microseconds. We need communication that happens near in time, but not in real-time

6. Privacy and confidentiality User data like the user profile and user history that is stored on the server should be protected from unauthorised access. We think that the user should have a chance to agree to whether their profile will be shared with other ("foreign") servers, e.g., during the registration process. The user profile is sensible data. However, everyone in a user's vicinity can eavesdrop on data that is sent to them from the server. This makes it rather easy to guess their interest profile. As countermeasure, the filtering of the data could be moved to the node. The server would transfer all information and recommendation data to the client, where a filtering machine would match them against the user profile. Indeed, we think that the matching of data and user profile ought to take place on the server, so that no unnecessary data is transferred. Another solution is to use encrypted communication between the server and client.

Our extended user scenario showed a number of features that a mobile tourist information system should provide. Based on the scenario, we defined requirements addressing issues like communication between services or how services are managed. We identified a number of open questions: interaction and communication between services or between services and the TIP server, privacy-related issues or safety-related issues. The following section discusses related work and compares them to the requirements.

4 Related Work

In this section we analyse related work: mobile tourist information systems, a rule-based middleware for sensor networks. In the first section, we compare mobile tourist information systems. Section 4.2 discusses a rule-based and event-driven middleware for mobile sensor networks and general basics on event-driven architectures. Lastly we summarise the results of our comparisons and compare them to our requirements.

Although pull systems, such as the Google search engine or other search engines available on the internet, may be used by tourists, we do not include them for several reasons: With search engines, the user has to request information. Unlike TIP, the information is not displayed automatically, without any actions on the user's side. Search engine results and Google's catalogues have to be filtered by the user, a task that can be rather tedious. In TIP, the filter engine undertakes the task of filtering. The user has to know the name of the sights, or at least that a sight exists, and where it is located. Otherwise, it is difficult to obtain good results from search engines. Imagine a tourist in front of the Berliner Dom in Berlin-Mitte, who does not know that the huge church is called Berliner Dom. A search at google.de on "Lustgarten Berlin" returns the google map, the URL for a sex-bar, several homepages that offer pictures from the Lustgarten, and some pages that supply information on the actual place. This makes search engines rather useless to tourists.
Neither do we look at technologies as dynamic mash-ups. Dynamic mash-ups combine information and knowledge from several sources into a new product. Dynamic mash-ups can open up existing information sources and internet sites for TIP, while they preserve TIP's advantages. A dynamic mash-up could filter and personalise Google search results. Dynamic mash-ups can be used by TIP services. They are no rivalling system, but a complementing technology.

4.1 Mobile Tourist Information Systems

Mobile tourist information systems usually are context-aware systems. They have to be highly adjustable, i.e., they adjust themselves to a change of context. It is desirable that they are personalisable as well. We are convinced that context-awareness, personalisation and exibility add to an enhanced user experience. Studies have shown that users benefit from mobile tourist information systems. They discover and visit more sights than without the guiding system [22]. Users are ready to accept and use mobile tourist information systems, even those who have little previous experience with computers.

In this section, we present three mobile tourist information systems: the Dynamic Tour Guide, the George Square System and GUIDE. We choose the systems as they represent different facets in the field of mobile tourist information systems. The Dynamic Tour Guide [27], [21], [22] is an event-driven, context-aware tourist information system. It has been used by many real tourists in an on-field study. GUIDE [5] is a well-known mobile tourist information system that is often referred to as standard tourist information system. The George Square System [20] comprehends a real-time communication system. It concentrates on the sharing of experiences with other users. TIP 2 and TIP 3* are the previous versions of the TIP system.

In this section, we firstly present a system's functionality. The following part introduces the user interface, i.e., how the user can interact with the system, and what features it offers. In the last part, we explain how the system works.

4.1.1 The Dynamic Tour Guide

The Dynamic Tour Guide [21, 22, 27] (DTG) is a mobile tourist information system that was developed at the University of Applied Sciences Zittau/Görlitz, Germany. It offers two different usage modes:

- the Explorer mode: A map shows the tourist's surroundings, with all surrounding sights. The Explorer mode reacts dynamically to the user's behaviour: whenever they enter a sight, or stops at a sight, a presentation is started. The presentation stops when the user leaves the sight or stops it by hand.

- the Planner mode: The aim was to imitate a human personal guide. The Planner mode asks tourists about their interests and develops a city tour adapted to the user profile and other user constraints. Only the sights in the personalised tour are shown on the map. The information presentation on a sight starts when the user arrives at that sight. The Planner mode offers the chance to adapt the tour while walking, by adding or dropping sights.

Location data is provided through a GPS device connected to the mobile client. Sights are stored in Tour Building Blocks (TBB). A TBB contains the sight address, some categorisation data, picture and audio files, and additional information on the sight. Every TBB has a Web service where its data is stored and offered to users. The Web services register with a UDDI registry. The audio hints and the navigation map are stored on the mobile client.

Comparison to the Requirements We analysyse to what extend the Digital Tour Guide meets our requirements (Section 3.2):

1a. Prompt Communication The DTG immediately responds to the user's location, as it either adjusts the map and shows the sights nearby (Explorer mode) or displays information on a visited sight (both modes). The map and information displayer have to be notified about the current location, that is ascertained through the connected GPS device. We therefore assume that the DTG offers prompt communication. However, this prompt communication mainly takes place on the mobile device. The mobile device and the server communicate only when the user is asked about their interests, and when the server computes a route for the Planner mode.

1b. Local Communication The DTG disposes of several services: a navigation service, tour planner, tour adapter, navigation software, among others. Most of the services are located on the mobile device. The tour planner is an exception. The services that are located on the device, e.g., the GPS service and the map, communicate locally.

2. Service Management The DTG offers several services, e.g., the tour planner, the display of information, the map, or the location service. However, it does not supply any means to replace a failing service through another. Even more important is that the DTG cannot include new services, a characteristic that makes it difficult to introduce new features. We therefore argue that the DTG does not meet our requirements.

3. Rule-based subscriptions The Planner mode shows information on sights, if the sight matches the user profile and is part of the proposed tour. The sight data and other information are wrapped in Web services. However, to the best of our knowledge this architecture includes a publish-subscribe aspect. Therefore we suppose that the DTG contains some rule-based data forwarding component. However, this does not guarantee that a service always subscribes the best data available.

4. Server Management The DTG architecture, such as it is described in [21], uses only one server. The server's main tasks are to compute the guided tours, and to store the user profiles and the data that was gathered during the field study. We assume that it cannot handle more than one server at a time, though it should not be to difficult to extend the system.

5a. Data ClassificationSight and information data is classified in several ways: sights are distinguished from such things as restaurants. An ontology was used to model the sight information. The sights are classified into several interest topics. DTG obviously provides some means of data classification.

5b. Data Trustworthiness and Data QualityThe issue of data trustworthiness is not raised in any of the papers about DTG. We assume that data trustworthiness is not implemented in the DTG, especially as it seems to be a one-server system administered by trustworthy staff. The DTG was implemented for a field study with the goal to investigate and analyse user acceptance and the usage of mobile tourist information services by everyday users. The data upon sights and other points of interest, such as restaurants or cafés, was put in at once, by trustworthy personnel. User input was not arranged for, therefore there was no need to handle or edit it in any way. We think that this applies for data quality and data evaluation as well.

6. Privacy and Confidentiality The DTG project's aim was to examine and analyse the acceptance and usage of mobile tourist information systems by usual tourists. The users' movements and interactions with the system were recorded, so that they could be analysed later on. Users were asked for their interests when they used the DTG Planner. All data that were produced by users – interest profiles, gps track, system interactions etc. – were stored on the DTG server.

4.1.2 The George Square system

The George Square system [20] (GS) is a tourist information system that focuses on the sharing of experiences. It was developed at the University of Glasgow. Users are shown a map of the visited town, either the users current neighbourhood, or the location where the user placed her avatar. Sights, recommendations and other users' avatars are shown on the map. The map data are downloaded from a map server on the internet. Users can talk to other users via IP telephony. This gives the users the chance to interact with other users, e.g., a friend at home, and to immediately show photos taken with the camera to someone else. Users can place their photos on the map. They can also browse the web for information, or visit web pages recommended by the Recommender. The George Square System offers two different usage modes, either on-site, i.e., touring a town or off-site, when the user prepares her journey, shares it with other users, or shares user's experiences while they are on-site.

The George Square system has an event-based architecture. It uses the EQUIP middleware, which provides communication between devices or sensors on one network node and between different network nodes. The communication model supports peer-to-peer communication. Location data is provided through a GPS device connected to the Tablet PC. A camera is also connected to the PC.

Comparison to the Requirements The George Square System does not meet all of the requirements defined in Section 3.2:

1a. Prompt CommunicationThe George Square system cannot perform without a network connection. The mobile device has to be connected to some network, otherwise the system does not provide information on the sights. This permanent network connection obviously supports prompt communication.

1b. Local Communication The George Square system uses the EQUIP middleware, which offers local communication.

2. Service Management The EQUIP software is an event-based system. To the best of our knowledge, it does not offer any kind of service management.

3. Rule-based subscriptions The EQUIP middleware, that is used for the system, is an event-based, distributed tuple space system. In a tuple space system, the producers publish their data in the tuple space. The consumers use rules to retrieve data from the tuple space. The George Square system fulfils our requirements.

4. Server Management The used EQUIP software offers peer-to-peer communication and does not rely on a central server. We conclude that the George Square system to some extent provides means of server management. However, the client needs to be connected to one or several peers that offer the requested data, e.g., map tiles.

5a. Data Classification The George Square system offers some rough data classification, as it distinguishes between sights, recommendations and user avatars. The authors do not go into detail on the implementation, as the research focus lay on sociability. Indeed, the data classification is static and not dynamic.

5b. Data Trustworthiness and Data Quality The described system relies on and lives on user input. A user sees other users' photographs, or the websites other users have visited on a location. However, the George Square system does not provide any means of securing that the users' input really is interesting, or trustworthy. In the same way, this system does not furnish its users with any instruments to assess the offered data. The George Square system does not meet our requirements.

6. Privacy and ConfidentialityThe emphasis of the system lays on the sharing of the users' experiences, ideas and knowledge. Privacy and confidentiality are somewhat opposed to the thought of sharing all information and impressions that a user has gained during her visit, and are not an issue with the George Square system.

4.1.3 GUIDE

GUIDE [5] is a mobile tourist information system, targeted at pedestrians. It was developed at Lancaster University, United Kingdom. GUIDE can be used to create a tailored guided tour to the city, though the tourists have to select the sights they want to visit out of a list. The GUIDE user interface looks and acts like a common web browser. On site, GUIDE provides access to information on a sight. Using this browser, the user can choose a web page containing information about the visited sight. When the user leaves a sight, GUIDE shows them how they reache the next sight on her tour. The tour is rescheduled dynamically, e.g., when a sight closes before the user has visited it, or when the user skips one or more sights on the tour. The user can access some online services like room booking, or buying a cinema ticket through GUIDE. When a user needs to ask something of the tourist information, they can send a message via the GUIDE system.

GUIDE knows two modes, online and off-line. The off-line mode restricts the functionality, for example, the online booking services are not available. Information on sights is only available if it was downloaded and cached beforehand. As the GUIDE system uses WaveLan cells to identify the user's location, location is not available while off-line. Every WaveLan cell is equipped with a cell server that is connected to a web server. The cell servers provide the clients in their cell with informations about the cell, i.e., with information on sights that are located in the cell.

Comparison to the Requirements We now discuss to what extend the GUIDE systems fulfills the requirements defined in Section 3.2:

1a. Prompt Communication While the client is connected to a cell server, GUIDE provides for prompt communication.

1b. Local Communication The GUIDE client mainly consists of a browser and some caching mechanisms. Whenever cached information is requested, the browser and the cache communicate locally. However, this is an exception, and most communication is not local.

2. Service Management The GUIDE system offers a static set of services. Some services, like the tour guide, are available both in the online and of ine mode. Other services like the online booking service are available only in the online mode. To the best of our knowledge, the GUIDE system does not offer any kind of service management that go much farther than the differentiation of online and of ine availability. The GUIDE system does not meet our requirements on service management.

3. Rule-based subscriptions GUIDE is an event-based system, therefore it probably provides rule-based data forwarding. However, to the best of our knowledge GUIDE does not assure that services always subscribe the best data available.

4. Server Management The GUIDE system, as it is described in [5], probably would allow for more than one web server. However, the clients cannot connect to more than one cell server.

5a. Data Classification The sights are arranged in different categories, e.g., “historical sights”. GUIDE provides basic means of data classification, however it does not distinguish between information data and recreation data, as the TIP project does (see Sections 4.1.4 and 4.1.5)

5b. Data Trustworthiness and Data Quality The information stored on the GUIDE web server was provided by tourist information personnel. However, GUIDE offers its users the possibility of browsing web pages that lay outside the tourist information’s responsibility. GUIDE does not offer any mechanisms to ensure trustworthiness neither for the intern information nor for the external web pages. GUIDE users cannot evaluate or rank the information.

6. Privacy and Confidentiality Privacy and confidentiality are no issues with the GUIDE project, as no user data was collected.

4.1.4 TIP 2

The Tourist Information Provider [14], [15] (TIP) is a context-aware mobile tourist information system. It was developed at the Freie Universität Berlin, Germany and the University of Waikato, New Zealand.

The user interacts with TIP 2 through the browser on a mobile device. On first usage, they register with the TIP server and creates her profile. In the profile, the user specifies the topics that interest them, e.g., history, archeology and bicycling. A map shows the user’s current location. While the client is connected to the TIP server, icons on the map indicate adjacent sights matching the user’s profile. When the user visits a sight, the browser shows information on the sight.

TIP 2 is an event-based system. It provides a communication infrastructure, so that the different services can communicate directly with each other. However, messages or events are always sent via the TIP server. At the moment, the basic services are a location service, an information service, a recommendation service and a map service. The services are described in detail in Section 2. When the location service observes a new location, it notifies the TIP server and sends the new location to the TIP server. On the TIP server, the filter engine and the location engine select the sights and informations matching the user’s profile and location and forward them to the mobile client.

Comparison to the Requirements TIP 2 meets some, however not all of the requirements defined in Section 3.2:

1a. Prompt Communication TIP 2 provides prompt communication as long as the mobile client is connected to the server. However, as soon as the mobile device and the server are not connected any longer, prompt communication between services is no longer available.

1b. Local Communication TIP 2 does not offer local communication. All messages are sent through the TIP 2 communication infrastructure relying on the server.

2. Service Management TIP 2 does not offer any means of service management. The services are tightly coupled, and if a service fails, its consumers have to find a replacing service by themselves. However, at the moment nearly all services are unique, i.e., a replacement does not exist. TIP 2 does not offer any infrastructure that facilitates the locating of replacement services.

3. Rule-based subscriptions The TIP communication infrastructure forwards events from the event producer to the event consumer. Nevertheless, the consumer cannot provide its own rules or conditions on the data, therefore TIP 2 does not fully meet our demands in this issue.

4. Server Management TIP 2 does not offer the management of multiple servers, it is a single-server system.

5a. Data Classification Sights are grouped in several topics. However we need a classification at runtime: The same piece of data can be classified as ”recommendation data” when sent to one client, and as ”information data” when it is sent to another client, depending on the sender service.

5b. Data Trustworthiness and Data Quality The current TIP 2 system does not offer any means for users to put in any data, except for the user profile. Data that was put in by TIP personnel was regarded as trustworthy by the designers therefore services that help ensure the trustworthiness of data were not planned. TIP 2 does not provide mechanisms that help ensure the quality of the stored data, or its evaluation.

6. Privacy and Confidentiality All user data are stored on the server, more precisely in the TIP 2 database. They are neither shared with or distributed to other servers, as TIP 2 handles only one server at a given time, nor are the user profiles shared with or distributed to other clients. However, if someone eavesdrops on the transmission between server and client, she may soon detect the main interest topics.

4.1.5 TIP 3*

TIP 3* continued the development of TIP. In a first step, [17] proposed the concept of a service-oriented architecture for TIP and introduced a new component to the architecture, the broker. The functionality of TIP 2 was maintained. The design is still a client-server architecture, where several TIP 3* services run on the server, such as an information service. The TIP 3* client provides a set of services, e.g., an information service, a location service or a map service. When a service has been started, it registers with the broker, advertises the provided information and subscribes to the information it needs. Services do not communicate directly with each other, but publish their messages to the broker. The broker forwards messages to the respective subscribers.

A small prototype of TIP 3* was implemented by Michel [26]. Michel focused on the introduction of a caching service to reduce duplicate downloads. The prototype implements an event-based communication middleware, the broker. However, no service management has been implemented yet. TIP 3* does not solve the problem of starvation: a subscriber can wait for its subscribed data indefinitely, if there is no respective publisher.

Comparison of TIP 3* to TIP 2 TIP 2's functionality is largely preserved, even though the underlying software architecture changes. The shift to a new architecture brings along the need for re-implementing the TIP services as the existing TIP 2 services cannot be used. A translating service is needed alternatively. Nevertheless, from a functional point of view the main services will be an information service, a location service, a recommendation service and a map service. A caching service has been added to reduce downloads from the server. The TIP user interface was not changed.

Discussion of TIP 3* The TIP 3* architecture is not completely event-based, even though services react to incoming location events. The remaining communication between co-operating services indeed uses the event notification middleware, however the communication between co-operating services resembles communication between services in a service-oriented architecture. TIP 3* services are loosely coupled. However, once two services are coupled, they stop looking for other cooperation partners. A service may well starve when its cooperation partner vanishes for some reason. TIP 3* took first steps towards service-orientation, although some service-oriented principles like the service contract were not considered.

Comparison to the Requirements We defined our requirements on the new TIP version in Section 3.2. We now analyse how TIP 3* meets these requirements:

1a. Prompt Communication TIP 3* provides for near-in-time communication. The broker filters and forwards data as soon as it receives them, so that subscribers receive their respective data soon. We consider that TIP 3* meets our requirement in this point.

1b. Local Communication In TIP 3*, services communicate indirectly via the local broker. The broker forwards data to the respective subscribing services. If a local service subscribes to data from a local publisher, the broker forwards the data directly to the subscriber, without a detour via the server. TIP 3* meets our requirement.

2. Service Management In TIP 3*, there is no service discovery process. A service subscribes to all the event types needed. If a necessary event type is not available, the service cannot function. On the one hand, this makes the service discovery process that is typical for a SOA redundant, and facilitates the broker. On the other hand, a service is useless if it cannot provide its functionality. The service itself should be notified that necessary data is not available. TIP 3* does not meet our demands in this aspect.

3. Rule-based subscription TIP 3* offers simple means of rule-based data forwarding: the broker forwards events from the event publisher to one or many event subscribers. Indeed, subscribing services cannot inform the broker about the rules selecting the appropriate events. TIP 3* does not fully meet our demands in this aspect.

4. Server Management TIP 3* can handle several servers – it can connect to and use services from several servers at once.

5a. Data Classification TIP 3* offers basic means of data classification, as the events are categorised. However, data cannot be tagged as information and recommendation data at the same time, it has to be sent twice in an information event and a recommendation event.

5b. Data Trustworthiness and Data Quality TIP 3* does not offer mechanisms to ensure and enable data trustworthiness. Data Quality is not an issue with TIP 3*.

6. Privacy and Confidentiality Privacy and confidentiality were not an issue with TIP 3*. User profiles are stored on one or more servers. TIP 3* clients are not able to access them. As with TIP 2, if someone intercepts the transmission, she probably will be able to reconstruct at least the user's interests.

4.2 FACTS – A Rule-based Middleware for Distributed Data Processing

In this section we discuss FACTS, a rule-based middleware for sensor networks as an example for a middleware for rule-based systems.

FACTS [31] is a rule-based middleware architecture for wireless sensor networks. It was developed at the Freie Universität Berlin. It offers an event-based programming interface to application programmers. An application can communicate with other applications through the FACTS middleware. Applications have to provide a rule set, so that they can select the information needed. Even though FACTS uses rules in a different way than TIP 3 does, it was the main in uence for our concept of rule-based communication.

The FACTS architecture consists of three main constituents: *facts*, *rules* and *functions*. Information is stored as facts, irrespective of what the information itself is about. The facts are stored in the fact repository. Every fact has an owner node, an unique identifier, a non-unique name and a set of properties.

Rules express algorithms, i.e., they define how the application responds to external events. A rule consists of a set of conditions and a set of statements. The conditions specify when a rule is triggered. The statements declare what should be done by the rule. The rule engine examines the rules periodically. It triggers a rule when the rule's conditions are satisfied. The functions call a firmware or operating system procedure. Functions are called by the Rule Engine.

Comparison to the Requirements We compare FACTS to the requirements defined in Section 3.2:

1a. Prompt Communication FACTS provides prompt communication through the wireless communication interface.

1b. Local Communication Services, or event producers and consumers, communicate through the change of facts. The facts are stored in the local facts repository situated on the node. Two or more local services can thus communicate through shared facts in the local repository, so that local communication is available.

2. Service Management Service management in the notion that another service – or rule – with the same functionality should be chosen as a substitute when a service fails is not implemented. The rule engine checks the applications conditions periodically, and triggers their rules when all conditions are met. However, the rule engine does not search for replacements of failing applications.

3. Rule-based subscriptions The designers of FACTS found that event-driven programming re ects the changing and dynamic nature of sensors' environments, i.e., the real world.

   FACTS provides rule-based subscriptions.

4. Server Management FACTS does not know off any servers. On the other hand, it is intended for peer-to-peer networks.

5a. Data Classification Data – or facts – can be classified through their properties. However, FACTS does not provide an ontology, or other means towards data classifying. We therefore consider FACTS to meet this requirement partly only.

5b. Data Trustworthiness and Quality FACTS is intended for a sensor network, where sensors produce data. A sensor may be prone to errors, i.e., it's data is not necessarily correct. However, FACTS does not offer any mechanisms to ensure data trustworthiness. We think that this issue could be a task for an application that is implemented with the FACTS middleware. FACTS does not offer any mechanisms to ensure data quality or the evaluation of data.

6. Privacy and Confidentiality Because FACTS is a middleware for wireless sensor networks, it does not address issues as privacy or confidentiality.

4.3 Summary of Related Work

Table 1 summarises to which extent the different approaches and systems meet our requirements. Our first requirement is prompt communication (Requirement 1a), i.e., that communication between two partners takes only a certain amount of time. This requirement is more or less met by nearly all related works and approaches we examined. However we have to add that we simply analysed whether the communication would be likely to be completed within a given period of time. We did not measure this. The next requirement, for local communication (Requirement 1b), is also met by most of the examined systems.

Requirement 2, service management, is met by the GUIDE system, TIP 3* and the event-driven architectures and FACTS. Most of the analysed works do not provide for rule-based subscriptions (3), indeed only FACTS and the George Square System let the subscribers or event consumers provide rules selecting the interesting events. The + indicates that the system provides for rule-based event or data forwarding, however, it lacks the feature that the event consumer provides the rules that select the events. Server management (Requirement 4), i.e., the ability to handle more than one server, is met partly by the George Square System and GUIDE. The designers of TIP 3* foresaw the need, although they did not implement this feature.

The most important point is if and how services are managed. Service management describes that a service is notified when a subscribed data type is no longer available. Services should be notified when new data types are made available, enabling them to select new co-operation partners if necessary. None of the examined systems meets our demands completely. Indeed, service-oriented and event-driven architectures give attention to that matter, out of different points of view. In the next section, we discuss service-oriented architectures, web services and event-driven architectures.

Some way of data classification (5a) is provided by nearly all examined systems, however, no system is able to change or complement the classification on the y. The issue of data trustworthiness and quality (5b) is not addressed by any system. Privacy and confidentiality (6) is solved to some parts by some systems, TIP 2 and TIP 3*.

The three requirements that we identify as the most important are (2) service management, (3) rule-based subscriptions and (4) server management. The different systems examined only partly meet our requirements to rule-based subscriptions. FACTS is the only system that partly implements service management in the meaning that co-operating software applications are notified when a new possible co-operation partner connects, or disconnects. Most of the analysed systems fail to meet our requirements as to server management. TIP needs an architecture that combines the functionality from TIP 3* with the enhanced exibility that service management and rule-based subscriptions provide. The new TIP system also needs means to connect to several servers at once.

Data classification and trustworthiness (requirements 5a and 5b) and privacy and confidentiality (6) are important issues, especially in a mobile tourist information system that relies on content input from its users. Users feel the importance of privacy and confidentiality, so that their private profiles and user data are not accessible by anyone. Although these issues are interesting and important, we cannot address theses issues in this paper. This is a task for a future implementation of the TIP 3 design, but a model cannot solve them.

Table 1: Related Work and our Requirements: ++ = completely met; + = mostly met; − = partly met; −− = not met.

Requirements	Related Work					
	Dynamic Tour Guide	George Square System	GUIDE	TIP 2	TIP 3*	FACTS
1a. Prompt Communication	++	+	++	+	++	++
1b. Local Communication	++	++	−	−−	++	++
2. Service Management	−−	−−	−	−−	−−	+
3. Rule-Based Subscriptions	−	++	−	−	−	++
4. Server Management	−−	+	−	−−	+	−
5a. Data Classification	+	+	−	+	+	+
5b. Data Trustworthiness	−−	−−	−−	−−	−−	−−
6. Privacy and Confidentiality	−−	−−	−−	+	+	−−

5 Service-oriented and Event-driven Architectures

In this section, we present the main features of service-oriented and event-driven architectures. TIP 2 was conceptualised as an event-driven system. However, the initial implementation [25] followed a request-response design. With the TIP 3* middleware first steps towards an event-driven architecture were made. The new design of TIP 3 brings together service-oriented design and event-driven architectures.

5.1 Service-oriented Architectures

This section presents the basic principles behind service-oriented architectures (SOA) and their main components. TIP is a modular system, where the different services co-operate with one another. The TIP map service displays a sight's location on the map. The map is centred around the user's current location. The map service co-operates both with the TIP location service and the TIP information service. The location service informs the map about the current location, and the information service informs the map service about a sight's location. The question of how TIP provides the map service's functionality can be answered in several ways. One possibility is to design it as a monolithic service that only can co-operate with the information and location service. However, the map service should be able to display more than icons on sights: a tourist might be looking for a hotel room, a restaurant or even for a at to rent or buy. All this information could be supplied by one all-encompassing information service or by several services providing information data. If the map service is designed as a monolithic service that can only co-operate with one other service, it cannot display points of interests provided by other services. This could be solved by several monolithic map services, where each map service displays another kind of information. Another way to solve the problem is to re-design the map service. Functionality and data would be split – the map service could co-operate with services that provide two data types: location data for points of interest and icons the map should display on the location. The map service's functionality is to display the icons on a given location.

Service-orientation is an approach to split a problem, and the solution to this problem, into several independent and autonomous logical entities, or services. An SOA consists of services that encapsulate functionalities. An SOA does not depend on an operating system, a programming language or a certain transmission protocol. We now discuss the main principles in SOA: (1) service re-usability, (2) service contracts, (3) loose coupling, (4) service abstraction, (5) service composition, (6) service autonomy, (7) service statelessness and (8) service discoverability (see [7]).

(1) Service re-usability means that one service's interface should not be tailored to another service. Service interfaces should be designed so that each service may interact with many other services, especially when the underlying service logic could be needed by several services.

(2) The service contract binds a specific functionality to a specific service. It defines the service's endpoint, the service's operation, i.e., its functionality, the in- and outgoing messages and the service's rules and properties. It also defines which functionality a message corresponds to.

(3) Loose coupling means that services are not tightly bound to one another. It prevents that services depend on each other. Services should be able to collaborate with several services, or to use several services' functionalities.

(4) Service abstraction means that a service encapsulates its underlying logic. The service requester does not need to know how the service performs its functionality. Especially, the requester does not know whether the operations were completed by the service provider, or if the provider on

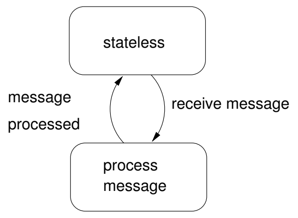


Figure 4: A stateless service. Upon receiving a message, the stateless service is stateful, until it has finished processing of the message.

its part used other services.

(5) Service composition implies that a service can use other services to provide its functionality. A service provider may request other services, and combine them with its operations into a new service. Service composition requires service re-usability, and at the same time it enhances it.

(6) Services are autonomous or self-governed and do not depend on another service. However, the service designers have to balance service autonomy against service granularity and against service composability. Service granularity and service composability can bring forward a service's dependence on other services, .

(7) Services are stateless, unless they receive, send or process a message. When a service receives a message, it is stateful, until the message has been processed, as shown in Figure 4. Service statelessness is useful to prevent that the service is blocked by a task, e.g., while waiting for a requested service. Statelessness enhances service re-usability and system scalability. To acquire statelessness, the messages between services have to be almost self-describing. They need to carry some information for the receiver as to how the message content should be processed, i.e., carry some kind of state information.

(8) Services are discoverable. This means that an SOA provides ways for a service to know about, locate and contact other services. This helps to avoid redundant services and enhance service re-usability. Service discoverability can be achieved through UDDI service registries, or service repositories.

An SOA consists of cooperating services and a service repository. A service is offered by the *service provider* and requested by the *service client*. The service repository is a central point in the SOA. Services belong to a service provider that is called their owner. Services have a simple, well defined interface that helps avoiding artificial dependencies, and enables loose coupling [34].

Services register with a repository, thus advertising their services. Whenever a service needs another service's functionality, it can find this service via the repository. A Universal Description Discovery and Integration (UDDI) registry or service repository provides

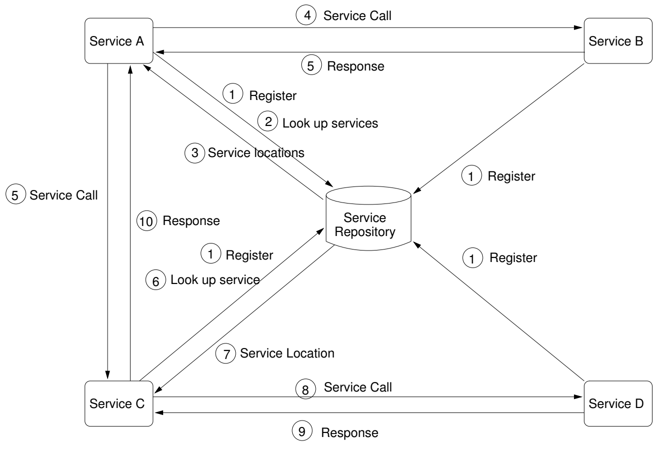


Figure 5: Service interaction in an SOA

- **White Pages** providing general information on service providers (business), such as the business name, description, contact data – the White Pages answer the question "Who am I"?
- **Yellow Pages** classifying the service provider, or the services – the Yellow Pages address the question "What do I offer"?
- **Green Pages** offering technical information on the services, i.e., the technical specification and the service address – the Green Pages provide instruction on "how to call me and my phone number"

In an SOA, the service repository publishes the yellow pages in which service providers announce their services and the offered functionalities. The white pages can be referred to when general information on the service provider is needed. The green pages are useful for programmers of other services that should co-operate with an existing service. White and green pages are stored by the UDDI service repository. Whenever a service disconnects, its customers – i.e., the service requesters – have the possibility to look up another service provider in the repository.

However, if a service provider fails for some reason, and does not disconnect properly, its customers can wait indefinitely. The SOA approach does not require the customers to stop waiting. In a worst case scenario, they will wait, although other service providers offering the same functionality may be available. When a service has found another service providing the expertise needed, the requesting service will typically collaborate with the service provider, without checking whether a new, possibly better service provider has registered with the repository in the meantime.

Figure 5 shows the example interaction between four services and a service repository. In Step 1, the services register with the service repository. Service A needs external expertise. It asks the service repository which service or services can solve its problem (Step 2). The repository responds with a message containing the service locations of services B and C (Step 3). The repository recommends these services because they registered with the service repository, and their service contracts fit service A's demands. In our example, service A calls service B and service C, in Step 4 and 5. Service B replies in Step 5, while service C cannot reply immediately. Service C submits a service lookup to the service repository in Step 6. The service repository replies with service D's location (Step 7). In Step 8, service C calls on service D. After service D has replied (Step 9), service C can reply service A's service request (Step 10). Whenever service A needs the expertise provided by services B and C from now on it will call them directly, without asking the service repository if other services are available.

In an SOA, the typical communication pattern mostly is call-response, i.e., the requester calls a service and waits for a response. SOA mostly apply synchronous communication.

The messages between services should describe the calling service's problem, i.e., what it wants the called service to do, but not how to do it [34]. An SOA is easier to expand, and it is easier to introduce new services if the message are easy to understand. The message grammar should be extendable and allow for additions. Software designed following the SOA principle can provide a message grammar to assure extendable messages.

The main concepts of service-oriented architecture address issues of how software applications co-operate. More technical issues such as prompt or local communication are not necessarily required of an service-oriented architecture. For example, a service provider does not have to send its reply immediately after it has finished its computations. Services communicate and co-operate through a common network. Service-oriented architectures do not define how this network is organised. Although services located on the same computer could communicate locally, they could also use an intranet for communication.

The UDDI repository provides information for service location. Whenever a service disconnects, its customers – i.e., the service requesters – have the opportunity to look up another service provider in the repository. However, if a service provider for some reason fails and does not disconnect properly, the SOA approach does not require the customers to stop waiting after a certain amount of time. In the worst case, they will wait forever although other service providers that offer the same functionality are available. Service-oriented architecture does not meet our requirements on service management.

Services in a service-oriented system are loosely coupled. They communicate directly with one another, without intermediate, and do not use rule-based subscriptions.
In an SOA, the issue of server management addresses the question of how the service repository is organised. It may be distributed, or there may be more than one repository, or there is only one service repository. Hence, this issue is not solved, or addressed, by general SOA. Service-oriented architectures need some way of service classification, otherwise they could not offer a really working service repository; at least the service descriptions would be inconsistent, so that service consumers possibly would not find the service provider they were looking for. While service-oriented architectures offer reasonable means of service classification, they do not provide for data or event classification.
Data trustworthiness is not an issue of service-oriented architectures in general. Designers could conceptualise a service-oriented system that comprises data trustworthiness. Data trustworthiness may be provided for through trust services in an SOA. Similarly, data quality is not addressed by service-oriented architectures in general. Data quality and evaluation are not addressed by SOA in general. They may be offered by services that are part of a particular SOA. Privacy and confidentiality do not pose an issue with SOA per se. However, they may be important in a certain service oriented system.

5.2 Web Services

Over the last years, Web Services have become an important way of integrating web-based applications. Web Services often are mentioned in an SOA context, nevertheless they are not automatically designed following the SOA paradigms. A Web Service is offered by the service provider to customers through the internet, or any other network. The W3C Working Group [33] defines

> A Web Service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web Service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialisation in conjunction with other Web-related standards.

A Web Service is self descriptive, i.e., it provides a description of its functionality and information on needed parameters. A Web Service description contains information on a service's public interfaces, on what data types are needed in order to communicate with the service, the transport protocol and the service location, i.e., the address where the service can be called. A Web Service description is typically written in the Web Service Description Language (WSDL). WSDL is independent of a platform or a programming language.

Web Services are modular. They do not depend on an operating system or a certain programming language, but on an usable and accessible communication platform like the internet.

As in an SOA, Web Services can use UDDI repositories for the service discovery process. However, unlike services in a service-oriented system, Web Services can be tightly coupled as well as loosely coupled. They do not necessarily implement a service-oriented architecture.

Figure 6 illustrates how Web Services interact. Web Service A requests Web Service B (Step 1). Web Service B receives the request in Step 2. B processes the request and replies (Step 3). Although Web Service A and B are located on the same machine, the reply is transferred through the network. At the same time, Web Service A sends another request to Web Service C (Step 4). It then receives B's answer in Step 5. C receives and processes the request and replies to it

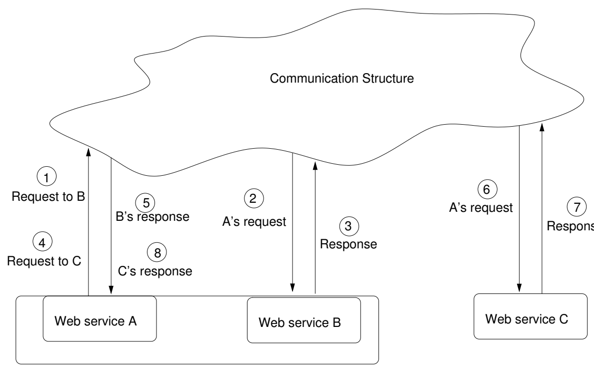


Figure 6: Web Service interaction

(Step 6 and 7). Some time later, Web Service A receives C's answer (Step 8).

Unlike services in a service-oriented architecture, Web Services are not necessarily loosely coupled. A Web Service can use remote procedure calls to invoke other Web Services. Similar to SOA, Web Services may enhance service re-usability. If a Web Service implements a service-oriented design, it will have a service contract, however not every Web Service has a service contract. Web Services may, but do not have to abstract from the underlying logic. Neither are Web Services necessarily stateless. A Web Service can discover another Web Service through a request to a Web Service repository, however, the co-operating Web Service's address and technical details as to how the Web Service should be contacted can also be stored by the calling Web Service.

However, Web Services face several problems. One main problem concerns the subject of service discovery and integration. Automated service discovery and integration can be difficult, as the service description can be interpreted ambiguously [29, p. 42]. Indeed, service discovery, location and invocation are mainly handled by humans and are difficult to automate [35, p.64]. During the service discovery, the user may not find the service that meets the requirements best simply because they do not know enough about the service [24, p. 162]. If a service disconnects, its partners have to search a replacement by themselves. Web Services communicate through a network whose characteristics, such as transmission times, can be unknown and unpredictable.

Web Services may, but do not necessarily have to use prompt communication. As with service-oriented architectures, a Web Service could transfer its reply to a request immediately, or it could wait for some time. Web Services usually are distributed on the network. Even if two co-operating Web Services were hosted on the same server, they would use the network for communication, and not communicate locally.

As in service-oriented architectures, Web Services may use UDDI repositories for service discovery. If a service disconnects, its partners have to look for a replacement by themselves. The kind of service management we identified where co-operating services are notified if a service is disconnected is not an issue of Web Services.

Co-operating Web Services communicate directly with each other. If two Web Services communicate through an intermediate, for example, a Web Service that translates the output from one Web Service into the input format the other service can process, this intermediate Web Service could of course use rules to determine the receiver. However, rule-based subscriptions are not an issue of Web Services in general.

Web Services may use a service repository for the localisation of other Web Services. If they use an UDDI repository, our requirement on server management addresses the issue of how this repository is managed. On the other hand, a web service can locate its co-operation partner through a built-in address. Similar to service-oriented architectures, our demands on server management are not met.

Co-operating Web Services need some kind of shared or common data classification, otherwise they could not co-operate. Data trustworthiness and data quality are not issues with Web Services as such. A web service may provide for data trustworthiness, or implement a rating mechanism that helps to ensure data quality. Web Services can provide privacy and confidentiality. They can also contribute to the violation of privacy and confidentiality depending on the implemented functionality.

5.3 Event-driven Architectures

Event-driven architectures (EDA) are another approach to distributed computing where operations are divided into independent logical units that collaborate with one another. The communication between collaborating applications is indirect and exible.

An EDA consists of event publishers and event consumers. The publishers and consumers do not communicate directly with each other but indirectly through an event notification system or event manager. Before we go into further details on EDA, we will define some terminology.

An *event* is something interesting that happens and can be observed [28]. The datum that describes the event and represents it in a computer system is defined as *notification* or *event notification*. However, we use the term event when we refer to the event notification. We are convinced that the intended meaning will be apparent. We expanse the definition of event by the notion of an *absence event*. An absence event occurs whenever something was expected to happen during a specified time frame and did not happen, e.g., a theatre ticket was bought, however, the payment was not received during the following five minutes, and the purchase was cancelled.

An *event producer* is software that publishes an event. A *consumer* is software that subscribes to and consumes events. An application can be both publisher and consumer of events. Publishers advertise the event types with the *broker*, or *event manager*, and publish the events to the broker. Consumers subscribe to events at the broker. The broker filters incoming events and forwards them to the subscribers.

The communication between producers and consumers is asynchronous.

Figure 7 illustrates the transmission of events. In a first step, event consumer 1 connects to the event manager and subscribes to event A. The event manager now creates a *filter*

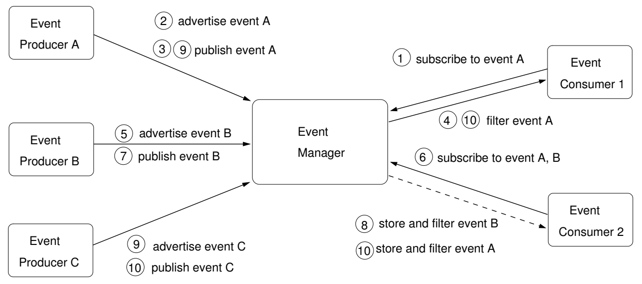


Figure 7: Event Producers and Consumers in an Event-driven Architecture

from the subscription. The filter specifies that events of type A should be forwarded to event consumer 1. In Step 2, event producer A connects to the event manager and *advertises* that it publishes event A. Event producer A then publishes event A in Step 3. The event manager applies the filter to the incoming event and forwards it to event consumer 1 in Step 4. In Step 5, event producer B advertises event B. Event consumer 2 then subscribes two events, event A and B (Step 6). Again, the event manager creates a filter from the subscriptions. Event producer B publishes event B (Step 7). Event consumer 2 is unreachable when the event manager tries to filter event B, so that it stores the event and filters it when B is reachable (Step 8). In the following Step 9, two things happen at once: event A is published, and event producer C advertises event C. In Step 10, the event manager filters event A, i.e., it forwards it to event subscriber 1 and stores it for the disconnected event subscriber 2. At the same time, event producer C publishes event C.

EDA can easily adapt to asynchronous or unpredictable environments, as an event consumer does not depend on a certain event producer. Event-driven design and development have several advantages, the most important ones are that new and existing applications can easily be (re)composed and reconfigured, and that existing applications and components may easily be re-used. EDA do not comprise services. However, if we – in this case – regard the applications that form an event-driven system as services, the event-driven approach does not involve replacement of a failing or disappearing application. EDA do not necessarily rely on a server. The event manager, an EDA component that could be compared to a server, can be decentralised. Therefore, EDA does not meet our requirements in this respect. The events need to be classified in some manner, so that producers and consumers can communicate, in the first place. The actual decision of how the events are classified is left to the system designers. EDA meets this requirement.

5.4 Comparison of SOA and EDA

Event-driven and service-oriented architectures are two different architectural styles for the co-operation of applications. We compare them in this section, and elaborate on their strengths and weaknesses. Table 2 summarises the results of our discussion.

In event-driven systems, applications react to incoming events. Consumers and producers are extremely loosely coupled, nearly decoupled. They communicate indirectly with each other via the event manager. This advances the exibility of the whole system. Services react to a service request from another service in a service-oriented system. They communicate directly with each other. Even though they are loosely coupled, service-oriented systems are not as exible and do not adapt to changes in the service environment as event-driven systems. We argue that event-driven systems are more suited for a highly dynamic environment of changing services, such as the TIP system.

In service-oriented systems, services share a service contract. This provides for a certain degree of reliability – services know who their co-operation partners are. Event-driven architectures do not provide for something like a service contract per se. Applications in an event-driven system have to rely on the fact that the event notification middleware only allows reliable publishers. The service contract of service-oriented architectures provides several advantages that event-driven architectures lack. TIP would benefit from the inclusion of service contracts.

The service discovery process in a service-oriented architecture uses a service repository where services look up collaboration partners. In event-driven systems, event publishers simply publish their events to the event manager. The event consumers subscribe to their needed events at the event manager even if there is no event publisher who publishes this event. Both approaches have advantages: in an SOA, a service can only request services from registered publishers, however, once it has selected a co-operation partner it will not necessarily select another. In EDA, the event manager forwards events to the respective subscribers. Even if a publisher registers with the event manager after a subscriber has registered, the publisher's events will be forwarded to the subscriber.

We propose that subscribing services register for the event types needed if they are available. Services may specify certain conditions on the publisher and the events, such as the quality of data, or a specific data format. Subscribing services are notified whenever a new publisher registers with the middleware, so that they may adjust their subscriptions. This ensures that subscribers always subscribe the best events available.

Both service-oriented and event-driven architectures facilitate service abstraction. In an EDA, the event consumers do not necessarily know which event producer actually publishes the events. In a service-oriented architecture, services know their co-operation partners, however, they do not know how their co-operation partners solve the problem. Service composition is a main principle of SOA, but service composition is feasible with EDA, as well. Service-oriented systems enable the autonomy of services. In an EDA, some event producers may be autonomous, however, the event consumers depend on event producers. In EDA, applications are stateless, as services should be in a service-oriented architecture.

	needed by	characteristic of	
	TIP 3	EDA	SOA
event-based communication	++	++	−−
extremely loose coupling	++	++	−−
exibility	++	++	−−
communication style	indirect and asynchronous	indirect and asynchronous	direct and synchronous
reliability	++	−−	++
service discovery process	−−		++
service re-usability	++	++	++
service abstraction	++	++	++
service composition	++	++	++
service autonomy	++	++	++
service statelessness	++	++	++

Table 2: Comparison of the characteristics of event-driven and service-oriented architectures and the demands of TIP 3. ++ indicates that the architecture provides the characteristics or that TIP needs it. −− shows that the respective architecture does not offer the characteristic. - indicates that the characteristic can be achieved with the respective architecture, but that it is not built-in.

5.5 Summary of the Comparison

In this section we have discussed service-oriented architectures, Web Services and event-driven architectures. Service-oriented architectures and event-driven architectures share several characteristics that the TIP system would benefit from. The following section shows how we have integrated service-orientation into an event-driven system.

6 Architecture and Design of TIP 3

TIP consists of several co-operating software applications. Event-driven and service-oriented architectures are two approaches as to how software applications can co-operate. Both approaches have their advantages and disadvantages as we discussed in Section 5. A composition of the two combining the advantages of service-oriented architectures with those of event-driven architectures meets the requirements on a future TIP architecture. This section discusses the architecture and the new design for TIP 3.

6.1 The Architecture of TIP 3

The TIP 3 architecture is a peer-to-peer architecture. It is shown in Figure 8. A TIP 3 peer may either be a client peer that exchanges events with other peers, or a server peer. We refer to the server as *server peer* and to the client as *client peer* to distinguish between server and client. The client peer provides services and user interface to the services. The client peer provides a location service. The server peer provides services, such as the information service, or other services that access the TIP database. When the client peer is connected only to a server peer and not to other client peers, the architecture resembles a client-server architecture.

A TIP 3 peer hosts several services. The services communicate indirectly via the local broker. They are administered by the observer. Auxiliary services offer functionalities such as converting data from one data format to another. Every service registers to the local broker, i.e., it advertises the events it publishes, and subscribes to the event it needs. Later on, it publishes its events to and receives events from the broker. The broker provides the functionality of a publish/ subscribe middleware. It filters the events and forwards them to the respective subscribers. The observer observes the connection between broker and services and other brokers respectively. The observer notifies the broker when a service or peer disconnects. It also re-evaluates and adjusts a service's subscriptions if necessary.

TIP 3 services are 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*. 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 different publishers and

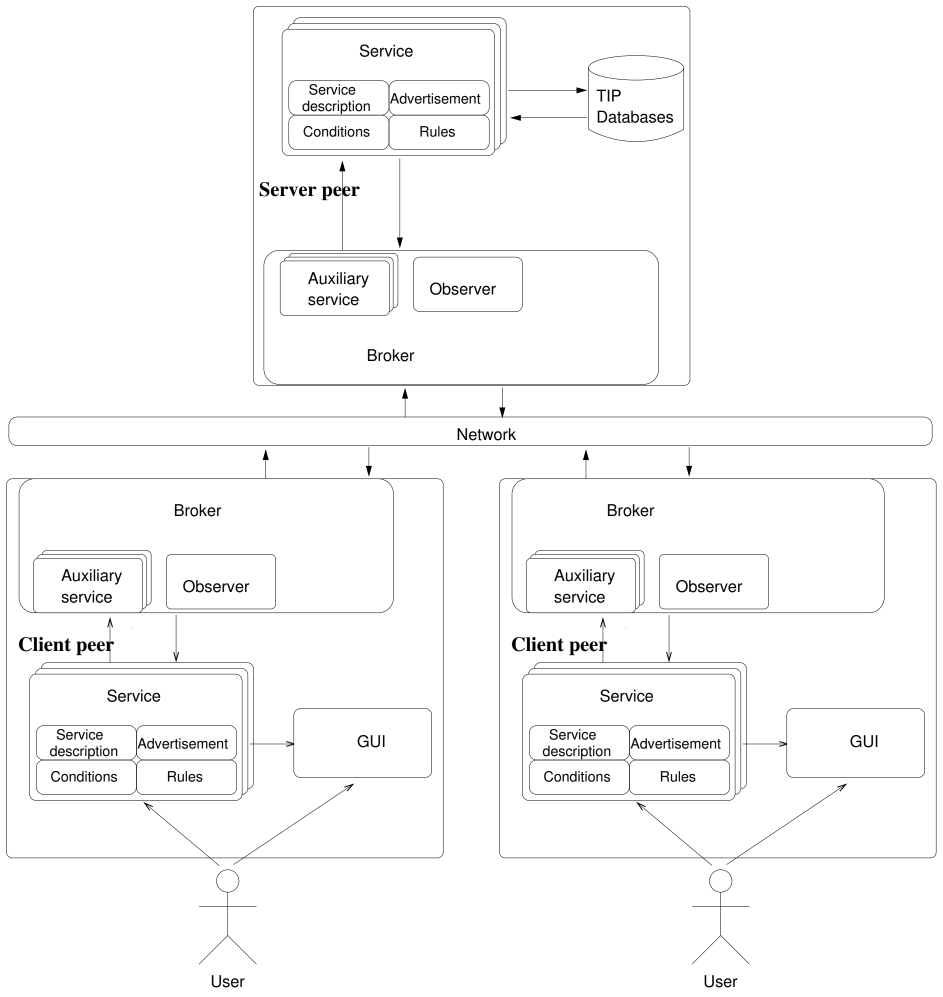


Figure 8: The TIP 3 architecture. The main components are the services, the TIP databases and the broker. Services provide a service description, an advertisement, a set of functional conditions and of subscription rules. The broker provides auxiliary services and the service observer. The observer's task is to evaluate the service rules and conditions, and to monitor the availability of services and external brokers. The auxiliary services convert between data formats, e.g., from metric to imperial units of measurement. The TIP databases are located on the server peer.

to treat them accordingly. A location event belonging to the "information event" category is treated differently by the map service than a location event belonging to the "location event" category, for example.

Figure 8 shows the TIP 3 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 peer accommodates services, the broker and the observer. The TIP 3 server peer additionally accommodates the TIP databases. The TIP 3 client peer does not necessarily accommodate a database. It provides a user interface, so that the user can interact with the TIP 3 services. Most services on the TIP 3 client peer provide a graphical user interface, so that the user can interact with the TIP system, e.g., click on an information icon on the map. The main difference between client peer and server peer is that the client peer does not provide a database.

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 of 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 define 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. Rules are grouped into groups of three. 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 subscribes 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 on 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 on 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.

6.2 The Design of TIP 3

This section introduces the design of TIP 3. We then brie y discuss the design constraints.

6.2.1 The Design

The main actors in TIP 3 are the brokers, the observers and the services. The actors take part in several interactions: When a service is started, it registers with the broker. Publishing services advertise their events to the broker and then publish events to the broker. The broker filters events and forwards them to subscribing services. Subscribing services transmit their functional conditions and their subscription rules to the broker. The observer evaluates a service's functional conditions and stops the registration process if the conditions are not satisfied. The observer also evaluates a subscriber's subscription rules and subscribes the service to events. A service disconnects and unregisters from the broker. The observer may also unregister a service from the broker if this service is disconnected without unregistering.

When two TIP 3 peers connect, brokers interact with other brokers. They advertise their events and subscribe to the other broker's events. They forward events to one another and receive events from remote brokers. The observer notifies the broker when a remote broker disconnects, so that the remote broker's advertisement and subscriptions are removed. The observer re-evaluates the subscribing services' subscriptions when a remote broker has connected or disconnected, so that services always are subscribed to the best available events.

Service registration When a service is started, it registers with the broker located on the same device. We call this broker the *local broker*. During the registration process, the service first publishes its service description to the broker. When a service subscribes to events, its functional conditions are evaluated in the next step. This ensures that a service is only registered if the absolutely essential event categories are available. The event categories have to be known to the service providers so that the service conditions can be formulated properly. If the conditions are satisfied, the service is subscribed to the best possible event types. For this purpose, every subscribing service brings a set of subscription rules. A subscription rule specifies the characteristics of an event type the service needs to subscribe to. It also delivers information about the publisher. In case the service can co-operate only with a special, named service the subscription rule directly identifies the publisher. When a subscribing service also publishes data, it publishes its advertisement to the broker after the evaluation of the subscription rules. The registration process of a subscribing service is then completed.

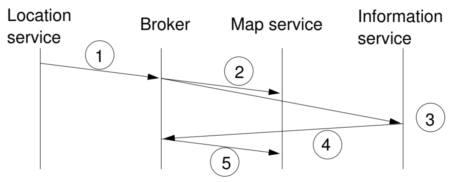


Figure 9: Interaction of services in TIP 3. When the location service publishes a location event to the broker, the event is forwarded to the subscribers. The information service processes the location event and publishes an information event that is filtered and forwarded to the map service. For reasons of simplification the broker middleware in this figure unites both the local and remote broker as well as the network between them.

When the service only publishes events, it transmits its advertisement directly after the service description. The registration process is completed directly after the advertisement.

Interaction between services TIP 3 services communicate and co-operate with each other. Figure 9 illustrates how TIP 3 services interact. When the location service publishes a new location event to the broker in Step 1, the broker filters it and forwards it to every subscribing service (Step 2). The subscriber, e.g., the map service or information service, reacts to the incoming event according to the designers' definition. The map service may centre the map at the received location. The information service may process the location in Step 3 and may publish an information event (Step 4). The map service subscribes to information events. Therefore the broker filters the information event and forwards it to the map service in Step 5, enabling the map service to display sights on the map.

Service deregistration When a service disconnects from the broker, the broker removes the service's advertisement and subscriptions. Services may announce to the broker that they disconnect. If the service disconnect without announcement, the observer notifies the broker. The broker then removes the advertisement and subscriptions. The subscription rules of subscribing services are re-evaluated and subscriptions are updated if necessary.

Broker registration Services on the TIP peers communicate indirectly via their respective broker. When two brokers connect (see Figure 10), they first exchange advertisements, i.e., they exchange information about the event types that local services published and also the service descriptions from the local services. In Step 2, each broker examines the received advertisement and notifies the observer. The observer evaluates the subscription rules from the services (Step 3) and if necessary adjusts the service subscriptions (Step 4). The broker then submits its subscriptions to the other broker and receives the subscriptions from the other broker (Step 5). From now on, published events are forwarded to the external broker if it has subscribed to them.

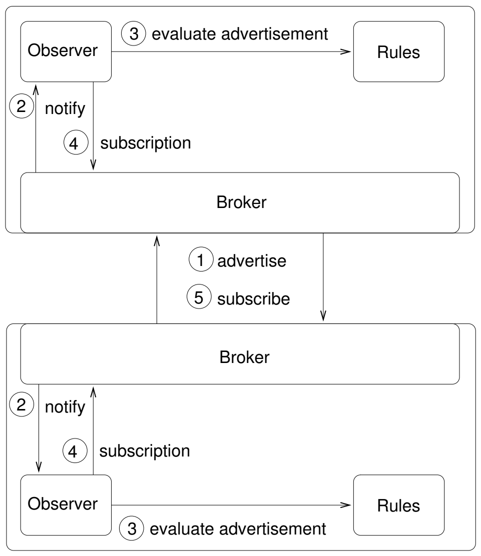


Figure 10: When two brokers connect, they first advertise their local events to one another. The observers then evaluate the advertisements and re-evaluate the local services' subscription rules.

Interaction between brokers When a broker has connected to another broker and both brokers have subscribed to event types from one another, one broker forwards incoming events to the other if the latter has subscribed to them. Brokers may forward events in a multi-hop peer-to-peer network. This scenario could be further explored.

Broker deregistration Broker A may announce to a connected broker B that it disconnects. Broker B then removes broker A's advertisement and subscriptions. When broker A simply disconnects without a prior announcement, broker B is notified by its local observer. Broker B removes broker A's advertisement and subscriptions. The subscription rules of subscribing services are re-evaluated and subscriptions are updated if necessary.

User interaction with TIP 3 There are two ways the user can interact with the TIP 3 system: active or passive. (1) They can actively use the TIP user interface, e.g., click on icons or click on the map. When the user clicks on an information icon on the map, the map may publish an information event to the broker. Thus, user input can result in a new TIP event. (2) When the user moves, the location service publishes a new location event to the broker. We call the second kind of interaction "passive interaction" because the user does not actively interact with the TIP 3 user interface, but simply moves.

6.2.2 Design Constraints

The design of TIP 3 does not address issues as how services may contact the broker, or how a broker connects to another broker. We disregarded these technical aspects of TIP. We also assume that the event types, and event and service categories and their respective descriptions are known to the actors. These simplifications are made for reasons of modelling. We revisit them in Section 9.

6.3 Service-orientation and Event-drivenness in TIP 3

TIP 3 integrates service-oriented aspects into an event-driven architecture. The TIP 3 services are extremely loosely coupled, although loose coupling is possible if necessary. The communication between services, between services and broker and between brokers is event-driven, as services react to incoming events and publish events. This enhances the exibility of the TIP 3 system: de-coupled services can easily co-operate with several services, without having to locate them first. When a publisher connects or disconnects, the subscriptions are re-evaluated, and adjusted if necessary, resulting in subscribers receiving events from alternative event publishers.

Although the communication style is asynchronous and event-driven, important aspects from service-oriented architectures are maintained. The service conditions and rules together with the advertisements and service description are used in stead of the service contract from service-oriented architectures. Subscribers can define important features the subscribed events should meet. The service owner and the responsible for the service are defined in the service description. This enhances the reliability of TIP 3 services. The service conditions specify the functional requirements from a service that are typically defined in the service contract as well. The service type is defined via the service category. TIP 3 integrates service-oriented aspects, such as the service contract, into an event-driven architecture.

6.4 Comparison of TIP 3 and TIP 3*

TIP 3* offers a simple event-driven middleware that services subscribe with and publish their events with. The TIP 3* middleware forwards events to the subscribers. TIP 3* services are loosely coupled, however, once they have selected a co-operation partner, the services remain coupled . TIP 3* cannot react and adapt to a dynamic environment of changing services. TIP 3 addresses this deficiency. It provides an event-driven middleware. The design enables service-orientation through the service conditions and rules, advertisements and the service descriptions. In contrast to TIP 3*, TIP 3 can easily adapt to changing service availability and a highly dynamic set of available events. The services' subscription rules ensure that TIP 3 services always subscribe to the best possible set of events available.

Figure 11 shows how the difficulty and complexity in TIP and in the development of TIP have changed. In the first versions, the implementations were not easily extendable. TIP 3* was implemented in [26] as a small prototype, however, the main purpose was to provide a framework for the implementation of caching mechanisms. Our design includes service conditions and service rules, so that TIP 3 provides major service-orientated principles. On the one hand, this facilitates the co-operation of services. On the other hand, the design

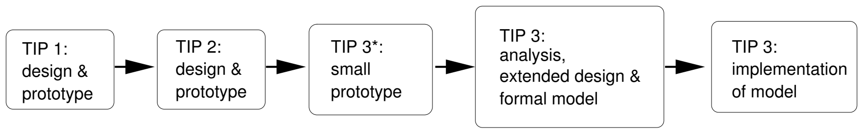

TIP 1: design & prototype
TIP 2: design & prototype
TIP 3*: small prototype
TIP 3: analysis, extended design & formal model
TIP 3: implementation of model

Figure 11: The difficulty and complexity in the development of the TIP system have grown bigger with each new version. The size of the boxes corresponds to the complexity of the problem.

and especially the examination and the analysis of the design are more complex than in previous versions of TIP as the services' interactions have become more complex.

We have presented a new design for TIP 3 that easily adapts to a dynamic environment. We have added several components to the TIP architecture, such as the observer, the functional conditions, and the subscription rules for services. They contribute both to the high adaptivity and to the enhanced service-orientation of TIP 3.

7 Methodology

In this section, we discuss methods of modelling software systems. There are several ways of exploring and analysing software design and architecture. One of them is examining and analysing formal models. A model can be used to detect aws of existing software systems [9]. A formal model can also be used to ascertain that a design has certain properties, or that a design solves the problem posed.

Section 7.1 begins with an informal definition of the term *formal model*. This is followed by an introduction to modelling terminology and the modelling tool UPPAAL[4]. Here the terms *system state* and *trace* are discussed, using examples to illustrate the concepts. We show how a simulation helps to understand the model's behaviour. We then examine if our example model satisfies a simple property, and explain what verification means in the context of formal modelling. Section 7.2 explains why we decided to implement a formal model instead of a software implementation. The last section gives a more detailed overview on UPPAAL and its modelling and query languages. We discuss timed automata, their semantics and the semantics of networks of timed automata in detail. We explain how UPPAAL extends timed automata. We show how processes can communicate with each other. We explain the different types of queries that UPPAAL offers. Lastly we demonstrate the different effects of urgent and committed locations with an example. This section serves as background to Section 8, where we present and discuss the model of the TIP 3 architecture.

7.1 Introduction to Formal Modelling

Formal modelling is an acknowledged method to examine and analyse software systems [1].

A *formal model* of a system represents the system attributes and characteristics, using a specified modelling language. A formal model can be used to simulate and thereby explore the system behaviour. A modelling tool like UPPAAL [2–4, 23] helps developing formal models of a system. UPPAAL provides several utilities: a modelling language, a user interface to create the model, means for automated simulation, a query language that helps formulating properties, means for automated verification and automatically generated traces if the verification of a property fails. We explain the meaning of the different terms later in this section.

UPPAAL uses *timed automata*, or *clocked automata*, as modelling language. A timed automaton is a finite state machine extended by clock variables. A finite state machine consists of several vertices, here called locations, and edges between locations. A clock variable evaluates to a real number. All clocks advance synchronously. Clock variables can be used in conditions on edges and locations. Every timed automaton represents a process. In UPPAAL, concurrent processes are represented by a network of several timed automata that run in parallel. Whenever we use the term *system* in this section, it refers to such a network of timed automata with variables and other extensions to timed automata offered by UPPAAL. The theory of timed automata and the extensions that UPPAAL provides are discussed in more detail in Section 7.3.

The *state of the system* is defined by the location of each automaton, the clock constraints for every location, and the variable values. When an automaton fires an edge, separately or synchronously with one or several other automata, the *system state* can change.

A *trace* or *state trace* is a sequence of system states. One system state can lead to several different state traces, as is shown later.

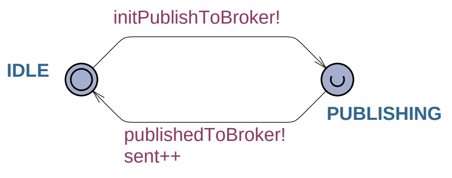
We now discuss these concepts with the help of a simple model of a sender and receiver.

Example

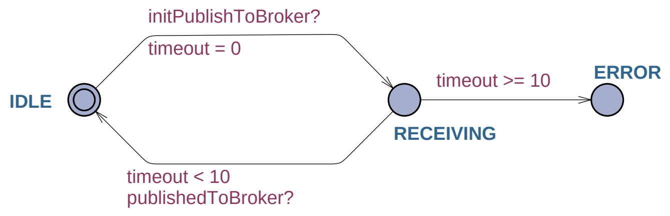
Figure 12 shows a small system that models a sender and a receiver. Sender and receiver run in parallel. The sender is modelled in Figure 12(a). It has two locations, IDLE and PUBLISHING. The initial location of an automaton is marked with a double circle. In both automata the initial location is named IDLE. Figure 12(b) shows the receiver. It has three locations, IDLE, RECEIVING and ERROR.

When the sender initiates the sending of a message, it fires the edge that is labelled `initPublishToBroker!`. Thereby it synchronises with `initPublishToBroker?`, switches to the PUBLISHING location and starts to

[4]http://www.uppaal.com



(a) Sender.



(b) Receiver.

Figure 12: A simplified example of sender and receiver.

send. The sender uses a *binary synchronisation channel* to synchronise and communicate with the receiver. Binary synchronisation channels are a pair, `initPublishToBroker!` and `initPublishToBroker?`, where the exclamation mark indicates the initiating process, and the question mark the responding. Only the sender can start the synchronisation, because it fires the edge labelled `initPublishToBroker!`.

The receiver switches to the RECEIVING location. The system state changes, from the state where both automata are in their initial locations, to the state where the sender is in the PUBLISHING location and the receiver in the RECEIVING location. In the synchronisation, the receiver's clock is reset to 0. The receiver now waits to synchronise with `publishedToBroker!`. However, if the sender does not finish the transmission during a specified interval (10 time units in our example), the receiver detects an error. It switches to the ERROR location when the timeout of more than 10 time units has been detected by the clock `timeout` and the receiver consequently fires the edge that is labelled with the condition `timeout` $\geq$ `10`. In our example, one of several possible traces consists of the state sequence (IDLE, IDLE), (PUBLISHING, RECEIVING), (PUBLISHING, ERROR), another is the state sequence (IDLE, IDLE), (PUBLISHING, RECEIVING), (IDLE, IDLE), (PUBLISHING, RECEIVING). We have omitted the clock variables in the examples.

During the examination of this system, the designers need to ensure that the sender always sends the whole message, i.e., that the sender always reaches the IDLE location again. Reformulated, this property is "A message transmission is always completed". This cannot be guaranteed in our model. The receiver has the possibility to detect timeouts with its clock and to detect transmission errors with the help of its clock. In our example, this can of course be detected easily. However, f the model is more complex, a property may not be checked simply by sharp scrutiny. The property has to be checked by using an algorithmic model checking tool that verifies the model, i.e., it examines if the model satisfies one or several properties. Thus *model verification* means that a *model checking tool* examines and analyses a model with respect to specified properties. The model checking tool examines every possible reachable system state for property satisfaction.

7.2 Formal Model vs Implemented Prototype

Modelling a software design and implementing a prototype of the same design are different ways to explore, examine and analyse the design. We decided to implement a formal model of the TIP architecture. Such a formal model describes the system characteristics accurately [11]. A model helps the designers to detect and identify the properties of a ubiquitous system [12]. A model makes it easy to explore alternative designs. Model checking tools typically provide some means of simulation, so that traces can be generated and evaluated. These traces can be used for scenario development. A model has to be detailed enough so that properties can be checked and traces can be generated. System states that were not anticipated in the scenarios or design can be detected through the traces of unsatisfied properties. New scenarios can be developed or the design can be adapted. A software implementation can be analysed with software testing tools. However, the analysis can only examine a limited number of scenarios that hopefully cover all significant cases. Model checking tools are an accepted and recognised automated algorithmic technique.

TIP is a location-aware mobile tourist information system. It is composed of several extremely loose coupled services. Service availability cannot be taken for granted. While the user moves, they can walk in and out of the area where a service is available (cf. Figure 13). The set of available services changes dynamically as the user moves. Services subscribe and unsubscribe, advertise and unregister. This feature, although desired, complicates the analysis and examination of the architecture [1] and of the behaviour of a single service. A single service can be examined alone, however, the examination and analysis of service co-operation in a dynamically changing publish/subscribe system is difficult and can soon become quite complex. A model of the publish/subscribe middleware that connects publishers and subscribers can help to understand the behaviour of the system [1]. From the point of view of a service designer, a model of the TIP middleware, such as we provide, can be more than helpful to fully understand the behaviour of the service.

TIP is a ubiquitous system. Ubiquitous systems are complex to design [12]. A model allows the designer to assess how a decision or change of design affects the system. The analysis of a model gives the developers the chance to change the design before implementing the prototype, thereby saving time and money. The system analysis should consider the interactions between user and system, the user experience, the usability of different hardware devices and software usability.

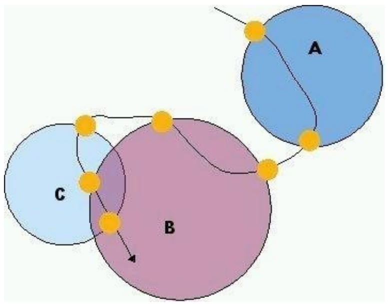


Figure 13: The movement of the user is shown in the drawn line. While they move, the user first comes into the area of service A. They walk out of the reach of service A and into the availability of service B. After they have walked out of the area of service B, they walk into service C, and then into the area of service B. They then leave the reach of service C.

Problems The model scalability may pose some problems. Memory and time limitations of the UPPAAL model checking tool make it difficult or impossible to examine and verify large models as thorough as necessary. A general problem is the mapping of informal formulated properties to a model, and the mapping of the model to an implementation.
The modelling process is an extra step in the system development process that may induce higher costs and that may take more time. With a prototype implementation, the user interaction can be examined. For instance, the frequency of how often features are used, can be determined. This cannot be obtained with a model.

7.3 UPPAAL

UPPAAL is a tool-box for the verification of real-time systems. It is being developed by Uppsala University and Aalborg University. [2] give several examples where UPPAAL has been used to check systems, such as an audio/video protocol [13] and a commercial field bus protocol [6]. In this section, we introduce UPPAAL's modelling language and query language.

7.3.1 UPPAAL s Modelling Language

UPPAAL uses timed automata to model processes. We now define the syntax and semantics for timed automata, following [2], before introducing UPPAAL's extensions to timed automata.

A *timed automaton* is a finite directed graph with a set of conditions over integer and clock variables and resets of clock variables. C denotes a set of clocks, and $B(C)$ is the set of conjunctions over simple conditions, such as $x \odot c, x - y \odot c$, where $x, y \in C, c \in \mathbb{N}$, and $\odot \in \{<, \leq, \geq, >\}$.

De nition 1: Timed Automaton [2] A timed automata is a tuple (L, l_0, C, A, E, I), where

L denotes the set of locations,

l_0 is the initial location,

C denotes the set of clock variables,

A is a set of actions, co-actions and τ-actions,

$E \subseteq L \times A \times B(C) \times 2^C \times L$ is the set of edges that connect locations, with the assigned actions, conditions and the set of clocks that are reseted when the edge is fired, and

$I : L \rightarrow B(C)$ assigns invariants to locations.

The *semantics of timed automata* describe the transition relation between states. There are two types of transitions, as Definition 2 shows: (1) the delay transition and (2) the action transition. In a delay transition, the automaton does not switch to a new location. It stays in the current location, but the automaton's clock progresses, i.e., time passes. In an action transition the automaton fires an enabled edge. Time does not necessarily pass during an action transition.

The function $u : C \rightarrow \mathbb{R}_{\geq 0}$ denotes a clock valuation that evaluates a clock $c \in C$ to a non-negative real number. Let $\mathbb{R}^C$ be the set of all clock valuations. Let $u_0(x) = 0$ for all $x \in C$. Guards and invariants are considered as sets of clock valuations. $u \in I(l)$ means that u satisfies $I(l)$.

De nition 2: Semantics of timed automata [2] Let (L, l_0, C, A, E, I) be a timed automaton. The semantics is defined as a labelled transition system $\langle S, s_0, \rightarrow\rangle$ where $S \subseteq L \times \mathbb{R}^C$ is the set of states, $s_0 = (l_0, u_0)$ is the initial state, and $\rightarrow \subseteq S \times \mathbb{R}_{\geq 0} \cup A \times S$ is the transition relation. The transition relation is defined by:

1 $(l, u) \xrightarrow{d} (l, u + d) if \forall d' : 0 \leq d' \leq d \Rightarrow u + d' \in I(l)$, and

2 $(l, u) \xrightarrow{a} (l', u')$ if there exists an edge $e = (l, a, g, r, l') \in E$ so that $u \in g, u' = [r \mapsto 0]u$, and $u' \in I(l)$,

where for $d \in \mathbb{R}_{\geq 0}$, $u + d$ maps each clock $x \in C$ to the value $u(x) + d$. $[r \mapsto 0]u$ identifies the clock valuation that maps every clock in r to 0 and agrees with u over $C \backslash r$.

A *network of timed automata* consists of n timed automata that share a set of clocks and actions: $A_i = (L_i, l_i^0, C, A, E_i, I_i), 1 \leq i \leq n$. A location vector $\bar{l} = (l_1, \ldots, l_n)$ is a vector where the individual locations $l_i, 1 \leq i \leq n$ are the current locations of the respective automata. The invariant functions for the automata's locations are composed into a common function over location vectors $I(\bar{l}) = \wedge_i I_i(l_i)$. $\bar{l}[l_i'/l_i]$ identifies the vector where the ith element l_i is replaced by l_i'.

De nition 3: Semantics of a network of timed automata [2] Let $\bar{l}_0 = (l_i^0, \ldots, l_n^0)$ be the initial location vector that consists only of the automata's initial states. The semantics of a network of timed automata is defined as a transition system $\langle S, s_0, \rightarrow\rangle$, where $S = (L_1 \times \ldots \times L_n) \times \mathbb{R}^C$ is the set of states, $s_0 = (\bar{l}_0, u_0)$ is the initial state, and $\rightarrow \subseteq S \times S$ is the transition relation. The transition relation is defined by

- $(\bar{l}, u) \longrightarrow (\bar{l}, u + d)$ *if* $\forall d' : 0 \leq d' \leq d \Rightarrow u + d \in I(\bar{l})$ – no process in the network fires an edge that leads to another location, or: all processes remain in their previous locations, if the locations' invariants are still satisfied.
- $(\bar{l}, u) \longrightarrow (\bar{l}[l'_i/l_i], u')$, *if* $\exists l_i \xrightarrow{\tau g r} l'_i$, *so that* $u \in g, u' = [r \mapsto 0]u$ *and* $u' \in I(\bar{l})$.
- $(\bar{l}, u) \longrightarrow (\bar{l}[l'_j/l_j, l'_i/l_i], u')$, *if* $\exists l_i \xrightarrow{c?g_i r_i} l'_i$ *and* $l_j \xrightarrow{c!g_j r_j} l_j$ *so that* $u \in (g_i \wedge g_j), u' = [r_i \cup r_j \mapsto 0]u$ *and* $u' \in I(\bar{l})$. This defines the synchronisation between two processes.

UPPAAL s extensions to timed automata

UPPAAL adds several features to timed automata. A meaningful example that includes every of the listed extensions would be very comprehensive and complex. We therefore opted to refer to this explanation in Section 8, whenever one of the features actually is used in our model. Here we limit ourselves to a general description of the extensions.

Templates In UPPAAL, a process is modelled as an automaton which can have one or several parameters. If the automaton does not have any parameters, it can only be instantiated once in the system declaration. However, if the automaton has parameters, it may serve as a *process template*. Process templates can be instantiated several times in the system declaration. The parameter values are then assigned in the process instantiation. A template parameter can be of any type, e.g., `int`. Templates enable the reuse of the model in several processes.

Constants The scope of a constant can be global or local, i.e., for the instantiation of one automaton. Constants must have integer values. Constants are declared `const name value`.

Bounded integer variables A bounded integer variable `int [min, max] x` is a variable that has `min` as lower and `max` as upper bound. UPPAAL checks automatically upon verification if the bound is violated. This might as well be expressed as a guarding expression, or a location invariant, but then the guard or invariant would have to be checked on every edge and in every location. We explain the concepts of guarding expressions and location invariants later. Bounded integer expressions can be used in expressions of the type guard, assignment and invariant.

Arrays can be of the data type (bounded) integer, clock, or constants and channels. Arrays can be multi-dimensional. They are declared as `chan n[3]; clock c[10]; int[1,4] i[5];`.

Binary synchronisation An edge that is labelled with `name!` synchronises with an edge that is labelled with `name?`. If several combinations are possible, a synchronisation pair is selected non-deterministically. The process that fires the synchronisation with the exclamation mark (`name!`) is usually called the sender or initiator of the synchronisation, while the process that reacts to the synchronisation by firing the synchronisation with the question mark (`name?`) is called the receiver or listener. Only the sender can initiate a synchronisation. Binary synchronisation channels can block the process if only one of sender and listener is available. Binary synchronisation channels are declared as `chan name`.

When two processes synchronise with a channel from an array of channels, they have to use an array index to decide what channel to synchronise on, e.g., `synch[2]!`.

Broadcast channels enables a sender to synchronise with zero or many listeners at once, i.e., they allow 1-to-many synchronisations. An edge that is labelled with `name!` synchronises with none or any number of listeners `name?`. If a listener is able to synchronise in its present state, it has to. The broadcast channel cannot block the sending process, i.e., if there is no listener, the sending process will still send the broadcast. A broadcast channel blocks the listening process if no sender synchronises with it. A broadcast channel is declared as `broadcast chan name`.

Urgent synchronisation An urgent synchronisation that is enabled must not be delayed, i.e., the respective edges have to be fired before the following system state change. Urgent synchronisation channels cannot have clock guards. Urgent synchronisation channels can be used for binary synchronisation and for broadcast synchronisation. When an urgent synchronisation is used, the clocks are not increased before and during the synchronisation, i.e., it is impossible to delay in a location when it has an outgoing edge that is labelled with an urgent synchronisation. However, an alternative edge, where the guarding conditions are satisfied, can be fired instead of the urgent synchronisation. Urgent synchronisation channels are declared as `urgent chan name` respectively `urgent broadcast chan name`.

Urgent locations are equivalent to adding a clock constraint `t` to a location and adding an invariant `t` $\leq$ `0`. `t` is reset on all incoming edges. When the automaton is in an urgent state, the time does not pass, i.e., the clocks are not incremented. This means that the operation does not take time in respect to the clocks, and time does not pass in an urgent location.

Committed locations If any location of the current system state is committed, then the state itself is committed. In the next step, an edge must be fired that leaves one of the committed locations. Committed locations are more stern than urgent locations. Committed locations can be used to model atomic operations and to model synchronisations between more than two processes if the synchronisations should "happen at once". However, if several processes are in a committed location at the same time, they can interleave, so that the atomicity of the operations is lost.

Initialisers initialise integer variables and arrays of integer variables, e.g., `int i = 0; int z[3] = 0,2,4;`.

Expressions in UPPAAL

Expressions label the edges in an automaton. They are either evaluated or executed before or when the edge is fired. UPPAAL distinguishes between four types of expressions:

Guards are evaluated before an edge is fired. A guarding expression has no side effects. It evaluates to a boolean value. Guards reference only clock or integer variables and constants, or arrays thereof. Clocks and clock differences can only be compared to integer expressions.

Synchronisation labels either have the form `expr!` or `expr?` or they are empty. They must not have side effects and evaluate to a channel. Synchronisation labels only reference integer variables, constants or channels.

An assignment is a list of comma-separated list of expressions. The expressions in an assignment must have side effects. Assignments reference clock and integer variables and constants. A clock variable can only be set to a non-negative integer value.

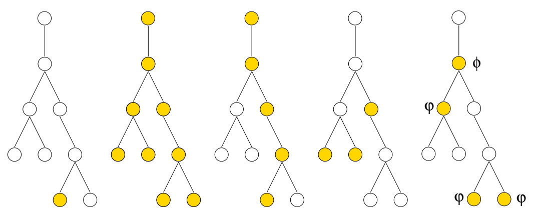
Invariants do not have side effects. They reference clock or integer variables and constants. An invariant is a conjunction of conditions of the form `x < e` or $x \leq e$, where `x` is a clock reference and `e` evaluates to an integer.

In addition to the extensions to timed automata and expressions, UPPAAL also offers the possibility to define data types and methods. The data types and methods are defined in a subset of the C programming language.

7.3.2 UPPAAL s Query Language

Formal models are used to verify that certain properties hold, i.e., that the model satisfies their requirements. A formal language wherein the verification properties can be expressed is necessary. If the verification should be conducted by computers, the properties should be expressed in a well-defined and machine readable language. UPPAAL uses a subset of computational tree logic. UPPAAL's query language features state and path formulae, however not nested path formulae. Path formulae can verify three types of properties, reachability, safety and liveness properties, which we explain later.

State formulae A state formula is an expression that is evaluated for a state. The overall behaviour of the model is not considered. A state formula can simply check if a process is in a certain location. It may be an expression, or examine if the system can deadlock. The syntax of state formulae is a superset of the syntax of guards. Disjunctions are allowed. Like guards, state formulae must not have any side effects. A state formula in our example in Section 7.1 is `Receiver.RECEIVING && Receiver.timeout > 10`. It formulates that the Receiver should be in the RECEIVING location, and that the receiver's clock timeout should have a value `> 10`, i.e., `u(timeout) > 10`. This formula holds whenever these conditions are satisfied. Another state formula, `Sender.sent > 100` is true whenever the sender has sent a message to the receiver more than 100 times. UPPAAL uses a special state formula `deadlock` to check for deadlocks. `deadlock` is satisfied if a deadlock can occur in a reachable state.



(a) E $\Diamond\phi$ (b) A $\Box\phi$ (c) E $\Box\phi$ (d) A $\Diamond\phi$ (e) $\phi \leadsto \varphi$

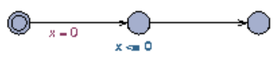
Figure 14: The property types that UPPAAL can verify.

Reachability properties Let ϕ be a state formula. Reachability properties examine if there exists a state where ϕ is true (cf. Figure 14(a)). Reachability properties are often used for sanity checks during the modelling process. In our model, many processes send messages to other processes, and the receivers possibly reply to them. A typical sanity check is to verify that the sender really can send the first message, that the intended receiver indeed receives the message, and is able to reply, or react in some other ways. This does not prove that our design is correct, or that the model really represents the design, but examines the essential characteristics of the model.

A reachability property, i.e., that some state satisfying the state formula ϕ should be reachable, is expressed with the path formula E $\Diamond\phi$. In UPPAAL, this is expressed with `E<>` ϕ. In our example on page 24 we would like to check that both the receiver's ERROR and IDLE locations can be reached. The query for this property is `E<> Receiver.IDLE || Receiver.ERROR`. This property is satisfied.

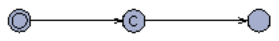
Safety properties Safety properties are used to prevent that something undesirable happens, e.g., that a message is delivered to more than the intended receivers. Let ϕ be a state formula. Then $A\Box\phi$ expresses that ϕ should be true in every reachable state (cf. Figure 14(b)). $E\Box\phi$ expresses that there exists a maximal path, where ϕ is always true (cf. Figure 14(c)). A maximal path is an infinite path, or a path where the last state cannot be left by any leaving edges. In UPPAAL, safety properties are expressed as `A []` ϕ respectively `E []` ϕ.

In our example, we do not want that the receiver switches to the ERROR location before the timeout. The safety property `A[] !(receiver.ERROR&& receiver.timeout < 10)` expresses that the receiver cannot detect an error when the timeout has not been reached yet. Our model satisfies this property.



(a) A normal location with an invariant.

(b) An urgent location.



(c) A committed location.

Figure 15: UPPAAL features three location types: normal, urgent and committed locations. The urgent and committed locations can be identified through the u respectively c.

Liveness properties Liveness properties ensure that something will happen, however without any notions as to when it will happen. For example, when a service has registered with the broker and advertised certain event types, the event types will be made available to other services. A liveness property can be expressed in two ways. Let ϕ, φ be state formulae. Then `A` $\Diamond\phi$ means that ϕ will be true eventually (cf. Figure 14(d)). A more complex form is the *leads to* or *response* property $\phi \rightsquigarrow \varphi$: if ϕ is satisfied, then φ will be satisfied eventually (cf. Figure 14(e)).
In UPPAAL, these properties are expressed as `A<>` ϕ and ϕ`-->` φ.

7.3.3 Time in UPPAAL

Time is modelled with a continuous time model in UPPAAL. Clock constraints in guards and location invariants have different effects. A clock constraint in a location invariant forces the process to leave that location, i.e., to fire an outgoing edge before the invariant is not satisfied any longer. If this is impossible, the system reaches a deadlock state. A clock reference in a guard prevents that the process fires the corresponding edge if the guard is not satisfied. It does not force the process to leave a location. For example, let `x` be a clock variable and let the location invariant be `x < 10`. This means that an outgoing edge must be fired within 10 clock ticks. A guard expression `x < 10` on an outgoing edge simply prevents that this edge is fired if `x` is greater than 10. Another edge might be fired, or the process does not or cannot leave the current location at all.

Committed and urgent locations UPPAAL knows three types of locations, normal, urgent and committed. Figure 15 shows that urgent and committed locations are marked `u` respectively `c`. A normal location is not marked.
An *urgent location* is the same as a location where the incoming edge resets a clock y, and the location is labelled with the invariant `y` $\leq$ `0`. Whenever a process is in an urgent location, the system is in an *urgent state*. If the system is in an urgent state, time cannot progress, however interleavings with normal states are allowed. Clock updates are not allowed in an urgent state.

A *committed location* is more restrictive than an urgent one. Whenever a process is in a committed location, the next transition has to fire an edge that leaves the committed location. A state that has a committed location is called committed. In a *committed state*, the committed location has to be left in the successor state. If a committed state has several committed locations, at least one of the committed locations has to be left in the successor state.

We demonstrate the different effects of urgent and committed locations in an example.

We return to our previous example (cf. Figure 12, page 24) and extend it by adding a processor and change the sender and receiver processes. The result can be seen in Figure 16: The sender (16(a) and 16(b)) synchronises with the receiver to initiate a transmission (`send!`). The transmission can either turn out to be faulty, or free of errors. When the transmission is errorless, the receiver uses the enqueue-method to enqueue the received message in the in-queue. The enqueue-method and the in-queue are not declared in the automaton, therefore the in-queue cannot be seen in Figure 16. The enqueue-method is called on the edge that leads from the RECEIVING location back to the IDLE location.

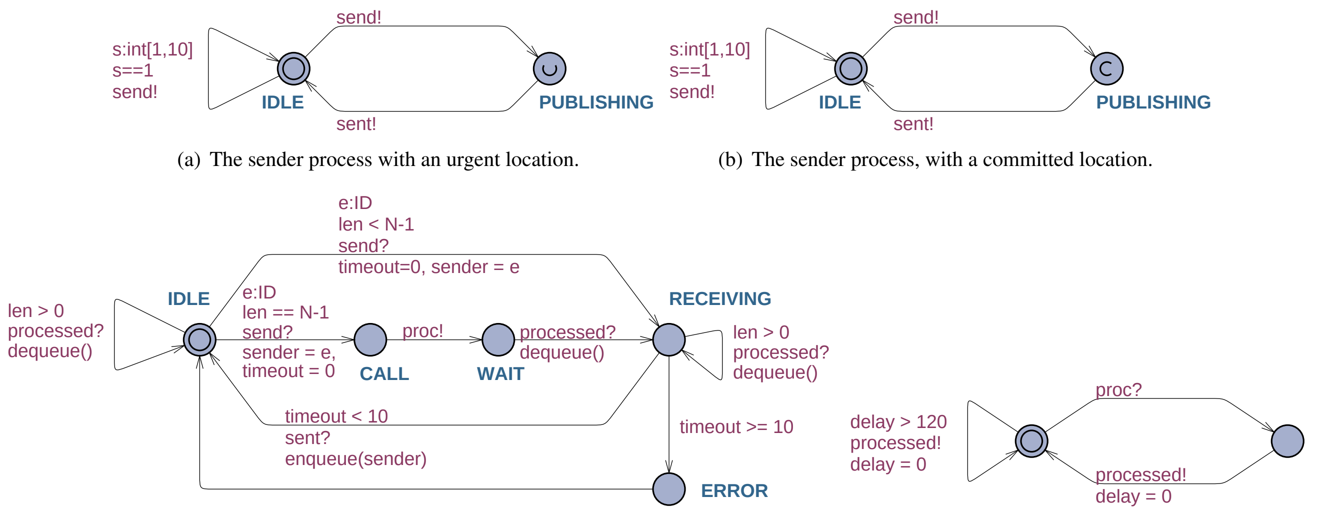
When the in-queue is full (`len == N-1`), the receiver starts a processor that processes the messages in the in-queue (`proc!`). When the processor has processed a message it synchronises with `processed!`. The receiver can now remove a message from its in-queue and receive the message that was sent by the sender.

The processor can randomly process messages, as well.

We now discuss the two different systems. Each system consists of a sender, a receiver and a processor, as above. The receiver and processor processes are the same in the two systems. In the system that we discuss firstly, the sender process' PUBLISHING location is an urgent location. The sender and the receiver synchronise with `send`. The sender randomly selects the looping edge that immediately leads back to the IDLE location, or the edge that leads to the PUBLISHING location. Both edges are labelled `send!`. When the sender switches to the PUBLISHING location, the receiver switches to the RECEIVING location if its in-queue is not full yet. If its in-queue is full, the receiver starts the processor (`proc!`) and waits in the WAIT location until the processor signals that it has finished (`processed!`).

The receiver then dequeues the processed message in the in-queue and switches to the RECEIVING location. The receiver's RECEIVING location has three outgoing edges. One is fired when the processor synchronises with `processed!` to signal that it has processed a message from the in-queue. The second is fired before a timeout has been reached and the sender signals that it has completed the message transmission (`sent!`). The message is then enqueued. The last outgoing edge is fired when the timeout has been reached. The receiver then switches to the ERROR location and consequently to the idle location. Whenever the sender process has reached the urgent PUBLISHING location, the system is in an urgent state. This means that time cannot progress until the system leaves that state. However, other processes than the sender can fire edges, update variables and change their locations, unless clock variables are updated. If the in-queue from the receiver is full, the receiver can wait for the processor until it has processed a message. The system is deadlock free.

In the second system, the sender process' PUBLISHING



(a) The sender process with an urgent location.

(b) The sender process, with a committed location.

(c) The receiver.

(d) The processor.

Figure 16: This extended example shows the difference between committed, urgent and normal locations. The sender's PUBLISHING location can either be an urgent location as in 16(a), or a committed location (16(b)). We have combined two different systems in this figure: one system consists of a sender with an urgent location, a receiver and a processor, the other system consists of the sender that has a committed location, a receiver and a processor.

location is a committed location. Whenever the system state contains a committed location, the next edge that is fired has to leave at least one of the committed locations. Normal actions cannot interfere. This means in our case that the sender has to fire the edge that is labelled `sent!` in the next step when it is in the committed PUBLISHING location. However, the sender cannot synchronise with `sent!` if the receiver has to wait for the processor to process messages from the in-queue. The second system is not deadlock free.

8 Model Description and Validation

We created a formal model of the TIP 3 architecture that we present and discuss in this section. 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.

There are several concepts that have to be treated in a different way in a model than in a software implementation, e.g., the user movement. The first section discusses the modelling and verification process and how these processes often interleave with each other. Section 8.2 presents how different real world concepts are represented in the model. In Section 8.3 we present the model of the client and the services running on the client. The model of the server and its services is discussed in Section 8.4. Section 8.5 introduces the model of how brokers communicate with one another. In Section 8.3, 8.4 and 8.5 we first present and discuss the model and then show how we have examined and analysed it. In Section 8.6 we argue how the three models could be combined into one and how they would be able to communicate with each other if they were combined. Section 8.7 discusses the problems encountered during the modelling process. In Section 8.8 we return to the requirements formulated in Section 3 and discuss how our model meets them.

The modelling terminology that is used in this section was explained in Section 7.

8.1 The Modelling and Veri cation Process

The design and modelling process comprises three interleaving steps. In the first step, the usability requirements should be recognised, as they are later needed 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. It may even lead to a completely new model, if the model did not satisfy a property. Indeed, 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 why. As a consequence, either the model, the underlying assumptions or the property have to be revisited. The model can be adjusted and complemented. It has to be examined whether the assumptions on which the model is based are correct or if they can be improved. The property can be adjusted and complemented. A property that

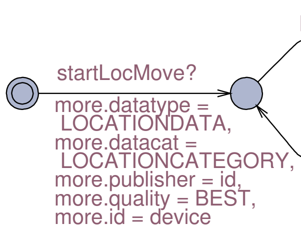


Figure 17: The automaton simulates user movement by publishing location events.

does not hold shows weaknesses of the model and should result in a better model or property.

8.2 Real World Concepts

Several phenomena that occur in the real world have to be treated differently in a model than in a software implementation. The most important ones are discussed in this section.

User movement When the user moves and changes their location, the TIP location service publishes a location event. In a software implementation, user movement may be simulated through a playback of recorded geo-data.

In a model, user movement is simulated by the publishing of location events by the location service, not by a sequence of coordinates. The automaton shown in Figure 17 publishes location events. It is randomly selected out of several automata belonging to the same network of timed automata.

Modelling events A model has to abstract from the actual data published in an event. The fact that a new event is published is important. Hence we model that a new event is published and abstract from the content of the event. User mobility and movement, for example, are not modelled through a trace of the user's geographical movements, but through the publishing of a location event by the location service. As a consequence, the model cannot simulate database queries containing a specific SQL query or the database's response. The model can simulate a generic database query and a generic database response. However, this level of abstraction is necessary. The model still represents the modelled system.

The graphical user interface (GUI) TIP services use the GUI of the mobile client for interaction with the the user. When the user selects an element providing some functionality, analogue to clicking on a button with a mouse on a desktop computer, this may result in a new event published by the respective service. In a software implementation, only the application being in focus may react to user input. This service is typically identified by the operation system. In our model, we had to model this part of the operation system. Otherwise, several services could react to mouse clicks and similar.

8.3 Model of the Mobile Client Peer

In this section, our model of the mobile client peer is presented and discussed. It is the first part of the model of the TIP 3 architecture. This model implements the client-side services and the user interaction with the system. It captures the behaviour of the TIP 3 system on the user's mobile client. Services that are located on the mobile client react to each others' events. They interact with the user and with services running on the server.

8.3.1 The Model

Actors and processes involved Before discussing the model, we identify the actors and the situations that have to be modelled. The actors are the broker, the observer, the user and the client-side services. They were introduced and discussed in Section 6. The actors interact with each other in different situations, such as (1) the service registration, (2) when services publish events to the broker, (3) the broker filters the events and forwards them to the subscribers, (4) the user interacts with the TIP system and (5) a service deregisters from the broker.

In our model, one process represents an actor in a certain situation. Therefore the broker is represented by several processes in the model, as is the observer, as are the services. We now describe how we have modelled the different situations. The respective automata may be found on the accompanying CD. As an example, the user interaction (4) is described in detail discussing the automata.

(1) Service Registration At startup, a service registers with the local broker. A service publishes its service description to the broker. A service may provide a set of functional conditions and a set of subscription rules. We introduced the concept of functional conditions and subscription rules in Section 6. The observer evaluates the service's functional conditions. If they are satisfied, the service's subscription rules are evaluated by the observer. The observer subscribes the service to the specified events. If the conditions are not satisfied, the registration process is stalled until an appropriate advertisement has been published to the broker so that the conditions are satisfied.

The service advertises its published event types.
The service process interacts both with the broker process and the observer process. In a programmed implementation, the observer should simply observe the communication between the broker and the service and react to it. The service and its observer should not communicate with each other. However, this cannot be modelled in UPPAAL. Therefore we decided to add an observer process communicating with the processes it needs to observe in order to obtain the same result.

During the service registration, the service's conditions and subscription rules are evaluated by the observer. From a conceptual point of view, the observer is part of the broker.

(2) Publishing events to the broker A service publishes its events to the local broker, i.e., it notifies the broker that it will publish a message, and notifies the broker when the transmission of the message has been completed. We decided to split this operation into two steps, enabling the verification that services may publish events to the broker. A typical query is `E <> locPublish.PUB`. The query examines the property ascertaining that the process representing the location service's publishing component may indeed reach the PUB location. This location can only be reached when synchronising with the broker, thus publishing an event to the broker. The broker then enqueues the message in its in-queue.

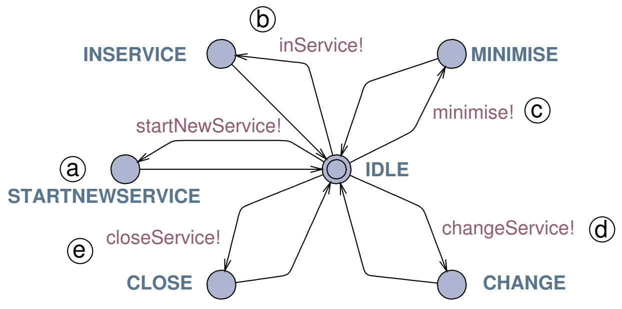
(3) Filter events Whenever a service has published an event to the broker, the broker filters the event. It checks if any services have subscribed the event. The broker then synchronises with `sendToService!` and notifies the respective services of the transmitted an event. The service receive the event and enqueue it in their respective in-queue. If the receiver is a second broker, the sender would notify it with `b2bPublish!` and `publishedToBroker!`. The broker distinguishes remote brokers from subscribing (local) services for modelling reasons. The filtering automaton is nearly the same in the three models. However, we wanted to examine whether a broker really filters events to remote brokers in the model of the communication between brokers discussed in Section 8.5. Brokers, services and observers are identified by a number, their respective identity number. Service processes synchronise with the respective observer processes using a synchronisation channel identified by their shared identity number. If the broker would filter events to services and brokers using one synchronisation channel, the receiver could not be identified unambiguously. Especially in a model containing both several brokers and services, this would lead to an unexpected and undesired behaviour of the model.

(4) User interaction with TIP The user interaction with TIP 3 services is modelled by the automata shown in Figure 18, Figure 19 and Figure 20. Figure 18(a) shows the automaton modelling how a user may interact with the TIP GUI. For reasons of verification we decided to use respectively named locations. We now discuss the different user interactions shown in Figure 18(a) and how they are modelled in detail. We refer to the automaton shown in Figure 18(a) as Input. The automaton shown in Figure 20 is referred to as GUI-Handler.

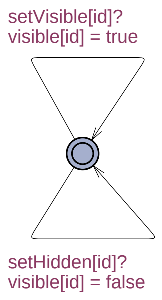
The first described interaction, the user starting a new service, is modelled by the input automaton, shown in Figure 18(a). The discussion refers to the letters in Figure 18(a).

(a) Input models the user starting a new service by synchronising with `startNewService!`. The GUI-Handler replys with `startNewService?` and moves to the START_SERVICE location. It then randomly[5] identifies a TIP 3 service (`requested : int[1, NUMBEROFSERVICES-1]`). If the selected service is not yet running

[5]The model selects a service, that would otherwise have been selected by the user. The user would not select a random service.



(a) input.



(b) Service visibility.

Figure 18: Model of user input. 18(a) models user interactions through the GUI. We refer to this automaton as input. 18(b) shows a service switching between being visible and unseen.

(`!registered[requested]`), the GUI-Handler starts the service by synchronising with `startReqService[requested]!`. The currently visible service is set to identify the requested service and the GUI-Handler moves to the SET_ANOTHER_VISIBLE location.

We now brie y explain how the GUI-Handler (see Figure 20) handles previously visible services that are hidden behind the currently visible one. The explanation is necessary for understanding the automaton. The model is based on the assumption that previously used services are still maximised though hidden when a new service is maximised, i.e., the only service visible on the mobile device's screen. If the currently visible one would be minimised, another service would be seen. Hence, the automaton needs to remember the previously visible services that were neither stopped nor minimised, but simply hidden by another service. This is achieved by the `oldVisibles` variables, a queue storing the services identifications. The integer variable `visibleService` identifies the currently visible service.

If the GUI-Handler may select an already registered and running service when leaving the SET_ANOTHER_VISIBLE location. It then only enqueues the currently visible service's identification number in the `oldVisibles` queue and updates the variable `visibleService` so that it identifies the currently visible service. GUI-Handler then moves to the SET_ANOTHER_VISIBLE location.

If the `oldVisibles` queue is empty (`head(oldVisibles) <= 0`), no action is taken and the automaton moves to the DONE loca-

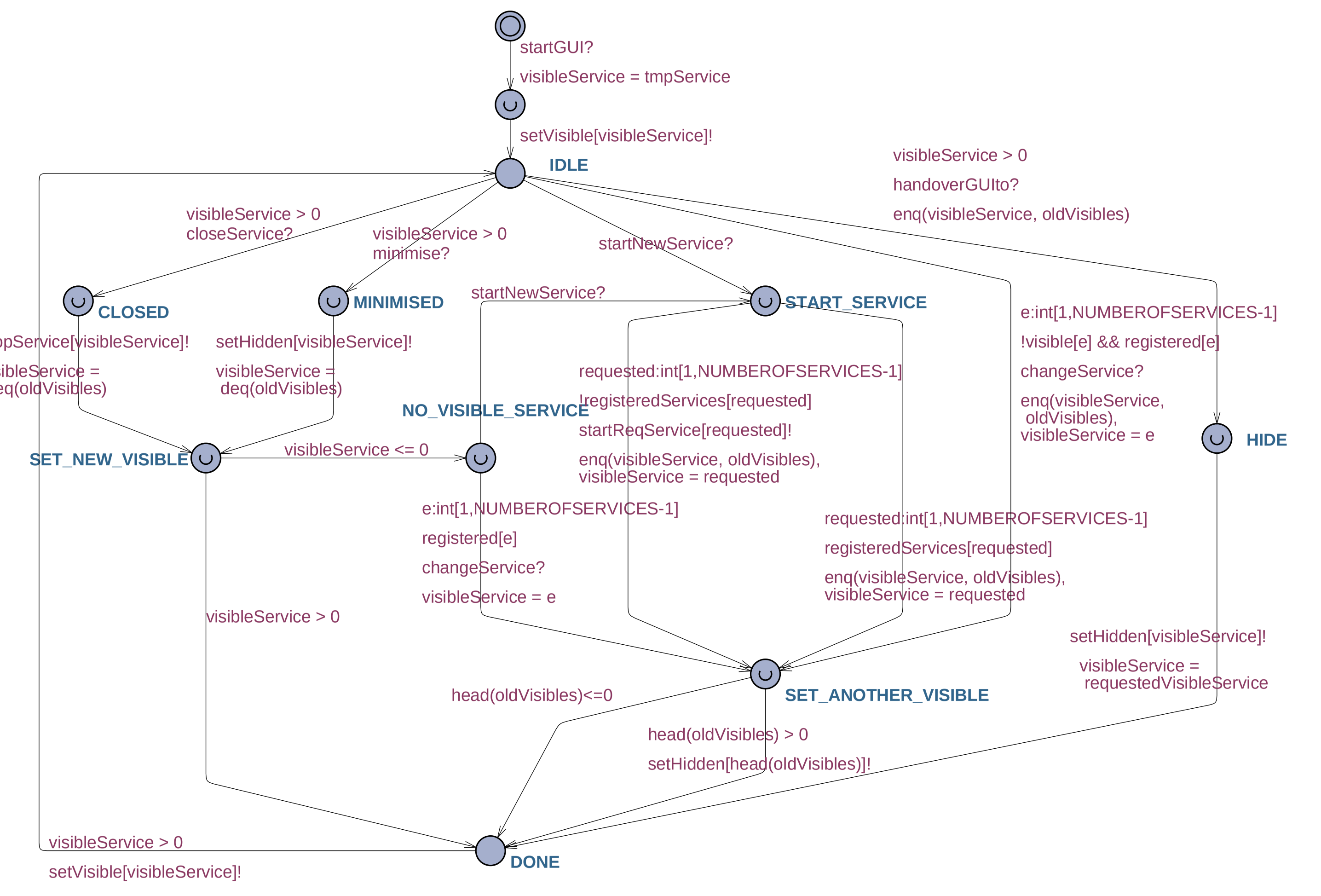
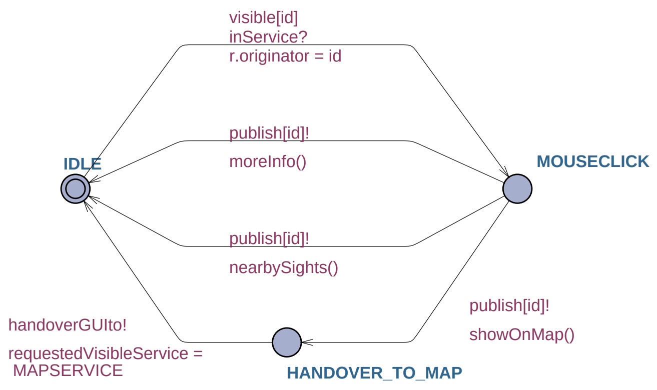
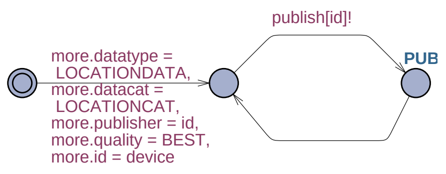

startGUI?
visibleService = tmpService
setVisible[visibleService]!
IDLE
visibleService > 0
handoverGUIto?
enq(visibleService, oldVisibles)
visibleService > 0
closeService?
visibleService > 0
minimise?
startNewService?
startNewService?
CLOSED
MINIMISED
START_SERVICE
e:int[1,NUMBEROFSERVICES-1]
!visible[e] && registered[e]
changeService?
enq(visibleService, oldVisibles), visibleService = e
stopService[visibleService]!
visibleService = deq(oldVisibles)
setHidden[visibleService]!
visibleService = deq(oldVisibles)
requested:int[1,NUMBEROFSERVICES-1]
!registeredServices[requested]
startReqService[requested]!
enq(visibleService, oldVisibles), visibleService = requested
NO_VISIBLE_SERVICE
HIDE
SET_NEW_VISIBLE
visibleService <= 0
e:int[1,NUMBEROFSERVICES-1]
registered[e]
changeService?
visibleService = e
requested:int[1,NUMBEROFSERVICES-1]
registeredServices[requested]
enq(visibleService, oldVisibles), visibleService = requested
visibleService > 0
setHidden[visibleService]!
visibleService = requestedVisibleService
head(oldVisibles)<=0
SET_ANOTHER_VISIBLE
head(oldVisibles) > 0
setHidden[head(oldVisibles)]!
visibleService > 0
setVisible[visibleService]!
DONE

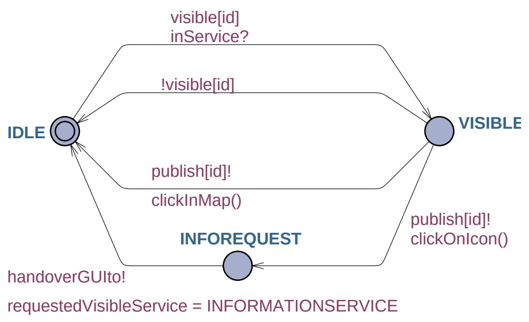
Figure 20: GUI-Handler: This process describes how the model identifies the currently visible service. It also shows how minimisation, maximisation, service start and service termination are handled from the system's point of view.



(a) The information service



(b) The location service



(c) The map service

Figure 19: The three processes describe how the services react to a user's interactions and movement.

tion. Otherwise, the automaton synchronises with `setHidden[head(oldVisibles)]!`, i.e., with a process like that shown in Figure 18(b) and moves to the DONE location. In our model, every running service providing a GUI needs such a process. This process accesses a boolean array `visible[]`. The array keeps track for each service if it is visible and may react to user input.

The GUI-Handler then leaves the DONE location by synchronising with `setVisible[visibleService]!`, setting the service previously requested visible. The GUI-Handler to the IDLE location. This synchronisation is guarded (`visibleService > 0`) for modelling reasons.

(b) The Input process models the user selecting an item in the currently visible service's user interface by synchronising with `inService!`. One of the automata shown in Figures 19(c) and 19(a) synchronises with `inService?`, depending on which service is currently visible. If the map service is the currently visible service, the automaton shown in Figure 19(c) moves to the VISIBLE location. The map service modelled can react to two user actions. The user may either centre the map on another location, or select an information icon on the map. In the first case, the map service publishes an event and returns to the IDLE location. In the second case, the map service publishes an event synchronising with `publish!` and moves to the INFOREQUEST location. It then synchronises with `handoverGUIto!` and the GUI-Handler. The map service sets the global variable `requestedVisibleService` so that it identifies the information service.

The GUI-Handler synchronises with `handoverGUIto?` and enqueues the map service in the `oldVisibles` queue. It then synchronises with `setHidden[visibleService]!` and updates the `visibleService` variable, so that it identifies the information service. The GUI-Handler then moves to the DONE location and returns to the IDLE location as explained above.

(c) The input automaton models the user minimising the currently visible service by synchronising with `minimise!`. This synchronisation is enabled only if at least one service is visible: the synchronisation `minimise?` in the GUI-Handler is guarded (`visibleService >= 0`). The GUI-Handler moves to the MINIMISED location. It then synchronises with `setHidden[visibleService]!`, setting the currently visible service invisible and moves to the SET_NEW_VISIBLE location. `visibleService` is set so that it identifies the service lastly visible. If there is no such service (`visibleService <= 0`) the automaton switches to the NO_VISIBLE_SERVICE location and waits for the `startNewService?` or the `changeService?` synchronisation. When the input automaton synchronises with `startNewService!`, the GUI-Handler moves to the START_SERVICE location and proceeds as shown above (see (a)).

When the input automaton synchronises with `changeService!`, the GUI-Handler randomly selects a running service, updates `visibleService` so that it identifies the selected service and moves to the SET_ANOTHER_VISIBLE location. From here, the GUI-Handler proceeds as shown above (see (a)).

(d) The input automaton models the user changing to another service by synchronising with `changeService!`. The GUI-Handler randomly selects a running service and enqueues `visibleService` in the `oldVisibles` queue. `visibleService` is then update to identify the service selected by the user. The GUI-Handler moves to the SET_ANOTHER_VISIBLE location and proceeds as shown above (see (a)).

(e) The input automaton models the user closing the currently visible service by synchronising with `closeService!`. The GUI-Handler then stops the currently visible service by synchronising with `stopService[visibleService]!`, updates the `visibleService` variable and moves to the SET_NEW_VISIBLE location. From there it proceeds as shown above (see (c)).

(5) Service deregistration A service is deregistered from the broker if the service is shut down or if the service is disconnected. When a service disconnects, the respective observer deregisters the service from the broker after a certain time interval. The broker simply removes the service from its list of subscriptions and publishers and stops forwarding events to the service. Eventually the subscriptions of other services are re-evaluated and changed if necessary.

The observer does not deregister the service immediately after detecting the lost connection. Hence, a service that only disconnects for a short time and then reconnects again, does not have to register again and subscription rules do not have to be evaluated.

8.3.2 Veri cation of the Model

We could verify basic properties, for example, "the location service can register", through simulations. Our simulations showed that the model functions properly: 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 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 a only the currently visible service reacts to user input.

However, owing to UPPAAL's limitations, we were not able to completely verify the model. We wanted to examine queries like `E <> registeredServices[MAPSERVICE] and registeredServices[INFORMATIONSERVICE] and !subscriberIndex[2][2][0][2][1][1]`. The query assures that the map service always subscribes to events published by the information service if both services are registered. While examining the query, UPPAAL needed more memory than it is able to address. Another query that could not be verified for the same reason is `(input.CHANGE and visibleService[INFORMATIONSERVICE]) --> (!visibleService[INFORMATIONSERVICE] and visibleService[MAPSERVICE])`. It examines the GUI-Handler automaton. We wanted to assure that the map service is always set to visible if a) the information service is visible and b) the map service is registered and c) the service is changed. When the service is changed, the input automaton moves to the CHANGE location.

A complete set of the queries can be found on the accompanying CD.

Simulation runs helped examining the model's functionality. They showed that the model provides basic functionality. More complex and important queries such as the ones shown above could not be verified.

8.4 The Server Peer and its Services

In this section we describe how we have modelled the server peer and its services. The server peer's model resembles the client peer's.

8.4.1 The Model

Actors and processes involved In this model, the main actors are services, the broker and the observer. The main processes that these actors are involved in include (1) service registration, (2) service deregistration, (3) publishing events to the broker and (4) filtering events. Unlike the client peer, the server peer provides a TIP database that services can access. Server-side services do not provide a GUI.

We now discuss how the different situations are modelled. The respective automata may be found on the accompanying CD. The filtering of events is exemplarily described in detail.

(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 service's 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. The publishing process is identical to that on the client peer.

(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.

Figure 8.4.1 and Figure 8.4.1 show the participating processes. In Step 1, a service (21(a)) synchronises with the broker (21(b)) through `initPublishToBroker!` and sends its message (`publishedToBroker!`). The broker receives the event and enqueues it in its in-queue. Sometime later the broker in Figure 8.4.1 dequeues the message again (Step 2) and first checks if there are any subscribers (Step 3) or if the

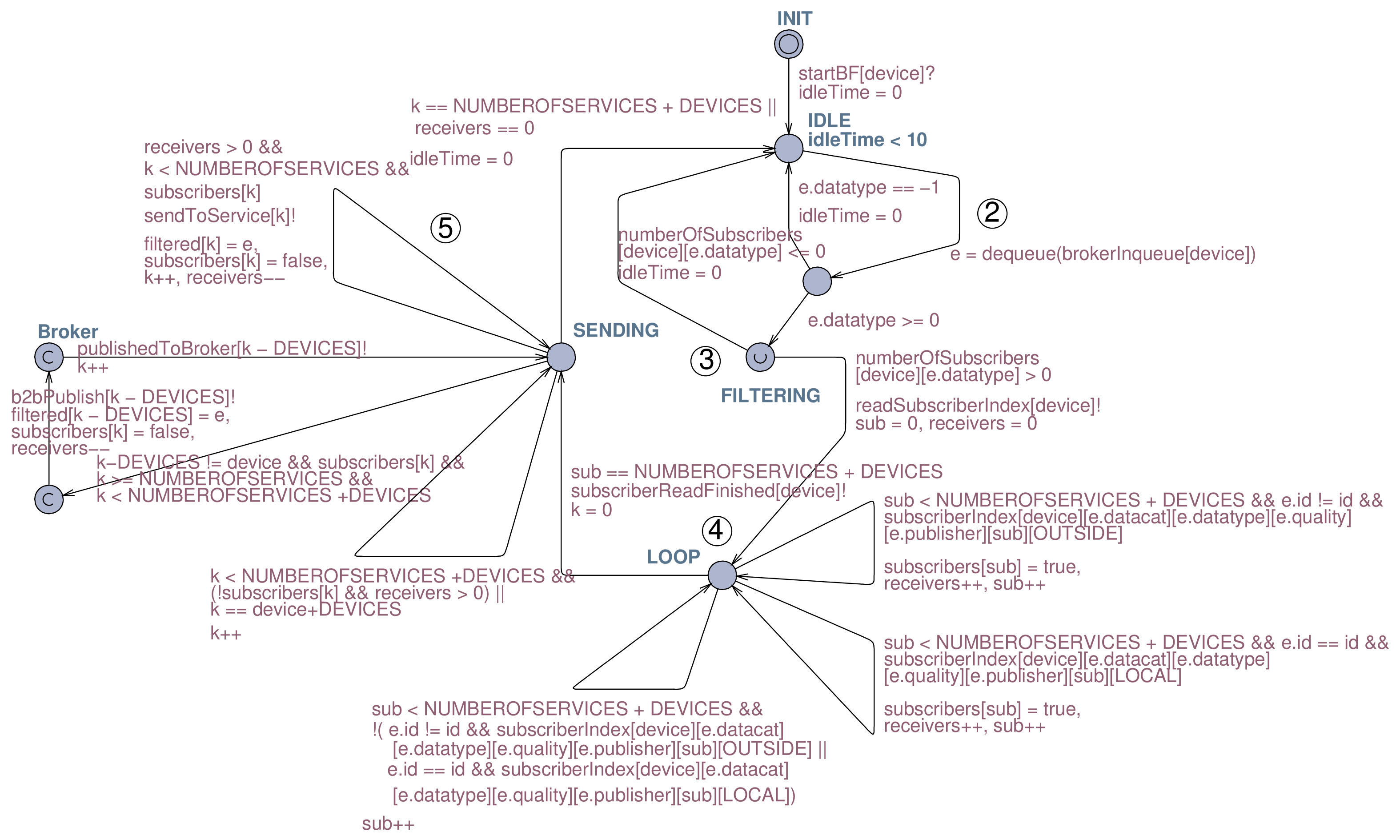
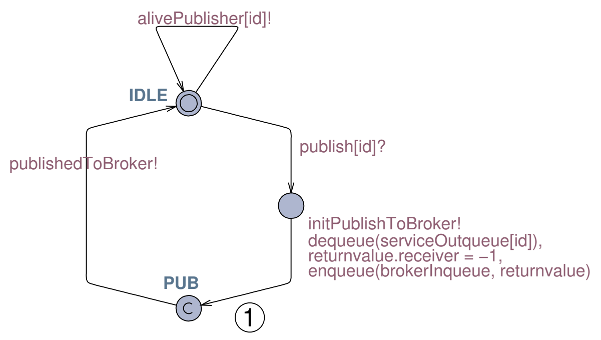
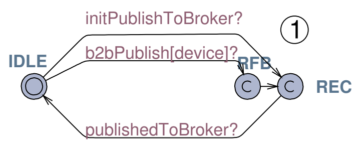

INIT
startBF[device]?
idleTime = 0
IDLE
idleTime < 10
e.datatype == –1
idleTime = 0
2
e = dequeue(brokerInqueue[device])
e.datatype >= 0
numberOfSubscribers
[device][e.datatype] <= 0
idleTime = 0
3
FILTERING
numberOfSubscribers
[device][e.datatype] > 0
readSubscriberIndex[device]!
sub = 0, receivers = 0
sub == NUMBEROFSERVICES + DEVICES
subscriberReadFinished[device]!
k = 0
4
LOOP
sub < NUMBEROFSERVICES + DEVICES && e.id != id &&
subscriberIndex[device][e.datacat][e.datatype][e.quality]
[e.publisher][sub][OUTSIDE]
subscribers[sub] = true,
receivers++, sub++
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++
SENDING
k == NUMBEROFSERVICES + DEVICES ||
receivers == 0
idleTime = 0
5
receivers > 0 &&
k < NUMBEROFSERVICES &&
subscribers[k]
sendToService[k]!
filtered[k] = e,
subscribers[k] = false,
k++, receivers––
k < NUMBEROFSERVICES +DEVICES &&
(!subscribers[k] && receivers > 0) ||
k == device+DEVICES
k++
Broker
publishedToBroker[k – DEVICES]!
k++
b2bPublish[k – DEVICES]!
filtered[k – DEVICES] = e,
subscribers[k] = false,
receivers––
k–DEVICES != device && subscribers[k] &&
k >= NUMBEROFSERVICES &&
k < NUMBEROFSERVICES +DEVICES

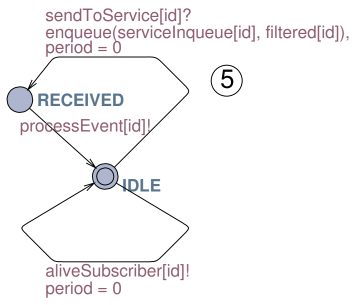
Figure 22: The automaton modelling how the broker filters incoming events.


alivePublisher[id]!
IDLE
publish[id]?
publishedToBroker!
initPublishToBroker!
dequeue(serviceOutqueue[id]),
returnvalue.receiver = -1,
enqueue(brokerInqueue, returnvalue)
PUB
C
1

(a) A service publishes an event.

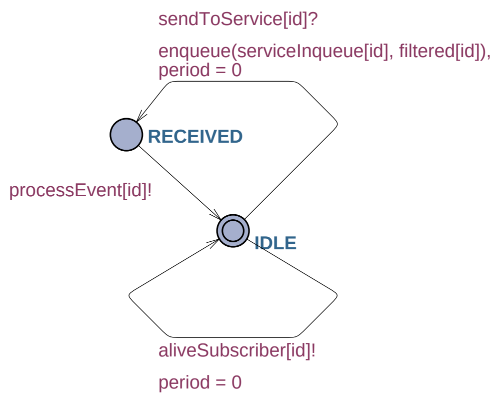


(b) The broker receives an event.

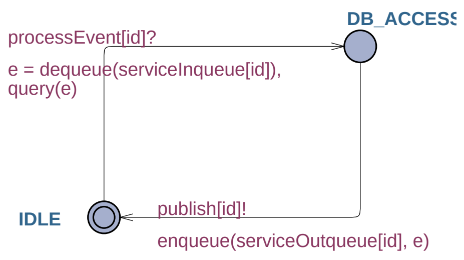


(c) A service receives an event.

Figure 21: The processes modelling how a service publishes events to and receives events from the broker. 21(a) shows the service publishing an event. 21(b) shows how the broker receives an event. 21(c) shows how a service receives an event from the broker.



(a) The database receives a query.


DB_ACCESS
processEvent[id]?
e = dequeue(serviceInqueue[id]),
query(e)
IDLE
publish[id]!
enqueue(serviceOutqueue[id], e)

(b) The database processes a query.

Figure 23: The automata modelling access to a TIP database. 23(a) model's the database service receiving a query and calling 23(b) by synchronising with process[id]!. 23(b) processes the query.

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, 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 (Figure 21(c)).

The TIP 3 Database There is a major difference between a server peer and a client peer: The server peer hosts the TIP database and a service providing access to a database, while a client peer interacts directly with the TIP user. We modelled the database as a service subscribing to database queries, cf. Figure 23. A service needing access to the TIP database publishes a database query to the broker. The broker filters the query and forwards it to the database service, cf. Figure 23(a). The database service executes the query (Figure 23(b) and enqueues the resulting event in its out-queue. It then publishes the result to the broker. The broker then forwards it to the requester.

Additional source and sink Services running on the server peer typically react to events published by client peers. The services publish events that are forwarded to client peers. We decided to add a source and sink process. The source process replaces the client peers that would be connected to the broker. It publishes events to the server broker, just like a client broker would do. The client process is represented by an automaton. It uses the same synchronisation channels to send messages to the server broker as the broker processes in the model of the communication between brokers do (compare Section 8.5).

8.4.2 Veri cation of the Model

Rather simple properties, assuring that a certain location can be reached, were verified during the modelling process, either through formal queries, e.g. `E<> iReg.REGISTERED`. This property verifies that the information service can register with the broker. When a subscribing service has registered with the broker, its functional conditions have successfully been evaluated by the observer. Likewise, its subscription rules have been evaluated by the observer and it has been subscribed to the events needed.

More complex properties examining the co-operation of services, e.g., `mapService.RECEIVED --> sink.RECEIVED` could not be verified. The property examines the co-operation between the server peer's map service providing map tiles to the client and the client peer's map service, needing map tiles. We wanted to ascertain that the server peer's map service provides a client peer's map service with map tiles. When the server peer's map service receives an event, it should eventually publish an event that is forwarded to the client. In our model, the sink represents the client.

A complete set of queries can be found on the accompanying CD.

8.5 Communication between Brokers

This section presents and discusses the model of communication between several brokers. Brokers subscribe to events from and forward their local events to other brokers.

8.5.1 The Model

Actors and processes involved In this model, brokers are the only actors. Brokers participate in several interactions: they connect to each other and (1) advertise and (2) process another brokers advertisement, that is they subscribe to event types. They (3) receive and filter events and (4) disconnect from one another.

(1) Advertising When two brokers connect to each other, each first creates an up-to-date advertisement by aggregating the advertisements published by its local services. The broker then advertises them to the other broker.

(2) Processing an advertisement When a broker receives an advertisement from another broker, it notifies the observer. The observer evaluates the subscription rules introduced in Section 6, from every local subscribing service. It then informs the broker about the needed event types. The broker sends the services' subscription to the foreign broker. When a broker subscribes to events from another broker, the observer monitors the connection and notifies its broker when the connection is lost.

Figure 24 shows the advertising and subscription automata and how the brokers communicate in detail. Minor automata, as those modelling the indices of publishers and subscribers, are not included for clarity reasons. The shown automata co-operate with them. Figure 25 shows a diagram of the actions and messages transmitted between the two brokers.
In Step 1, broker A (24(a)) aggregates the event types published by local services. In the second step, it synchronises with broker B through the binary synchronisation channel `advertise[device][id]!`. Figure 24(b) shows broker B receiving the advertisement. At first it waits for the synchronisation `advertise[e][id]?` in the IDLE location. After the synchronisation with broker A (Step 2) it updates its event indices. It then calls for the observer to re-evaluate the services' subscriptions (Step 3). When the observer returns in Step 4 (`updateAD?`), the broker checks if any data types from the external broker are needed by the local services. In Step 5, it synchronises with broker A via the synchronisation

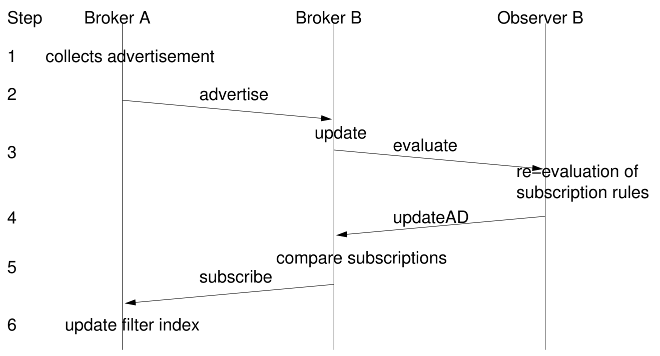


Figure 25: Broker A advertises its events to broker B. This sequence diagram shows messages transmitted and actions executed during the advertising process.

`subscribe[forDevice][id]!` and transmits its subscriptions. Broker A (Figure 24(c)) subsequently updates its filter index in Step 6.

(3) Receiving and ltering messages When a broker receives events from a service or another broker, it filters the events and transmits them to the subscribers.

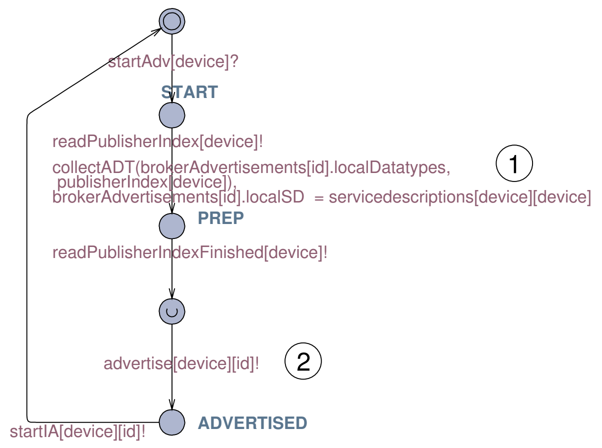
(4) Disconnecting TIP 3 is an information system for mobile users. The mobile devices move in and out of each others reach. When two TIP 3 peers loose contact their observers detects this after some time. The observer then signs off the remote peer. During the deregistration process, the advertisements and subscriptions from the remote broker are removed.

When a broker becomes aware of an imminent loss of connectivity or has to end the connection for some other reason, it first notifies the remote broker, so that it can deregister.

8.5.2 Veri cation of the Model

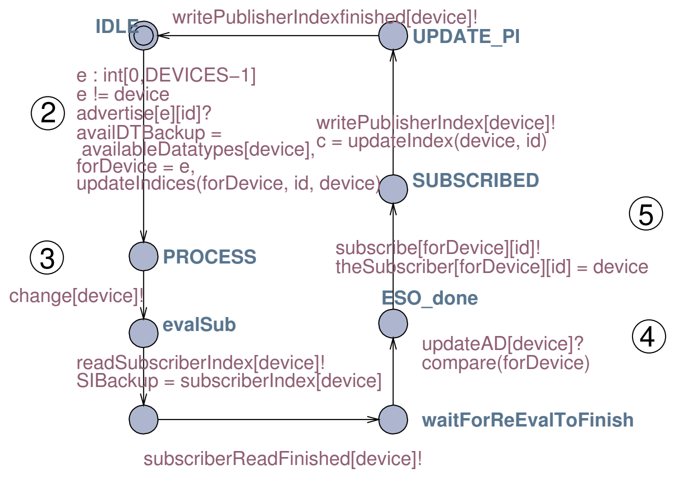
Simulation runs assured that simple properties, e.g., that a broker can indeed advertise to another broker, are verified. Another query, `E <> client1BR.RFB`, verifies the property that a client broker can receive events from a connected broker. The property holds for our model. However, even seemingly simple properties like this could only be verified by simulation runs. While examining this property, UPPAAL reached its memory limitations and aborted the verification. Therefore, we could not verify more complex queries, such as `A<> !(serverSource.sending and (serverBR.IDLE or serverBR.RFB)` are satisfied. The query examines whether a server peer service may publish an event to the broker while the broker either is idle or receives an event from a remote broker.

Our simulations showed that brokers may advertise to one another. Advertisements are processed and result in subscriptions. When a remote broker disconnects, its advertisements are removed. The observer re-evaluates the subscription rules and updates the subscriptions.



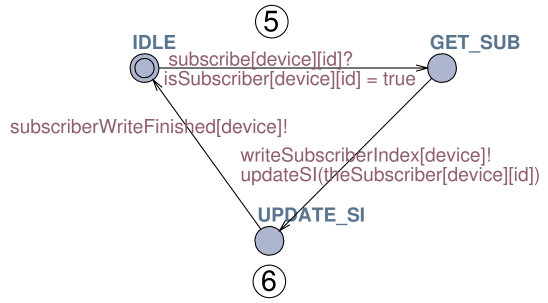
startAdv[device]?
START
readPublisherIndex[device]!
collectADT(brokerAdvertisements[id].localDatatypes,
publisherIndex[device]),
brokerAdvertisements[id].localSD = servicedescriptions[device][device]
1
PREP
readPublisherIndexFinished[device]!
advertise[device][id]!
2
startIA[device][id]!
ADVERTISED

(a) The advertising process from broker A.



IDLE
writePublisherIndexfinished[device]!
UPDATE_PI
e : int[0,DEVICES−1]
e != device
advertise[e][id]?
availDTBackup =
availableDatatypes[device],
forDevice = e,
updateIndices(forDevice, id, device)
2
writePublisherIndex[device]!
c = updateIndex(device, id)
SUBSCRIBED
5
3
PROCESS
subscribe[forDevice][id]!
theSubscriber[forDevice][id] = device
change[device]!
evalSub
ESO_done
updateAD[device]?
compare(forDevice)
4
readSubscriberIndex[device]!
SIBackup = subscriberIndex[device]
waitForReEvalToFinish
subscriberReadFinished[device]!

(b) Broker B processes the advertisement from broker A.



5
IDLE
GET_SUB
subscribe[device][id]?
isSubscriber[device][id] = true
subscriberWriteFinished[device]!
writeSubscriberIndex[device]!
updateSI(theSubscriber[device][id])
UPDATE_SI
6

(c) Broker A receives the subscriptions from broker B.

Figure 24: The three main automata modelling the advertising and subscription process.

8.6 Joining the Models

Although we would have liked to create one single model, we had to split it into several parts due to UPPAAL's limited memory, discussed in the following section. The models of the server peer, the client peer and the communication between brokers were presented in the previous sections. These three models may be joined to one single model. Their common interface are the filter and receive processes, shown in Figure 26 and Figure 27.

The automaton representing the filter process is basically the same in all three models. In Figure 27, the relevant part of the automaton modelling the filter process for communication between brokers is shown. The brokers from the client peer (27(a)) and server peer model (27(b)) are identical. The only differences between Figures 27(a) and Figure 27(b) on the one hand and Figure 27(c) on the other hand are how the synchronisation channel `b2bPublish[]!` is selected, and the broker in Figure 27(c) avoiding to forward events to itself.

Let us consider a situation with two brokers A and B, located on peer A respectively peer B. They already have subscribed to each others data, as described in Section 8.5. When a service on peer A publishes data to broker A, the broker filters the data. If peer B subscribed to this event, broker A filters the data and synchronises with broker B through the synchronisation `b2bPublish[B]!`. If a local service on peer A subscribed to the event, broker A synchronises with `send-ToService!` and forwards the event to the service. These two synchronisations are available in all three automata presenting the filter process.

The automata that model how the broker receives data are the same in all three models. They are shown in Figure 27. When a broker publishes an event to another broker, it initialises the transmission with the `publishToBroker[k]!` synchronisation. The receiver uses the same synchronisation and switches to the RFB (Received From Broker) location. The transmission is completed when both processes synchronise with `publishedToBroker`. The received event is then enqueued in broker B's in-queue.

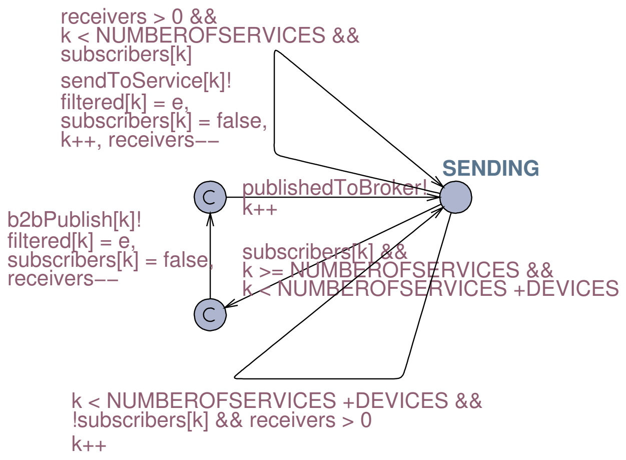
A service cannot receive a message that was sent to a broker, because services cannot synchronise with `publishTo-Broker[k]?`.

The automata that model how a broker filters and receives events require minor adjustments if the three models are merged. No other automata have to be changed.

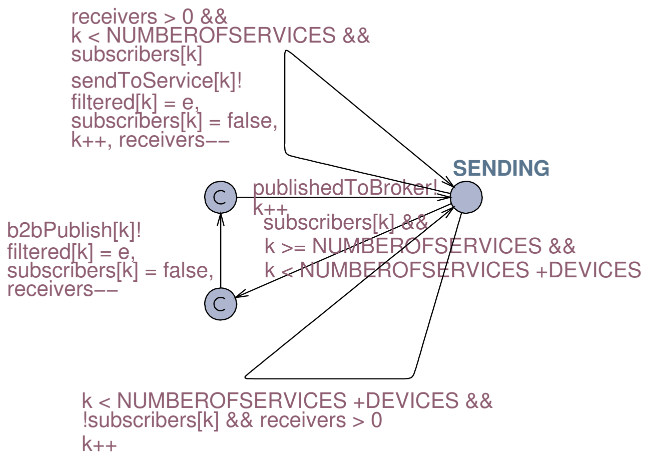
8.7 Issues encounteres

This section discusses some major issues we encountered during the modelling process. The problem that affected us most is the fact that UPPAAL can address only 4GB in memory[6].

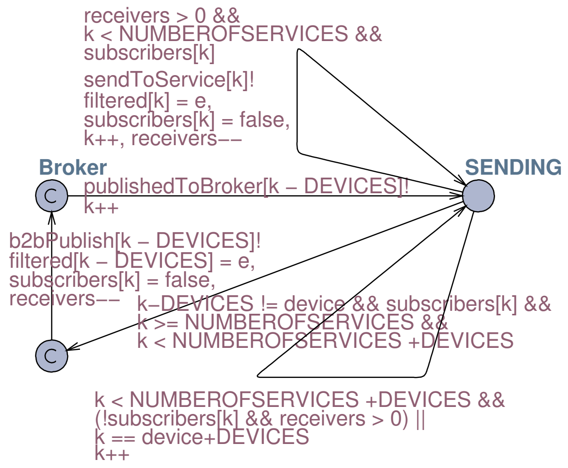
[6]see the statement made by Gerd Behrmann, associate professor at Aalborg University and UPPAAL developer, `http://bugsy.dominic.auc.dk/cgi-bin/bugzilla/show_bug.cgi?id=63`: "(...) First, uppaal is a 32-bit process (even on 64-bit Solaris). This means that there is no way that uppaal can address more than 4GB of memory.(...)"



(a) The client peer model.



(b) The server peer model.



(c) The broker communication model.

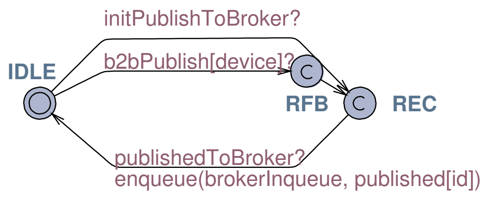
Figure 27: The relevant part from the filter processes. 27(a) from the client peer model, 27(b) from the server peer model and 27(c) from the communication between brokers.

During the verification process, UPPAAL explores every possibly reachable state of the model and examines if the property still holds. Especially when verifying safety and liveness properties UPPAAL often needed more than 4GB of memory. It then aborted the verification process before reaching a result. Therefore we decided to split the model into several parts early in the modelling process. However, even the split models soon became too big for UPPAAL. The execution of a simple sanity check could take several hours on a Sun Fire 480R with four 900 MHz Sparc III CPUs and 32 GB of memory. During the simulated execution of a model that had run awlessly in earlier simulation runs, suddenly a deadlock could occur.
Hence we could not prove that the model is deadlock free. Some properties could not be examined, so we often had to rely on the simulation of the models.

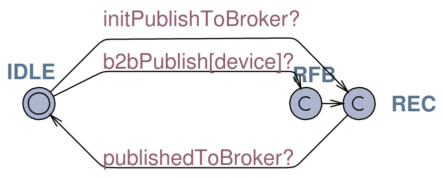
The model could have been split into more parts, e.g., one model of the service registration process, one for the GUI and user interaction, one for the filtering and receiving of events etc. However, joining the models would have become more difficult. Also the models share common data, e.g., all models of the client peer share the the indices of published events and subscriptions. These shared data are modified by several processe such as the service registration or filtering of incoming events by the broker. Hence the shared data have either to be created in the different models, or to be passed as a template parameter. Passing the shared data as a parameter to a process is a very likely source of error.

Also the different processes interact with one another. When a publisher registers with the broker, other service's subscription rules are evaluated by an observer process and subscriptions may be changed. The same observer process evaluates a service's subscription rules during the service registration process or when a publisher disconnects from the server. Several processes would be part of more than one model, the models would not be as small as aimed for.

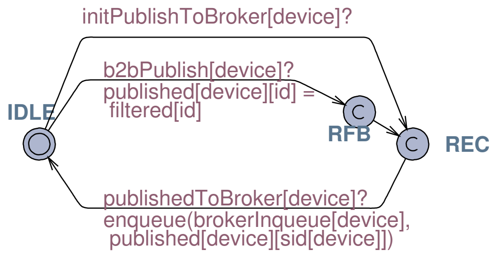
Another problem we encountered is that UPPAAL cannot model the observer pattern. Instead of simply observing the communication between brokers and services or brokers and brokers, the observed objects have to actively synchronise and communicate with the observer.



(a) The client peer model.



(b) The server peer model.


initPublishToBroker[device]?
b2bPublish[device]?
published[device][id] =
filtered[id]
IDLE
RFB
REC
publishedToBroker[device]?
enqueue(brokerInqueue[device],
published[device][sid[device]])

(c) The broker communication model.

Figure 26: The receiver processes from the three models. 26(a) from the client peer model, 26(b) from the server peer model and 26(c) from the model of the communication between brokers.

8.8 Comparison to the Requirements

In Section 3.2 we defined our requirements for a new architecture and design for the TIP 3 system. We identified six requirements whose fulfilment we now discuss in detail:

(1 a) Prompt communication TIP 3 services do not communicate directly. They communicate indirectly, i.e., a service publishes its event to the broker. The broker then filters the event and forwards it to the respective subscribers. In our model, the broker regularly checks for new events in its in-queue. Figure 28 shows a part of the filtering automaton. We added a clock `timeout` to the automaton and the invariant `timeout < 10` to the filter automaton's IDLE location. This forces the automaton to leave the location and check for new events before the timeout is reached.

The FILTERING location is urgent. This means that time cannot pass as long as the automaton stays in this location.

The LOOP location and the transitions leaving it do not depend on any other processes.

Several synchronisations leave the SENDING location. When the broker filters a message to a local service, the service immediately synchronises with `sendToService[id]?`. The service cannot block the broker, as it receives messages only from the broker. When the broker needs to forward an event to another broker, it first moves to a committed location, firing the edge with the guarding expression `subscribers[k] && k >= NUMBEROFSERVICES && k < NUMBEROFSERVICES +DEVICES`. When a process is in a committed location, the whole system is in a committed state. While the system is in a committed state, the transitions leaving a committed location are prioritised. Transitions leaving normal or urgent states are not fired until the system has left the committed state. Hence, publishing an event to a remote broker does not block the broker.

The internal transition returning from the SENDING to the IDLE location does not depend on another process. We therefore argue that the filtering of events takes limited time. Hence, TIP 3 meets this requirement.

(1 b) Local communication Services located on the same device communicate locally without detour to the server peer. The broker filters the evens and forwards them directly to the subscribers. TIP 3 meets this requirement.

(2) Service management After a publisher disconnects, its event types are removed from the index of available events. Every subscriber's subscription rules are evaluated and subscriptions are renewed. When a new publisher registers with the broker, its event types are added to the index of available events. The subscription rules of subscribing services are re-evaluated by the observer. Subscriptions are renewed when necessary. Subscribers always subscribe to those events they prefer the most among those available. TIP 3 meets this requirement.

(3) Rule-based subscriptions Rule-based subscriptions are the basis for the TIP 3 design. Subscribing services provide a set of subscription rules, enabling the observer to select the most appropriate events out of those available. TIP 3 clearly meets our requirements on rule-based subscriptions.

(4) Server management A TIP 3 broker may connect to several other brokers at once, both to server peers and to client peers. An implementation might enable the user to choose whether to connect to several brokers, and to choose between a peer-to-peer mode and a client-server mode. TIP 3 meets our requirements on server management.

(5) Event classification The data, or events, in TIP 3 are coarsely classified into event categories. TIP 3 mostly meets this requirement.

(6) Privacy and confidentiality This model of TIP 3 does not address issues such as privacy and confidentiality, beyond the characteristics provided by the publish subscribe scheme.

9 Summary and Future Work

The focus of this paper lay on the re-design of the TIP system. We identified the basic features TIP should offer: (1) TIP

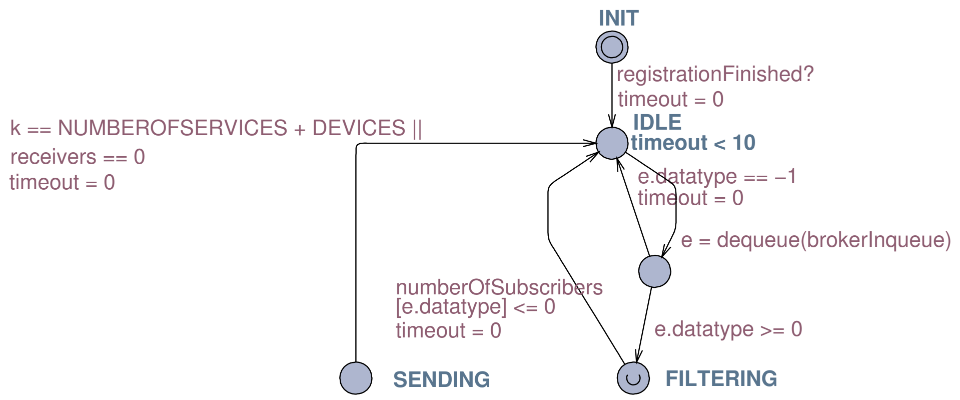


Figure 28: A part of the automaton representing the filtering process, c.f. Figure 8.4.1. The IDLE location must be left before the clock "timeout" reaches its timeout of 10.

Table 3: Related Work, TIP 3 and our requirements: ++ = completely met; + = mostly met; − = partly met; −− = not met.

Requirements	Related Work						
	Dynamic Tour Guide	George Square System	GUIDE	TIP 2	TIP 3*	FACTS	TIP 3
1a. Prompt Communication	++	+	++	+	++	++	++
1b. Local Communication	++	++	−	−−	++	++	++
2. Service Management	−−	−−	−	−−	−−	+	++
3. Rule-Based Subscriptions	−	++	−	−	−	++	++
4. Server Management	−−	+	−	−−	+	−	++
5a. Event Classification	+	+	−	+	+	+	+
5b. Data Trustworthiness	−−	−−	−−	−−	−−	−−	−−
6. Privacy and Confidentiality	−−	−−	−−	+	+	−−	+−

Table 4: Features the TIP system should provide.

	TIP 2	TIP 3*	TIP 3
(1) extendability	−	−	++
(2) extremely loose coupling	−	−	++
(3) combine event-driven and service-oriented architectures	−	+	++

should be easily extendable, enabling new services to cooperate with existing ones. (2) Services should be extremely loosely coupled. (3) The design should combine event-driven communication with service-oriented features. These three characteristics enable the TIP system to adjust to a dynamic environment and to the changing availability of services and events. Table 4 summarises the comparison of TIP 2, TIP 3* and TIP 3. The TIP 2 implementation did not realise any of these features, although the design included some. TIP 3* extended the TIP 2 design. It made first steps towards service-orientation. However, the main focus of TIP 3* lay on the introduction of caching and pre-fetching mechanisms. TIP 3 is easily extendable. Its services are extremely loosely coupled. It combines the asynchronous communication style typical for event-driven architectures with the advantages of service oriented architectures.

The project consisted of theoretical and practical parts. In the first theoretical part, Section 1 defined our understanding of location based systems and context-awareness. Section 2 discussed the current TIP 2 system. Section 3 analysed the requirements. Section 4 reviewed related work. In Section 5, service-oriented and event-driven architectures were discussed and compared. The first practical part, Section 6, introduced and discussed the architecture and design of TIP 3. The following theoretical part, Section 7, introduced formal modelling. The last Section 8 presented and discussed the model.

All major goals of the project were achieved. TIP 3 services provide functional conditions that are evaluated during the service startup and registration process. Functional conditions and subscription rules are explained in Section 6.1, see page 20. If events essential for the service's functionality have not been advertised, the service cannot register. TIP 3 services may subscribe to events published by several services. When a publisher deregisters or a new publisher registers to the broker, the services' subscription rules are re-evaluated. The service may then be subscribed to new events. Old subscriptions may be revoked. Services therefore always subscribe to the best events available. We consider the goals service management and rule-based subscriptions to be obtained.
The TIP broker can communicate with several remote brokers at the same time. We consider the goal "server management" to be reached, as well. A future model or implementation should enable the user to decide if the broker should connect to a remote broker. The user should be able to choose between a peer-to-peer mode and connecting to one or many server peers (client-server mode).
The goal to provide a framework for the analysis and examination of services could only be partly achieved. Future work may include using a future 64bit version of UPPAAL, or a different modelling tool with different memory management or higher memory limitations. This would enable researchers to join our three models into one global model, and to verify the remaining properties. We also developed a design for TIP combining service-orientated aspects with event-driven communication.

A future model or implementation may help researchers learn more about the effect of differently formulated rules. However, the overall behaviour of the system and the interaction of services have already been described by the model.
A future model or implementation will have to decide on an ontology for the categorisation of events and services. We supplied a coarse categorisation for a small number of event types and services. However, the design assumes that event and service categories are known (cf. Section 6). The application of an ontology in our model would have enlarged the model. It would have unnecessarily complicated the service advertisements and service descriptions as well as the services' functional conditions and subscription rules. We therefore decided not to use one of the existing ontologies. However, there are several ontologies available that may be used in TIP, such as the OWL[7] ontology. Another possibility is the application of adaptive group-based service discovery as suggested by Katharina Hahn [10].
A future model or implementation may examine how services contact the broker, or how a broker connects to another broker. In Section 6, we abstracted from these technical details.
A future model or implementation may have to address the problem arising out of the way subscription rules are re-evaluated every time a new publishing service advertises to the broker. At the moment, an advertisement to the broker inevitably leads to a re-evaluation of all subscription rules. When several services register consecutively, the subscription rules are evaluated several times. A solution could be to re-evaluate subscription rules of already registered services when every current service registration is completed.
A future model or implementation should address the requirements the model did not address, i.e., rating of data, privacy and confidentiality. The implementation may be used to explore both the usability of the TIP user interface as well as the usability of the design.

The model could be extended, enabling further exploration and analysis of the system's behaviour in a MANET or multi-hop peer-to-peer network.
We successfully introduced the concepts of functional conditions and subscription rules to TIP. This paper reports on a proof-of-concept, demonstrating the applicability of the concepts of rules and conditions. Future works could focus on a more detailed model of functional conditions or subscription rules. The effect of different kinds of rules or conditions may be examined in a more detailed model. However, this task goes beyond the scope of this paper.

The development of a model checking tool targeted at the examination and analysis of large-scale publish/subscribe systems would be an interesting and challenging project. UPPAAL showed its limitations during the modelling process, as many other modelling tools would have done as well. Our future research aims at developing new formal methods and new modelling techniques, that is, new modelling languages, logic and methods, for collaborating services in context-aware mobile systems. This paper contributes to the project with a thorough understanding of the strengths and weaknesses of current modelling methods when applied to event-based ser-

[7]http://www.w3.org/TR/owl-ref/

vice collaboration.

References

[1] L. Baresi, C. Ghezzi, and L. Zanolin. *Testing Commercial-off-the-Shelf Components and Systems*, chapter Modeling and Validation of Publish/Subscribe Architectures, pages 273–291. Springer, Berlin Heidelberg, Dec. 2005.

[2] G. Behrmann, A. David, and K. G. Larsen. A Tutorial on UPPAAL. In *Proceedings of Formal Methods for the Design of Real-Time Systems(SFM-RT 2004)*, number 3185 in LNCS, pages 200–236, September 2004.

[3] G. Behrmann, A. David, K. G. Larsen, O. Möller, P. Pettersson, and W. Yi. UPPAAL - Present and Future. In *Proc. of 40*th *IEEE Conference on Decision and Control*, 2001.

[4] J. Bengtsson and W. Yi. Timed Automata: Semantics, Algorithms and Tools. In J. Desel, W. Reisig, and G. Rozenberg, editors, *Lecture Notes on Concurrency and Petri Nets*, number 3098 in LNCS. Springer–Verlag, 2004.

[5] K. Cheverst, N. Davies, K. Mitchell, and A. Friday. Experiences of developing and deploying a context-aware tourist guide: the GUIDE project. In *Mobile Computing and Networking*, pages 20–31, Boston, USA, Aug. 2000.

[6] A. David and W. Yi. Modelling and Analysis of a Commercial Field Bus Protocol. In *Proceedings of the 12th Euromicro Conference on Real Time Systems*, pages 165–172. IEEE Computer Society, 2000.

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

[8] 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, Department of Computer Science, August 2008.

[9] W. Fokkink, A. Kakebeen, and J. Pang. Adapting the Uppaal Model of a Distributed Lift System. In *Proceedings of the 2nd IPM Symposium on Fundamentals of Software Engineering*, volume 4767 of *Lecture Notes in Computer Science*, pages 81–97. Springer–Verlag, 2007.

[10] K. Hahn. Exploring Mobility and Service Discovery in MANETs. PhD thesis – work in progress, 2008.

[11] M. D. Harrison, J. C. Campos, and K. Loer. *Research Methods in Human Computer Interaction*, chapter Formal Analysis of Interactive Systems: Opportunities and Weaknesses. Cambridge University Press, Cambridge, United Kingdom, August 2008. To appear.

[12] M. D. Harrison, C. Kray, and J. C. Campos. Exploring an option space to engineer a ubiquitous computing system. In P. Curzon and A. Cerone, editors, *Pre-proceedings of FMIS'07*, pages 67–82, 2007.

[13] K. Havelund, A. Skou, K. G. Larsen, and K. Lund. Formal Modelling and Analysis of an Audio/Video Protocol: An Industrial Case Study Using Uppaal. In *Proceedings of the 18th IEEE Real-Time Systems Symposium*, pages 2–13, San Francisco, California, USA, 3-5 December 1997.

[14] A. Hinze. Location-based system TIP: Challenges of a multi-faceted project, 2006. talk held at Freie Universität Berlin, August 2006.

[15] A. Hinze and G. Buchanan. The Challenge of Creating Cooperating Mobile Services: Experiences and Lessons Learned. In *Proceedings of the 29th Australasian Computer Science Conference (ACSC 2006)*, volume 48 of *CRPIT*, pages 207–215, Hobart, Australia, 2006. ACS.

[16] A. Hinze and S. Junmanee. Advanced Recommendation in a Mobile Tourist Information System. Technical Report 04/2005, Department of Computer Science, University of Waikato, May 2005.

[17] A. Hinze, P. Malik, and R. Malik. Towards a TIP 3.0 Service-Oriented Architecture: Interaction Design. Technical Report 08/2005, Dept. of Computer Science, University of Waikato, Aug. 2005.

[18] A. Hinze and A. Voisard. Combining Event Notification Services and Location-based Services in Tourism. Technical report, Freie Universitaet Berlin, 2003.

[19] A. Hinze and A. Voisard. Location- and Time-based Information Delivery in Tourism. In *Proceedings of Conference in Advances in Spatial and Temporal Databases (SSTD 2003)*, volume 2750 of *LNCS*, Santorini Island, Greece, July 2003.

[20] N. Hristova, G. O'Hare, and T. Lowen. Agent-based Ubiquitous Systems: 9 Lessons Learnt. In *Proceedings of UbiSys '03, System Support for Ubiquitous Computing Workshop*, Seattle, Washington, USA, October 2003. UbiComp 2003, The Fifth Annual Conference on Ubiquitous Computing.

[21] R. Kramer, M. Modsching, and K. ten Hagen. Development and Evaluation of a Context-driven, Mobile Tourist Guide. *International Journal of Pervasive Computing and Communication*, March 2005.

[22] R. Kramer, M. Modsching, K. ten Hagen, and U. Gretzel. Behavioural Impacts of Mobile Tour Guides, 2007.

[23] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, Oct. 1997.

[24] K. Li, K. Verma, R. Mulye, R. Rabbani, J. A. Miller, and A. P. Sheth. *Semantic Web Services, Processes and Applications*, chapter Designing Semantic Web Processes: the WSDL-S Approach. Springer, New York, NY, USA, 2006.

[25] K. Löf er. User-Adapted Information Delivery in Context-Aware Systems. Master's thesis, Freie Universität Berlin, Department of Computer Science, Jan. 2004.

[26] Y. Michel. Location-aware caching in mobile environments. Master's thesis, Freie Universität Berlin, Department of Computer Science, June 2006.

[27] M. Modsching, R. Kramer, K. ten Hagen, and U. Gretzel. Effectiveness of Mobile Recommender Systems for Tourist Destinations: A User Evaluation, 2007.

[28] G. Mühl, L. Fliege, and P. R. Pietzuch. *Distributed Event-Based Systems*. Springer, Berlin, Heidelberg, 2006.

[29] M. Nagarajan. *Semantic Web Services, Processes and Applications*, chapter Semantic Annotations in Web Services. Springer, New York, NY, USA, 2006.

[30] P. Ottlinger. Design and Implementation of an extensible Software Architecture for Distributing Context-sensitive Information (in German). Master's thesis, Freie Universitaet Berlin, Department of Computer Science, June 2004.

[31] K. Ter oth, G. Wittenburg, and J. Schiller. FACTS - A Rule-Based Middleware Architecture for Wireless Sensor Networks. In *Proceedings of the First International Conference on COMmunication System softWAre and MiddlewaRE (COMSWARE'06)*, New Delhi, India, Jan. 2006.

[32] K. Virrantaus, J. Markkala, A. Garmash, V. Terziyan, J. Veijalainen, A. Katanosov, and H. Tirri. Developing GIS-Supported Location-Based Services. In *Proceedings of WGIS 2001 - First International Workshop on Web Geographical Information Systems*, pages 423–432, Kyoto, Japan, 3-6 December 2001.

[33] Web Services Architecture Group. Web Services Architecture. W3C Working Group Note 11 February 2004, February 2004.

[34] A. Wenzler. Web Services and Service Oriented Architecture, June 2004.

[35] M. Zaremba, M. Kerrigan, A. Mocan, and M. Moran. *Semantic Web Services, Processes and Applications*, chapter Web services Modeling Intology. Springer, New York, NY, USA, 2006.