University of Groningen
Department of Computing Science

LogicaCMG, BU Groningen
EAI Competence Team

The Software Architecture of RFID Systems

Master of Science Thesis

Niels Heinis

March 2005

Supervisors:
R. Smedinga, University of Groningen
R. Boverhuis, LogicaCMG
J. Jongejan, University of Groningen

Table of Contents

Management summary

Introduction

The last few years, Radio Frequency Identification has been an enormous hype. Especially trading companies are very interested. One of the major issues with RFID is the enormous amount of events and data that needs to be processed when implementing RFID company or supply chain wide. This issue is the basis for the master's research presented in this report. It focuses on the handling of data from sensor-based systems, and RFID systems in particular, from a software architectural point of view.

Problem statement

The primary goal of this thesis is to compare two architectural styles to determine which one could best be used for sensor-based systems: a centralised or a distributed architecture. Criteria are: performance, reliability and flexibility. Also costs will be taken into account.

Problem analysis

Centralised architectures

A centralised architecture is defined as the architecture in which all sensors are directly connected to the central system and that central system does all the processing. A central database is used for storage.

Distributed architectures

In a distributed architecture, functionality is spread over multiple smaller subsystems, that are often geographically dispersed. Typically, each subsystem services only a part of the readers of the whole system. Often a centralised part will still be needed, but only for part of the processing. Depending on the functionality, databases can also be added to the subsystems.

Performance

In the centralised architecture all events are published to the central system that needs to do all the work. In a distributed architecture, the work load can be divided over multiple distributed components by publishing events to subsystems only. In both cases performance optimisation techniques like load balancing can be applied.

A very valuable part of the processing work is correlating events to other events which gives it a more important meaning. This requires extra processing, but also retrieval of data. In a centralised situation, all this data is centrally available, in a distributed situation it probably must be fetched from another database or multiple databases. A solution to this is an Object Naming Service that centrally stores where which data is available. This does, however, require quite some overhead.

Finally, there is a real-time aspect. In a centralised system it will be relatively hard to achieve real time responses. In a distributed architecture this will be easier.

Reliability

In the centralised architecture, there will be one component receiving all incoming events, at least for a specific type of event. If that component fails, the system will not receive any more events. Failure of other components will also harm the correct functioning of the system, but

then the impact is lower. If a distributed component fails, only that part of the system stops functioning. The most important difference is that failure of a distributed component has only a limited impact compared to the failure of a central component. In both cases, certified messaging and fault tolerant or load balancing techniques can increase reliability. Also, the reliability of databases highly influences the reliability of a system.

Flexibility

When presuming a component-based architecture, both architectures can be very maintainable and scalable. In a distributed architecture, one subsystem can easily be stopped for maintenance, while the rest of the system keeps operating correctly. In the centralised solution this can only be achieved by duplicating all components so that one can always be operating. The drawback of the distributed architecture is that there are many components sharing the same functionality, so that, for changes, many components must be updated.

Costs

The number of machines needed in a centralised solution is much less than in a distributed solution, which positively influences the costs: hardware, infrastructure and maintenance costs are relatively low for a centralised system. A distributed solution is much more expensive. More machines are involved, which requires a larger and better infrastructure, more hardware and thus higher maintenance costs. Also extra database systems are quite expensive.

Case

For validation purposes a prototype has been built. It is a prototype of what public transport could look like when RFID is incorporated in it. The goal is to no longer use all different kinds of public transport tickets, but just give everyone an RFID smart card. By registering where and when a person hops on and off, pricing and billing can take place automatically.

Advantages for companies are personal services, advertisements and accurate statistical information and billing procedures. The main advantage for travellers is not having to think about buying the correct ticket and having the right card for the used transport types.

Validation

In the prototype, the following events were defined: access a station, enter a train, exit a train and leave a station. These events, in turn, cause other events. If the prototype system was implemented for Dutch Railways, a message rate of 350 messages per second should be met. In the centralised architecture, the central application needs to handle all of these messages. In the distributed architecture, the central part only needs to handle about 200 messages per second when only the first and the last events mentioned above are handled by distributed components. That means that using a distributed architecture can reduce the message load by 40% in the presented situation. Reliability and flexibility tests showed the expected behaviour.

Conclusion

The final conclusion of this research must be that there is not one answer to the question whether a centralised or a distributed architecture is best for an RFID-based system. The choice of which style to use as a foundation for such a system merely depends on the size of such a system, its expected growth in the future and the number of locations. It would also be a possibility to combine both styles. Concluding, it can be said that it is very likely that the cost aspect will be decisive and that the final choice will be a business rather than a technical decision.

1 Introduction

1.1 About this thesis

The last few years Radio Frequency Identification has been an enormous hype. The subject is in the news every day and many pilot projects are started. Especially trading companies are very interested and have the feeling they have to join the hype in order to gain knowledge about RFID.

One of the major issues with RFID is the enormous amount of events and data that needs to be processed when implementing RFID company wide or supply chain wide. This issue is the basis for the master's research presented in this report.

This research will focus on the handling of data from sensor-based systems, and RFID systems in particular, from a software architectural point of view. Two architectural styles will be defined that could be used for such systems. An analysis of both styles will follow, where especially performance, reliability and flexibility are important quality attributes.

This report starts with some background information on software architectures and architectural styles, a technical description of RFID technology, and a few words on two modern architectural styles in chapter 2. In chapter 3, the precise problem is stated, which is then analysed in chapter 4. In chapter 5 a case is presented, that has also been implemented as a demo, to be able to validate the research results in chapter 6. Chapter 7 concludes the research by presenting answers to the research questions. Appendix A provides more detailed information on RFID and Appendix B finally describes the built demo in detail.

1.2 Internship

This research has been carried out externally at the LogicaCMG office in Groningen. LogicaCMG offered the author an internship position and technological expertise needed to carry out the research and build a demo set-up for testing purposes.

1.3 Acknowledgements

The author would like to thank:

- René Boverhuis (LogicaCMG) for his ideas, support, constructive feedback and the pleasant collaboration during my internship.
- Jan Bosch (University of Groningen) for his support and ideas during the start of this project.
- Rein Smedinga (University of Groningen) for his feedback and for guiding me throughout this project.
- LogicaCMG / BU Groningen for offering me an internship position for ten months.
- LogicaCMG / RFID CC Rotterdam for providing RFID hardware and showing their interest.
- Karin for her patience and loving support.

2 Background & Related work

2.1 Software architecture

The architecture of a software system is a detailed description of that system's main components and their mutual relations and connections, according to which an application is implemented. It is the software architect's job to make sure a software architecture supports all functional and technical requirements and quality criteria that must be met by the system to be built.

The goal of this research is not to create yet another architecture. It will focus on a higher, less detailed level: software architectural styles.

2.1.1 Architectural styles

This thesis is about comparing two software architectural styles. Before starting on this, a little bit of common knowledge about architectural styles is required. Therefore, the following questions will need to be answered first.

- *What is an architectural style?* - Bosch [1] describes architectural styles as follows: "An architectural style (or pattern) is the main, characteristic organisation of a software system aiming to satisfy the most essential requirements of the system". So, an architectural style is a description of the most important components of a software architecture for a certain type of system.
- *Does an architectural style give enough information to just use it as the architecture of a system?* - No, an architectural style only gives a rough sketch of a system. It is meant to be a starting point that at least needs to be refined (detailed design) but should probably also be modified to suit the designer's needs.
- *What advantages do architectural styles provide?* - Since there are a number of different styles which all support specific kinds of systems, one can, by determining what kind of system needs to be designed, choose a certain style as the starting point of the new system. This way, the most important parts of the system are already described, so that they cannot be forgotten.

Software systems are seldom based on only one architectural style. Often several styles are combined.

2.1.2 Assessment of software architectures

So, an architectural style is not the same as a software architecture because the latter is much more detailed. That influences the way software architectural styles can be assessed. For assessing software architectures, formal methods like Pasa [2] and Architecture Based Software Reliability [3] were developed but since architectural styles do not provide enough details, these methods are not usable for assessing architectural styles. They have to be assessed on a higher level, using a more analytical approach.

2.1.3 *Evolution of software architectures*

Although often seen as monolithic, a centralised software system may still consist of a number of components that are connected to each other in some way. It is the way these components are connected that make the system monolithic. The separate components often interact with each other by direct process calls due to which the processes, if separate, of all the components need to run on the same machine. Also shared memory might be used for inter-component communication.

Over time, the design of software has changed a lot and systems became less monolithic. One of the first steps taken to design more flexible systems was by designing multi-tiered architectures [4], which provided the possibility to place different components on different machines. Although this was a step forward, it was still necessary to exactly know well each part of the system to be able to use it. To overcome this, techniques like message passing were developed for communication, which made coupling of components less tight. For example, by using remote method invocation (RMI) it is possible to communicate with other components (processes) that not necessarily reside on the same physical system. It is then sufficient to know only the interfaces of the components.

The following subsections present loosely coupled architectures and the way communication is implemented between the architecture's components.

2.1.4 *Implicit invocation & Certified messaging*

To loosen the coupling between components of a software systems in order to decrease dependencies and increase maintainability, implicit invocation can be used. In a system based on implicit invocation the system is organised in components that generate and consume events. Components register themselves as having an interest in receiving certain events. This way communication between components is merely event handling. [5]

Implicit invocation is often implemented by using message passing. There are roughly two methods for message-based communication: uncertified and certified. The latter is also called guaranteed messaging and provides functionality for ensuring that each message will be delivered successfully (messages are for example re-sent if delivery fails or takes too long). Since every single message must be acknowledged, certified messaging does take overhead.

2.1.5 *Service-Oriented software architectures*

Essentially, a service-oriented software architecture is a collection of services that, together, form an application. By making sure that every service is clearly outlined and has a specific function, other components or services of the system can use these services if they require its specific functionality.

Because services have a strictly defined functionality, they can easily be put in separate processes. This enables the possibility to run a service on another machine or on a completely different place in the network. This can, for example, be done to gain a performance increase through load balancing. Also, a service can easily be re-used.

A service-oriented architecture is very suitable for a system based on implicit invocation. A service typically listens to specific request messages, processes it (probably using the data available in the message) and publishes a response.

2.1.6 Event-driven software architectures

Every tag that is read in an RFID system, represents an event. An event typically carries some data and/or has a certain meaning. In combination with earlier or upcoming events, the meaning of an event becomes more valuable. So, events are especially valuable when they are linked to others. Therefore, event correlation techniques were developed.

An event-driven architecture typically supports event-based systems and can thus be very valuable for RFID systems. According to Gartner [6], the combination of a service-oriented architecture and an event-driven architecture is most valuable.

2.2 Radio Frequency Identification

2.2.1 Introduction

Radio Frequency IDentification (RFID) is basically a technique that enables wireless and automated identification of objects. These objects need to have a so called "tag", which actually is a small transponder, to be able to identify them with a reader using radio waves. The tag can, depending on its type, contain a certain amount of data that can be handled by a system behind the reader. RFID is very often compared to the legacy bar code that is almost on every product. Indeed, one could think of replacing or extending these bar codes with a tag containing at least the same information, which would make automated scanning of products easier. But RFID offers a lot more features, which allows it to be used for numerous other applications as well.

At the time of writing this report, Radio Frequency Identification is a real hot item. It is in the news very often and a lot of (large) companies are doing pilots with RFID technology in various environments, although very often in supply chains. A benchmark study performed by LogicaCMG [7] shows that companies think that RFID can solve a lot of business pains they go through. It is quite surprising to see that the choice of starting to explore RFID technology is most of the times based on intuition only. Often a cost-revenue analysis is not performed and technical issues are believed to be solved within reasonable time. So, people are convinced that RFID will be widely adopted within a few years and feel the need to invest in knowledge of and experience with it, all based on intuition.

In this project there are of course other reasons to incorporate RFID technology. First of all LogicaCMG in Groningen wanted to know more about this technology in general and would also like to be able to easily integrate RFID in the existing IT systems of its customers. Another reason is that one of the known problems with RFID is that an enormous amount of data is generated and people do not really know how to handle that adequately. The last reason makes RFID very useful for validation of the research conclusions stated in section 4.7.

The following subsections provide some basic information on available standards and show some examples of known applications and issues. Also, a few words are dedicated to the privacy aspect. For more details on RFID technology, please refer to Appendix A.

2.2.2 EPC Global

EPCGlobal [8] is an international standards organisation that provides a global service for electronic product codes (EPC). EPCGlobal has a mission to enable true visibility of information about items in the supply chain. Therefore the EPCGlobal Network is being developed.

The EPCGlobal Network employs EPC and RFID technologies and provides a global standard framework for product information exchange. The goal is to be able to "identify any object anywhere automatically". An important part of the EPCGlobal Network is the Object Naming Service (ONS) that provides a way for business information systems to match the EPC to information about the associated item. By querying the ONS, providing an EPC that was read from a tag, the address of the machine that has more information on the product belonging to the tag is retrieved. Using that address, the specific product information can be retrieved.

2.2.3 Known applications

Although most companies are currently only experimenting with RFID and widespread implementation will still take a few years, there are already some systems in production. Here is a list of some examples.

- From January 2005 the USA started to put RFID tags in their passports to prevent fraud.
- In 1999 the city of Vejle, Denmark, successfully implemented an RFID Automatic Vehicle Location System to track all their buses and use this information to inform passengers about actual arrival and departure times.
- For security reasons 160 employees of the anti-crime centre in Mexico City have been implanted with RFID tracking chips.

2.2.4 Known issues

RFID is a hype and many companies want to start using it for various applications, but there are a number of issues that still must be solved or that need to be worked around.

- Standards on communication, used frequencies and data structures are not fixated yet. Authorities like EPCGlobal work on such standards. For world wide implementations of RFID, such standards must be clearly defined.
- Defining standards is complicated. Every country manages its own frequency ranges which results in the same frequencies not being available in all countries. Still, there is a need for standard frequencies.
- RFID tags are quite large, especially when knowing that the chip itself has a size of only one square millimetre. The attached antenna causes the large size, which cannot easily be reduced.
- Although prices are decreasing, RFID tags are still quite expensive. When real mass production starts, prices will drop. Generally, a price of one or two cents is seen as the final goal.
- Radio waves have some limitations. The environment highly determines how well radio waves can travel in a certain space. For example, radio waves cannot go through water, so reading the tag on a bottle of milk or water is not trivial.

2.2.5 Privacy

Besides all technical and business aspects, there is also an ethical aspect on which opinions differ. Some people think usage of RFID technology will lead to a "big brother" society, others think privacy will not become a major issue, arguing that, nowadays, we can already be traced by our cell phones. The difference here is that RFID tags will probably be readable by every single reader and anyone could purchase a reader.

Without going into much detail, here are some thoughts and ideas to protect people's privacy:

- Store only a unique id on a tag; the data belonging to the tagged product is only available to the authority that provided the tag. Other people can at most read the unique number, but then cannot acquire the accompanying data.
- Encrypt data on tags. Initiatives have started to encrypt data on tags so that only authorised people and authorities can read the data on a tag.
- Destroy a tag after use. RFID tags on for example cloths could be destroyed after a customer has paid for it. Techniques to do that are scrambling the tag's memory or simply sending too much power to the tag.

3 Problem Statement

3.1 Introduction

Companies continuously generate more and more information [9]. This is caused by the definition and implementation of new processes, connections to external parties, etcetera. So more and more data must be managed. Storage of this data, most often in (large) databases, is one thing but, the data from a certain source system has a certain meaning and might need to be correlated with data from other systems to provide even more information. Handling all this data efficiently is quite a challenge already. By introducing Radio Frequency Identification (RFID, see section 2.2) in a company this challenge is getting even bigger since each RFID reader raises an enormous amount of events (at least one for every tag that is read), the number of readers and tags can be very large and, most importantly, RFID adds most value when the events of several readers are correlated.

An example of such a situation is the introduction of RFID technology in public transport. We think that an ideal situation for travellers as well as transport companies could be that all legacy tickets were replaced by a smart card, say of credit-card size, containing an RFID tag that holds enough information to enable the owner to just jump onto and off buses, trains, undergrounds and so on without having to think about buying tickets. The traveller only needs to make sure that he has his RFID smart card on him, for example in his wallet.

This implies that each person that is going on or off will have to be detected and registered so that, based on the traveller's places of departure and destination, billing can take place. A system like this enables companies to get information about, for example, the travel behaviour of their clients but there are also great advantages for clients, like for example not having to buy tickets each time. The sketched situation will be the starting point of the prototype accompanying this research.

Such a system, but also other systems using RFID technology like (large) supply chains, would consist of many readers that read enormous amounts of tags. Each tag will contain a certain amount of data, which can vary from only an identification number to more detailed information about the object a tag belongs to. In case of only a number, a database probably needs to be queried for extended information. If the tag itself contains more information, it might not be necessary to contact other systems or services for more information.

So, a system like described above will generate much data that needs to be handled by one or more processes. More generally, we could speak of a system consisting of many sensors that generate much data. To understand what type of data this is, it is important to know that every time a tag is read, an event is raised that contains the data belonging to the event.

In summary, we can say that the amount of data in companies is growing and that this will get an extra boost when RFID technology is introduced [10]. Therefore, it is important to think about how this data can be handled efficiently.

3.2 Software architecture

This thesis discusses how that flow of data can best be handled from a software architectural point of view. While brainstorming about this subject three aspects of software architectures came up that we find particularly interesting: performance, flexibility and reliability. These aspects were chosen for the following reasons:

- More and more pilots with RFID technology are launched and a lot of companies think about using RFID. But before being able to implement RFID on a large scale, a very important issue needs to be solved: a lot of data will be generated but how should it be handled without losing information or having to wait very long for it? Thus, an architecture for such a system must support a *high performance*.
- No one knows what the future will bring but we do want to design an architecture that can support future types of tags, can be scaled to support a larger number of readers and tags and can be easily maintained. All this without having to completely re-design the system. So, the architecture should be as *flexible* as possible.
- As an RFID system grows it becomes more important to be *reliable*. In the public transport situation described in section 3.1, one can imagine that for example losing all data of a few trains full of people every week has a high financial impact, while missing a tray of soft drink cans once in a while may not be a very big problem.

Furthermore, we want the system to respond real-time, or at least near real-time. For example, when a person tries to get on a train and his RFID tag is read, it is important that within a very short time the system allows or denies access (the latter could for example happen if the person has a large debt by the transport company). It is not acceptable if it takes ten seconds or so for each person since that would heavily extend boarding times and thus cause delays. This real-time aspect influences performance as well as reliability since it puts an even higher demand on the architecture and the system only operates correctly if responses do come in real time.

Summarising we can say that we need a high performance software architecture that gives us enough flexibility but is also very reliable. The degree of flexibility and reliability highly depends on the type of system. In our case of using RFID in public transport, mainly the latter is very important. These three requirements should not strongly influence response times in a negative way.

3.3 Architectural styles

For our public transport system we can roughly choose between two types of software architectures, or better, between two architectural styles (see also 2.1.1), which we define as follows:

- Centralised – This is the traditional style of designing software architectures. The system has a central component that handles all data and therefore contains (almost) all intelligence and is often closely coupled to some sort of data storage.
- Distributed – This is the more radical solution. Instead of centralising all intelligence and data handling, this is distributed over several components of the system, RFID readers for example, that might even be geographically distributed.

Besides those two there are numerous other architectural styles which all have their own characteristics and often suite specific needs. The two styles compared here are more or less two extremes, they are each other's opposites.

3.4 Primary research question

For the design of a software architecture for the system described in 3.1 the two architectural styles, centralised and distributed, will be compared to find out which one provides the best foundation. An in-depth analysis of both styles will be done, focussing on performance,

flexibility and reliability. The point of view will not be the proposed public transport system but "an RFID system where large amounts of data are generated and need to be handled efficiently and reliably".

Beforehand it is not excluded that the final architecture of the prototype (see chapter 5) will consist of the best of both worlds.

3.5 Secondary research question

Besides focussing on the three aspects mentioned in 3.4, another question will be answered as well: What are the financial differences between both architectural styles?

4 Problem Analysis

4.1 Introduction

In this chapter the problem stated in the previous chapter will be analysed. Two software architectural styles that can be applied to design the software architecture of an RFID system will be compared. The first style has all its functionality and intelligence in a central place of the system; the second distributes functionality and intelligence over multiple components of the system, it is decentralised or distributed. These styles are more or less two extremes.

A number of quality attributes for software architectures exist that might be important in various types of systems. Examples are maintainability, flexibility, performance, security, usability, safety and reliability. Depending on which quality attributes are important for a system that must be designed, i.e. the system's quality requirements, it is the software architect's job to design an architecture that meets these requirements as well as possible. In this research, focus will be on performance, reliability and flexibility. Both architectural styles will be analysed on these three aspects.

To be able to assess quality attributes correctly, it is important to know what kind of data is involved. The problem stated in 3.1 is about the software architecture of RFID systems. As described there, the data that circulates in such systems consists of many *events*, each containing a small amount of data, that come from the RFID readers of the system. Additional information may need to be collected from other data sources. The number of tags that can be read, and thus the number of events and the amount of data to be handled, depends on the readers, the environment and the used frequency. The exact amount of data per event mainly depends on the type of tag that is read and the amount of data stored in it. It varies between only a few bytes to a few hundred bytes. For details on RFID tags and equipment, please refer to section 2.2.

4.2 Definitions

In the previous section a number of quality attributes were mentioned. Three of them will be considered in this research, which are defined as follows.

- Performance – Performance refers to the responsiveness of the system – the time required to respond to stimuli (events) or the number of events processed in some interval of time. Performance qualities are often expressed by the number of transactions per unit of time or by the amount of time it takes to complete a transaction with the system. Performance measures are often cited using benchmarks, which are specific transaction sets or workload conditions under which the performance is measured [11].
- Reliability – Reliability is the ability of the system to keep operating correctly over time [11]. Here, correctly means that no data or events are lost, responses come in near real-time and behaviour is always predictable.
- Flexibility – The ability to adapt separate components of a system without having to redesign the whole system. This is especially useful for maintenance purposes and thus for being able to adapt a system in the future when new requirements arise [5]. Scalability is an important part of a system's flexibility too.

4.3 Requirements and characteristics

Now, having defined the three quality attributes generally, what precisely do they mean in this research? This section describes what they mean and how they are influenced by the software architecture of a sensor-based system. The information in this section will be used to judge performance, reliability and flexibility potential of the centralised as well as the distributed architectural style.

Performance

Sensor-based systems often need to handle large amounts of data (it is estimated that a supermarket company like Wallmart would produce a few million terabytes of data each day if all product movements were monitored in the complete supply chain using RFID technology). This data comes in very small portions that are all related to an event. The occurrence of an event needs to be stored in a database just like the data belonging to it. But that is not all. As stated in 4.1, the correlation between events is very important and adds much value to the system. So, besides just registering events, also some processing needs to be done which requires previously stored data. And last but not least, often a reply to an event is expected. If for example an employee scans a product in a supermarket to retrieve its expiry date, he expects to get the answer within about a second or less, he should not have to wait minutes.

Reliability

The degree of reliability of a sensor-based system highly depends on how valuable an event and its data are to a company. The more valuable each event is, the more important it is that the system has a high availability. If the system is not very reliable, it is very likely that a lot of events are not registered every now and then, which causes loss of data which can, in turn, cause for example loss of income or revenue.

The reliability of a system can be influenced by a failure of one or more components of the system, failure of a data store or failure of communication between components. Problems could for example be caused by the system having such a large load that operation becomes unpredictable.

Flexibility

To be able to maintain the system well and to allow future extensions to the system, both without having to re-design the whole application, a high flexibility is important. There are three reasons for changing software: updates, bug fixes and adaptations [12]. So every now and then, one or more components need to be changed for one of these reasons. This should have as little influence on the entire system as possible.

4.4 Centralised software architectures

4.4.1 Description

There is not an architectural style called "centralised" but there is a number of styles that have a centralised character, like for example 'layered' or 'component based' [5]. In this case a centralised architectural style is meant as an architectural style where all intelligence and all data is on a central system.

The term "centralised" may be associated with monolithic systems, but that is not the kind of

software architecture that is considered here. What is meant is a modern software architecture, like for example a service-oriented architecture (2.1.5), that is suitable for handling events (2.1.6). Figure 1 presents a schematic overview of a centralised software architecture.

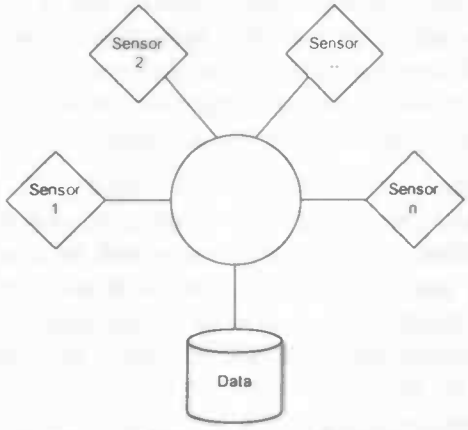


Figure 1 - Schematic representation of a centralised software architecture

The figure clearly shows that all the sensors of the system are directly connected to the central software system. Also a data storage component is attached to it. It is important to notice that the number of sensors may grow very large, depending on the system's purpose.

A centralised architecture is characterised by the fact that all components of the system reside relatively close to each other. Together, they can therefore best be seen as one big application. That application is the heart of the system and it has to handle all events and the accompanying data that are brought in by the sensors. Since there is only one data store and one application, it is very clear where all data resides.

4.4.2 Performance

In the centralised situation there is only one application to do all the work, so the work load cannot be divided over multiple applications. But there are some other ways to improve performance. In 4.4.1 it is pointed out that a modern architecture is assumed, where tasks are assigned to specific components of the application. By smartly choosing a communication method between those components, it is possible to tear the application apart and run components on different machines. Inter-component communication by implicit invocation (see 2.1.4) does enable this. The drawback is that using events requires event handling mechanisms that need a little computation time that is unrelated to the actual functionality [5]. This negatively affects performance. Still it is a logical choice because of the data coming from sensors is based on events.

By spreading components, the application becomes quite well scalable. Performance optimising techniques, like load balancing, then become possible that can easily improve performance in handling the large amounts of data. But still, this one application is the only one to receive all events and data and thus needs to do all the work.

The work that needs to be done may consist of a few parts: (1) receive events / event data, (2) store relevant data in the database, (3) correlate (a part of) the events to other already stored events and (4) send back a reply to the event source.

The first two operations will most likely occur in every system. The other two are in fact optional, but especially (3) adds much value to the system. Listening for and receiving events has quite some overhead, but since that is the same for the centralised and the distributed style, it will not be taken into account here. What is important, is that the system needs to receive (1) *any* event that is thrown by any sensor. It may, however, not be necessary to store all incoming data (2). So filtering can be part of (1). In fact, since the amount of data can be enormous, it might simply be impossible to store all incoming data so filtering is necessary. It is not very efficient though, if merely irrelevant data still must be processed by the application. But since there is only one application, it must be done there, which badly influences performance.

There are two more performance related issues that need to be addressed. First, (2) and (3) intensively use the database to store and retrieve data, at least once per event. Although the amount of data per transaction may be small, there will be many transactions. To optimise database communication, a communication method with as little overhead per transaction as possible will be best to use. Further, the number of transactions can be reduced by thinking well about which information really is relevant and by keeping the number of correlations as small as possible. Always having all data in the right place (there is only one data store) makes event correlation easier and is a positive thing for the performance of a centralised system.

When using dedicated event correlation software, the number of transactions may be reduced. Such software could keep all necessary events in memory so that they are directly available when they are needed. When finished, those events can simply be removed again and only the (most) valuable information can then be stored in a database for future use.

Second, there is the real-time aspect. Incoming events may require a response. That response needs to be received with the smallest possible response time. Here, an important issue of the centralised architecture appears. Every single event needs to go all the way to the central system, must be processed there and a response must be sent back or a signal must be sent to another system. It is very hard to design the system in a way that, no matter what, events are handled nearly real-time, even in peak situations. This has to do with the limitations of scaling. The more events are raised, the harder it becomes to respond in real-time. To keep up as well as possible, the system can be scaled, load balanced and so on, but only up to the limitations of hardware and infrastructure. For systems with a relatively small number of sensors, this will not be an issue. And, of course, it highly depends on the type of system how 'real-time' should be defined.

Overall we can say that by using loosely coupled components, a centralised system can be quite scalable which improves performance. Scalability is mainly restricted by hardware and infrastructure boundaries. Performance is negatively influenced by the overhead of events and by the fact that every single event needs to be handled, at least for filtering. The (value adding) event correlation lays a heavy load on the application as well as the database. It is important to think about which events are interesting and which are not, to keep this manageable. Further, responding in real-time becomes harder when the system grows. Finally, not having to replicate data to other data stores is an important pro.

4.4.3 Reliability

In a completely centralised system, all events from sensors are sent directly to the central application which is responsible for processing the event. That means that the reliability of the system depends on the reliability of that application and the data store[1].

1 One could argue that also the infrastructure has influence on reliability, but that is not different from a distributed environment and is therefore not taken into account here.

Let us first have a look at the reliability of the application. Assuming a component-based architecture, there is one component that receives all incoming events, at least for one type of message. It is more or less the front door of the application. That means that the system is as reliable as this front door. If this fails, no more events are received by the application and thus nothing can be processed and or stored any more. Other than that one component, failure of other components of the system does not influence reliability so seriously. If events are received by the front component, they are inside the application. There they can be stored until failing components become available again after which they can still be delivered and processed. This, however, is also possible for the front door by implementing guaranteed messaging.

This, of course, does influence performance and response times. When saying that too large response times mean that the application does not run correctly, reliability is harmed, but at least no data is immediately lost. If it takes too long before failing components become available again, the application may not be able to store all incoming events any more, which may cause unpredictable behaviour. Even so, if a component starts running again, a usage peak might occur that may be hard to handle.

Second, there is the data store that influences the correct operation of the application. If the data store becomes unreachable, storage of incoming events and data is not possible. The same holds for event correlation. This means that the application cannot run correctly and that data might be lost. Potential failure of the data source may thus be considered as large a risk for the system's reliability as failure of the front component.

4.4.4 Flexibility

In 4.4.3 we saw that if a component of the application fails, the whole system may not work correctly any more. So if maintenance of (part of) the application is necessary, there is the serious problem that taking down one component, will eventually stop the system from working. As pointed out in 4.4.2 it is possible to duplicate components to be able to share the load. This technique can also be used when maintenance is needed. By maintaining only one instance of each component at a time, down time is reduced. The other instance(s) just have to take some extra work. Moreover, certified communication [13], although it takes some overhead, can be used as protection against the loss of data. So, a component may be taken down, upgraded, and brought up again, and then receive the events that were sent to it in the mean time.

So, a centralised architecture can be quite flexible, but it does take some extra measures to reach it. Further, if a change is needed in some component which has only to do with a small part of the sensors, the change still affects the complete system. So even a minor change brings quite some risks to the whole system.

4.5 Distributed software architectures

4.5.1 Description

Opposite to the centralised style, in a distributed architecture functionality and data are spread over multiple smaller systems. Those small systems all process only a part of the events of the whole system. At that point, data can be filtered and stored and maybe further intelligence can

be added. It is important to think about where all data will be stored. If each subsystem has its own database, how can another system use that data? Figure 2 shows a schematic representation of the distributed architectural style. All circles represent a part of the system, so there is one central system with a number of smaller subsystems that are distributed over a network. Such a smaller system may serve as a central point for other subsystems as well. It is important to notice that there is not one data store any more, but that subsystems also have a data store. This is more or less a design decision, but to enable the storage of data at a subsystem, a data store must be available so it is very likely that the system needs more than one data store.

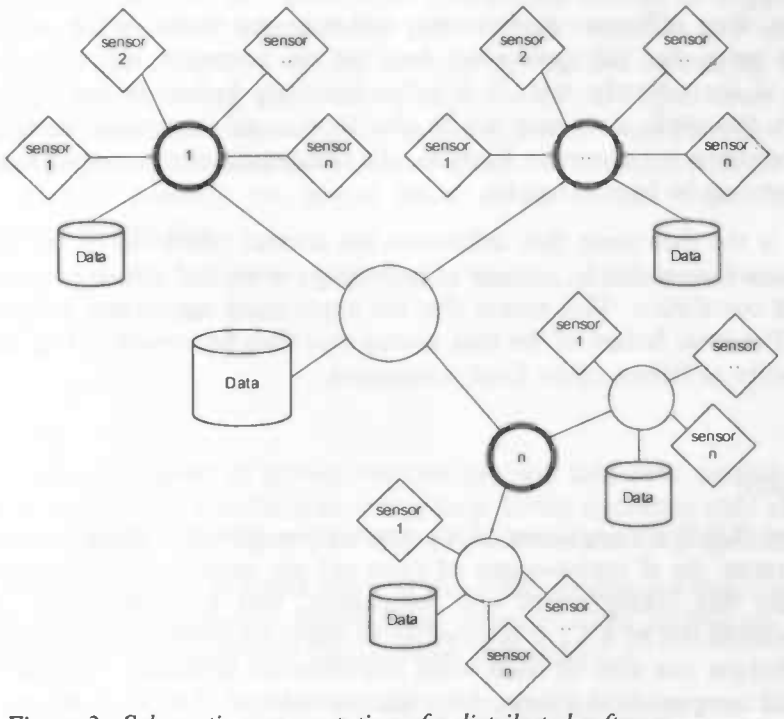


Figure 2 - Schematic representation of a distributed software architecture

What is called a distributed architecture in this research, is sometimes also called a network-based architecture [14]. A distributed system is then seen as one that looks centralised to the user and is thus transparent. A network-based architecture is not necessarily transparent to the user. In this research we call a non-centralised architecture distributed, but in fact, it is the same as a network-based architecture. Note that, in such an environment, message passing-like communication is the only realistic option for inter-process communication. For communication between data stores (data replication etc.), there are several other options, like for example (proprietary) techniques of database vendors [15] or Daffodil Replicator [16]. However, message passing may be a logical choice since it is also used for inter-process communication.

4.5.2 Performance

In a distributed architecture functionality can be spread over multiple systems that are on different places in a network. Where in a centralised situation the centralised system has to

process every single event, there are a number of possibilities here to prevent that. First of all, a subsystem that is connected to only a part of the sensors can do some processing on events coming from those sensors. The simplest operation is filtering. By filtering useless[2] events the load on the central part of the system already decreases.

Besides filtering, also more advanced functionality can be done in the subsystem. Referring to the work breakdown we made in 4.4.2, a number of improvements can be achieved on the four types of work. The first activity was receiving events. As mentioned in the above paragraph, every subsystem only receives events from the sensors that are connected to it. By adding a data source to each (group of) subsystem(s), those events and their accompanying data can also be saved. This way subsystems need not be very large and the load on the central part of the system is decreased.

Another part of the work was correlating events. And that is where problems arise in comparison to a centralised system. If every subsystem stores incoming data in its own data store only, how can another subsystem correlate its data with it? There are a few options to address this issue.

- The first is using only one central database for the whole system, but that will not perform very well since a lot of communication will be necessary to gather or store information.
- A second option is to also store data that might be needed for correlation purposes in the central database and use that database for correlation.
- The third option is to replicate data such that all databases have the same data.
- The fourth option is to store data in the distributed data stores and store where the data resides centrally. This way, because the event data itself does not need to be sent over the network, only a minimal amount of data has to be sent over the network and data is only stored in one place.
- A completely different option is to reduce the number of transactions on the databases by using dedicated event correlation software that can keep events in memory instead of storing it to a database. In the centralised architecture, this can directly improve performance. In this situation, however, events may still be needed in distributed subsystems. So the events must still be available to other parts of the system, which can even decrease performance in comparison to the above mentioned options.

The fourth option mentioned in the list above would work the same as a Domain Name Service (DNS). In fact, EPC Global has defined a standard Object Naming Service (ONS) for this purpose that is based on DNS [17]. The ONS option requires the least network traffic and prevents having to access a single database for all data. The other options all require quite some network traffic. Which solution is best mainly depends on the type of application and the amounts of data that circulate in it.

The final part of the work is sending responses. If all necessary data is available in the data store of a subsystem, real-time responses are very well possible. But if data needs to be gathered from other data stores, response time can easily grow higher than in a centralised system.

Besides functional differences in performance, there is also an important difference in scalability. In a distributed architecture processing load is already spread over multiple systems, but those systems can of course all be load balanced again if that would be necessary. A distributed system is thus much better scalable.

2 The definition of useless will depend on the requirements of the system

Overall we can say that, in comparison to a centralised architecture, a distributed architecture is very scalable and processing load can be very well spread because fewer events traverse down to the central system. That all increases performance. On the other hand, it is very likely that a lot of traffic is generated between all the components to acquire all necessary data, which has negative impact on performance.

4.5.3 Reliability

In a distributed system there are a lot of subsystems and connections between them. So, there are more parts that could fail than in a centralised system. But instead of stopping the whole system from operating correctly, only a small part will not function any more. When a subsystem is broken, events sent to it will be lost. Again, certified messaging could prevent that. In case of a database becoming unreachable, impact is a little higher since other systems may rely on it. It could mean that response times increase or that event correlation is not fully possible any more. The impact of a database failure depends on how databases are positioned in the architecture but it is possibly the largest threat to the system. Techniques like data replication can be used to ease this.

So, although the chance of a failure is much higher than in a centralised architecture, the average impact of a failure is much lower.

4.5.4 Flexibility

For flexibility it is also very good to spread functionality and data over several systems. If a change is needed, it can be done one subsystem at a time. Every operation only affects a rather small part of the system so that the biggest part of the system just can keep operating correctly.

An even higher flexibility can be reached when al those subsystems are duplicated. Then the system and its subsystems can always be kept running correctly when performing maintenance. Also certified messaging can be used to at least prevent loss of data.

Besides these pros, there is also a disadvantage. Since many subsystems have the same functionality, there are many copies of each component of the software. Every time a change is needed, many systems must be updated in stead of only one (or maybe a few), so this requires a lot more work. This only gets worse when duplicating systems. However, for some kinds of maintenance, like a small patch for example, techniques like the well known auto-updates of virus scanners may be applicable so that the amount of work can be reduced.

Concluding we can say that a distributed architecture is more flexible, mainly because changes only affect small parts of the system.

4.6 Cost aspect

Most often not only suitability determines which type of architecture will be chosen to implement a certain system. The final decision is often a business decision where the cost aspect is very important. Therefore, a short insight in costs of both architectural styles will be given here.

Centralised

Theoretically a centralised system could run on a single machine connected to a single database

machine. To connect sensors, network connections and switches could be used. If systems grow or have to be duplicated for reliability reasons, more machines are needed. Still, not many machines are needed. That fact has a positive influence on the costs of a centralised system: hardware and infrastructure costs are relatively low. The same holds for maintenance costs.

Distributed

A distributed solution is much more expensive. More machines are involved, which automatically requires a larger and better infrastructure, more hardware and thus larger maintenance costs (maybe, the machines used in a distributed system can be a little less high-end and thus a little cheaper, but since machine management and maintenance are more expensive, that cost advantage is ignored here). Also, extra database systems are quite expensive. Sensors could for example be directly connected to the distributed machines or, again, using network connections.

4.7 Conclusions

Having analysed both architectural styles and their strengths and weaknesses, which one can one choose best, on a theoretical basis, to design a software architecture for a sensor based system?

First, from a performance perspective we can say that a centralised architecture can be used quite well for sensor based systems and that its performance is mainly restricted by the scalability of hardware and infrastructure. The big drawback is that every single event, relevant or not, must be handled by the system. Responding real-time gets harder when the system gets larger. Finally, there will be a heavy load on the central database but knowing where all data resides is a pro. In comparison to a centralised architecture, a distributed architecture is much better scalable and processing load can be very well spread because fewer events traverse down to the central part of the system. That all increases performance. On the other hand, it is very likely that a lot of traffic is generated between all the components to acquire all necessary data. An Object Naming Service is a good solution for this. Real-time responses can be achieved easier if the necessary data is locally available.

Second, a centralised system is as reliable as the central application and data store where the application's 'front door' is the weakest spot. Events in the front door application are in the system and thus not lost. Between components, certified messaging can prevent data loss. Since sensor based systems heavily rely on their data stores, data store failures are a large risk for reliability. In distributed systems communication between components is more sensitive for failures. This increases the chance of failures, but the average impact is lower. Again, data store failure is the most serious threat. A pro for distributed architectures is that because of better performance and probably better response times, the system can keep working correctly under a higher load.

Third, distributed architectures are more flexible than central architectures. The latter can be made quite flexible with some special measures. But still, if a change is needed that is only relevant for a small part of the readers, that change still affects the complete system. So minor changes bring quite some risks to the whole system. In a distributed architecture, small subsystems can be changed one at a time so the rest of the system keeps operating correctly. By applying the same measures that can be used for centralised architectures, flexibility can even become better. The only drawback is that there are a lot more systems to maintain.

Concluding, we can say that a distributed architecture will be the best choice if systems (have the potential to) become very large. For smaller systems a centralised architecture will suit fine.

The big drawback of a distributed system is that costs are higher, as well for hardware and infrastructure as for maintenance. Because of the large amounts of data and the high number of queries on databases, it is very important to implement the databases as efficiently and fail-safe as possible. The availability and performance of the databases highly determine the performance and reliability of the chosen architecture.

5 Case

To be able to validate the research results of the previous chapter, a prototype has been built. The main goal of the prototype is to get some experience with RFID technology, to integrate RFID functionality into an Enterprise Application Integration (EAI) environment (see 5.4) and to have an implementation to validate the research results.

5.1 Introduction

The built system is a prototype of what public transport could look like when Radio Frequency Identification is incorporated in it. The goal is to no longer use all different kinds of public transport tickets, but just give everyone a smart card, say of credit card dimensions, containing an RFID tag. People only need to take this smart card, for example in a person's wallet, and can just hop on or off any type of public transport any time they want. By identifying a person when he or she hops on and again when he or she hops off, in combination with the start and end stations, pricing and billing can take place automatically. The main advantage for travellers is not having to think about buying the correct tickets when wanting to use public transport and having one card for all transport types.

The use of RFID can also enable additional services. For example, if, when a person orders an RFID smart card, the person's mobile phone number is registered, the public transport company could send SMS messages to all persons on a particular train or bus regarding current delays, arrival times, transfer options etc. Another option is to monitor a person's travel behaviour and recommend to him what kind of subscription would be most profitable. It could also be easy to get a lot of statistical information about occupancy of trains, most or least travelled routes, etc. Even discounts for delayed trains can be automatically applied. Because it is known where a person is, public transport companies have the possibility of personal advertising for nearby shops. So, advantages for companies are personal services, advertisements and accurate statistical information and billing procedures.

5.2 Functional requirements analysis

This section provides a functional specification of the prototype, which will function as the basis of the technical specification.

Since the system is a prototype and thus will not function in a real world situation as is, the context is self-defined and not completely real. Nevertheless the prototype will be as realistic as possible. This will be accomplished by taking the current Dutch transport system as a basis and integrating RFID technology into it.

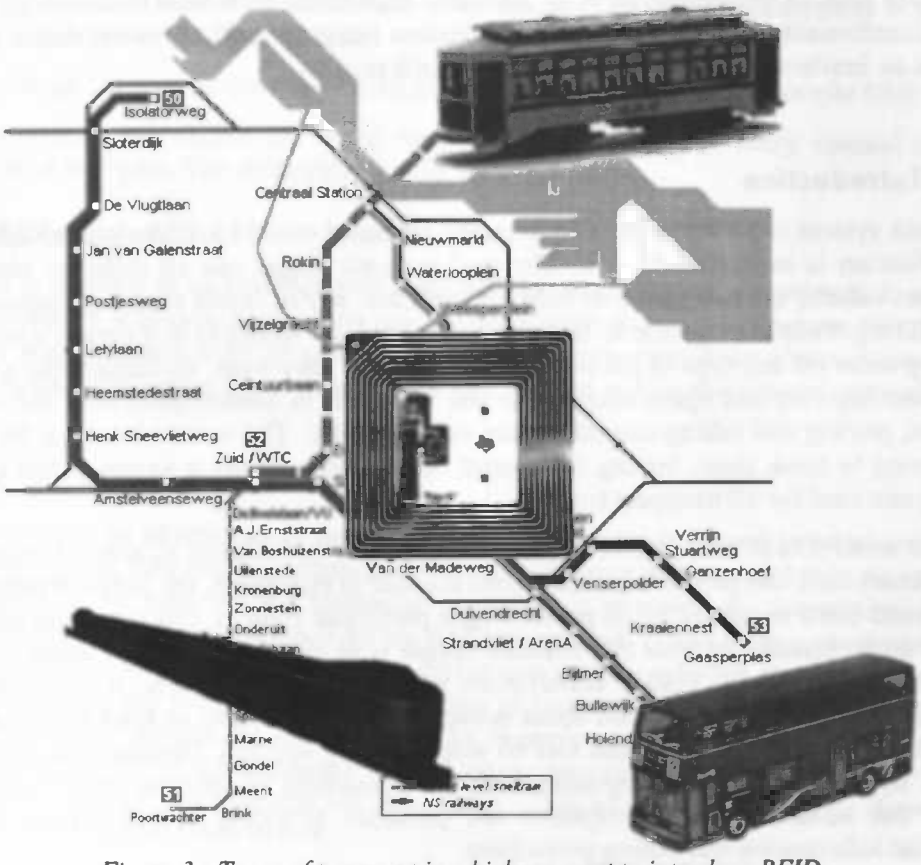


Figure 3 - Types of transport in which we want to introduce RFID

Figure 3 shows a part of a public transport map, some transport vehicles and in the centre the piece of technology that we want to give a central function in tomorrows public transport systems. Figure 4 is an example of a trip that a person, carrying an RFID smart card, can make without having to do anything but just hop on or off vehicles. The trip consists of four parts where each part is carried out by a different type of transport. At all mentioned stations, except Arena, a new trip is started. At the endpoint of each trip, pricing is done based on the trip length in kilometres and the type of transport. Each part of the trip is also billed separately.

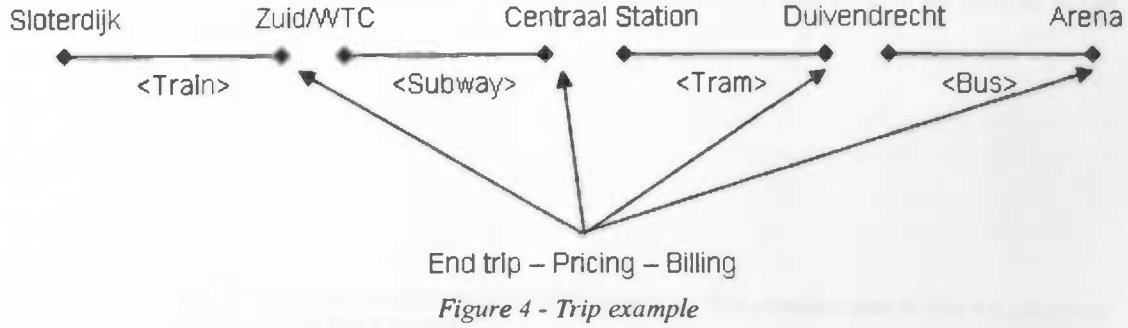


Figure 4 - Trip example

5.2.1 Detailed functionality analysis

There are a lot of public transport types. For this project they are grouped according to fixation to a network and the use of isolated stations. Public transport could then be grouped as shown in table 1.

Station-based (A)	Non-station-based (B)
Train	Bus
Subway	Tram
Ferry	

Table 1 - Public transport types

We need to distinguish these two groups since in the station-based group there are isolated stations that enable strict access control and communication networks to a central point are available. In the non-station-based group, these things are not available or are not usable for this system. The two groups result in two sub-systems.

Common functional requirements

The research that forms the basis of this prototype focuses on three quality aspects of software architectures. For this prototype those aspects are important as well: performance, reliability and flexibility. There are no quantitative values that need to be met. One of the goals is to measure these quality aspects of two different architectures for this system and compare them to each other.

Globally it can be said that the following functionality must be provided by the prototype system:

- Station access control – based on some criteria like a credit check, a person may or may not enter the public transport system. This check must be performed very fast, since passengers should not have to wait for seconds before they can enter. Further, every person that does really enter must be registered together with the entry location.
- Vehicle entry – every person that enters public transport must be registered together with the entry location.
- Vehicle exit – every person that leaves public transport must be registered together with the exit location.
- Validation & pricing – based on entry and exit locations the length of a trip can be calculated. A reference database is needed to determine the price to be paid for a trip. The price per kilometre also depends on the type of transport used. Before pricing, a trip must be validated. For example, if there is a trip that takes far too much time or if there is a round trip to and from a place that lasts too short to be a real trip, that trip might not be valid.
- Billing – each time a trip has been finished and pricing has been done, the trip must be billed. This results in a successful payment or an unpaid bill.
- Station exit – By registering persons leaving a station, it is known whether or not a certain person is still at the station.

- An additional service (SMS service for example) is added to the design but not implemented[3] – because of the access and exit controls it is possible to continuously know who is in which transport vehicle or on which station. This enables context-sensitive information services. A simulation of such a service must be built. A possibility is sending an SMS to each person in a certain train.
- To get an insight in data flows, monitoring must be done that reports to a simple GUI.

The two subsystems are almost the same; they only differ in station entry control and the availability of an exit gate. The differences are as follows:

Subsystem A (station-based)

Entry control is coupled to an entrance port that only opens when access is granted. It must be explicitly checked if a person really does enter when access is granted. Because this type of subsystem is station based, also an exit gate can be added that registers a person leaving a station and, optionally, shows some information like the last trip or the remaining credit on a screen.

Subsystem B (non-station-based)

Entry control cannot be as strict as in sub-system A. Since bus stops are not isolated enough to do access controls on, they must be done in the bus itself. Since there is probably no space to put access gates in the vehicles, the following procedure is proposed for access control: On entry, the traveller must identify himself by his RFID smart card. His id is then checked against a locally available blacklist in the bus. The bus driver gets a signal whether or not access should be granted. Since it is not easy to use gates in a bus or tram, other measures must be taken to make sure that persons getting off are scanned. For example, sanctions could be used to make travellers responsible for being registered well. Another possibility is to use historical data to determine where a person has probably left the vehicle. This, of course, can only be done when the person travelled on that route before.

System availability

As said before, the system that is built is a prototype, but it will be as realistic as possible. Focus of this project will be on performance, reliability and flexibility. Since availability influences the reliability of the system, also system availability must be very high. If this system is put into the real world, availability must be about 99,99..% of the operational time, where at night there might be a few non-operational hours that could be used for maintenance etc.

User interface

Since only a prototype is built, no extended GUI will be developed. The user interface will contain all that is needed to be able to test the system appropriately and to give an insight in the flow of actions in the system.

3 Implementing SMS services as proposed is beyond the scope of this project, but from a business point of view it is valuable to know about the possibilities of adding such services.

Use cases

Figure 5 shows a schematic view of the most important use cases that must be handled by the prototype system.

1. Station entrance -> access granted
2. Station entrance -> access denied
3. Vehicle entry
4. Vehicle exit
5. Pricing & Billing

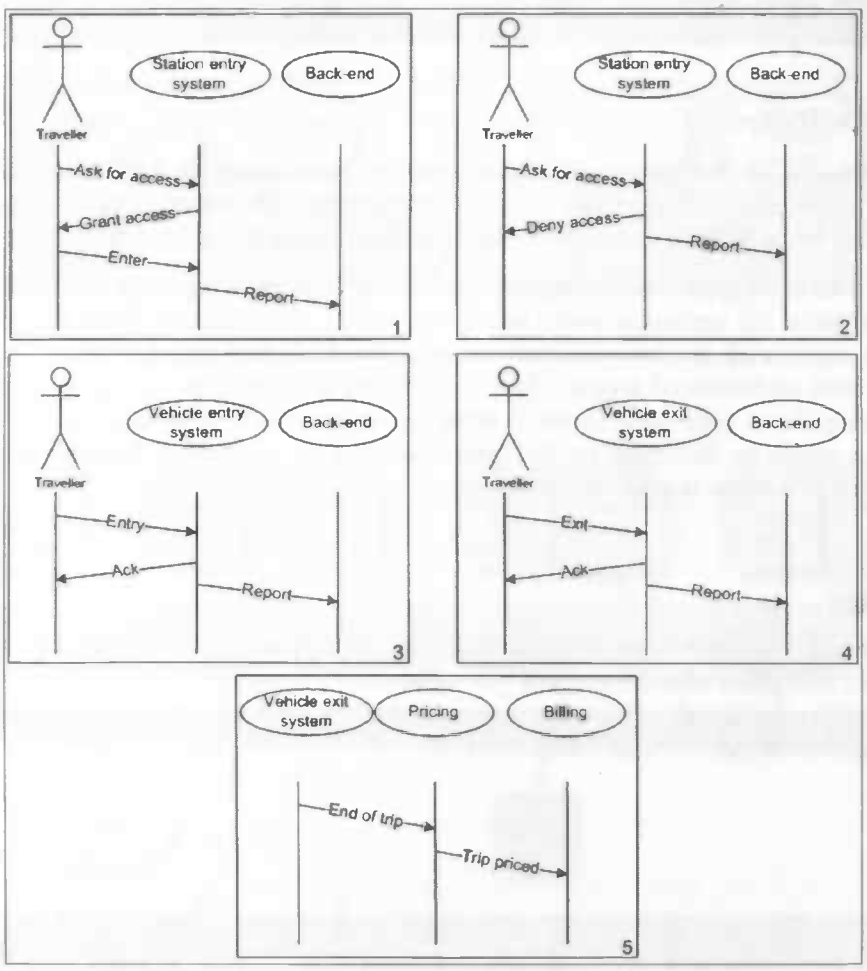


Figure 5 - The 5 most important use cases

This set of use cases is not exhaustive but the most important ones are shown here. Cases 1 and 2 hold for station-based systems only, 3,4 and 5 hold for both station-based and non-station-based systems. Functionally, a typical trip on system A will consist of cases 1-3-4-5 and on system B the sequence will be 3-4-5. As can be seen, the trip itself is the same for both systems, for type A only station access is added. Also the not shown use case "station exit" is added. In case of an "access denied" situation, only use case 2 occurs.

5.3 RFID

In the prototype three RFID readers were used together with a number of tags. The devices operate on 13.56 MHz and have a very short range of about 25 mm. which is very useful for testing purposes since there is no interference. The tags each have a unique 16 character id and 64 bytes of free, writeable space. To be able to use the RFID equipment in the prototype implementation, an interface to the devices was implemented. For this prototype, it is only important to know that the interface receives data from the devices through a serial port and publishes the same data as a TIBCO Rendezvous message (see 5.6.2 and [13]). Please refer to the interface's documentation for more detailed information.

5.4 EAI Software

One of the goals of this prototype implementation is to integrate RFID technology into an Enterprise Application Integration (EAI) environment. Therefore, implementation of the prototype will be in TIBCO BusinessWorks [18]. See section 5.6.1 for details.

BusinessWorks is an application integration suite that enables integration of numerous systems. It is very suitable for service-oriented and event-driven architectures. Since the prototype will consist of a number of services, it is not necessary to build two entire systems from scratch to implement both architectural styles. The two different architectures can easily be implemented by connecting these processes to each other according to both architectures. The different services can easily be deployed on different machines to simulate a distributed environment. This way, this prototype is also very scalable.

5.5 Design

The design of the prototype as described in this chapter must support all functional requirements and must also be suitable for validating the results of the research presented in this report. This section describes the design of both a centralised and a distributed solution, which are shown in figure 6 and 7 respectively.

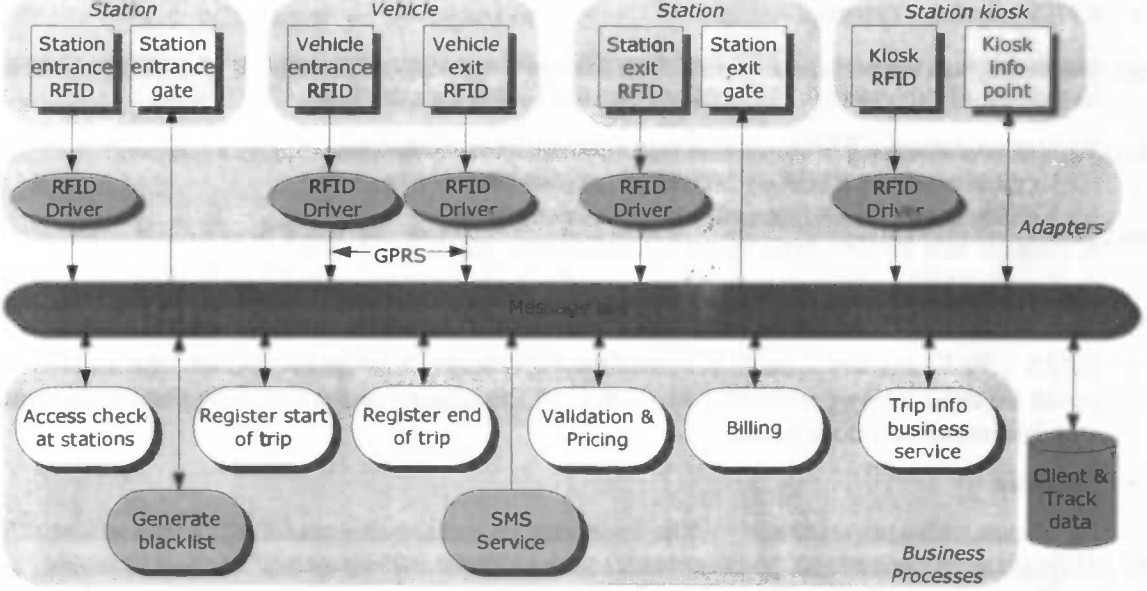


Figure 6 - Centralised architecture for the described prototype

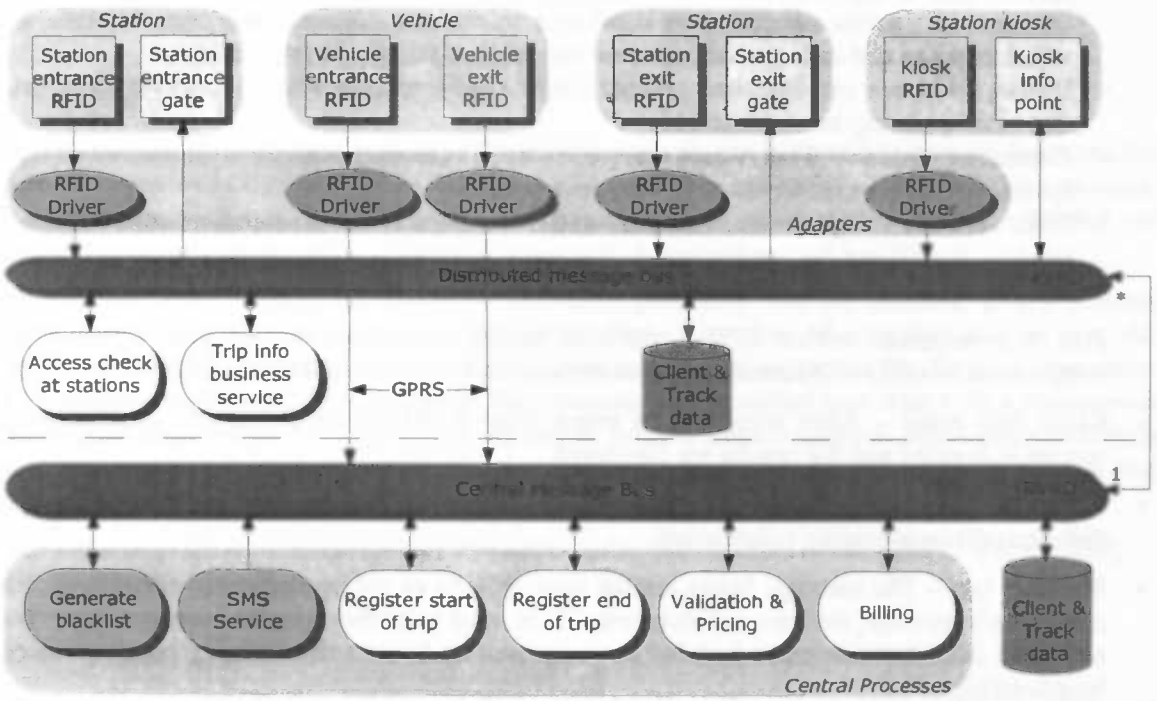


Figure 7 - Distributed architecture for the described prototype

Since both architectures consist of the same components, they are described once. Further on, the differences between the two architectures are described.

- RFID devices & drivers

  In the design there are 5 RFID readers which all have their own driver or adapter to connect to the rest of the system. The readers have the following purposes:

  1. Station entrance RFID – Scan tag to perform an access check
  2. Vehicle entrance RFID – Scan tag to register start of trip
  3. Vehicle exit RFID – Scan tag to register end of trip
  4. Station exit RFID – Scan tag to register person leaving a station
  5. Kiosk RFID – Scan tag at a kiosk to fetch information about recent trips, remaining credit etcetera.

- GPRS – RFID readers 2 and 3 are installed on vehicles, so communication with the system could be done using protocols like GPRS. In the prototype they will be connected using standard ether net connections.

- Services

  - Access check at stations – When receiving an event from reader 1, an access check is performed and the result is sent back to the exit gate which opens if access is granted.
  - Register start of trip – Store the start of a new trip for a certain person in the database.
  - Register end of trip – Store the end of a trip for a certain person in the database and publish a message that activates pricing of this trip.
  - Validation & Pricing – After a trip has been ended, a message is published to inform this service that a trip was finished. This service validates the trip according to specific criteria and queries one or more databases to retrieve distance and price information which is then used to calculate and store the price of the trip in the database.
  - Billing – Once a trip has been priced, a message is published to this service indicating that billing the trip can take place.
  - Generate Blacklist – This helper service is used to fill the blacklist of the access control systems at stations (and possibly vehicles as well).
  - SMS Service – This service is drawn in the architectures as an example of an additional service. It will not be implemented. For more possible additional services, see section 5.5.1.

- Trip info business service – The trip info business services can be used by multiple components to retrieve information about certain trips. One example is the kiosk info point.

- Kiosk info point – After receiving an event from RFID reader 5, the trip info business service is queried and the results are displayed.

- Client & Track data – Database containing information on clients, open and finished trips, a distance table and pricing information.

- Message bus – The message buses are the central parts of the systems and subsystems that connect all services, devices and databases. It is used to deliver rendezvous messages (see section 5.6.2) that are published by services and readers to the correct locations using broadcasting.

The difference between the two architectures is that in the central architecture there is only one message bus and in the distributed one, every subsystem has its own distributed message bus. To the distributed message bus, the access check and trip info services are connected. They are not connected to the central message bus any more, indicating that the trip info service and the access check are performed completely distributed. The same holds for the station exit events.

5.5.1 Possible additional personal services

Besides the example SMS service, many other additional services could be integrated into the described system:

- After finishing a trip, send an SMS message with information (price, etc.) regarding the just finished trip.
- When a person is at a station, provide the option to receive advertisements of nearby shops on his/her mobile phone.
- Enable travellers to plan a trip on the internet. Then send information to his/her mobile phone about up to date departure times, delays and transfers.
- Collect OV miles for every trip made.

5.6 Implementation

As pointed out in the previous section, the prototype is implemented using TIBCO BusinessWorks. First, a very brief introduction into BusinessWorks will be given. Then the implemented processes will be listed and the implementation of two of them will be described in detail as an example. Since it is beyond the scope of this research, not all processes are described here. Please refer to the software documentation for more detailed information on the implementation.

5.6.1 Implementing using BusinessWorks

In BusinessWorks (BW) everything is about processes. Every process has its own functionality and may also call other processes. By tightly separating functionality into different processes, they can be easily deployed on other systems. Also a number of settings can be adjusted per process at deploy time.

BW comes with a graphical designer in which processes can be defined. Every process typically has a starter and an ender between which a number of activities can be put. All activities are connected through conditional connectors that represent the flow of operations that the process must perform. Many of the processes implemented here start with a rendezvous subscriber that listens for rendezvous messages with a certain subject. Most processes then confirm having received a message, do some database queries and call a separate process that sends information to a GUI.

For more detailed information about BusinessWorks, please refer to the TIBCO website [19].

5.6.2 Rendezvous

Rendezvous is TIBCO's messaging software. In this project, it is used for all message based communication. The message buses specified in the architectures are responsible for delivering the right messages to the right listeners.

A rendezvous message is a structured message containing a subject, a time stamp and a number of custom fields. Messages can vary in size between, say, 10 bytes to megabytes. The messages used in this prototype are about 100 bytes in size.

5.6.3 *Processes*

The following processes were implemented to fulfil the functional requirements. It is easy to see that they represent the components as described in section 5.5.

- Access check at stations
- Register start of trip
- Register end of trip
- Station exit
- Validation & Pricing
- Billing
- Kiosk info point
- Trip info service
- Generate blacklist

The "Register start of trip" and "Register end of trip" processes will be described here.

Register start of trip

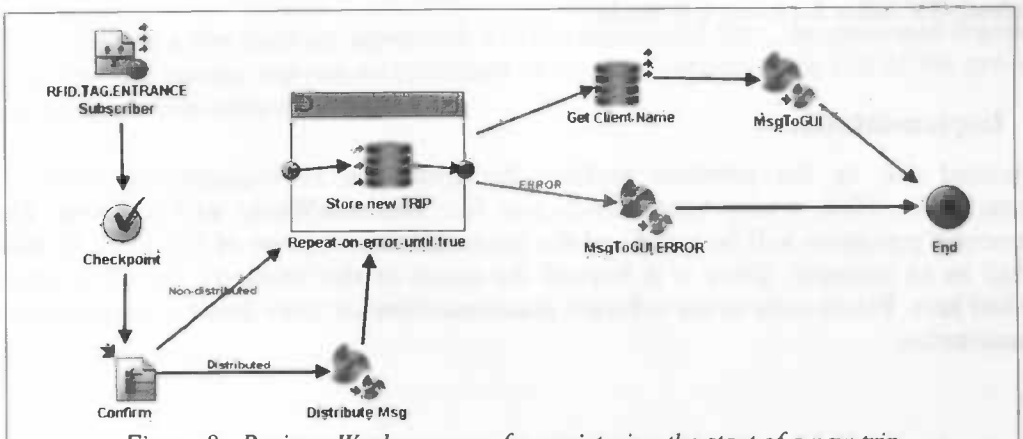


Figure 8 - BusinessWorks process for registering the start of a new trip

Figure 8 shows the implementation of the process for registering the start of a new trip. What follows is an overview of what each activity does in the order indicated by the arrows.

- RFID.TAG.ENTRANCE Subscriber – This is a rendezvous subscriber that listens for messages with subject "RFID.TAG.ENTRANCE". Such a message contains the fields "TimeStamp", "UID", "Data", "DeviceID" which respectively hold the time the tag was read, the unique id that is stored in the tag, the other data that was saved to the tag and the id of the RFID device that read the tag. The latter makes it possible to determine what the starting point of a person's journey is.
- Checkpoint – At this point, the process is saved to disk to make sure that, whatever happens, the received data cannot be lost any more. The process can be restarted from here.
- Confirm – Confirm to the sender that the rendezvous message has been received. This is necessary for certified messaging.
- Distribute Msg – This is a call to another process that can replicate data to other systems. It is added to be able to use the same processes for both scenarios. It is used in the distributed scenario only. Therefore, that process is only called when the "Distributed" condition holds.
- Store new trip – The start of a new trip is stored to the database here. The group around it is to catch any database errors: the group repeats the database activity at most 5 times when it fails and then suspends the process.
- Get client name – Database query to retrieve the name that belongs to the UID.
- MsgToGUI and MsgToGUI ERROR – These are again calls to another process. In these cases the called process sends information to the GUI.

Register end of trip

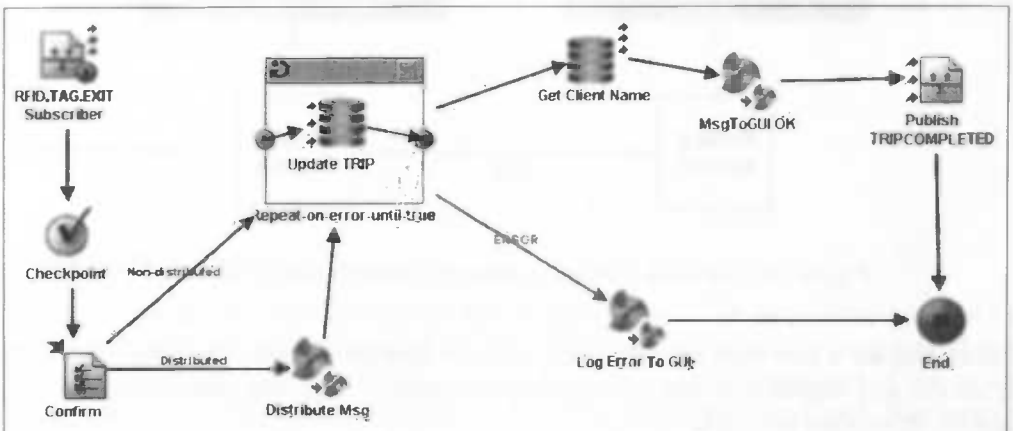


Figure 9 - BusinessWorks process for registering the end of a trip

Since this process is almost the same as the previous one, only two activities are described here.

- RFID.TAG.EXIT Subscriber – This is again a rendezvous subscriber, this time listening for the subject "RFID.TAG.EXIT". The fields of the message are still the same.
- Publish TRIPCOMPLETED – Once the ending of a trip has been stored to the database and has been reported to the GUI, a rendezvous message is sent. That message has the subject "RFID.TRIPCOMPLETED" and contains the id of the trip that was just completed. The Validation & Pricing process listens for such messages and processes them. This way pricing is done after the end of a trip.

5.7 Deployment

5.7.1 Centralised

By running all processes on the same machine or a few machines in the same network, the implemented system is centralised. Spreading processes over different machines can help to balance the load. Communication between all processes is done using messages, so it does not matter that processes run on other machines.

5.7.2 Distributed

By spreading the processes over multiple machines that are in different networks, a distributed set-up can be built. Rendezvous routing daemons are needed to send messages from one network to another. Those routing daemons can filter on message subjects, so that only messages that really must go to processes in another network will be routed. This reduces the number of messages per unit of time that is sent to each part of the system. Figure 10 shows how separate networks can be connected using routing daemons.

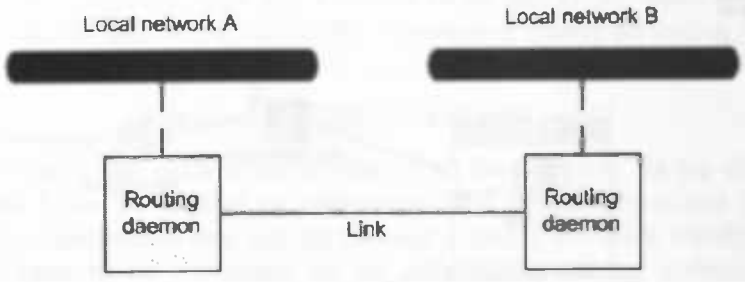


Figure 10 - Separate networks connected through routing daemons

In the next chapter a few tests are described. For the distributed tests, the processes registering station access and station exit run on machines in separate networks, the rest of the processes run together in another network.

6 Validation

In this chapter the conclusions stated in section 4.7 will be validated using the case described in the previous chapter.

6.1 Performance

6.1.1 Performance analysis

One of the most important conclusions was that if systems become very large (i.e. many events occur) a distributed architecture could be the best. The distributed architecture would share the load over a number of subsystems which reduces the load on the central part of the system. To see if there really is a significant difference, let us assume that the Dutch Railways would implement a system as described in chapter 5.

In the example case the following events (i.e. messages) were distinguished for a single train trip:

	Event	Instantiated by
1	Request station access	RFID reader
2	Reply on event 1	System
3	Enter train	RFID reader
4	Exit train	RFID reader
5	Request station exit	RFID reader
6	Reply to station exit	System

Table 2 - Events in example system

Since many people transfer during their journey, it is reasonable to say that the number of events per trip is a little higher than 6. Assuming that 50% of the travellers transfers once (the rest does not transfer), an extra instance of events 3 and 4 must be added for those trips, so that the average number of events per trip becomes

$$(6+8)/2=7$$

In the implemented case the "exit train" events instantiate two other events. So, the average total number of messages per passenger per journey is

$$((6+2)+(8+4))/2=10$$

Dutch Railways state in their annual report that they transport 1 million people every day. Assuming that almost all passengers travel between 7:00 and 23:00, such a day has 16 hours or

$$(16*60*60)=57.600$$

seconds. Now, the following calculation shows that, on average, 173 messages are generated every second.

$$(10*1.000.000)/57.600\approx 173$$

This number is probably not very realistic for every time of the day, so to get some feeling with peaks, let us assume that half of the journeys take place during rush hours. If rush hours are, for

example, defined to be between 7:00 and 9:00 and 16:00 and 18:00, there are four rush hours per day. That means that half of the 10.000.000 messages is generated during 4 hours giving an average of about 347 messages per second.

$$(10 * ½ * 1.000.000)/(4 * 60 * 60) \approx 347$$

In a centralised system, that system has to process all those events. In the distributed variant events 1, 2, 5, and 6 can be handled by subsystems. It is assumed here that those subsystems operate completely independent and that all necessary data is available locally. The rest of the events will still have to be processed by the central part of the system. The following calculations show how many messages still go to the central part.

Messages per trip (only from events 3 and 4, each occurring twice when transferring):

$$(2+4)/2=3$$

Again, the "exit train" events instantiate two other events, so that the average number of messages per passenger per trip becomes 6. This results in about 104 messages per second, equally spread over the day.

$$((2+2)+(4+4))/2=6$$

$$(6 * 1.000.000)/57.600 \approx 104$$

For rush hours the following holds:

$$(6 * ½ * 1.000.000)/(4 * 60 * 60) \approx 208\, msg/s$$

So, implementing the system described in the previous chapter for Dutch Railways would generate the following loads on the *central* part of the system, expressed in the number of messages to handle per second:

	Centralised	**Distributed**
Equally spread	173	104
Rush hour peak	347	208

Table 3 - Load on central system in messages per second

Analysing these numbers, it can easily be seen that in a distributed architecture, the load on the central part of the system would, in this case, be reduced by about 40%. The distributed subsystems have to process the rest of the messages (about 40% of all messages) , together. In this case, there are no additional messages needed to bring the right data to the right place. The subsystems can operate independently. Only incidentally an extra message is needed to update the access check blacklist.

6.1.2 Performance tests

To get a feeling about the number of messages that can reasonably be handled by the centralised solution, a few tests have been done with the following test system:

- machine 1: Intel Mobile Pentium4 2.0 GHz, 1 GB RAM
- machine 2: Intel Pentium4 2.80 GHz, 1 GB RAM
- machine 3: Intel Pentium3 700MHz, 320 MB RAM
- network: Regular 100 Mbit office network
- Communication protocol: TIBCO Rendezvous [13]

The machines and network are *not* dedicated.

Message rates and statistics about sent and received messages are taken from the rendezvous daemon's statistics on both machines.

Test 1

The first test had to give an insight in the number of messages that could reasonably be received by a centralised system. The test was very simple: send messages of about 100 bytes (enough to hold the information of a single RFID tag) from machine 1 to machine 2 at a rate of about 350 messages per second. On machine 2 only a listener was running that simply received all the messages, doing no further processing.

Test 2

For the second test the complete system as described in the previous chapter was stress tested. All processes and an Oracle database ran on machine 2. Machine 1 continuously sent 350 messages per second to machine 2.

Test 3

Test 2 was repeated with the following changes:
- The database ran on machine 1.
- Messages were sent from machine 3 at a rate of about 200 messages per second.

6.1.3 Performance test results

Test 1

The statistics from the rendezvous daemons showed that, after running for one hour, no messages were lost or needed to be re-sent. The stream of messages could thus be handled well.

Test 2 & 3

These tests lasted only about 5 minutes. Where machine 2 had no problems just receiving 350 messages per second, it could not handle all the processing. Although the system did not crash, it took about an hour before all messages were handled. In test 3 this took less time, but still more than half an hour. Machines 1 and 2 operated on a very high system load. Especially the database had a hard time performing so many small queries in such short time.

Because of these results, some experimenting has been done with much less messages per second (i.e. 10), but even then the systems became very stressed, so no reliable results could be taken from test 2 and 3. Also testing a distributed setup would not produce valuable results, so they were not performed.

6.2 Reliability

Reliability was tested in the central architecture by simply disabling the process that took care of receiving messages from the RFID readers. Of course, no more messages were received, no responses came back, no one was admitted at stations and no data was stored. That held for all RFID readers.

In the distributed solution, all readers were connected to their own receiving processes. By disabling one of those processes, i.e. the process responsible for receiving all messages for one station, the same happened to that station as in the centralised test. The other stations, however, kept functioning correctly.

Finally, response times were tested under a low system load. In both solutions a response came back immediately, i.e. at least within a second without any noticeable difference in response times.

6.3 Flexibility

Since the test systems were very small, flexibility was hard to test. During functionality tests, however, it was found out that the distributed system required more work to update all machines for bug fixes. On the other hand, while updating one instance, the rest of the system kept operating.

6.4 Costs

The test case of this research consists of three train stations. The processes for handling messages coming from all of them can all be run on a single machine. Assuming that a separate machine is used to run a database, the complete system consists of 2 machines and the network between them.

Now, if every station has its own machine to handle part of its events, the system consists of 3 distributed machines that all require a database machine and also the 2 centralised machines are still needed. So, the complete system would now consist of 8 machines and network connections between all of them.

If a station is being added, the centralised solution will probably be able to handle that with the available machines. In the distributed architecture, 2 extra machines and network connections are needed. Assuming that for every 5 stations 2 extra machines are needed in the central solution and 2 extra machines are needed in the central part of a distributed machine for every 10 stations, where each station requires 2 machines, the following table can be filled in:

# stations	1	2	3	4	5	10	15	20
Centralised	2	2	2	2	4	6	8	10
Distributed	4	6	8	10	14	26	38	50

Table 4 - Total number of machines needed

Table 4 clearly shows that a distributed architecture requires more machines than a centralised one. Although the machines in a distributed system can maybe be a little lighter, it will stay much more expensive, especially because usually maintenance costs are the biggest part of the total costs.

7 Conclusions

The goal of the research presented in this report was to determine if either a centralised or a distributed software architecture is the best choice for a sensor-based or, more specifically, an RFID-based system. Primary judging criteria were performance, reliability and flexibility. The cost aspect was a secondary criterion.

Before starting this research there were some discussions on which solution would suit best. Some thought a centralised architecture would be best, others, including myself, thought a distributed architecture would be most preferable, purely based on feeling. In this final section the conclusions of the problem analysis and the validation results will be used to answer the questions stated in the problem statement.

7.1 Performance

The conclusion of the performance analysis was that, although a centralised architecture could well be tuned and supported with hardware to perform very well, a distributed architecture would perform better for very large systems with many events.

In 6.1 it is shown that if some functional parts of a system are taken from the centralised system and put distributed, the load on the central part were indeed decreased. For the sample case, the difference was about 40%. This validates that a distributed architecture could help a system to perform better. On the other hand, it was also shown that a message rate of hundreds of messages per second was not a real problem. So, it could be well possible to build a centralised architecture that suffices for Dutch Railways. However, for the whole Dutch public transport system (about 5 million passengers per day), the message rates will probably exceed the maximum number of messages that can be handled by a single system.

For parts of the system that require real time responses, distributing those parts and the accompanying data can help achieve real time responses. Finally, because of the amounts of data, databases fulfil a very important role and highly determine the performance of a system. Although the test setup did not allow to test this, a DNS-like solution is most likely the best for data storage, since it requires little network traffic and the load is spread over multiple databases.

7.2 Reliability

The research showed that a centralised solution would have one weak spot that highly determined the reliability of the system. A distributed architecture has many more weak spots, but if such a part fails, only a relatively small part of the system is affected.

Section 6.2 shows that, in the distributed implementation of the public transport case a subsystem of one train station fails, the rest of the stations still keeps operating correctly. In the centralised implementation, the whole system stops functioning. It can thus be concluded that a distributed design has a higher chance of failure, but that the impact is lower. In both cases certified communication can be used to prevent data loss so that reliability is only harmed in the response time aspect.

7.3 Flexibility

A distributed architecture is more flexible than a centralised one. It is easier to change a part of

the system without harming the rest of it. The drawback is that maintenance requires more work, since more systems have to be maintained that will most likely reside on geographically different locations.

7.4 Costs

The differences in costs between the two compared architectural styles are in hardware, infrastructure and maintenance costs. For all three types of costs, a distributed system is more expensive: more machines are needed, there is more infrastructure between those machines and there are more machines and infrastructure to maintain. The table in section 6.4 confirms this.

7.5 Further thoughts

Thinking about the conclusions presented above, the following thoughts came to mind.

7.5.1 Choosing an architecture

This report presents a comparison of two architectural styles on performance, reliability and flexibility. The research shows that the choice for an architecture mainly depends on the size of a system, where size refers to the number of sensors and events. It is stated that network capacity, database performance and processing power (machines) limit the performance of a centralised system. However, if new technology comes available or existing technology can be significantly optimised, those limitations can probably be stretched so that a centralised architecture can be used with even larger systems. On the other hand, if, for example, wireless networks are used, the network capacity is reduced so that a distributed architecture would be the better choice for smaller systems as well.

Since there are too many dependencies, it is not possible to give a valuable threshold above which a centralised architecture would not be possible any more.

7.5.2 Combining both styles

The research presented in this report focussed on answering the question whether either a centralised or a distributed software architecture would be best suitable for sensor-based systems. But it might be very well possible to combine both styles (recall from 2.1.1 that most often not only one architectural style is applied). Below are some thought on how both styles could be combined in order to improve performance, reliability and/or flexibility. It is very well possible that a measures improves performance, but decreases reliability. It depends on the system to be built which aspect is most important and if such measures can or cannot be used.

Since the choice for an architectural style highly depends on the size of a system, it may be possible to split up a large system into a few smaller centralised systems instead of using a fully distributed architecture.

Another possibility could be to use a distributed architecture for only a few parts of a system, and implement the rest in a centralised way. For example, in the presented case, some busy stations can be built in a distributed way where the rest of the country is done centralised.

Finally, simply adding distributed filtering to a centralised architecture could already increase performance of a system.

7.6 Final conclusion

Looking at the sub conclusions presented above, the final conclusion of this research must be that there is not one answer to the question whether a centralised or a distributed architecture is best for an RFID-based system. The choice of which style to use as a foundation for such a system merely depends on the size of such a system and its expected growth in the future. It might, however, be possible to combine both styles. Concluding, we can say that it is very likely that the cost aspect will be decisive and that the final choice will be a business rather than a technical decision.

8 Appendix A – RFID basics

8.1 Introduction

Radio Frequency IDentification (RFID) is basically a technique that enables wireless and automated identification of objects. These objects need to have a so called "tag", which actually is a small transponder, to be able to identify them with a reader using radio waves. The tag can, depending on its type (see chapter 3), contain a certain amount of data that can be handled by a system behind the reader. RFID is very often compared to the legacy bar code that is almost on every product. Indeed, one could think of replacing or extending these bar codes with a tag containing at least the same information, which would make automated scanning of products easier. But RFID offers us a lot more features which allows it to be used for numerous other applications as well.

At the time of writing of this paper Radio Frequency IDentification is a real hot item. It is in the news very often and a lot of (large) companies are doing pilots with RFID technology in various environments, although very often in supply chains. A benchmark study performed by LogicaCMG [7] shows that companies think that RFID can solve a lot of business pains they go through. It is quite surprising to see that the choice of starting to explore RFID technology is most of the times based on intuition only. Often a cost-revenue analysis is not performed and technical issues are believed to be solved within reasonable time. So, people are convinced that RFID will be widely adopted within a few years and feel the need invest in knowledge of and experience with it, all based on intuition.

In this case study of course there are other reasons to incorporate RFID technology to the project. First of all LogicaCMG in Groningen wanted to know more about this technology in general and also would like to be able to easily integrate RFID in the existing IT systems of its customers. Another reasons is that one of the known problems with RFID is that an enormous amount of data will be generated and people do not really know how to handle that adequately.

In the remainder of this section a closer look will be taken at the RFID technology, its known applications and issues and the way we will use it in this case.

8.2 History and technical details

Although RFID is currently very hot, the technique was already invented in the nineteen-seventies. The father of this technology, Charles Walton, registered the first patent on this subject in 1973. He developed "an electronic identification and recognition system for identifying or recognizing an object carrying an electrically passive circuit" [20].

An RFID system consists of two parts, a usually fixed reader and a tag put on an object. The reader and the tag communicate with each other through radio waves. The tag therefore is equipped with an antenna. This enables the reader to communicate with the tag even when it is out of sight although some conditions need to be met like the tag being within the range of the reader and both parts speaking the same language (i.e. Frequency).

Figure 11 diagrammatically illustrates an identification-recognition system as proposed by Walton. The left part of the system is the reader, an active network, and the right part represents a passive network on a moving object. The latter contains two passive circuits (10A and 10B) each of which contains an inductor and a capacitor that form an electrical resonant circuit. Both inductors are coupled to a sensing coil of the active network inductively. The values of the

components 10A and 10B determine the resonant frequency that the passive network is sensible to. In the schema the detector is a network that interprets the incoming signal from the passive network, which eventually leads to the ID of the object being displayed on a digital display.

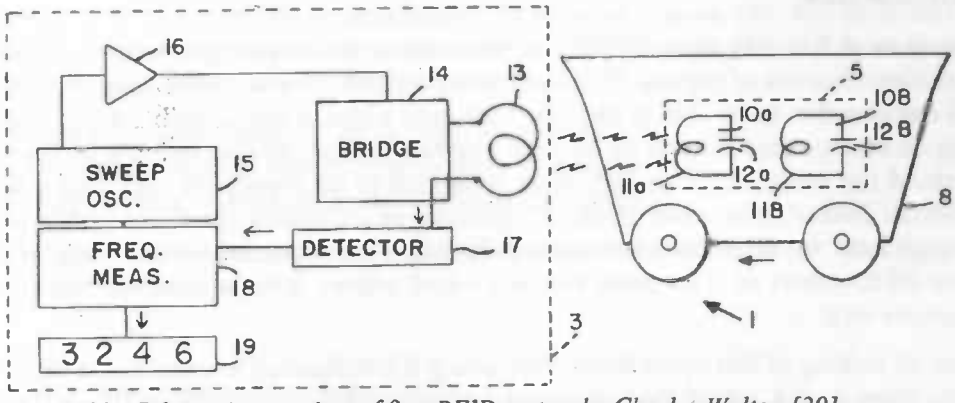


Figure 11 - Schematic overview of first RFID system by Charles Walton [20]

After Walton had invented this technique, he tried to sell it to various companies but was not very successful doing that. Only after more than one year he sold a license to a door lock maker who built the first and maybe still best known application based on RFID technology: electronic door locks. Of course a lot has changed since then, but the system is still based on the same principles. The key card for the described system contained a 36 square-inch circuit board containing various electrical components. Nowadays all these electronics can be put into one single microchip approximately as small as the head of a needle which makes the total size of a tag fully dependent on the geometry of the antenna.

8.3 Different types of tags and their characteristics

Since the invention of RFID some standardisation has been realised on the technique which has resulted in a number of different types of tags. In this section an overview will be given of these types and their specific characteristics like range, reading speed and costs [21]. Besides the tags also RFID readers differ strongly. The choice of which reader to use depends mainly on the environment in which it has to operate and of course on the type(s) of tags it must be able to communicate with.

Active versus passive tags

First of all we need to distinguish between active and passive tags. Walton's invention was based on detecting objects with passive tags but nowadays there is also an active variant. There are quite some differences between these two types, but they are all based on the difference in power supply.

Active tags are powered by a small battery that is integrated in the tag. Usually this is a cell battery that has a high power-to-weight ratio. The use of a battery has a number of advantages:

- great communication range
- high data transmission rates
- operate in quite extreme conditions, for example a temperature range of -50°C to +70°C
- high immunity to noise

These advantages of active tags give them a high potential but one needs to keep in mind that

the extent to which they apply is highly influenced by other characteristics of the tags like the operating frequency. Furthermore, the inclusion of a battery in an RFID tag has some drawbacks in comparison to passive tags as well:

- the battery has a finite lifetime that limits the tag's lifetime; depending on the usage this may still be ten years or even more
- greater size and weight
- greater costs

Passive tags derive their power from the electromagnetic field that is generated by the RFID reader. Therefore, they do not need a separate battery and thus all the drawbacks of the active tags are overcome. Passive tags have the following properties:

- a virtually unlimited lifetime
- much lighter than active tags
- very inexpensive; it is expected that their price will drop to only a few cents within the next five years. This should be mainly accomplished by mass production.

Also passive tags have some disadvantages:

- relatively short read range
- higher-powered reader needed
- more vulnerable to noisy environments
- limited data capacity since only a small amount of power is available to transport data to the reader

Although these disadvantages seem to strongly limit the usage of passive tags the long lifetime and especially the low costs make this type of tags very popular.

In the remainder of this section I will highlight some other characteristics of RFID tags which, unless otherwise indicated, hold for active as well as passive tags.

Radio frequencies

RFID readers and tags communicate through radio waves which are part of the electromagnetic spectrum. The used frequencies can be divided in three frequency bands that all have their own characteristics.

- Low frequency (LF)
  100 to 500 kHz – short/medium distance – slow communication – low costs
- High frequency (HF)
  10 to 15 MHz – short/medium distance – medium communication speed – low/medium costs
- Ultra-High frequency (UHF)
  850 to 950 MHz and 2.4 to 5.8 GHz – long distance – fast communication – high costs

Unfortunately, the specific frequencies that are used in each of these bands differ per region around the world. This is caused by the fact that there are not many frequencies that are available worldwide. Governments regulate the use of frequencies in their countries and there has not been a lot of consistency between countries in that respect. Sometimes also regulations regarding power levels and interference apply. There are efforts to achieve some global standardisation before the year 2010. In October 2004 UHF frequencies between 865 and 868 megahertz have been approved to be used for RFID tags [22]. These frequencies are also available in the United States and Japan. Up to now, the frequencies that are most often used for RFID implementations are 125kHz, 13.56MHz and 2.45GHz respectively.

Reading distance

The reading distance is defined as the maximum distance that the antenna of a tag can be removed from a reader while effective communication is still possible. This distance depends on a number of things like the operating power of the reader and the tag, the geometry of the tag's antenna, the tag's read speed, interference from metal objects, etc. We have also seen that the reading distance depends on the communication frequencies, varying from short to long distance. An a little more narrow specification follows:

- low frequency / short distance: up to a few decimetres
- high frequency / medium distance: up to about one metre
- ultra-high frequency / long distance: up to about 6 metres

When longer distances are needed, active tags might be used to be able to read tags from 100 metres or even more.

Data capacity

The data capacity of RFID tags differs from 1 bit up to 64 kilobits. The amount of data that needs to be stored highly depends on the application. If it is enough to only detect a tag, one bit will do. In case a tag needs to be identified by an identification number or string that is saved on it, 128 to 512 bits will be sufficient. Further data might be stored in a data warehouse from which information can be retrieved that belongs to the identification number. Storage capacities above that can be used to store that extra information on the tag in stead of in a data warehouse. Such tags are said to carry portable data files. Besides just the data, a small part of the capacity can be used for encoding or error detection facilities.

Not all possible capacities apply to all types of tags. One can imagine that a tag that supports only slow communication (low frequency) may be able to carry relatively much data, but it might very well not be able to send it to the reader. If there is enough time, it maybe would not be a problem, but since tagged objects often are moving when they are identified, they cannot transmit the data before they get out of range.

We saw before that active tags have a high data transmission rate. That makes them suitable for all data capacities.

Read and write capabilities

Up to now we only talked about reading data from RFID tags but there are also possibilities to write data on a tag. We can, in this field, distinguish three types of tags: read-only, read/write and write once read many.

Read-only tags are programmed during production and contain only an identification number. This is the type of tag that is most often used when only identification of objects is involved. It can be best compared to the legacy bar codes. Since for read-only tags only one-way communication is satisfactory, they have a very simple layout and low fabrication costs. Theoretically it would be possible to provide these tags with worldwide unique codes, but to achieve that there needs to be an authority that manages these codes and assigns them to the manufacturers.

Write once read many (WORM) tags are shipped blank and can be written exactly once by the user. The data is burned into the transponder. After that they can only be read. This type of tags can for example be used to give objects identification numbers that are unique within the company only or to put more information on a tag than only a number.

Read/write tags contain some sort of programmable memory that can hold its data for years

without needing power. RAM, FRAM or EEPROM are examples of such memory technologies, all having their own specific properties. These properties are outside the scope of this paper.

There is one other aspect on reading RFID tags that is quite important: the number of tags that can be read within a unit of time. First of all this depends on what needs to be read from a tag (only a number or quite some more data) or, in other words, how many bits need to be read from each tag. Next, the transfer rate, which as we saw before highly depends on the frequency, strongly determines how many tags can be read in a unit of time. Finally, a reader can only read data from a limited number of tags concurrently. This may vary from readers being able to communicate with only one tag in their field to about 256.

Overview

An indication of possible transfer rates and reading distances per frequency band is shown in table 5.

Frequency band	**LF**	**HF**	**UHF**
Operating freq	125kHz	13.56MHz	2.45GHz
Transfer rate	1 - 10 KB/s	1 - 3 KB/s	1 KB - 10 MB/s
Distance			
Theoretical	< 2 meters	< 1 meter	1-100 meters
Typical	1 cm - 1.5 m	1 - 70 cm	1 - 3 m

Table 5 - Transfer rates and reading distances of RFID tags per frequency.

8.4 Known applications

Although most companies are currently only experimenting with RFID and wide spread implementation will still take a few years, there are already some systems in production. Here is a list of some examples.

- From January 2005 the USA started to put RFID tags in their passports to prevent fraud.
- In 1999 the city of Vejle, Denmark, successfully implemented an RFID Automatic Vehicle Location System to track all their buses and use this information to inform passengers about actual arrival and departure times.
- For security reasons 160 employees of the anti-crime centre in Mexico City have been implanted with RFID tracking chips.

8.5 Known issues

RFID is a hype and many companies want to start using it for various applications, but there are a number of issues that still must be solved or that need to be worked around.

- Standards on communication, used frequencies and data structures are not fixated yet. Authorities like EPC Global work on such standards. For world wide implementations of RFID, such standards must be clearly defined.
- Defining standards is complicated. Every country manages its own frequency ranges which results in the same frequencies not being available in all countries. Still, standard frequencies are needed.
- RFID tags are quite large, especially when knowing that the chip itself has a size of only one

square millimetre. The attached antenna causes the large size, which cannot easily be reduced.

- Although prices are dropping, RFID tags still are quite expensive. When real mass production starts, prices will drop. Generally, a price of one or two cents is seen as the final goal.
- Radio waves have some limitations. The environment highly determines how well radio waves can travel in a certain space. For example, radio waves cannot go through water, so reading the tag on a bottle of milk or water is not trivial.

8.6 Privacy

Besides all technical and business aspects, there is also an ethical aspect on which opinions differ. Some people think usage of RFID technology will lead to a "big brother" society, others think privacy will not become a major issue, arguing that, nowadays, we can already be traced by our cell phones. The difference here is that RFID tags will probably be readable by every single reader and anyone could purchase a reader.

Without going into much detail, here are some thoughts and ideas to protect people's privacy:

- Store only a unique id on a tag; the data belonging to the tagged product is only available to the authority that provided the tag. Other people can at most read the unique number, but then cannot acquire the accompanying data.
- Encrypt data on tags. Initiatives have started to encrypt data on tags so that only authorised people and authorities can read the data on a tag.
- Destroy a tag after use. RFID tags on for example cloths could be destroyed after a customer has paid for it. Techniques to do that are scrambling the tag's memory or simply sending too much power to the tag.

9 Appendix B - Software Documentation

This appendix presents a more detailed description of the software that was built for the prototype of a public transport system based on Radio Frequency Identification (RFID) technology, that was used for validation purposes in the research presented in this report.

9.1 System description

The built system is a prototype of what public transport could look like when Radio Frequency Identification is incorporated in it. The goal is to no longer use all different kinds of public transport tickets, but just give everyone a smart card, say of credit card dimensions, containing an RFID tag. People only need to take this smart card and can just hop on or off any type of public transport any time they want. By identifying a person when he or she hops on and again when he or she hops off, in combination with the start and end stations, pricing and billing can take place automatically. The main advantage for travellers is not having to think about buying the correct tickets when wanting to use public transport and having one card for all transport types.

Usage of RFID can also enable additional services. For example, if, when a person orders an RFID smart card, the person's mobile phone number is registered, the public transport company could send SMS messages to all persons on a particular train or bus regarding current delays, arrival times, transfer options etc. Another option is to monitor a persons travel behaviour and recommend to him what kind of season ticket would be most profitable. It could also be easy to get a lot of statistical information about occupancy of trains, most or least travelled routes, etc. Even discounts for delayed trains can be automatically applied. Because it is known where a person is, public transport companies have the possibility of personal advertising for nearby shops. So, advantages for companies are personal services, advertisements and accurate statistical information and billing procedures.

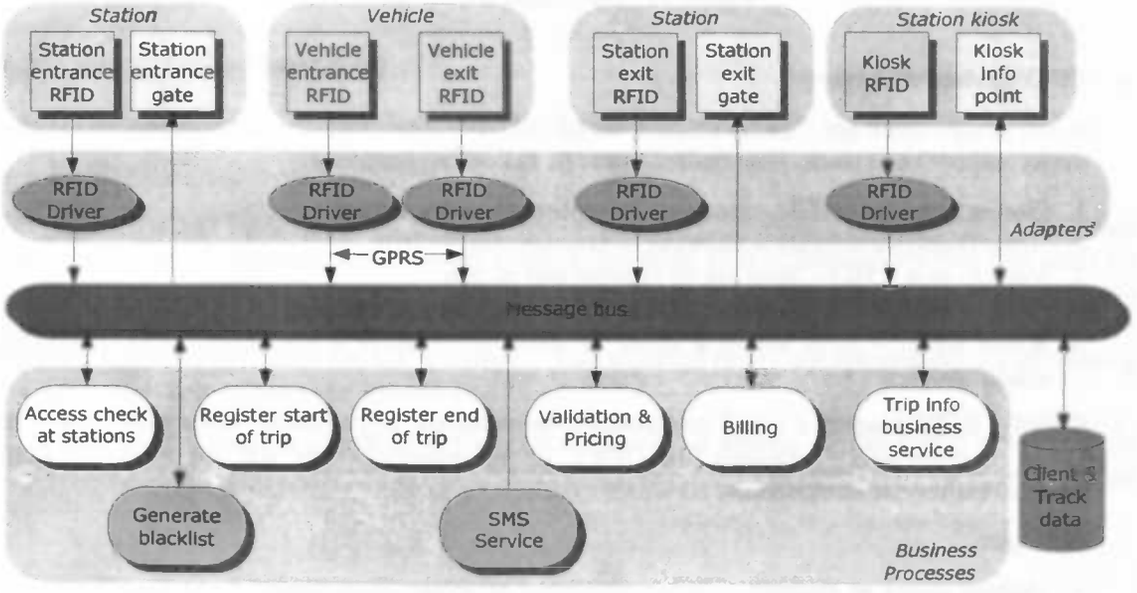


Figure 12 - Centralised architecture

9.2 Software architecture

Two architectures, both service-oriented, have been designed for the prototype: a centralised and a distributed architecture. Both architectures are shown in figures 12 and 13. Since both architectures consist of the same components, there will be no distinction between the two in the rest of this document. On the next page, the components will be described briefly.

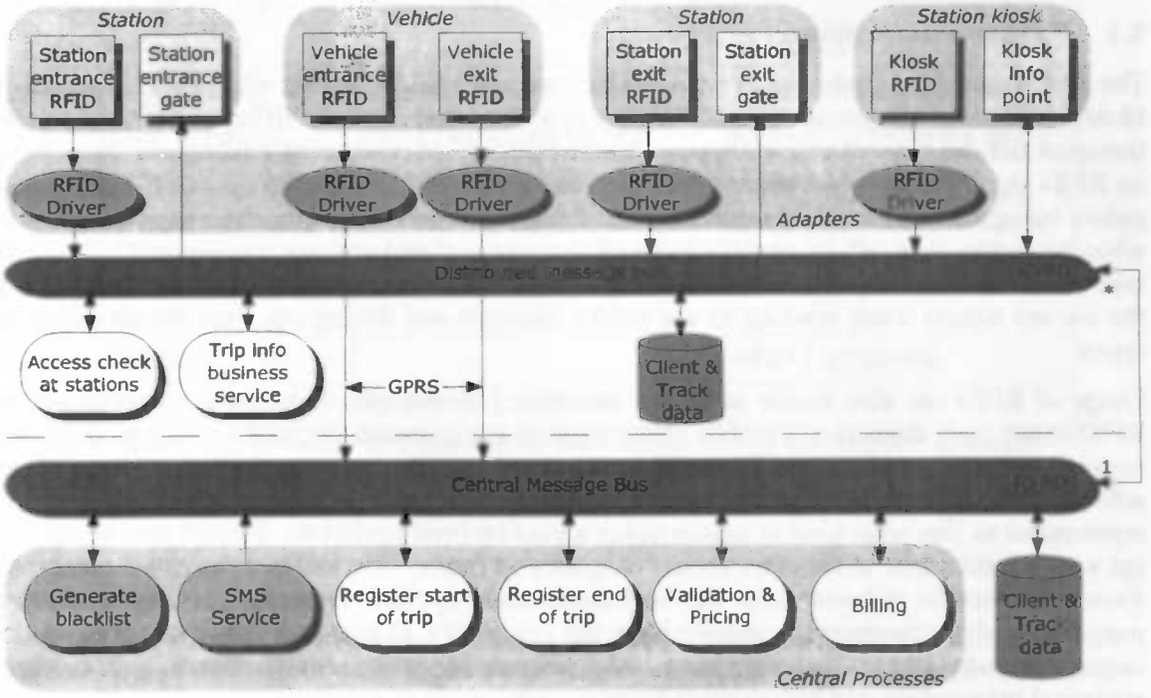


Figure 13 - Distributed architecture

- RFID devices & drivers

  In the design there are 5 RFID readers which all have their own driver or adapter to connect to the rest of the system. The readers have the following purposes:

  1. Station entrance RFID – Scan tag to perform an access check
  2. Vehicle entrance RFID – Scan tag to register start of trip
  3. Vehicle exit RFID – Scan tag to register end of trip
  4. Station exit RFID – Scan tag to register person leaving a station
  5. Kiosk RFID – Scan tag at a kiosk to fetch information about recent trips, remaining credit etcetera.

- GPRS – RFID readers 2 and 3 are installed on vehicles, so communication with the system could be done using protocols like GPRS. In the prototype they will be connected using standard ether net connections.

- Services

  - Access check at stations – When receiving an event from reader 1, an access check is performed and the result is sent back to the exit gate which opens if access is granted.
  - Register start of trip – Store the start of a new trip for a certain person in the database.
  - Register end of trip – Store the end of a trip for a certain person in the database and

publish a message that activates pricing of this trip.
    - Validation & Pricing – After a trip has been ended, a message is published to inform this service that a trip was finished. This service validates the trip according to specific criteria and queries one or more databases to retrieve distance and price information which is then used to calculate and store the price of the trip in the database.
    - Billing – Once a trip has been priced, a message is published to this service indicating that billing the trip can take place.
    - Generate Blacklist – This helper service is used to fill the blacklist of the access control systems at stations (and possibly vehicles as well).
    - SMS Service – This service is drawn in the architectures as an example of an additional service. It will not be implemented. For more possible additional services, see section 5.5.1.
- Trip info business service – The trip info business services can be used by multiple components to retrieve information about certain trips. One example is the kiosk info point.
- Kiosk info point – After receiving an event from RFID reader 5, the trip info business service is queried and the results are displayed.
- Client & Track data – Database containing information on clients, open and finished trips, a distance table and pricing information.
- Message bus – The message buses are the central parts of the systems and subsystems that connect all services, devices and databases. It is used to deliver rendezvous messages that are published by services and readers to the correct locations using broadcasting.

The difference between the two architectures is that in the central architecture there is only one message bus and in the distributed one, every subsystem has its own distributed message bus. To the distributed message bus, the access check and trip info services are connected. They are not connected to the central message bus any more, indicating that the trip info service and the access check are performed completely distributed. The same holds for the station exit events.

9.3 Rendezvous messaging

9.3.1 About Rendezvous

The prototype has been designed as a service-oriented architecture. Therefore, message-based communication between the components is used. Because TIBCO BusinessWorks was used for implementation, TIBCO Rendezvous (RV) was chosen as communication protocol. Everything connected to a message bus (see figures 12 and 13), communicates through RV messages. Also the GUI application listens for RV messages. The message buses are responsible for delivering the right messages to the right listeners.

A rendezvous message is a structured message containing a subject, a time stamp and a number of custom fields. Messages can vary in size between, say, 10 bytes to megabytes. The messages used in this prototype are about 100 bytes in size.

Subjects

Every service listens for an RV message with a specific subject. The following subjects are used:

Component*	Listens for	Sends
Station entrance RFID		RFID.TAG.PRE-ENTRANCE
Access check at stations	RFID.TAG.PRE-ENTRANCE	
Station entrance gate	**	RFID.
Vehicle entrance RFID		RFID.TAG.ENTRANCE
Register start of trip	RFID.TAG.ENTRANCE	
Vehicle exit RFID		RFID.TAG.EXIT
Register end of trip	RFID.TAG.EXIT	RFID.TRIPCOMPLETED
Validation & Pricing	RFID.TRIPCOMPLETED	RFID.RFID.TRIPPRICED
Billing	RFID.TRIPPRICED	
Station exit RFID		RFID.TAG.STATIONEXIT
Station exit service	RFID.TAG.STATIONEXIT	
Station exit gate	**	
Kiosk RFID		RFID.TAG.KIOSK
Kiosk info point	RFID.TAG.KIOSK	

* The drivers of the RFID components are responsible for sending RV messages

** Gates are implemented as "send a message to the GUI". Those messages are sent using a separate process MsgToGUI.

Message structures

All rendezvous messages that have a subject that starts with "RFID.TAG." have the same structure, which is shown in table 6. All other messages only contain one or more ID's that can be used to retrieve the required data from a database.

Field	Type	Description	Example value
UID	String	Tag's unique ID	70C9D701000104E0
Timestamp	DateTime	Time of reading the tag	2005-02-25T13:46:27+01:00
Data	String	Free data field (client name for example)	Jan Jansen
DeviceID	String	Contains type of transport + station/position	Train.Groningen

Table 6 - RV message structures of RFID.TAG.> messages

Routing daemons

When implementing the distributed architecture, rendezvous routing daemons are needed to connect different networks to each other and to make sure that (only) messages with specific subjects are routed to networks that have services that listen for those messages. Figure 14 schematically shows how the daemons should be positioned. For more details about the rendezvous routing daemons, please refer to the TIBCO rendezvous documentation.

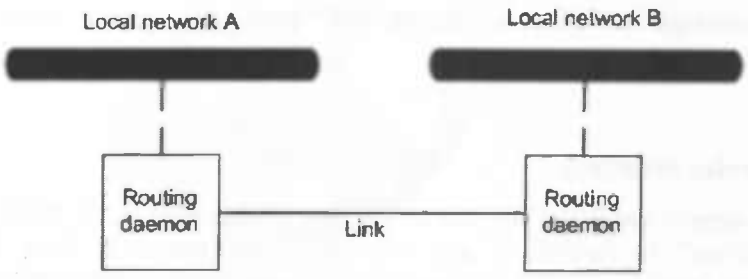


Figure 14 - Separate networks connected through routing daemons

9.4 TIBCO BusinessWorks Processes

About BusinessWorks

In BusinessWorks (BW) everything is about processes. Every process has its own functionality and may also call other processes. By tightly separating functionality into different processes, they can be easily deployed on other systems. Also a number of settings can be adjusted per process at deploy time.

BW comes with a graphical designer in which processes can be defined. Every process typically has a starter and an ender between which a number of activities can be put. All activities are connected through conditional connectors that represent the flow of operations that the process must perform. Many of the processes implemented for the prototype start with a rendezvous subscriber that listens for rendezvous messages with a certain subject. Most processes then confirm having received a message, do some database queries and call a separate process that sends information to a GUI.

Implemented processes

The following processes were implemented to fulfil the functional requirements. The processes represent the components as described in the software architecture.

- Access check at stations
- Register start of trip
- Register end of trip
- Station exit
- Validation & Pricing
- Billing
- Kiosk info point
- Trip info service
- Generate blacklist

In the BW project, every process and every activity in it have a description. Please refer to the supplemented "Detailed Software Documentation" for detailed descriptions of all processes.

Centralised versus Distributed deployment

The TIBCO system integration suite contains an Administrator application that can be used to deploy the BW processes that were implemented using the Designer application. In the Administrator it is possible to deploy processes on (several) different machines. Also, several parameters can be set deploy time or run time.

By deploying processes on different machines, and making sure those machines are in different networks, the prototype can be deployed in a distributed way. Rendezvous routing daemons are then necessary to route all the rendezvous messages to the right networks and machines.

For more information on how to deploy BW processes, please refer to the TIBCO Administrator documentation.

9.5 RFID device interface

To be able to connect the RFID devices to the rest of the system, an RFID device interface (called "RFIDdriver" in figures 12 and 13) was implemented in Java. In the following paragraphs, a high level description is presented, for more detailed information on the interface, please refer to the Java documentation that comes with this report as a supplement.

Devices

The RFID devices used in the prototype are: Omron V720S_HMC73 High Frequency readers. The devices support full read and write capabilities. They are, however, restricted to read only one tag that is in range at a time and have only a short communication range of about 25 mm. For more information about the used RFID devices, please refer to their documentation.

Architecture

The device interface was implemented according to the architecture shown in figure 15. An application can use the interface in two ways: (1) it registers itself as an "RFIDResponseListener" to the interface, or (2) it listens for rendezvous messages with specific subjects, that can be defined in the configuration of the interface.

For a detailed description of the components and their methods please refer to the Java documentation.

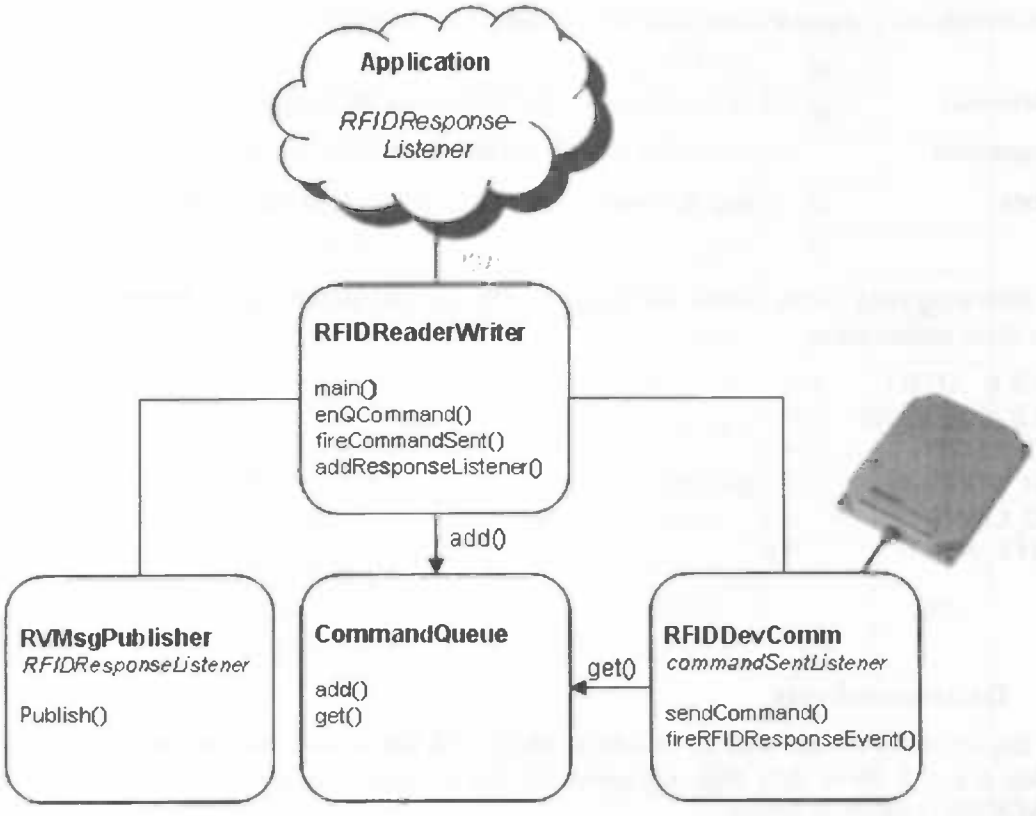


Figure 15 - Architecture of RFID device interface

Configuration

The RFID device interface can be configured using either command line options or a configuration file (the latter overrides the first). The following options can be configured in a plain text configuration file, with one settings per line. The '#' character can be used to insert comments.

- useRV - whether or not to publish rendezvous messages
- daemon - rendezvous settings, please refer to the rendezvous documentation
- service - rendezvous settings, please refer to the rendezvous documentation
- network - rendezvous settings, please refer to the rendezvous documentation
- cmname - rendezvous settings, please refer to the rendezvous documentation
- comport - serial communication port the device is connected to
- subject - subject that published rendezvous messages should get
- errorsubject - subject that published rendezvous messages should get when a read or write error occurs

- initsubject - subject that the interface's initialisation rendezvous messages should get
- deviceID - id of the device to the rendezvous messages
- command - command to send to the device, see listing below
- data - string that must be written to a tag's free space (write modes only)

The following read / write modes are available. Please refer to the device documentation to see what these modes mean.

SINGLE_AUTO = 1
SINGLE_REPEAT = 2
FIFO_AUTO = 3
FIFO_REPEAT = 4 (default)
FIFO_CONT = 5
WRITE_AUTO = 6

9.6 Database schema

The implemented system uses an Oracle database with the database tables described in table 7. Tables 8 and 9 show data that was used for the prototype system in the PRICEINFO and TRACKINFO database tables.

Table	Field	Nullable	Data type	Length	Default
BLACKLIST	TAGUID	N	VARCHAR2	16	
	REASON	Y	VARCHAR2	50	
CLIENT	TAGUID	N	VARCHAR2	50	
	SURNAME	Y	VARCHAR2	50	
	FIRSTNAME	Y	VARCHAR2	50	
	STREET	Y	VARCHAR2	50	
	NR	Y	VARCHAR2	4	
	ZIP	Y	VARCHAR2	6	
	CITY	Y	VARCHAR2	50	
	PHONE	Y	NUMBER		
	MOBILE	Y	NUMBER		
	MONEY	Y	NUMBER		1000
	COMMENTS	Y	VARCHAR2	500	
PRICEINFO	TRANSTYPE	N	VARC	50	
	KMPRICE	N	NUMBER		20
TRACKINFO	STATION1	N	VARCHAR2	50	
	STATION2	N	VARCHAR2	50	
	KM	N	NUMBER		
TRIP	ID	N	NUMBER		
	TAGUID	N	VARCHAR2	16	
	STARTTIME	N	VARCHAR2	50	
	STARTPOS	N	VARCHAR2	50	
	ENDTIME	Y	VARCHAR2	50	
	ENDPOS	Y	VARCHAR2	50	
	TRANSPORTTYPE	N	VARCHAR2	10	
	PRICE	Y	NUMBER		
	PRICED	Y	NUMBER		0
	BILLED	Y	NUMBER		0
	INVALID	Y	NUMBER		0

Table 7 - Database tables for prototype

TRANSTYPE	KMPRICE
TRAIN	20
BUS	10
SUBWAY	15
FERRY	50

Table 8 - PRICEINFO data for prototype

STATION1	STATION2	KM
Groningen	Zwolle	100
Groningen	Leeuwarden	85
Zwolle	Leeuwarden	90
Groningen	Assen	30
Assen	Zwolle	70
Groningen	Amersfoort	170
Amersfoort	Schiphol	60
Rotterdam	Amersfoort	85
Groningen-Grote Markt	Groningen-CS	2
Groningen-CS	Groningen-Kranenburg	4
Rotterdam-CS	Rotterdam-Alexander	10
Schiphol-CS	Amsterdam-CS	20
Rotterdam-CS	Rotterdam-Ahoy	6

Table 9 - TRACKINFO data for prototype

References

1. Bosch, Koskimies, *The Architecture of Software Systems,* Unpublished, 2004
2. Williams, Smith, *PASA: A Method for the Performance Assessment of Software Architectures,* CMG Proceedings, 2002
3. Goseva-Popstojanova, Trivedi, *Architecture Based Software Reliability,* Elsevier Science, 2001
4. Tanenbaum, Van Steen, *Distributed Systems - Principles and Paradigms,* Prentice Hall, 2002
5. Bosch, *Design & Use of Software Architectures,* Addison-Wesley, 2000
6. Gartner, *Service-Oriented Architecture Scenario,* Gartner, 2003
7. *RFID Benchmark Study,* LogicaCMG, 2004
8. *http://www.epcglobalinc.com*
9. King, *The Tsunami of Data Growth,* www.windowsitpro.com, 2004
10. Van der Kooij, *RFID veroorzaakt data-lawine,* Computable, 2004
11. Clements, Kazman, Klein, *Evaluating a Software Architecture,* Addison Wesley2001,
12. Bosloper, Siljee, Nijhuis, *Modeling Dynamic Software Systems with Variation Points,* Unpublished, 2004
13. TIBCO, *Rendezvous,* http://www.rv.tibco.com/datasheet.html
14. Fielding, *Architectural Styles and the Design of Network-based Software Architectures,* 2000
15. Aronoff, *High Speed Replication for Oracle,* Oracle, 2000
16. *Daffodil Replicator,* Daffodil (http://www.daffodildb.com/oracle-replication.html)
17. Verisign, *EPC Network Architecture,* Verisign, 2004
18. BusinessWorks, *http://www.tibco.com/software/business_integration/businessworks.jsp,* TIBCO
19. *http://www.tibco.com*
20. Ch. A. Walton, *US Patent 3,752,960,* US Patent Office, 1973
21. *RFID - A basic primer,* AIM, 2001
22. *Europese goedkeuring voor RFID in UHF-band,* Automatiseringgids 19-10-2004, 2004

Document history

date	Issue	Status	author
2005-02-04	0.1	First draft, presented for review	Niels Heinis
2005-02-10	0.2	Added draft version of design and implementation parts of chapter 5 Processed René Boverhuis' review of issue 0.1	Niels Heinis
2005-03-07	0.3	Processed Rein Smedinga's review of issue 0.2 Added conclusion chapter 4 Completed chapters 2 and 5, added chapters 6 and 7	Niels Heinis
2005-03-08	0.4	Added introduction and management summary	Niels Heinis
2005-03-14	0.5	Added title page Processed René Boverhuis' review of issue 0.4	Niels Heinis
2005-03-16	0.6	Processed Rein Smedinga's review of issue 0.4	Niels Heinis
2005-03-30	0.7	Some layout changes (blank pages etc.) Processed Rein Smedinga's review of issue 0.6 Processed Jan Jongejan's review of issue 0.6	Niels Heinis
2005-03-31	0.8	Added Appendix B	Niels Heinis
2005-03-31	1.0	Made final	Niels Heinis

Bijlage
Niels Heinis

WORDT
NIET UITGELEEND

Package rfidDriver

Interface Summary	
RFIDCommandSentListener	Title: RFIDDriver - RFIDCommandSentListener Interface Description: Interface that must be implemented in order to receive RFIDCommandSent events V720S Copyright: Copyright (c) 2004 Company: LogicaCMG Author: Niels Heinis
RFIDResponseListener	Title: RFIDDriver - RFIDResponseListener Interface Description: To be able to receive RFIDResponseEvents, this interface must be implemented.

Class Summary	
Globals	Title: RFIDDriver - Globals Description: Class containing a number of static constants and variables Copyright: Copyright (c) 2004 Company: LogicaCMG Author: Niels Heinis
RFIDCommandSentEvent	Title: RFIDDriver - RFIDCommandSentEvent Description: Event that is raised when a command is added to RFIDReaderWriter's command queue Copyright: Copyright (c) 2004 Company: LogicaCMG Author: Niels Heinis
RFIDDeviceCommunication	Title: RFIDDriver - RFIDDeviceCommunication Description: Maintains low-level connecting with serial port, sends and receives raw data and converts it Copyright: Copyright (c) 2004 Company: LogicaCMG Author: Niels Heinis
RFIDQueue	Title: RFIDDriver - Generic queue Description: Multi purpose queue, used for command and response queues Copyright: Copyright (c) 2004 Company: LogicaCMG Author: Niels Heinis
RFIDReaderWriter	Title: RFIDReaderWriter Description: Java interface for Omron RFID reader/writer V720S_HMC73 Copyright: Copyright (c) 2004 Company: LogicaCMG Author: Niels Heinis
RFIDResponseEvent	Title: RFIDDriver - RFIDResponseEvent Description: This event is fired when a response from the RFID device is received and decoded by RFIDDeviceCommunication Copyright: Copyright (c) 2004 Company: LogicaCMG Author: Niels Heinis
RFIDRVMsgPublisher	Title: RFIDDriver - RFIDRVMsgPublisher Description: Class containing methods to publish TIBCO RendezVous messages Copyright: Copyright (c) 2004 Company: LogicaCMG Author: Niels Heinis

rfidDriver

Interface RFIDCommandSentListener

All Known Implementing Classes:
RFIDDeviceCommunication

public interface **RFIDCommandSentListener**

Title: RFIDDriver - RFIDCommandSentListener Interface

Description: Interface that must be implemented in order to receive RFIDCommandSent events V720S

Copyright: Copyright (c) 2004

Company: LogicaCMG

Author: Niels Heinis

Method Summary

void	**commandSent**(RFIDCommandSentEvent cse)

Method Detail

commandSent

void **commandSent**(RFIDCommandSentEvent cse)

rfidDriver

Interface RFIDResponseListener

All Known Implementing Classes:
RFIDDeviceCommunication, RFIDRVMsgPublisher

```
public interface RFIDResponseListener
```

Title: RFIDDriver - RFIDResponseListener Interface

Description: To be able to receive RFIDResponseEvents, this interface must be implemented.

Copyright: Copyright (c) 2004

Company: LogicaCMG

Author: Niels Heinis

Method Summary

void	**responseReceived**(RFIDResponseEvent r, boolean ok) To be able to receive RFIDResponseEvents, you need to implement this interface.

Method Detail

responseReceived

```
void responseReceived(RFIDResponseEvent r,
                      boolean ok)
```

To be able to receive RFIDResponseEvents, you need to implement this interface. The boolean parameter indicates that the command was succesful (true) or that an error ocurred (false).

Parameters:
r - RFIDResponseEvent
ok - boolean

rfidDriver

Class Globals

```
java.lang.Object
  └ rfidDriver.Globals
```

```
public class Globals
extends java.lang.Object
```

Title: RFIDDriver - Globals

Description: Class containing a number of static constants and variables

Company: LogicaCMG

Author: Niels Heinis

Field Summary

static int	**BCC_ERROR** Constant for BCC error Corresponds to error code from device
static java.lang.String	**cmname**
static int	**command**
static int	**COMMAND_ERROR** Constant for command error.
static int	**COMMUNICATIONS_ERROR** Constant for communications error.
static int	**comport**
static java.lang.String	**configFile**
static int	**CUSTOM**
static java.lang.String	**daemon**
static java.lang.String	**data**

static java.lang.String	**deviceID**
static java.lang.String	**errorSubject**
static int	**FIFO_AUTO** Constant for read mode FIFO Auto
static int	**FIFO_CONT** Constant for read mode FIFO Continuous.
static int	**FIFO_REPEAT** Constant for read mode FIFO Repeat
static int	**FORMAT_ERROR** Constant for format error.
static int	**FRAME_ERROR** Constant for frame error.
static int	**FRAMING_ERROR** Constant for framing error.
static java.lang.String	**initSubject**
static java.lang.String	**network**
static int	**NOTAG_ERROR** Constant for no tag error.
static int	**OVERRUN_ERROR** Constant for overrun error.
static int	**PARITY_ERROR** Constant for parity error.
static int	**RESET** Constant for Reset command
static java.lang.String	**service**
static int	**SINGLE_AUTO** Constant for read mode Single Auto
static int	**SINGLE_REPEAT** Constant for read mode Single Repeat
static java.lang.String	**subject**
static int	**TEST** Constant for Test command.
static int	**UNKNOWN**
static boolean	**useRV**

static java.util.TimeZone	**UTCTimeZone**
static int	**WRITE_AUTO** Constant for Single Auto Write mode
static int	**WRITE_ERROR** Constant for write error.

Constructor Summary

Globals()

Method Summary

static void	**loadSettingsFile**()
static void	**printSettings**() Print settings during initialisation
static void	**setGlobal**(java.lang.String var, java.lang.String val)

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

SINGLE_AUTO

public static final int **SINGLE_AUTO**

Constant for read mode Single Auto

See Also:

Constant Field Values

SINGLE_REPEAT

public static final int **SINGLE_REPEAT**

Constant for read mode Single Repeat

See Also:

Constant Field Values

FIFO_AUTO

```
public static final int FIFO_AUTO
```

Constant for read mode FIFO Auto

See Also:

Constant Field Values

FIFO_REPEAT

```
public static final int FIFO_REPEAT
```

Constant for read mode FIFO Repeat

See Also:

Constant Field Values

FIFO_CONT

```
public static final int FIFO_CONT
```

Constant for read mode FIFO Continuous. This mode requires an ACK to be sent to the device after a response is received

See Also:

Constant Field Values

WRITE_AUTO

```
public static final int WRITE_AUTO
```

Constant for Single Auto Write mode

See Also:

Constant Field Values

TEST

```
public static final int TEST
```

Constant for Test command. Response of the device should be 00 and the same data as was sent to the device

See Also:

Constant Field Values

RESET

```
public static final int RESET
```

Constant for Reset command

See Also:
Constant Field Values

CUSTOM

```
public static final int CUSTOM
```

See Also:
Constant Field Values

UNKNOWN

```
public static final int UNKNOWN
```

See Also:
Constant Field Values

PARITY_ERROR

```
public static final int PARITY_ERROR
```

Constant for parity error. Corresponds to error code from device

See Also:
Constant Field Values

FRAMING_ERROR

```
public static final int FRAMING_ERROR
```

Constant for framing error. Corresponds to error code from device

See Also:
Constant Field Values

OVERRUN_ERROR

```
public static final int OVERRUN_ERROR
```

Constant for overrun error. Corresponds to error code from device

See Also:
Constant Field Values

BCC_ERROR

```
public static final int BCC_ERROR
```

Constant for BCC error Corresponds to error code from device

See Also:

Constant Field Values

FORMAT_ERROR

```
public static final int FORMAT_ERROR
```

Constant for format error. Corresponds to error code from device

See Also:

Constant Field Values

FRAME_ERROR

```
public static final int FRAME_ERROR
```

Constant for frame error. Corresponds to error code from device

See Also:

Constant Field Values

COMMUNICATIONS_ERROR

```
public static final int COMMUNICATIONS_ERROR
```

Constant for communications error. Corresponds to error code from device

See Also:

Constant Field Values

WRITE_ERROR

```
public static final int WRITE_ERROR
```

Constant for write error. Corresponds to error code from device

See Also:

Constant Field Values

NOTAG_ERROR

```
public static final int NOTAG_ERROR
```

Constant for no tag error. Corresponds to error code from device

See Also:
Constant Field Values

COMMAND_ERROR

```
public static final int COMMAND_ERROR
```

Constant for command error. Corresponds to error code from device

See Also:
Constant Field Values

---

configFile

```
public static java.lang.String configFile
```

---

comport

```
public static int comport
```

---

useRV

```
public static boolean useRV
```

---

service

```
public static java.lang.String service
```

---

network

```
public static java.lang.String network
```

---

daemon

```
public static java.lang.String daemon
```

---

cmname

```
public static java.lang.String cmname
```

---

deviceID

```
public static java.lang.String deviceID
```

subject

```
public static java.lang.String subject
```

errorSubject

```
public static java.lang.String errorSubject
```

initSubject

```
public static java.lang.String initSubject
```

command

```
public static int command
```

data

```
public static java.lang.String data
```

UTCTimeZone

```
public static java.util.TimeZone UTCTimeZone
```

Constructor Detail

Globals

```
public Globals()
```

Method Detail

loadSettingsFile

```
public static void loadSettingsFile()
```

setGlobal

```
public static void setGlobal(java.lang.String var,
                             java.lang.String val)
```

printSettings

```
public static void printSettings()
```

Print settings during initialisation

rfidDriver

Class RFIDCommandSentEvent

```
java.lang.Object
  └java.util.EventObject
      └rfidDriver.RFIDCommandSentEvent
```

All Implemented Interfaces:
java.io.Serializable

```
public class RFIDCommandSentEvent
extends java.util.EventObject
```

Title: RFIDDriver - RFIDCommandSentEvent

Description: Event that is raised when a command is added to RFIDReaderWriter's command queue

Copyright: Copyright (c) 2004

Company: LogicaCMG

Author: Niels Heinis

See Also:
Serialized Form

Field Summary

Fields inherited from class java.util.EventObject

source

Constructor Summary

RFIDCommandSentEvent(java.lang.Object source)

Method Summary

Methods inherited from class java.util.EventObject

getSource, toString

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

Constructor Detail

RFIDCommandSentEvent

public **RFIDCommandSentEvent**(java.lang.Object source)

rfidDriver

Class RFIDDeviceCommunication

```
java.lang.Object
  └rfidDriver.RFIDDeviceCommunication
```

All Implemented Interfaces:

java.util.EventListener, javax.comm.SerialPortEventListener, RFIDCommandSentListener, RFIDResponseListener

```
public class RFIDDeviceCommunication
extends java.lang.Object
implements javax.comm.SerialPortEventListener, RFIDCommandSentListener, RFIDResponseListener
```

Title: RFIDDriver - RFIDDeviceCommunication

Description: Maintains low-level connecting with serial port, sends and receives raw data and converts it

Copyright: Copyright (c) 2004

Company: LogicaCMG

Author: Niels Heinis

Constructor Summary

`RFIDDeviceCommunication(int comport, RFIDQueue commandQ)`	Constructor to create a new RFIDDeviceCommunication object that maintains the real serial port connection and the command and response queues.

Method Summary

void	`addResponseListener(RFIDResponseListener rl)` Registers rl to listen to RFIDResponseEvents.
void	`close()` Close in- and outputstream and serial port connection
void	`commandSent(RFIDCommandSentEvent cse)` When an RFIDCommandSentEvent is received checks if device is busy.
boolean	`isConnected()` This method may be called to check if a connection to the hardware device is available.
void	`removeResponseListener(RFIDResponseListener rl)` Unregisters rl; no more RFIDResponseEvents will be received.
void	`resetDevice()` The resetDevice() method is called on creation of this object, before eventlisteners can be added and sends a reset command to the hardware device
void	`responseReceived(RFIDResponseEvent r, boolean ok)` This method is invoked when an RFIDResponseEvent is raised

void	serialEvent(javax.comm.SerialPortEvent event) This object listens to events from its SerialPort through this event listener

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

RFIDDeviceCommunication

```
public RFIDDeviceCommunication(int comport,
                               RFIDQueue commandQ)
```

Constructor to create a new RFIDDeviceCommunication object that maintains the real serial port connection and the command and response queues. The latter is not yet used.

Parameters:

comport - int
commandQ - RFIDQueue

Method Detail

resetDevice

```
public void resetDevice()
```

The resetDevice() method is called on creation of this object, before eventlisteners can be added and sends a reset command to the hardware device

---

isConnected

```
public boolean isConnected()
```

This method may be called to check if a connection to the hardware device is available. This function is not implemented very well yet and thus may not give the right result.

Returns:

boolean

---

close

```
public void close()
```

Close in- and outputstream and serial port connection

---

serialEvent

```
public void serialEvent(javax.comm.SerialPortEvent event)
```

This object listens to events from its SerialPort through this event listener

Specified by:

serialEvent in interface javax.comm.SerialPortEventListener

Parameters:

event - SerialPortEvent

---

commandSent

```
public void commandSent(RFIDCommandSentEvent cse)
```

When an RFIDCommandSentEvent is received checks if device is busy. If so, lets command wait, else gets command from queue and sends it to the device

Specified by:

commandSent in interface RFIDCommandSentListener

Parameters:

cse - RFIDCommandSentEvent

---

responseReceived

```
public void responseReceived(RFIDResponseEvent r,
                             boolean ok)
```

This method is invoked when an RFIDResponseEvent is raised

Specified by:

responseReceived in interface RFIDResponseListener

Parameters:

r - RFIDResponseEvent

ok - boolean

---

addResponseListener

```
public void addResponseListener(RFIDResponseListener rl)
```

Registers rl to listen to RFIDResponseEvents.

Parameters:

rl - RFIDResponseListener

---

removeResponseListener

```
public void removeResponseListener(RFIDResponseListener rl)
```

Unregisters rl; no more RFIDResponseEvents will be received.

Parameters:

rl - RFIDResponseListener

---

rfidDriver

Class RFIDQueue

```
java.lang.Object
  └rfidDriver.RFIDQueue
```

```
public class RFIDQueue
extends java.lang.Object
```

Title: RFIDDriver - Generic queue

Description: Multi purpose queue, used for command and response queues

Copyright: Copyright (c) 2004

Company: LogicaCMG

Author: Niels Heinis

Constructor Summary

RFIDQueue() Creates a new empty RFIDQueue.

Method Summary

void	add(java.lang.Object o) Add any object o to the end of this RFIDQueue
java.lang.Object	get() Takes the first item of this RFIDQueue, returns it and deletes it from the RFIDQueue
boolean	isEmpty() Test is this RFIDQueue is empty
int	size() Returns the number of objects in this RFIDQueue

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

RFIDQueue

```
public RFIDQueue()
```

Creates a new empty RFIDQueue. RFIDQueue uses a private List object to maintain its data.

Method Detail

add

```
public void add(java.lang.Object o)
```

Add any object o to the end of this RFIDQueue

Parameters:
o - Object

---

get

```
public java.lang.Object get()
```

Takes the first item of this RFIDQueue, returns it and deletes it from the RFIDQueue

Returns:
Object

---

isEmpty

```
public boolean isEmpty()
```

Test is this RFIDQueue is empty

Returns:
boolean

---

size

```
public int size()
```

Returns the number of objects in this RFIDQueue

Returns:
int

---

rfidDriver

Class RFIDReaderWriter

```
java.lang.Object
  └ rfidDriver.RFIDReaderWriter
```

```
public class RFIDReaderWriter
extends java.lang.Object
```

Title: RFIDReaderWriter

Description: Java interface for Omron RFID reader/writer V720S_HMC73

Copyright: Copyright (c) 2004

Company: LogicaCMG

Author: Niels Heinis

Constructor Summary

RFIDReaderWriter(int comport) Constructor to create a new RFIDReaderWriter instance.

Method Summary

void	**addResponseListener**(RFIDResponseListener rl) By adding an RFIDResponseListener an RFIDResponseEvent can be received containing the response data
void	**close**() Close RFIDReaderWriter.
boolean	**isConnected**() Returns if a connection to the RFID reader/writer device is available
static void	**main**(java.lang.String[] args) Main method to use RFIDReaderWriter as a commandline, stand-alone application Stores all parameters in the Globals class
void	**removeResponseListener**(RFIDResponseListener rl) Remove RFIDResponseListener, no more RFIDResponseEvents will be received
void	**sendCommand**(int command, java.lang.String data) Send a command to the RFID Device

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

RFIDReaderWriter

```
public RFIDReaderWriter(int comport)
```

Constructor to create a new RFIDReaderWriter instance. Note that only one instance can connect to the hardware device at the same time

Parameters:

`comport` - int Defines to which port the RFID device is connected, default is 1

Method Detail

close

```
public void close()
```

Close RFIDReaderWriter. This method closes the connected RFIDDeviceCommunication and RFIDRVMsgPublisher object first.

---

sendCommand

```
public void sendCommand(int command,
                        java.lang.String data)
```

Send a command to the RFID Device

Parameters:

`command` - int This should be a command represented by one of the constant fields for a read or write mode

`data` - String In case of a writing command this String is written to the tag; an empty String is allowed

---

isConnected

```
public boolean isConnected()
```

Returns if a connection to the RFID reader/writer device is available

Returns:

boolean

---

addResponseListener

```
public void addResponseListener(RFIDResponseListener rl)
```

By adding an RFIDResponseListener an RFIDResponseEvent can be received containing the response data

Parameters:

`rl` - RFIDResponseListener

---

removeResponseListener

```
public void removeResponseListener(RFIDResponseListener rl)
```

Remove RFIDResponseListener, no more RFIDResponseEvents will be received

Parameters:

`rl` - RFIDResponseListener

---

main

```
public static void main(java.lang.String[] args)
```

Main method to use RFIDReaderWriter as a commandline, stand-alone application Stores all parameters in the Globals class

Parameters:

`args` - String[] Commandline arguments

---

rfidDriver

Class RFIDResponseEvent

```
java.lang.Object
  └java.util.EventObject
      └rfidDriver.RFIDResponseEvent
```

All Implemented Interfaces:
java.io.Serializable

```
public class RFIDResponseEvent
extends java.util.EventObject
```

Title: RFIDDriver - RFIDResponseEvent

Description: This event is fired when a response from the RFID device is received and decoded by RFIDDeviceCommunication

Copyright: Copyright (c) 2004

Company: LogicaCMG

Author: Niels Heinis

See Also:
Serialized Form

Field Summary

Fields inherited from class java.util.EventObject
source

Constructor Summary

RFIDResponseEvent(java.lang.Object source, java.util.Vector response) Constructor to create a new RFIDResponseEvent

Method Summary

java.util.Vector	**getResponse**() Returns the response Vector stored in this event

Methods inherited from class java.util.EventObject

getSource, toString

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

Constructor Detail

RFIDResponseEvent

```
public RFIDResponseEvent(java.lang.Object source,
                         java.util.Vector response)
```

Constructor to create a new RFIDResponseEvent

Parameters:

source - Object
response - Vector

Method Detail

getResponse

```
public java.util.Vector getResponse()
```

Returns the response Vector stored in this event

Returns:

Vector

rfidDriver

Class RFIDRVMsgPublisher

```
java.lang.Object
  └ rfidDriver.RFIDRVMsgPublisher
```

All Implemented Interfaces:
com.tibco.tibrv.TibrvMsgCallback, RFIDResponseListener

```
public class RFIDRVMsgPublisher
extends java.lang.Object
implements RFIDResponseListener, com.tibco.tibrv.TibrvMsgCallback
```

Title: RFIDDriver - RFIDRVMsgPublisher

Description: Class containing methods to publish TIBCO RendezVous messages

Copyright: Copyright (c) 2004

Company: LogicaCMG

Author: Niels Heinis

Constructor Summary

`RFIDRVMsgPublisher()`	Creates an RFIDRVMsgPublisher that listens to RFIDResponseEvents and publishes an RV message when such an event is received.

Method Summary

void	`Close()` Close connection to the RendezVous bus
void	`onMsg(com.tibco.tibrv.TibrvListener listener, com.tibco.tibrv.TibrvMsg msg)` This method must be implemented to implement the TibrvMsgCallback class.
void	`Publish(com.tibco.tibrv.TibrvMsg msg)` Publish TibrvMsg msg to the RendezVous bus
void	`responseReceived(RFIDResponseEvent r, boolean ok)` Listens for RFIDResponseEvents, turns them to a RendezVous message and publishes it

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

RFIDRVMsgPublisher

```
public RFIDRVMsgPublisher()
```

Creates an RFIDRVMsgPublisher that listens to RFIDResponseEvents and publishes an RV message when such an event is received.

Method Detail

Publish

```
public void Publish(com.tibco.tibrv.TibrvMsg msg)
```

Publish TibrvMsg msg to the RendezVous bus

Parameters:

msg - TibrvMsg

Close

```
public void Close()
```

Close connection to the RendezVous bus

---

responseReceived

```
public void responseReceived(RFIDResponseEvent r,
                             boolean ok)
```

Listens for RFIDResponseEvents, turns them to a RendezVous message and publishes it

Specified by:

responseReceived in interface RFIDResponseListener

Parameters:

r - RFIDResponseEvent

ok - boolean

---

onMsg

```
public void onMsg(com.tibco.tibrv.TibrvListener listener,
                  com.tibco.tibrv.TibrvMsg msg)
```

This method must be implemented to implement the TibrvMsgCallback class. It doesn't do

anything here.

Specified by:

`onMsg` in interface `com.tibco.tibrv.TibrvMsgCallback`

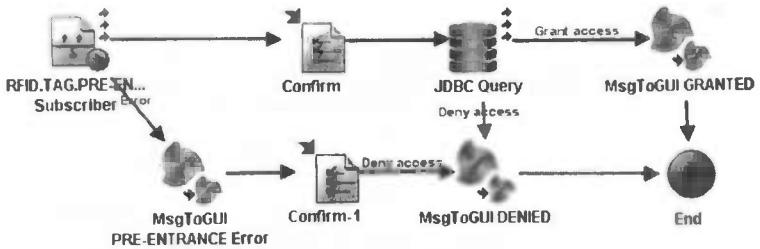
Parameters:

`listener` - TibrvListener

`msg` - TibrvMsg

TIBCO BusinessWorks processes

Access check at stations



- Name: Access check at stations
- Description:

> Perform an access check when person wants to enter a station. Based on his/her UID, it is checked if he/she is on the blacklist.

- Custom Icon File:
- Namespace:

End

- Name: End
- Description:

RFID.TAG.PRE-ENTRANCE Subscriber

Configuration

- Name: RFID.TAG.PRE-ENTRANCE Subscriber
- Description:
- Subject:
- Transport:
- XML Format: false
- Needs Output Validation: true
- Needs Output Filtration: true
- XML-Compliant Field Names: false
- Raw-RV-Object Mode: false

Misc

- Sequencing Key:
- Custom Id:

Confirm

- Name: Confirm
- Description:
- ConfirmEvent: RFID.TAG.KIOSK Subscriber

JDBC Query

Configuration

- Name: JDBC Query
- Description:

> Query database to see if person is on the blacklist. If so, deny access (also on error), else grant access which means opening the gate.

- JDBC Connection: /JDBC Connection.sharedjdbc
- SQL Statement:

```
SELECT count(*)
FROM BLACKLIST
WHERE TAGUID = ?
```

- Timeout(sec): 10
- Maximum Rows: 10
- Prepared Parameters: dm:node-kind()="element", dm:node-name()="Prepared_Param_DataType", dm:string-value()="UIDVARCHAR"

Advanced

- Override Transaction Behavior: false
- Use Nil: false
- Interpret Empty String as Null: false
- Process In Subsets: false

MsgToGUI PRE-ENTRANCE Error

- Name: MsgToGUI PRE-ENTRANCE Error
- Description:
- Process Name:
- Process Name Dynamic Override:
- Spawn: false

- Custom Icon File:

MsgToGUI DENIED

- Name: MsgToGUI DENIED
- Description:
- Process Name: /Processes/Logging/MsgToGUI.process
- Process Name Dynamic Override:
- Spawn: false
- Custom Icon File:

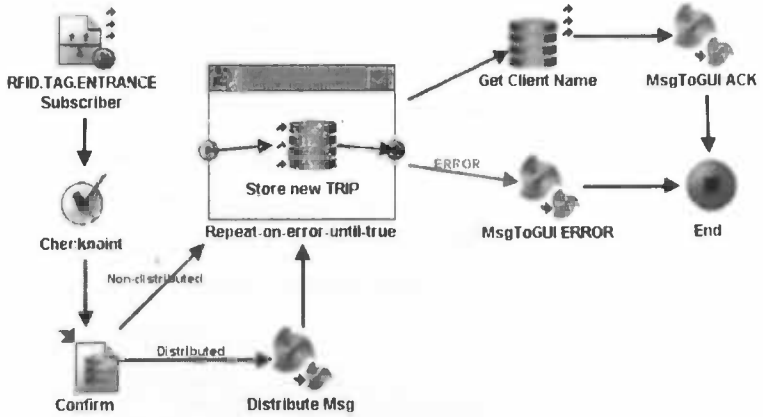
MsgToGUI GRANTED

- Name: MsgToGUI GRANTED
- Description:
- Process Name: /Processes/Logging/MsgToGUI.process
- Process Name Dynamic Override:
- Spawn: false
- Custom Icon File:

Confirm-1

- Name: Confirm-1
- Description:
- ConfirmEvent: RFID.TAG.KIOSK Subscriber

Register start of trip



- Name: Register start of trip
- Description: Register the start of a new trip for this UID.
- Custom Icon File:
- Namespace:

End

- Name: End
- Description:

RFID.TAG.ENTRANCE Subscriber

Configuration

- Name: RFID.TAG.ENTRANCE Subscriber
- Description: Rendezvous subscriber that listens for RV messages with the subject "RFID.TAG.ENTRANCE"
- Subject: RFID.TAG.ENTRANCE
- Transport: /RVConnection_Certified.rvtransport
- XML Format: false
- Needs Output Validation: false
- Needs Output Filtration: false
- XML-Compliant Field Names: false
- Raw-RV-Object Mode: false

Misc

- Sequencing Key:
- Custom Id:

MsgToGUI ERROR

- Name: MsgToGUI ERROR
- Description: Send message to the GUI.
- Process Name: /Processes/Logging/MsgToGUI.process
- Process Name Dynamic Override:
- Spawn: false
- Custom Icon File:

Confirm

- Name: Confirm
- Description:
- ConfirmEvent: RFID.TAG.KIOSK Subscriber

MsgToGUI ACK

- Name: MsgToGUI ACK
- Description: Send message to the GUI.
- Process Name: /Processes/Logging/MsgToGUI.process
- Process Name Dynamic Override:
- Spawn: false
- Custom Icon File:

Get Client Name

Configuration

- Name: Get Client Name
- Description: Retrieve client name from database for GUI purposes.
- JDBC Connection: /JDBC Connection.sharedjdbc
- SQL Statement:

```
SELECT SURNAME, FIRSTNAME
FROM CLIENT
WHERE TAGUID=?
```

- Timeout(sec): 10
- Maximum Rows: 1
- Prepared Parameters: dm:node-kind()="element", dm:node-name()="Prepared_Param_DataType", dm:string-value()="UIDVARCHAR"

Advanced

- Override Transaction Behavior: false
- Use Nil: false
- Interpret Empty String as Null: false
- Process In Subsets: false

Checkpoint

- Name: Checkpoint
- Description: Save the current job and all data belonging to it to disk to prevent loss of data.

Repeat-on-error-until-true

- Name: Repeat-on-error-until-true
- Description:
- Group Action: Repeat-On-Error-Until-True
- :

Store new TRIP

Configuration

- Name: Store new TRIP
- Description:
- JDBC Connection:
- SQL Statement:
- Timeout(sec): 10
- Prepared Parameters: dm:node-kind()="element", dm:node-name()="Prepared_Param_DataType", dm:string-value()=""

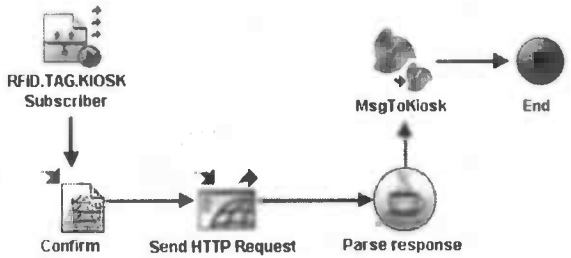
Advanced

- Override Transaction Behavior: false
- Interpret Empty String as Null: false
- Batch Update: false

Distribute Msg

- Name: Distribute Msg
- Description: When in distributed mode, replicate data to other systems as well.
- Process Name: /Processes/DistributeRV.process
- Process Name Dynamic Override:
- Spawn: false
- Custom Icon File:

Register end of trip



- Name: Register end of trip
- Description:

> When a person leaves a vehicle, he/she is scanned and a message with the subject "RFID.TAG.EXIT" is published. This process listens for that subject and processes such messages.

- Custom Icon File:
- Namespace:

End

- Name: End
- Description:

RFID.TAG.EXIT Subscriber

Configuration

- Name: RFID.TAG.EXIT Subscriber
- Description:
- Subject:
- Transport:
- XML Format: false
- Needs Output Validation: true
- Needs Output Filtration: true
- XML-Compliant Field Names: false
- Raw-RV-Object Mode: false

Misc

- Sequencing Key:
- Custom Id:

Log Error To GUI

- Name: Log Error To GUI
- Description: Send error message to GUI
- Process Name: /Processes/Logging/MsgToGUI.process
- Process Name Dynamic Override:
- Spawn: false
- Custom Icon File:

Confirm

- Name: Confirm
- Description:
- ConfirmEvent:

MsgToGUI OK

- Name: MsgToGUI OK
- Description:
- Process Name: /Processes/Logging/MsgToGUI.process
- Process Name Dynamic Override:
- Spawn: false
- Custom Icon File:

Publish TRIPCOMPLETED

- Name: Publish TRIPCOMPLETED
- Description:

> Publish RV message with subject "RFID.TRIPCOMPLETED". The message contains the UID of the person whose trip(s) just ended. The validation&pricing process listens to these messages.

- Subject: RFID.TRIPCOMPLETED
- Transport: /RVConnection_Certified.rvtransport
- Pre-register Listener:
- XML Format: false
- XML-Compliant Field Names: false

Get Client Name

Configuration

- Name: Get Client Name
- Description:
- JDBC Connection: /JDBC Connection.sharedjdbc
- SQL Statement:

```
SELECT SURNAME, FIRSTNAME
FROM CLIENT
WHERE TAGUID=?
```

- Timeout(sec): 10
- Maximum Rows: 1
- Prepared Parameters: dm:node-kind()="element", dm:node-name()="Prepared_Param_DataType", dm:string-value()="UIDVARCHAR"

Advanced

- Override Transaction Behavior: false
- Use Nil: false
- Interpret Empty String as Null: false
- Process In Subsets: false

tribute Msg

- Name: Distribute Msg
- Description: When in distributed mode, replicate data to other systems as well.
- Process Name: /Processes/DistributeRV.process
- Process Name Dynamic Override:
- Spawn: false
- Custom Icon File:

:ckpoint

- Name: Checkpoint
- Description: Save the current job and all data belonging to it to disk to prevent loss of data.

)eat-on-error-until-true

- Name: Repeat-on-error-until-true
- Description:
- Group Action: Repeat-On-Error-Until-True
- :

date TRIP

Configuration

- Name: Update TRIP
- Description:

```
Save the data from the scanned tag to the database. Note that, in this case, all open trips of this person are ended at this station.
```

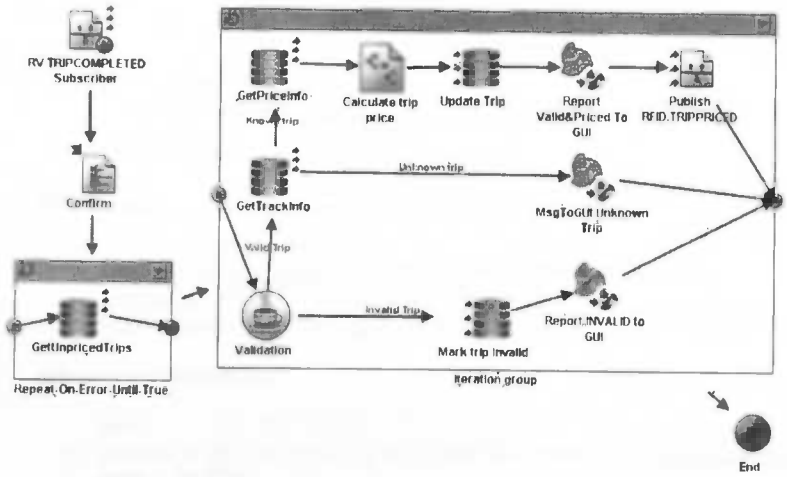
- JDBC Connection: /JDBC Connection.sharedjdbc
- SQL Statement: UPDATE TRIP SET ENDTIME=?,ENDPOS=? WHERE TAGUID=?
- Timeout(sec): 10
- Prepared Parameters:

```
dm:node-kind()="element", dm:node-name()="Prepared_Param_DataType", dm:string-value()="TimeStampVARCHARStationVARCHARUIDVARCHAR"
```

Advanced

- Override Transaction Behavior: false
- Interpret Empty String as Null: false
- Batch Update: false

Validation & Pricing



- Name: Validation&Pricing
- Description:

> After a trip is ended, an RFID.TRIPCOMPLETED message is published. This process listens for those messages and tries to validate and price the trip.

- Custom Icon File:
- Namespace:

End

- Name: End
- Description:

RV TRIPCOMPLETED Subscriber

Configuration

- Name: RV TRIPCOMPLETED Subscriber
- Description: Rendezvous subscriber that listens for RV messages with the subject "RFID.TRIPCOMPLETED"
- Subject: RFID.TRIPCOMPLETED
- Transport: /RVConnection_Certified.rvtransport
- XML Format: false
- Needs Output Validation: true
- Needs Output Filtration: true
- XML-Compliant Field Names: false
- Raw-RV-Object Mode: false

Misc

- Sequencing Key:
- Custom Id:

Iteration group

- Name: Iteration group
- Description: Iterate through all fetched trips and try to validate and price them.
- Group Action: Iterate
- :

Validation

Configuration

- Name: Validation
- Description: Custom java code to validate a trip.
- Alias Library:
- Input Parameters:

```
dm:node-kind()="element", dm:node-name()="inputData",
dm:string-value()="StartStationstringoptional
EndStationstringoptionalStartTimedateTimeoptional
EndTimedateTimeoptionalTransportTypestringoptional"
```

- Output Parameters: dm:node-kind()="element", dm:node-name()="outputData", dm:string-value()="validbooleanrequired"

Code

- Java Code:

```
package Processes.ValidationPricing;
import java.util.*;
import java.io.*;
public class ValidationPricingValidation{
/*** START SET/GET METHOD, DO NOT MODIFY **/
protected String StartStation = "";
protected String EndStation = "";
protected Date StartTime = null;
```

```
protected Date EndTime = null;
protected String TransportType = "";
protected boolean valid = false;
public String getStartStation() {
return StartStation;
}
public void setStartStation(String val) {
StartStation = val;
}
public String getEndStation() {
return EndStation;
}
public void setEndStation(String val) {
EndStation = val;
}
public Date getStartTime() {
return StartTime;
}
public void setStartTime(Date val) {
StartTime = val;
}
public Date getEndTime() {
return EndTime;
}
public void setEndTime(Date val) {
EndTime = val;
}
public String getTransportType() {
return TransportType;
}
public void setTransportType(String val) {
TransportType = val;
}
public boolean getvalid() {
return valid;
}
public void setvalid(boolean val) {
valid = val;
}
/**** END SET/GET METHOD, DO NOT MODIFY ***/
public ValidationPricingValidation() {
}
public void invoke() throws Exception {
/* Available Variables: DO NOT MODIFY
In : String StartStation
In : String EndStation
In : Date StartTime
In : Date EndTime
In : String TransportType
Out : boolean valid
* Available Variables: DO NOT MODIFY *****/
/* In this sample case, a trip is valid if all of the following criteria
hold:
- Departure and arrival stations are not the same
- Arrival time is later than departure time
- The trip was ended within 24 hours (86400000 ms) after the start
*/
valid = (!StartStation.equalsIgnoreCase(EndStation)) &
(StartTime.compareTo(EndTime) < 0) &
(EndTime.getTime()-StartTime.getTime() < 86400000);
}
}
```

GetTrackInfo

Configuration

- Name: GetTrackInfo
- Description: Get length of the trip from the database.
- JDBC Connection: /JDBC Connection.sharedjdbc
- SQL Statement:

```
SELECT KM
FROM TRACKINFO
WHERE (UPPER(STATION1)=UPPER(?) AND
UPPER(STATION2)=UPPER(?)) OR
(UPPER(STATION2)=UPPER(?) AND
UPPER(STATION1)=UPPER(?))
```

- Timeout(sec): 10
- Maximum Rows: 1
- Prepared Parameters:

```
dm:node-kind()="element", dm:node-
name()="Prepared_Param_DataType",
dm:stringvalue()="station1VARCHARstation2VARCHAR
station3VARCHARstation4VARCHAR"
```

Advanced

- Override Transaction Behavior: false
- Use Nil: false
- Interpret Empty String as Null: false
- Process In Subsets: false

Update Trip

Configuration

- Name: Update Trip
- Description: Store the calculated price to the database and mark the trip as priced.
- JDBC Connection: /JDBC Connection.sharedjdbc
- SQL Statement:

```
UPDATE TRIP
SET PRICED=1, PRICE=?
WHERE ID=?
```

- Timeout(sec): 10
- Prepared Parameters: dm:node-kind()="element", dm:node-name()="Prepared_Param_DataType", dm:string-value()="TRIPPRICENUMERICTRIPIDNUMERIC"

Advanced

- Override Transaction Behavior: false
- Interpret Empty String as Null: false
- Batch Update: false

Mark trip invalid

Configuration

- Name: Mark trip invalid
- Description: If the trip is invalid, set the "invalid" flag in the database.
- JDBC Connection: /JDBC Connection.sharedjdbc
- SQL Statement:

```
UPDATE TRIP
SET INVALID=1
WHERE ID=?
```

- Timeout(sec): 10
- Prepared Parameters: dm:node-kind()="element", dm:node-name()="Prepared_Param_DataType", dm:string-value()="TRIPIDNUMERIC"

Advanced

- Override Transaction Behavior: false
- Interpret Empty String as Null: false
- Batch Update: false

GetPriceInfo

Configuration

- Name: GetPriceInfo
- Description: Get price per kilometre for the type of transport from the database.
- JDBC Connection: /JDBC Connection.sharedjdbc
- SQL Statement:

```
SELECT KMPRICE
FROM PRICEINFO
WHERE UPPER(TRANSTYPE)=UPPER(?)
```

- Timeout(sec): 10
- Maximum Rows: 1
- Prepared Parameters: dm:node-kind()="element", dm:node-name()="Prepared_Param_DataType", dm:string-value()="TransportTypeVARCHAR"

Advanced

- Override Transaction Behavior: true
- Use Nil: false
- Interpret Empty String as Null: false
- Process In Subsets: false

Publish RFID.TRIPPRICED

- Name: Publish RFID.TRIPPRICED
- Description:

```
Publish an RFID.TRIPPRICED rendezvous message indicating that the
trip has been validated and priced successfully.
The Billing process listens for those messages.
```

- Subject: RFID.TRIPPRICED
- Transport: /RVConnection_Certified.rvtransport
- Pre-register Listener:
- XML Format: false
- XML-Compliant Field Names: false

Report Valid&Priced To GUI

- Name: Report Valid&Priced To GUI
- Description: Send message to GUI
- Process Name: /Processes/Logging/MsgToGUI.process

- Process Name Dynamic Override:
- Spawn: false
- Custom Icon File:

Report INVALID to GUI

- Name: Report INVALID to GUI
- Description: Send message to GUI
- Process Name: /Processes/Logging/MsgToGUI.process
- Process Name Dynamic Override:
- Spawn: false
- Custom Icon File:

MsgToGUI Unknown Trip

- Name: MsgToGUI Unknown Trip
- Description:
- Process Name: /Processes/Logging/MsgToGUI.process
- Process Name Dynamic Override:
- Spawn: false
- Custom Icon File:

Calculate trip price

- Name: Calculate trip price
- Description: number of kilometres * price per kilometre = trip price

Validation

Configuration

- Name: Validation
- Description: Custom java code to validate a trip.
- Alias Library:
- Input Parameters:

```
dm:node-kind()="element", dm:node-name()="inputData",
dm:string-value()="StartStationstringoptionalEndStationstringoptional
StartTimedateTimeoptionalEndTimedateTimeoptional
TransportTypestringoptional"
```

- Output Parameters: dm:node-kind()="element", dm:node-name()="outputData", dm:string-value()="validbooleanrequired"

Code

- Java Code:

```
package Processes.ValidationPricing;
import java.util.*;
import java.io.*;
public class ValidationPricingValidation{
/****** START SET/GET METHOD, DO NOT MODIFY *****/
protected String StartStation = "";
protected String EndStation = "";
protected Date StartTime = null;
protected Date EndTime = null;
protected String TransportType = "";
protected boolean valid = false;
public String getStartStation() {
return StartStation;
}
public void setStartStation(String val) {
StartStation = val;
}
public String getEndStation() {
return EndStation;
}
public void setEndStation(String val) {
EndStation = val;
}
public Date getStartTime() {
return StartTime;
}
public void setStartTime(Date val) {
StartTime = val;
}
public Date getEndTime() {
return EndTime;
}
public void setEndTime(Date val) {
EndTime = val;
}
public String getTransportType() {
return TransportType;
}
public void setTransportType(String val) {
TransportType = val;
}
public boolean getvalid() {
return valid;
}
public void setvalid(boolean val) {
valid = val;
}
/****** END SET/GET METHOD, DO NOT MODIFY *****/
```

```
public ValidationPricingValidation() {
}
public void invoke() throws Exception {
/* Available Variables: DO NOT MODIFY
In : String StartStation
In : String EndStation
In : Date StartTime
In : Date EndTime
In : String TransportType
Out : boolean valid
* Available Variables: DO NOT MODIFY *****/
/* In this sample case, a trip is valid if all of the following criteria hold:
- Departure and arrival stations are not the same
- Arrival time is later than departure time
- The trip was ended within 24 hours (86400000 ms) after the start
*/
valid = (!StartStation.equalsIgnoreCase(EndStation)) &
(StartTime.compareTo(EndTime) < 0) &
(EndTime.getTime()-StartTime.getTime() < 86400000);
}
}
```

GetTrackInfo

Configuration

- Name: GetTrackInfo
- Description: Get length of the trip from the database.
- JDBC Connection: /JDBC Connection.sharedjdbc
- SQL Statement:

```
SELECT KM
FROM TRACKINFO
WHERE (UPPER(STATION1)=UPPER(?) AND
UPPER(STATION2)=UPPER(?)) OR (UPPER(STATION2)=UPPER(?)
AND UPPER(STATION1)=UPPER(?))
```

- Timeout(sec): 10
- Maximum Rows: 1
- Prepared Parameters:

```
dm:node-kind()="element", dm:node-
name()="Prepared_Param_DataType",
dm:stringvalue()="station1VARCHARstation2VARCHAR
station3VARCHARstation4VARCHAR"
```

Advanced

- Override Transaction Behavior: false
- Use Nil: false
- Interpret Empty String as Null: false
- Process In Subsets: false

Update Trip

Configuration

- Name: Update Trip
- Description: Store the calculated price to the database and mark the trip as priced.
- JDBC Connection: /JDBC Connection.sharedjdbc
- SQL Statement:

```
UPDATE TRIP
SET PRICED=1, PRICE=?
WHERE ID=?
```

- Timeout(sec): 10
- Prepared Parameters: dm:node-kind()="element", dm:node-name()="Prepared_Param_DataType", dm:string-value()="TRIPPRICENUMERICTRIPIDNUMERIC"

Advanced

- Override Transaction Behavior: false
- Interpret Empty String as Null: false
- Batch Update: false

Mark trip invalid

Configuration

- Name: Mark trip invalid
- Description: If the trip is invalid, set the "invalid" flag in the database.
- JDBC Connection: /JDBC Connection.sharedjdbc
- SQL Statement:

```
UPDATE TRIP
SET INVALID=1
WHERE ID=?
```

- Timeout(sec): 10
- Prepared Parameters: dm:node-kind()="element", dm:node-name()="Prepared_Param_DataType", dm:string-value()="TRIPIDNUMERIC"

Advanced

- Override Transaction Behavior: false
- Interpret Empty String as Null: false
- Batch Update: false

GetPriceInfo

Configuration

- Name: GetPriceInfo
- Description: Get price per kilometre for the type of transport from the database.
- JDBC Connection: /JDBC Connection.sharedjdbc
- SQL Statement:

```
SELECT KMPRICE
FROM PRICEINFO
WHERE UPPER(TRANSTYPE)=UPPER(?)
```

- Timeout(sec): 10
- Maximum Rows: 1
- Prepared Parameters: dm:node-kind()="element", dm:node-name()="Prepared_Param_DataType", dm:string-value()="TransportTypeVARCHAR"

Advanced

- Override Transaction Behavior: true
- Use Nil: false
- Interpret Empty String as Null: false
- Process In Subsets: false

Publish RFID.TRIPPRICED

- Name: Publish RFID.TRIPPRICED
- Description:

> Publish an RFID.TRIPPRICED rendezvous message indicating that the trip has been validated and priced successfully. The Billing process listens for those messages.

- Subject: RFID.TRIPPRICED
- Transport: /RVConnection_Certified.rvtransport
- Pre-register Listener:
- XML Format: false
- XML-Compliant Field Names: false

Report Valid&Priced To GUI

- Name: Report Valid&Priced To GUI
- Description: Send message to GUI
- Process Name: /Processes/Logging/MsgToGUI.process
- Process Name Dynamic Override:
- Spawn: false
- Custom Icon File:

Report INVALID to GUI

- Name: Report INVALID to GUI
- Description: Send message to GUI
- Process Name: /Processes/Logging/MsgToGUI.process
- Process Name Dynamic Override:
- Spawn: false
- Custom Icon File:

MsgToGUI Unknown Trip

- Name: MsgToGUI Unknown Trip
- Description:
- Process Name: /Processes/Logging/MsgToGUI.process
- Process Name Dynamic Override:
- Spawn: false
- Custom Icon File:

Calculate trip price

- Name: Calculate trip price
- Description: number of kilometres * price per kilometre = trip price

Repeat-On-Error-Until-True

- Name: Repeat-On-Error-Until-True
- Description:
- Group Action: Repeat-On-Error-Until-True
- :

GetUnpricedTrips

Configuration

- Name: GetUnpricedTrips
- Description:
- JDBC Connection:
- SQL Statement:
- Timeout(sec): 10

- Maximum Rows: 100
- Prepared Parameters: dm:node-kind()="element", dm:node-name()="Prepared_Param_DataType", dm:string-value()=""

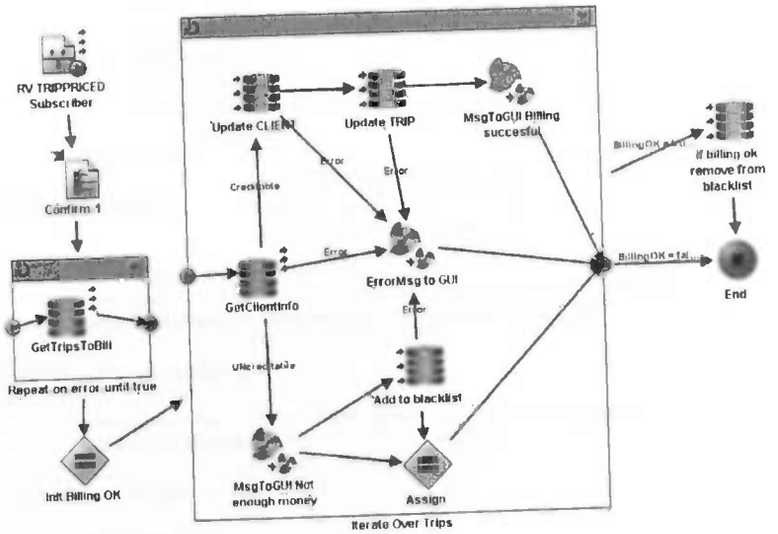
Advanced

- Override Transaction Behavior: false
- Use Nil: false
- Interpret Empty String as Null: false
- Process In Subsets: false

Confirm

- Name: Confirm
- Description:
- ConfirmEvent: RFID.TAG.KIOSK Subscriber

Billing



- Name: Billing
- Description:

> On receipt of RV message, which indicates a trip of this person was validated and priced, get all unpaid trips of this person and try to bill them. If billing fails, person is added to the blacklist.

- Custom Icon File:
- Namespace:

End

- Name: End
- Description:

RV TRIPPRICED Subscriber

Configuration

- Name: RV TRIPPRICED Subscriber
- Description:

- Subject:
- Transport:
- XML Format: false
- Needs Output Validation: true
- Needs Output Filtration: true
- XML-Compliant Field Names: false
- Raw-RV-Object Mode: false

Misc

- Sequencing Key:
- Custom Id:

Iterate Over Trips

- Name: Iterate Over Trips
- Description: For each un-billed trip, try to bill it.
- Group Action: Iterate
- :

Update CLIENT

Configuration

- Name: Update CLIENT
- Description: If client is creditable, take the price of current trip from his credit.
- JDBC Connection: /JDBC Connection.sharedjdbc
- SQL Statement:

```
UPDATE CLIENT
SET MONEY=?
WHERE TAGUID=?
```

- Timeout(sec): 10
- Prepared Parameters: dm:node-kind()="element", dm:node-name()="Prepared_Param_DataType", dm:string-value()="newMoneyNUMERICUIDVARCHAR"

Advanced

- Override Transaction Behavior: true
- Interpret Empty String as Null: false
- Batch Update: false

MsgToGUI Billing succesful

- Name: MsgToGUI Billing succesful
- Description:
- Process Name: /Processes/Logging/MsgToGUI.process
- Process Name Dynamic Override:
- Spawn: false
- Custom Icon File:

MsgToGUI Not enough money

- Name: MsgToGUI Not enough money
- Description: If client is not creditable, send message to GUI, leave trip unbilled.
- Process Name: /Processes/Logging/MsgToGUI.process
- Process Name Dynamic Override:
- Spawn: false
- Custom Icon File:

GetClientInfo

Configuration

- Name: GetClientInfo
- Description:
- JDBC Connection:
- SQL Statement:
- Timeout(sec): 10
- Maximum Rows: 100
- Prepared Parameters: dm:node-kind()="element", dm:node-name()="Prepared_Param_DataType", dm:string-value()=""

Advanced

- Override Transaction Behavior: false
- Use Nil: false
- Interpret Empty String as Null: false
- Process In Subsets: false

Update TRIP

Configuration

- Name: Update TRIP
- Description: After succesful payment, mark trip as billed.
- JDBC Connection: /JDBC Connection.sharedjdbc
- SQL Statement:

```
UPDATE TRIP
SET BILLED=1
WHERE ID=?
```

- Timeout(sec): 10
- Prepared Parameters: dm:node-kind()="element", dm:node-name()="Prepared_Param_DataType", dm:string-value()="TRIPIDNUMERIC"

Advanced

- Override Transaction Behavior: false
- Interpret Empty String as Null: false
- Batch Update: false

Add to blacklist

Configuration

- Name: Add to blacklist
- Description: If a person cannot pay for a trip, put him/her on the blacklist.
- JDBC Connection: /JDBC Connection.sharedjdbc
- SQL Statement: INSERT INTO BLACKLIST (TAGUID) VALUES (?)
- Timeout(sec): 10
- Prepared Parameters: dm:node-kind()="element", dm:node-name()="Prepared_Param_DataType", dm:string-value()="UIDVARCHAR"

Advanced

- Override Transaction Behavior: false
- Interpret Empty String as Null: false
- Batch Update: false

Assign

- Name: Assign
- Description:
- Process Variable to set: Schema0

ErrorMsg to GUI

- Name: ErrorMsg to GUI
- Description:
- Process Name: /Processes/Logging/MsgToGUI.process
- Process Name Dynamic Override:
- Spawn: false
- Custom Icon File:

if billing ok remove from blacklist

Configuration

- Name: if billing ok remove from blacklist
- Description: Only if billing of all trips was succesful, remove person from blacklist. Note: he is not necessarily on the blacklist.
- JDBC Connection: /JDBC Connection.sharedjdbc
- SQL Statement:

```
DELETE FROM BLACKLIST
WHERE TAGUID=?
```

- Timeout(sec): 10
- Prepared Parameters: dm:node-kind()="element", dm:node-name()="Prepared_Param_DataType", dm:string-value()="UIDVARCHAR"

Advanced

- Override Transaction Behavior: false
- Interpret Empty String as Null: false
- Batch Update: false

Init Billing OK

- Name: Init Billing OK
- Description: Set process variable to indicate that billing was succesful. Is set to false later on, if billing fails.
- Process Variable to set:

Repeat-on-error-until-true

- Name: Repeat-on-error-until-true
- Description: If database fails, suspend after 5 tries.
- Group Action: Repeat-On-Error-Until-True
- :

GetTripsToBill

Configuration

- Name: GetTripsToBill
- Description:
- JDBC Connection:
- SQL Statement:
- Timeout(sec): 10
- Maximum Rows: 100

- Prepared Parameters: dm:node-kind()="element", dm:node-name()="Prepared_Param_DataType", dm:string-value()=""

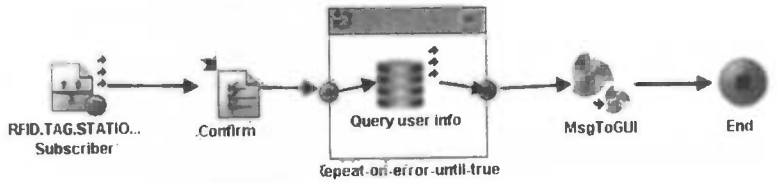
Advanced

- Override Transaction Behavior: false
- Use Nil: false
- Interpret Empty String as Null: false
- Process In Subsets: false

Confirm-1

- Name: Confirm-1
- Description:
- ConfirmEvent:

Station Exit



- Name: Station Exit
- Description: Dummy process to show in the GUI that a person left a station.
- Custom Icon File:
- Namespace:

End

- Name: End
- Description:

RFID.TAG.STATIONEXIT Subscriber

Configuration

- Name: RFID.TAG.STATIONEXIT Subscriber
- Description: Rendezvous subscriber that listens for RV messages with the subject "RFID.TAG.STATIONEXIT"
- Subject: RFID.TAG.STATIONEXIT
- Transport: /RVConnection_Certified.rvtransport
- XML Format: false
- Needs Output Validation: true
- Needs Output Filtration: true
- XML-Compliant Field Names: false
- Raw-RV-Object Mode: false

Misc

- Sequencing Key:
- Custom Id:

Confirm

- Name: Confirm
- Description:
- ConfirmEvent: RFID.TAG.KIOSK Subscriber

MsgToGUI

- Name: MsgToGUI
- Description: Send message to the GUI
- Process Name: /Processes/Logging/MsgToGUI.process
- Process Name Dynamic Override:
- Spawn: false
- Custom Icon File:

Repeat-on-error-until-true

- Name: Repeat-on-error-until-true
- Description:
- Group Action: Repeat-On-Error-Until-True
- :

Query user info

Configuration

- Name: Query user info
- Description: Get client's name from the database.
- JDBC Connection: /JDBC Connection.sharedjdbc
- SQL Statement:

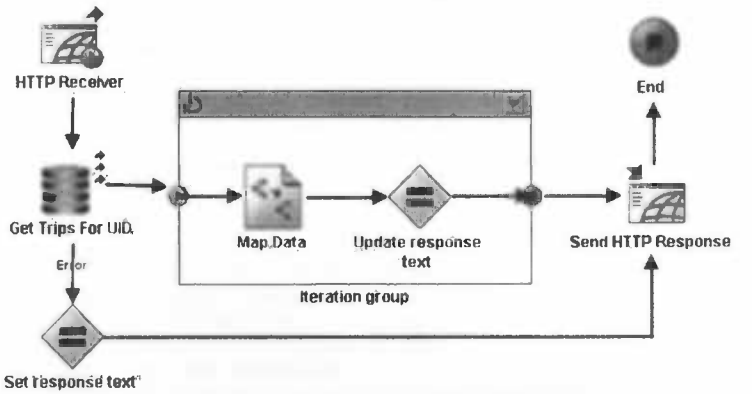
```
SELECT SURNAME,FIRSTNAME
FROM CLIENT
WHERE TAGUID=?
```

- Timeout(sec): 10
- Maximum Rows: 1
- Prepared Parameters: dm:node-kind()="element", dm:node-name()="Prepared_Param_DataType", dm:string-value()="TAGUIDVARCHAR"

Advanced

- Override Transaction Behavior: false
- Use Nil: false
- Interpret Empty String as Null: false
- Process In Subsets: false

TripInfoService



- Name: TripInfoService
- Description:

```
Business service that retrieves all trips for a certain UID from the database, puts them in a string, and returns the string in the HTTP response.
```

- Custom Icon File:
- Namespace:

End

- Name: End
- Description:

HTTP Receiver

Configuration

- Name: HTTP Receiver
- Description:
- HTTP Connection: /TripInfoService HTTP Connection.sharedhttp
- Output Style: String
- Parse Post Method Data: true
- Parameters: dm:node-kind()="element", dm:node-name()="customField", dm:string-value()="UIDstringoptional"
- HTTP Authentication: false
- Expose Security Context: false

- Default Encoding: ISO8859_1

Advanced

- Write to File: false

Misc

- Sequencing Key:
- Custom Id:

Get Trips For UID

Configuration

- Name: Get Trips For UID
- Description: Query database for all trips belonging to this UID.
- JDBC Connection: /JDBC Connection.sharedjdbc
- SQL Statement:

```
SELECT *
FROM TRIP
WHERE TAGUID=?
```

- Timeout(sec): 10
- Maximum Rows: 100
- Prepared Parameters: dm:node-kind()="element", dm:node-name()="Prepared_Param_DataType", dm:string-value()="UIDVARCHAR"

Advanced

- Override Transaction Behavior: false
- Use Nil: false
- Interpret Empty String as Null: false
- Process In Subsets: false

Send HTTP Response

- Name: Send HTTP Response
- Description: Respond to the HTTP request by sending the response text.
- Reply For: HTTP Receiver
- Close Connection: true

Iteration group

- Name: Iteration group
- Description: Iterate through all records and concatenate the data.
- Group Action: Iterate
- :

Update response text

- Name: Update response text
- Description:
- Process Variable to set: Schema0

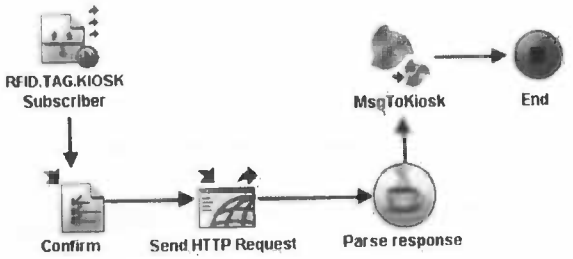
Map Data

- Name: Map Data
- Description:

Set response text

- Name: Set response text
- Description:
- Process Variable to set: Schema0

Kiosk info point



- Name: Kiosk info point
- Description:

> Process that listens for RFID.TAG.KIOSK messages, retrieves all known trips in the database for the UID of the read tag and sends them to the kiosk's GUI.

- Custom Icon File:
- Namespace:

End

- Name: End
- Description:

RFID.TAG.KIOSK Subscriber

Configuration

- Name: RFID.TAG.KIOSK Subscriber
- Description: Rendezvous subscriber that listens for RV messages with the subject "RFID.TAG.KIOSK"
- Subject: RFID.TAG.KIOSK
- Transport: /RVConnection_Certified.rvtransport
- XML Format: false
- Needs Output Validation: false
- Needs Output Filtration: true
- XML-Compliant Field Names: false
- Raw-RV-Object Mode: false

Misc

- Sequencing Key:
- Custom Id:

Confirm

- Name: Confirm
- Description:
- ConfirmEvent: RFID.TAG.KIOSK Subscriber

Send HTTP Request

- Name: Send HTTP Request
- Description: Send an HTTP request to the TripInfoService Business Service to retrieve trip information for the scanned UID
- Host: %%TripInfoBusinessServiceHost%%
- Port: 8888
- Use Proxy Setting:
- Accept Redirects: false
- Parameters: dm:node-kind()="element", dm:node-name()="customField", dm:string-value()="UIDstringoptional"
- HTTP Authentication: false
- SSL: dm:node-kind()="element", dm:node-name()="{http://www.tibco.com/xmlns/aemeta/services/2002}ssl", dm:string-value()=""

MsgToKiosk

- Name: MsgToKiosk
- Description: Send RV message to GUI for display
- Process Name: /Processes/Logging/MsgToGUI.process
- Process Name Dynamic Override:
- Spawn: false
- Custom Icon File:

Parse response

Configuration

- Name: Parse response
- Description: Custom java code to parse http response
- Alias Library:
- Input Parameters: dm:node-kind()="element", dm:node-name()="inputData", dm:string-value()="allTripsstringoptional"
- Output Parameters: dm:node-kind()="element", dm:node-name()="outputData", dm:string-value()="polishedTripsStringstringoptional"

Code

- Java Code:

```
package Processes.Kioskinfopoint;
import java.util.*;
import java.io.*;
public class KioskinfopointParseresponse{
/****** START SET/GET METHOD, DO NOT MODIFY
*****/
protected String allTrips = "";
protected String polishedTripsString = "";
public String getallTrips() {
return allTrips;
}
public void setallTrips(String val) {
allTrips = val;
}
public String getpolishedTripsString() {
return polishedTripsString;
}
public void setpolishedTripsString(String val) {
polishedTripsString = val;
}
/****** END SET/GET METHOD, DO NOT MODIFY
*****/
public KioskinfopointParseresponse() {
}
public void invoke() throws Exception {
/* Available Variables: DO NOT MODIFY
In : String allTrips
Out : String polishedTripsString
* Available Variables: DO NOT MODIFY *****/
/* The response to the HTTP request is a list of all trips that
re in the database for the current UID.
The following code parses the response and returns a
polished string that is ready for display.
*/
String[] entries = allTrips.split("&&");
for (int i=0; i<entries.length; i++) {
if (entries[i].length() > 0) {
String[] temp = entries[i].split(",");
String ps = "From : "+temp[0]+"\nTo : "+temp[1]+"\nBy :
"+temp[2]+"\nOn : "+temp[3]+"\nPrice:
"+temp[4]+"\nPayed: ";
if (temp[5].equals("payed")) {
ps += "yes\n\n";
} else {
ps += "no\n\n";
}
polishedTripsString += ps;
}
}}
```

MsgToGUI

- Name: MsgToGUI
- Description: Process used by most other processes to send an RV message to the GUI application.
- Custom Icon File:
- Namespace:

Start

- Name: Start
- Description:

End

- Name: End
- Description:

Publish RFID.GUI message

- Name: Publish RFID.GUI message
- Description: Send RV message to GUI application. Subject starts with RFID.GUI
- Subject:
- Transport: /RVConnection_Reliable.rvtransport
- Pre-register Listener:
- XML Format: false
- XML-Compliant Field Names: false