University of Twente
Electrical Engineering, Mathematics and Computer Science
Enschede, The Netherlands

and

Imtech ICT Technical Systems
Amersfoort, The Netherlands

Master's Thesis

SQLbusRT: Real time data distribution and storage

by

Bram Smulders

Supervisors:
dr. ir. Djoerd Hiemstra (University of Twente)
ir. Sander Evers (University of Twente)
ir. Hans Cremer (Imtech ICT Technical Systems)
ing. Carl Wolff (Imtech ICT Technical Systems)
ir. Evert van de Waal (Imtech ICT Technical Systems)

Amersfoort, 2007

Either write something worth reading or do something worth writing.

- Benjamin Franklin

To Cárin

Contents

Preface

This report is the written result of my final project at the University of Twente, executed externally at Imtech ICT Technical Systems in Amersfoort. With this project I have finished all the requirements for my academic course in Computer Science.

The project running time was from May 15th, 2006 till February 23rd, 2007.

It has been a very educative time for me. On of the most difficult problems I faced in this project was to find the right balance between research and practical activities. On one hand, the major goal of the project is to fulfill the requirements of the Master thesis. On the other hand, a project can not be fully successful when the wishes of the company are not fulfilled. I hope I have succeeded in finding this right balance, but this is up to the reader to decide.

The original assignment for this project, which was meant for two students, can be found in appendix A. It contains both the original assignment in Dutch, and a translated version. The assignment eventually had to be reduced to fit in one MSc project. The scope of this project is fully described in chapter 1.

Acknowlegdements

Several people have helped me to succeed in this project. Some have given me support with respect to the project contents, others have given me the support on a personal level. I would like to thank the following people for their help:

First of all, I would like to thank my supervisors at the University and at Imtech ICT. Djoerd, you've always been flexible, and you've motivated me when I was in doubt about certain approaches. Sander, you've asked the critical questions that I needed, making me look at my work critically. Evert, thanks to your technical input I've been able to write the prototypes in C++, a language which was still new to me at the time I started this project. Carl, your enthusiasm for this project was very infectious! You've also taught me a lot of new cool stuff on Linux, which will definitely be useful in future projects. Hans, thank you for being a great guide throughout the entire project. You knew how to get me

motivated again at times I missed the drive.

Secondly, I would like to thank some people which I haven't met in person, but who have given technical input by email or in online chats, which was very useful to me. They are: Jay Pipes (community manager at MySQL), Petr Smolik (author of ORTE) and Seppo Sierla(author of [21]).

My parents have been of great help. They have given me support not just during this project, but throughout all the past years that I've been studying computer science.

And last but definitely not least, I would like to thank Càrin. Despite the fact she is going through some busy times herself, she has been wonderful at encouraging me. Càrin, thank you for the wonderful days we spent together in Spain, and thanks for the great encouraging phone calls.

Amersfoort, February 6th, 2007

Bram Smulders

Abstract

The techniques used nowadays in a lot of research centers processing lots of sensor data, for instance wind tunnels, are often based on traditional relational databases. In some cases, no databases are used at all. No care is taken to deliver data under real time constraints. This does not have a negative effect in case data is merely stored for analysis after the measurement process has completed, but it can have disastrous effects when the data is needed for control of critical elements in the measurement process itself.

This report discusses a possible solution to real time delivery storage of data. It does not focus on sensor data. Instead, it should be flexible enough to suite any situation in which it is desirable to distribute data under real time constraints.

The newly created solution, carrying the name "SQLbusRT", is based on the blackboard architecture pattern, which will be explained in this report. A comparison is made on how the architecture of the new solution matches with the blackboard architecture. The choice for the blackboard pattern is mainly for its flexibility in the addition and removal of components to and from the system. System components will be able to work on a shared storage. This shared storage is called the blackboard, giving the name to the architecture pattern.

A prototype is developed by combining readily available open source products and creating new interfaces. The open source products which are used in this project are MySQL and ORTE. MySQL is a database management system which is known for its high performance and is used on a large scale worldwide. ORTE is an implementation of the RTPS protocol, which serves as a data communication channel over Ethernet, using a publish subscribe mechanism. An explanation of ORTE and the publish subscribe mechanism is given in this report.

This report discusses some tests which were executed to predict the *performance*, *reliability* and *scalability* of SQLbusRT in a simple setup. This set of tests can be extended in future research when SQLbusRT matures.

The report concludes with the answer to the question whether implementing a real time data distribution and storage solution is possible using the above mentioned components, and points out new fields of research.

Chapter 1

Introduction

This introduction gives the context and the goal of the SQLbusRT project. The problem description describes the research questions to be answered in order to reach the goal of the project. An approach is given on how this is to be done.

For the convenience of the reader, a report structure is added to this introduction, giving a short description of all the chapters in this report.

1.1 Context

Imtech N.V. is a European technical service provider in the field of information and communication technology as well as electrical and mechanical engineering. Imtech ICT is one of the six divisions of Imtech N.V., and focuses on the information and communication technology.

Imtech ICT Technical Systems is one of the nine subsidiaries of Imtech ICT. It develops technical and embedded software. The specialization of Imtech ICT Technical Systems is the development of distributed and real time systems.

The amount of data being processed in real time systems is growing vastly. An example of a project which Imtech ICT is currently working on is the automation of control systems in wind tunnels. The control systems in these wind tunnels gather all data from the sensors in the wind tunnel, and control the measurement process.

The techniques used nowadays in a lot of research centers like the wind tunnels, are often based on traditional relational databases. In some cases there is not even a database present at all. No care is taken to deliver data under real time constraints. This does not have a negative effect in case data is merely stored for analysis after the measurement process has completed, but it can have disastrous effects when the data is needed for control of critical elements in the measurement process itself.

Solutions for real time distribution of data exist, which offer flexibility for adding and removing data producing and consuming system components easily. Drawback of these solutions is that they often lack the possibility to keep history data available to the components, without the need for the components for keeping this data locally.

1.2 Project goal

The goal of the SQLbusRT project is to create a flexible solution for distributing data which is collected during a measurement process, meeting real time constraints. The solution should provide the possibility to store the data in a common storage place, which can later be retrieved by data consuming components in the system. The data should even be available to components if they were connected after this data was produced.

The solution should at least fit the context as described above. The intention however is to make the solution as general as possible, making it suitable to fit any situation where storage and real time distribution of data is desired, still offering the flexibility of easily adding and removing data producing and consuming components to the system.

Ideally, the solution should make use of "of-the-shelve" open source components, and result in an open source product.

Once a solution is found for data distribution and storage, it is desired to know how systems based on this solution will perform, before the system is fully implemented.

1.3 Problem description

Within the scope of this Master's thesis, the feasibility of developing such a solution is to be tested. Together with the wish for predicting its behavior, this leads to the following questions which are to be answered in this MSc project:

1. Is it possible to create a middleware solution offering storage and real time distribution of data, with the flexibility of easily adding and removing components to and from the system?

2. Can we predict *performance*, *reliability* and *scalability* of a system based on this middleware solution, before full implementation of that system?

1.4 Approach

In this project a prototype is developed of a middleware solution which contains the necessary elements to support real time delivery of data. The *System un-*

der Development carries the name SQLbusRT, where SQL stands for Structured Query Language (A widely used language to query databases) and RT stands for Real Time. "Bus" is included in the name since the SQLbusRT will have a bus structure for data communication.

The SQLbusRT prototype is built by combining, extending and, if necessary, modifying readily available open source components. It is used to experiment, form ideas and gain knowledge about the different components which are used, as well as to verify the chosen architecture for SQLbusRT.

Apart from the prototype, test programs are written which use the same components as SQLbusRT. The test programs are used in different setups to find how predictable the performance, reliability and scalability of systems based on these components are.

1.5 Report structure

The main part of this report is divided into six chapters. This following section gives a short description of the contents of these chapters:

Chapter 1: Introduction is this chapter, giving introductory information on the project, like the context in which the project is to be seen, the goal of the overall project, the problem description, the approach on how to solve the problem, and this report structure.

Chapter 2: Definitions explains some of the terms which are used in this report, which might have different meanings depending on the context. It discusses the terms "real time", as well as the metrics "performance", "reliability" and "scalability". Since finding values for these metrics is the major goal of the tests executed in this report, this chapter explains how they can be made quantifiable.

Chapter 3: SQLbusRT describes SQLbusRT in detail. SQLbusRT is the system under development which is to serve the goal of this project. It focuses on the architectural design. SQLbusRT uses "of the shelve components", which will be explained in more detail. It also identifies already existing projects with similar goals to those of SQLbusRT and makes a short comparison.

Chapter 4: Testing SQLbusRT describes the tests that have been performed during this MSc project. The results of these tests are discussed to draw conclusions about the performance, reliability and scalability of SQLbusRT.

Chapter 5: Conclusions and Recommendations gives conclusions to the findings in this project and it gives a list of recommendations for further research and development.

Chapter 2

Definitions

This chapter explains the meaning of terms which play an important role in this project. It focuses on the terminology which can have different meanings in different contexts.

2.1 Real Time

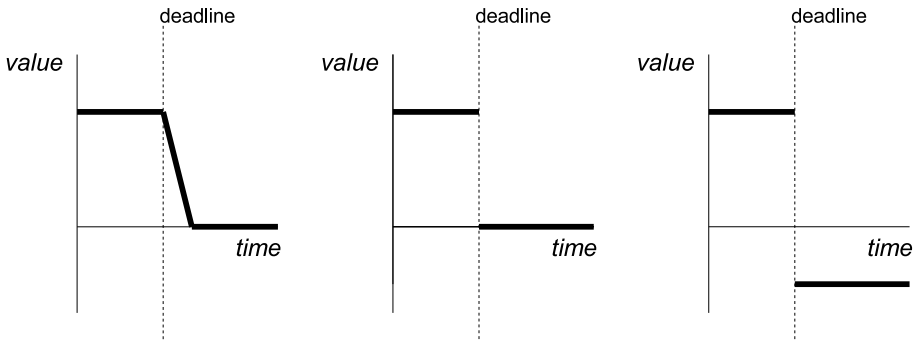
There are various definitions of real time, depending on the context. For instance, in case of video and audio editing, it can mean "at normal speed", so no fast forward or slow motion. It can also mean "live" in this context, meaning there is no noticeable delay.

In case of real time (database) systems, the term "real time" does not imply there can be no delay. Transactions might exist which take a minute, a day or even a year without violating the real time constraint. It's finishing the transaction before the specified deadline which makes a transaction real time or not. Two types of real time transactions are distinguished [17]:

Hard deadline transactions are those which may result in a catastrophe if the deadline is missed. One can say that a large negative value is imparted to the system once a hard deadline is missed.

Soft deadline transactions have some value even after their deadlines. Typically, the value drops to zero at a certain point past the deadline. If this point is the same as the deadline, we get *firm* deadline transactions, which impart no value to the system once their deadlines expire.

In addition to the explanation of hard deadline transactions the following can be said: "The right response after the deadline is just as bad as the wrong response in time". This might give some insight in how severe deadline occurrences are.



(a) Soft deadline (not firm) (b) Soft deadline (firm) (c) Hard deadline

Figure 2.1: Different deadline types and their transaction values

Figure 2.1 shows the transaction values for the different real time contraints. The soft deadline is shown twice: once as a non firm deadline (2.1(a)), and once as a firm deadline (2.1(b)).

2.2 Metrics used in this project

One of the goals of this project is to make a prediction on the *Performance*, *Reliability* and *Scalability* of systems based on SQLbusRT. These predictions can help system designers to decide whether SQLbusRT is the right choice as a middleware solution for their system.

Performance, reliability and scalability are not quantifiable metrics as such. The terms have different meanings in different situations. They can also be interrelated. For instance, reliability can be seen as a part of performance. Scalability on its turn might, depending on the chosen definition, say something about performance changes on changing load or hardware.

To discuss the performance, reliability and scalability of systems based on SQLbusRT, the terms first have to be made quantifiable. A discussion on how to make metrics quantifiable can be found in [7]. This section discusses different interpretations of the terms in general and chooses quantifiable scales of measure which are used for the discussion on performance, reliability and scalability.

Measurements are executed to find values for the metrics as chosen in this chapter. The description of these measurements and the results can be found in chapter 4.

2.2.1 Performance

In traditional RDBMS's, important performance metrics are throughput (records read or written in a fixed time interval) and response time. [24] states that in real time systems, finishing tasks before the deadline is most crucial. The perfor-

mance of real time databases could in that case be expressed as the percentage of transactions that succeeds before their deadline.

The percentage of succeeding transactions can indeed be seen as a performance measure, but it is a reliability measure as well. Section 2.2.2 will discuss how the reliability is measured using the percentage of succeeding transactions.

In [5] which describes a performance test on ORTE, the response times are taken as performance measure. A full roundtrip time is measured from a sending application to a receiving application, back to the initial sender again.

Also in this project, the roundtrip time is taken as the primary performance measurement. It is merely a time measurement which cannot be expressed in a formula, in contrast with the reliability, as described in the following section.

The approach in measuring the roundtrip time differs from one setup to another. More on the difficulties faced while measuring times, and how the roundtrip times are measured in the different setups in project can be found in chapter 4.

2.2.2 Reliability

The definition of the term *reliability* of software, found in different articles and books, are not as diverse as for terms like *real time*. Some examples:

> "The reliability of a software system expresses the probability that the system will be able to perform as expected when requested. This probability is interpreted in the context of an expected pattern of use and a specified time frame."[13]

> "Software reliability is one of the important parameters of software quality and system dependability. It is defines as the probability of failure-free software operation in a specified environment for a specified period of time."[12]

According to [9], a widely used approach to represent the reliability of a software product is by means of the *failure rate*. When N systems are observed and a total of F failures occur in time period T, then the failure rate λ is given as:

$$\lambda = \frac{F}{N \times T} \tag{2.1}$$

In this way of representing the reliability, assumptions are made which do not always hold when taking the system in practice. For instance, all failures are assumed to have the same impact, and all failures are assumed to be recorded.

An important matter is to classify the failures, since not all failures have the same impact. A malfunction of a single sensor doesn't necessarily give problems, while a malfunction of the database cluster likely makes the system inoperable.

The failure rate is therefore not a suitable metric to quantify the reliability in the case of this project.

In [10] which describes the performance measurement of a real time database, the primary performance metric used in the experiments is the *miss ratio*. The lower the miss ratio, the better the performance. Take N_{miss} as the number of transactions missing their deadline, and $N_{succeed}$ as the number of transactions that succeeded before their deadline. The miss ratio can now be expressed as:

$$miss\ ratio = \frac{N_{miss}}{N_{miss} + N_{succeed}} \tag{2.2}$$

In some occasions it is desirable to reject transactions, which gives other transactions with a higher priority to finish before their deadline. In this case, equation 2.2 is rewritten, adding $N_{rejected}$ as the number of rejected transactions. This gives the following equation:

$$miss\ ratio = \frac{N_{miss} + N_{rejected}}{N_{miss} + N_{rejected} + N_{succeed}} \tag{2.3}$$

The current stage of SQLbusRT and the test programs used in chapter 4 do not yet support rejection of transactions. This gives $N_{rejected} = 0$ in all cases, making equation 2.2 and 2.3 resulting in equal outcomes. Also, as discussed in section 2.2.1, it is desirable to mention the percentage of succeeding transactions. For this, the miss ratio formula is rewritten to the formula which is used as the reliability metric in this report:

$$\mathcal{R} = \frac{N_{succeed}}{N_{miss} + N_{succeed}} \times 100\% \tag{2.4}$$

2.2.3 Scalability

According to [8] no "useful, rigorous definition" of the term *scalability* currently exists. He encourages the technical community to "either rigorously define scalability or stop using it to describe systems".

Even though, if the assumption is made that a useful definition for scalability does exist, it is at least important to realize that different types of scalability are distinguished. The general definition that [3] gives for scalability is as follows:

> "Scalability is a desirable property of a system, a network or a process, which indicates its ability to either handle growing amounts of work in a graceful manner, or to be readily enlarged."

This definition is very general, but [3] also distinguishes different types of scalability, for instance: *load scalability*, *space scalability*, *space-time scalability* and *structural scalability*.

Even when these different types of scalability are distinguished, it has not made scalability quantifiable as such. The goal is to find a definition which is at least suitable for this project, and to make it quantifiable.

For SQLbusRT it is interesting to know how the SQLbusRT "scales" with increasing message size and with a growing number of components connected to the middleware. Ideally, this scaling shows linear behavior. Linear behavior provides makes the behavior more predictable.

Again, just as with performance, this metric cannot be put into a formula. However, with the creation of graphs, it is possible to discuss the linearity of the scalability. In these graphs, the change in roundtrip times is displayed against respectively the message size and the number of components.

Furthermore, the change in roundtrip times in relation to the hardware and varying network load is shown. The results for the scalability are, just as for the performance and reliability, gathered and discussed in chapter 4.

Chapter 3

SQLbusRT

This chapter describes SQLbusRT, the system under development which is meant to serve as a real time data distribution and storage middleware solution.

By the time of this writing, SQLbusRT is still in an alpha development stage. A prototype has been implemented. The description of SQLbusRT in this chapter contains ideas that are the result of experimenting with the prototype.

3.1 Goal of SQLbusRT

Chapter 1 describes the overall project goal, mentioning a real-time data distribution system provisioned with a database. The real time data distribution system with a database has received a name in this project: "SQLbusRT".

One of the goals of the project is to be able to make predictions about performance, reliability and scalability without fully implementing the system based on SQLbusRT. The goal of SQLbusRT should therefore be seen in two different scopes: the scope of the project, and the scope of a practical environment in which it will be fully functional. In the scope of the project, SQLbusRT is taken to the level of a prototype. The prototype is meant to be sufficient to draw conclusions about the performance, reliability and scalability of a fully functional SQLbusRT based system.

The remainder of this chapter describes SQLbusRT as how it is intended to be once fully functional, for as far as it is known and decided upon by now.

3.1.1 SQLbusRT in practice

SQLbusRT is not meant as a system that can simply be plugged in to start collecting and distributing data. It will serve as a middleware solution instead, to which data consuming and producing components can be connected.

Application programmers can create their own applications which will act as *Data consumers*, *Data producers*, or both. With the SQLbusRT API, the programmer can setup the data exchange between these applications.

The model used for exchanging the data is topic based *publish subscribe*. Data producing applications publish the data on the bus, and data consuming applications can subscribe to the data they are interested in. More about this publish subscribe model can be found in section 3.2.2.

An important addition to the publish subscribe communication which SQLbusRT offers is the ability to store and retrieve data to and from a database. The SQLbusRT *insertion interface* can be configured by the application programmer to subscribe to data which needs to be stored. The *selection interface* is responsible for handling requests by the data consumers, making sure the right data gets pulled from the database and published on the bus.

The data storage which is provided by SQLbusRT is not primarily intended for long term use, though the decision lies with the application programmer. The goal of this storage is mainly to have history data available in the data collection process. The application programmer can configure the insertion interface to store history data of choice.

A more thorough description of the SQLbusRT components is given in section 3.2.

3.2 Architecture

As mentioned before, SQLbusRT is to serve as a middleware solution. Its architecture is inspired by the blackboard architecture pattern. SQLbusRT is mainly an RTPS protocol implementation combined with a MySQL database, connected with each other and external components by means of newly created interfaces.

This section describes the blackboard architecture and the components in SQLbusRT, and how they form SQLbusRT together with the interfaces.

3.2.1 The blackboard architecture pattern

The philosophy behind the blackboard architecture is a collection of independent programs that work cooperatively on a common data structure. It is useful in situations where no deterministic solution to a task is yet known.

The independent programs are all specialized in a subtask. They can access the common data structure to fetch the data they need for executing their subtask. After completing a subtask, the program can post the results back on the blackboard so another knowledge source can fetch this data, process it, and so on. This process is repeated until the solution to the overall problem has been reached.

As described in [4], a blackboard architecture consists of three different types of components, which are:

Blackboard This is the common data structure. It holds all the data which is shared among the different programs. This data can be a partial solution for the task which the blackboard system has been set up for.

Knowledge source A knowledge source is an independent program which is specialized in a certain task. Executing this task should lead to a partial solution for the problem for which the blackboard system was set up.

Generally, a blackboard consists of multiple knowledge sources. The goal is to let these knowledge sources work together to the solution for the overall problem.

A knowledge source generally consists of two parts, the *condition part* and the *action part*. The condition part evaluations the current state of the solution process, and decides whether it can make a contribution to come closer to the overall solution. The action part is where this contribution is executed. It fetches the data from the blackboard and processes it.

Control The control decides which knowledge source can access the blackboard to fetch data and write (partial) solutions.

This is perhaps the most difficult part of a blackboard system to design. Since there's no deterministic approach known to solve a problem, it is quite a challenge to design a good control strategy.

Figure 3.1 shows a graphical representation of the blackboard architecture, as described in [4].

One of the benefits of the blackboard pattern is the flexibility for connecting and disconnecting independent programs. This is also known as *Loose coupling*. Programs do not have to be aware of each other. There's no direct connection necessary between the processes to exchange data, since the blackboard functions as an intermediate buffer for data. Having this loose coupling, it is very easy to add and remove applications to the system.

SQLbusRT in relation to the Blackboard architecture pattern

SQLbusRT is intended to fit in many different situations in which it is desirable to exchange data and to have a predefined amount of history data available. As stated before, the blackboard architecture was designed for domains where no deterministic solution to a problem is yet known. This is not necessarily the domain which SQLbusRT is intended for, but the benefits of the blackboard pattern have served as an inspiration for creating the SQLbusRT architecture.

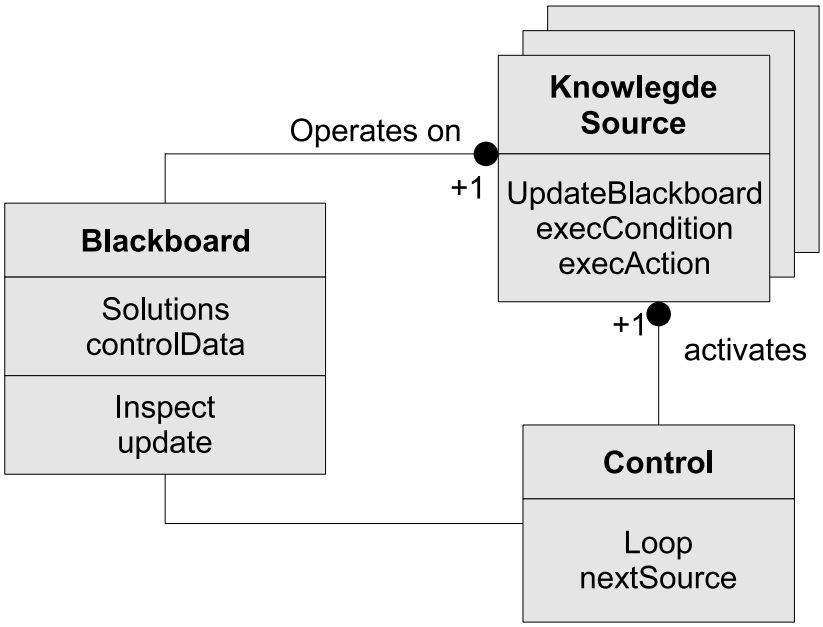


Figure 3.1: A graphical representation of the blackboard pattern

SQLbusRT can be seen as a generalized implementation of the blackboard architecture. The mapping between SQLbusRT and the blackboard architecture is as follows:

Blackboard The actual blackboard component, as it is present in the blackboard architecture, has been replaced in SQLbusRT with a storage space which can hold any kind of data. It is up to the application programmer to decide which data should be kept in the database, and for how long. This set of data does not necessarily have to be a (partial) solution to a problem that has to be solved with a couple of knowledge sources, as in the blackboard pattern.

Knowledge source The knowledge sources which are present in the blackboard pattern do not really exist in SQLbusRT as such. There are however elements in the SQLbusRT architecture which are very similar. These are the *data producers* and the *data consumers*, which are independent programs, just like the knowledge sources. There is no such restriction as with a knowledge source that a program has to post its results back to the blackboard. This decision lies again with the application programmer.

Another similarity between the knowledge sources in the blackboard architecture and the data producers and consumers in SQLbusRT is the presence of a *condition part*, as described earlier. A knowledge source contains this part to decide whether the blackboard contains data which it can process.

The data consumers have a similar construction. By *subscribing* to data of a certain topic, the data gets delivered to the data consumer once this data becomes available. This process is described in detail in section 3.2.2.

Control The control in SQLbusRT does not function as the control in the blackboard pattern. Where the control in the blackboard pattern makes the knowledge sources take turns on accessing the common data structure, the control in SQLbusRT lets the data producers and consumers access the data structure simultaneously. The data producers and consumers communicate with the data structure by means of a publish subscribe mechanism, which serves as some sort of control. It does however not include a control strategy, since there is no problem to solve.

3.2.2 Real Time Publish Subscribe

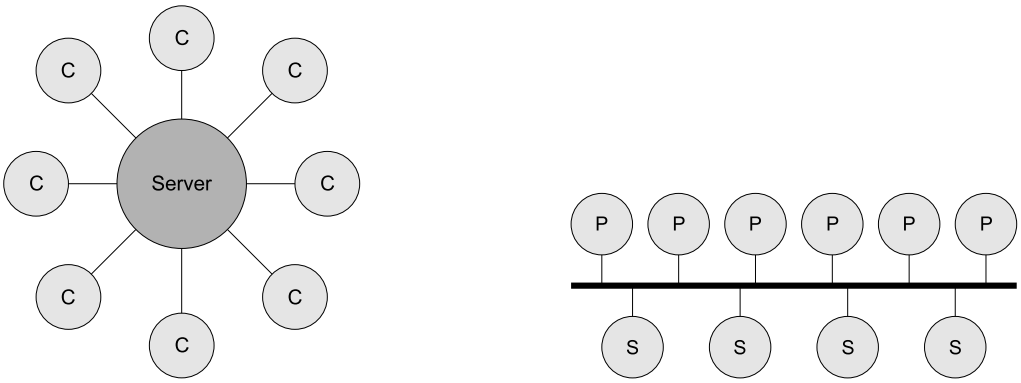
A widely used communication model on the Internet is the client server model. In this model, clients connect directly to the server. Connections among clients do not exist. Figure 3.2(a) is a graphical representation of this model. Since an actual connection is set up between the client and the server, the client and server are fully aware of each other.

An every day example which uses this model is browsing the web. The web browser acts as a client, and the system which it is connected to, containing the Web site, acts as a server. The client server works very well for this situation, since all data is centralized on the server. There is no need for data to be exchanged among clients.

In case many data producers and data consumers exist and the data needs to be distributed in a many-to-many fashion, the client server model is not a practical solution. When the client server model would still be used in this case, data producers would have to update the server, and the server would have to notify the clients which are interested in this new data. This situation might lead to a lot of unnecessary load for the server. It would be advantageous to bypass the server.

A solution to this is the publish subscribe mechanism. In this model, there is no server present. There are data producers, called *publishers*, and data consumers, called *subscribers*. Figure 3.2(b) shows the publishers and subscribers and the way they are connected. All publishers and subscribers are interconnected and treated as equal.

The publish subscribe mechanisms can be either topic based or content based. In topic based publish subscribe, publishers publish their data under a certain topic. Subscribers can subscribe to one or more topics. In content based publish subscribe, subscriptions are done on basis of characteristics of the actual message content. Both have their advantages and disadvantages. See [6] for a discussion



(a) Client Server (b) Publish Subscribe

Figure 3.2: Communication models: "Client Server" versus "Publish Subscribe"

on the benefits and the drawbacks of the different methods.

At Real-Time Innovations*(RTI) a protocol called RTPS [18] has been developed. RTPS is a protocol for publish subscribe communication on a closed ethernet network using the UDP protocol, which meets real-time constraints. RTPS is topic based. Because of the nondeterminism of the ethernet, it is said not to be suitable for real time data distribution. The research projects described in [20] and [5] deny this, and show that it is in fact a suitable medium.

RTPS is merely a protocol specification. An actual implementation of this real time publish subscribe mechanism is ORTE(Open Real Time Ethernet) [11]. The UDP protocol is used for data transmission, since it gives more control towards timing compared to the connection based TCP. SQLbusRT uses ORTE for setting up the communication channels between the data producers and data consumers.

ORTE provides an API which lets an application programmer create publishers and subscribers. Publishers provide data at predefined intervals, under a certain topic. This topic is used to give subscribers the possibility to receive only the data they are interested in. A manager application makes sure the right communication channels are set up between the publishers and the subscribers. In SQLbusRT, a new layer will be designed on top of the ORTE API to provide all mechanisms to communicate not only among data producers and consumers, but also the database.

3.2.3 SQLbusRT architecture

As said earlier, the SQLbusRT architecture is inspired by the blackboard architecture pattern. It is a composition of a readily available RTPS implementation with

*See `http://www.rti.com/`

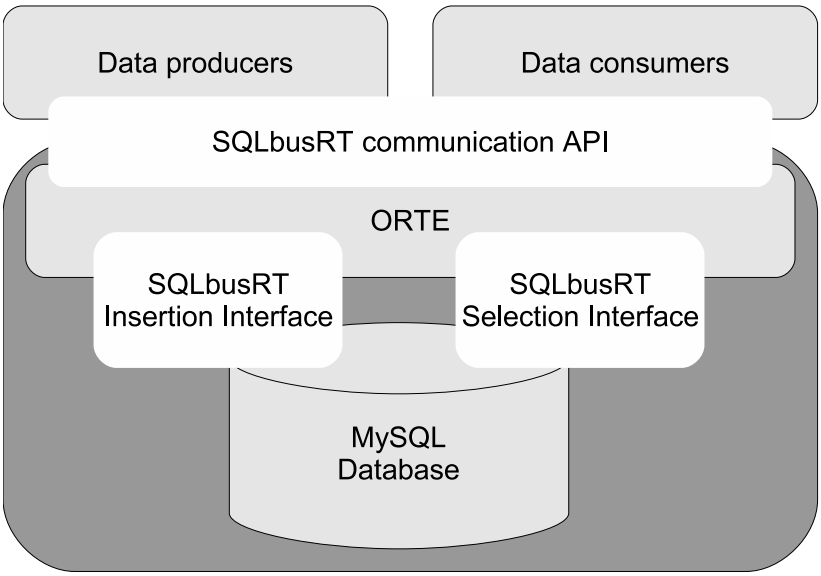


Figure 3.3: Basic architecture of SQLbusRT

a database engine, connected internally and externally with newly implemented interfaces. Figure 3.3 shows the different components and interfaces.

Data producers The data producers are not an actual component of SQLbusRT. They are external applications which produce data which is to be made available to data consumers directly, or to the database for later use. The data can be made available in two ways: periodically, or "on issue". When the data is produced periodically, the data is written to the bus on a predefined interval. With "on issue", the data is usually written to the bus on the occurrence of an external event.

An example of a data producer writing periodically would be an application reading an external thermometer. It publishes the temperature to the bus once every second, making it available to all data consumers which are subscribed to temperature data.

An example of a data producer writing "on issue" would be an application reading an external thermometer as well, but one which only publishes an event message when the temperature drops below a critical level. The data is directly available for data consumers which are subscribed to this alarm event.

Data consumers A data consumer is, just like a data producer, an external application. The data consumer subscribes to topics which it is interested in. With the examples given above, a data consumer could have subscribed to temperature data or temperature alarm events.

Data consumers subscribe to certain topics by means of SQL queries. This makes reading from either a data producer directly, or from history data in the database possible.

ORTE This is the implementation of the RTPS protocol. It has not been altered, but an interface has been build on top of it to make the connection with the data consumers and producers as intended in SQLbusRT possible.

SQLbusRT communication API This SQLbusRT component is an extra layer on top of ORTE. ORTE does not support MySQL subscriptions, so an interface had to be added on top of it to make this possible. It communicates with the SQLbusRT insertion and selection interfaces in the background, by using publishers and subscribers on the ORTE bus. Figure 3.4 shows how this is done. It is described in more detail under "selection interface".

MySQL Database MySQL is a readily available component. It is responsible for storing all the history for which it is configured in the selection interface. Presently the MySQL database is solely used for storage. With future research it might become clear that an "active database" which can send event notifications to external applications proof to be more useful. Therefore, it has been taken up as a recommendation in chapter 5.

Insertion interface This interface acts as a subscriber on the ORTE bus to fetch all the data from data producers that is to be stored in the database. In the current implementation, the interface subscribes to all topics, making all data available in the database. However, in future implementations the application programmer using SQLbusRT will be capable of designing an insertion strategy, which defines what data should be stored, at which interval, in which table and for how long.

Selection interface This interface is responsible for handling requests from data consumers. It consists of several components, which are: the request handler, one or more data publishers and the database interface.

The request handler is subscribed to requests that appear on the ORTE bus. When a data consumer sends a request, it is received by the request handler automatically. When this query was not requested by another data consumer before, the request handler creates a new data publisher which from then on will publish the requested data on the ORTE bus at the specified interval. After this, the request handler sends a message to the data consumer containing the topic on which the newly created data publisher will publish.

At present SQLbusRT does not yet recognize similar queries, but ultimately, SQLbusRT will not create a new data consumer for a query that has been

requested before. It will simply point the data consumer to the right data publisher that already exists.

Figure 3.4 is a graphical representation of how the data consumer (client), the request handler and the data publisher communicate. The idea is flexible and extendible to enable latter adjustments. The following happens on a request:

1. On startup of the total system, the Request Handler, which is part of the selection interface, opens a subscription to requests
2. The Data Consumer (Client) sends an SQL query together with its ID
3. Immediately after sending a query, the client opens a subscription for replies to its request
4. The Request Handler looks if a Data Producer (publisher) publishing data for a similar query is already active. If not, this Data Producer is created, as part of the selection interface
5. The Request Handler responds to the Data Consumer by sending an acknowledgment containing the topic and type of the Data Producer
6. On arrival of the acknowledgment, the Data Consumer closes its subscription to replies for its request, and opens a new subscription to the right Data Producer, using the data from the acknowledgment
7. All data sent by the Data Producer is received by the Data Consumer until the Data Consumer closes the subscription

3.3 Similar projects

This section describes OpenSplice, RTI Distributed Data Management and EDSAC21, which are existing projects with similar goals to those of SQLbusRT.

It makes a comparison, discussing some of the similarities and differences between these projects and SQLbusRT.

3.3.1 OpenSplice

Open Splice [23] is a product by PrismTech[†], and was formerly known as SPLICE-DDS, when it was still a product by Thales Naval Netherlands (TNL)[‡]. It is an implementation of the OMG-DDS specification, for which Thales was a co-author.

The OMG-DDS specification describes several layers, which are shown in figure 3.5. These layers are the following:

[†]See: `http://www.prismtechnologies.com/`
[‡]See: `http://www.thalesgroup.com/`

Minimum Profile This layer utilizes the publish subscribe paradigm, and uses topics to direct the information between the right publishers and subscribers.

Ownership Profile This layer offers support for publisher replication. Multiple publishers can provide data using the same topic. By specifying a *strength* for every publisher, the subscribers will only receive the data from the publisher with the highest strength which is currently present.

Content Subscription Profile This layer offers content awareness, which is useful for filtering information based on its content. It supports a subset of the SQL for querying data.

Persistence Profile This layer offers fault-tolerant availability of non-volatile information. The data is preserved outside the scope of the publishers. This gives access to the data to subscribers which join in after the data has already been published.

DLRL Profile This layer is optional. It adds an object oriented view the data centric publish subscribe layers.

SQLbusRT provides much of the functionality as offered in these OMG-DDS layers. Thanks to ORTE, it offers the publish subscribe mechanism with the support for replication as described in the minimum and ownership profile.

ORTE does not offer content awareness as mentioned in the content subscription profile, and SQLbusRT does not (yet) add this support on top of ORTE. It does however support SQL queries when data has to be retrieved from the database.

Handling SQL requests in order to combine database and published data is still a recommendation, as described in section 5.2.

The functionality of the persistence profile is partly present in SQLbusRT, and is made possible by adding a database to ORTE. Subscribers can subscribe to data which is present in the database.

Object orientation as provided by the DLRL (Data Local Reconstruction Layer) profile is not provided in SQLbusRT as such. However, SQLbusRT offers (de-)serialization, which can be used by the application programmer to flatten objects, so they can be transfered by means of the publish subscribe mechanism. The application programmer is always responsible for handling these objects correctly.

3.3.2 RTI Distributed Data Management

RTI Distributed Data Management [19], formerly known as SkyBoard, is a commercially available product by Real Time Innovations[§].

Of the three products discussed in this chapter, RTI Distributed Data Management (or RTI DDM) probably has to closest resemblance to what SQLbusRT is intended to become.

The component in RTI DDM responsible for the data distribution is RTI Data Distribution Service, or RTI DDS. [15]. RTI DDS is an implementation of the OMG-DDS specification, just like OpenSplice. The publish subscribe mechanism used in RTI DDS is again an implementation of the RTPS specification [18], just like ORTE.

A simple representation of the architecture can be found in figure 3.6. It shows three components, which can be fully mapped to components in SQLbusRT:

Applications These are the applications that use DDM for data exchange, but they are not really part of the DDM itself. They are like the data producers and data consumers in SQLbusRT.

Data Distribution Service This component is responsible for distributing the data among applications and the database. The DDS for most part is like ORTE in SQLbusRT.

DBMS The DBMS is responsible for the storage of data, an can be fully mapped to MySQL in SQLbusRT. The selection and insertion interface of SQLbusRT are the interface between ORTE and the database. This is also present in DDM, although they are not explicitly present in the figure.

To make a comparison, we can say SQLbusRT is similar to RTI DDM, and ORTE as a component of SQLbusRT is similar to RTI DDS as a component of RTI DDM.

3.3.3 EDSAC21

EDSAC21[2], wich stands for Event-Driven, Secure Application Control for the twenty-first century, is a project of the Opera group at the University of Cambridge Computer Laboratory.

Just like SQLbusRT and the other projects described in this chapter, it is based on publish subscribe communication. For this publish subscribe communication, EDSAC21 uses Hermes[16].

Although EDSAC21 shows similarities, mainly caused by the publish subscribe mechanism, like loose coupling and asynchronous communication, it also has a lot of differences.

[§]See: `http://www.rti.com/`

The Opera research group focuses on the design and deployment of open, large-scale, widely distributed systems. Where SQLbusRT is meant to be deployed in closed networks which are not geographically wide spread, EDSAC21 is meant for larger a larger scale of distribution, and uses a peer-to-peer routing network for data distribution.

Because of the wide scale which EDSAC21 is designed for and the openness of the network it has to function in, a major focus of this project is security.

Security is not considered to be an important issue in SQLbusRT which has to be tackled soon, since it is to be used in closed networks.

Because of the differences in focus between EDSAC21 and SQLbusRT, it is most likely more interesting to track the OpenSplice and RTI DDM projects.

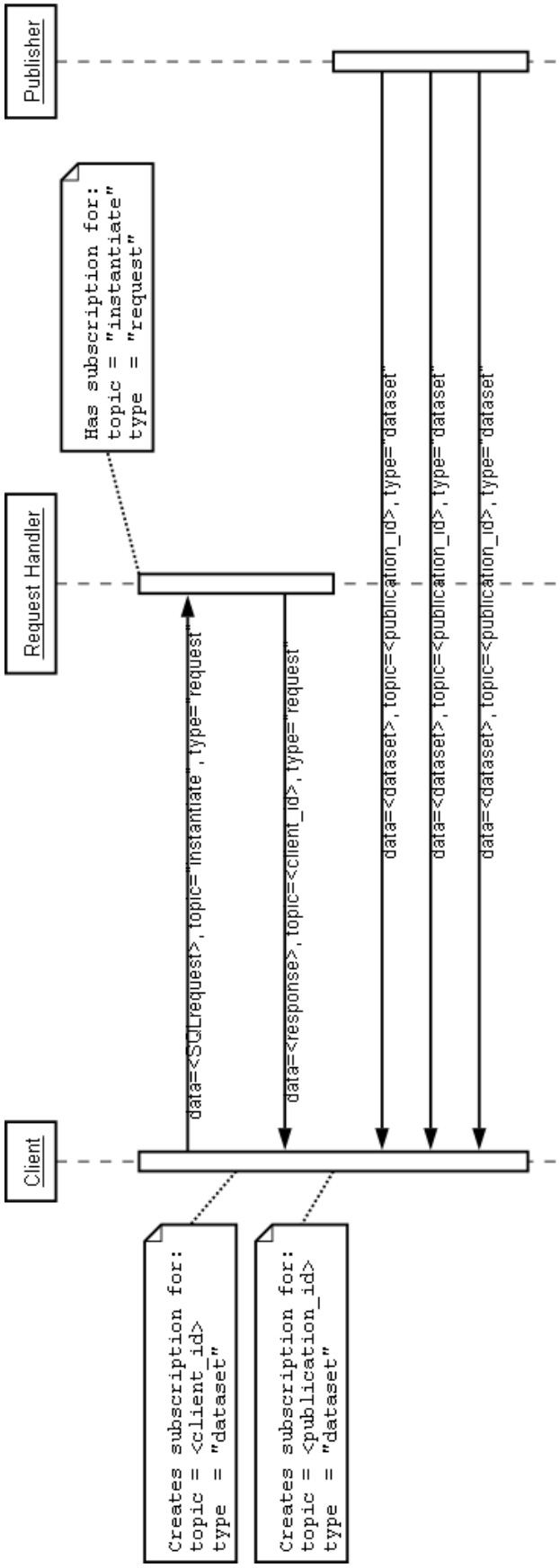


Figure 3.4: The way requests are handled in the current SQLbusRT prototype

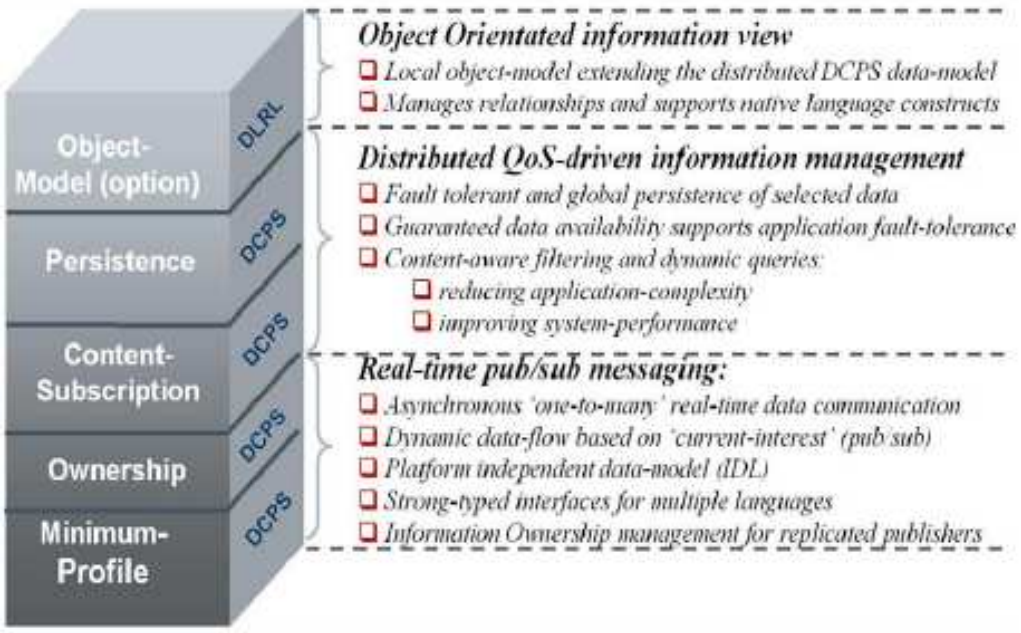


Figure 3.5: Layers in the OMG-DDS specification

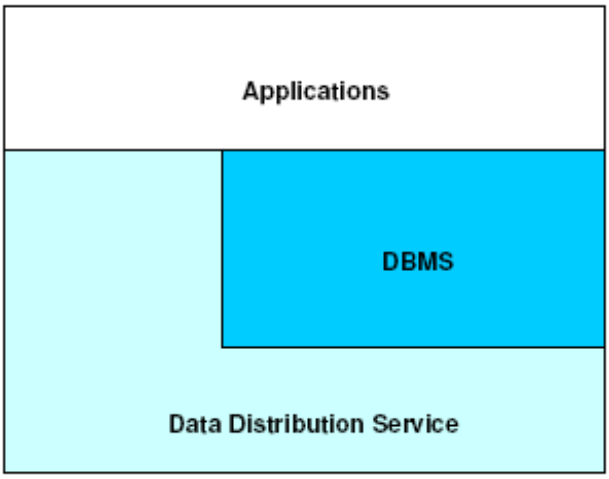


Figure 3.6: A layered architecture for Distributed Data Management

Chapter 4

Testing SQLbusRT

To be able to make predictions on the performance of systems to be built using SQLbusRT as middleware, SQLbusRT is put to the test using very simple sensors and clients. The following sections describe the tests that are performed.

The tests are performed in the order as they appear in this report. The approach is iterative. With the iterative approach, the following is meant:

- For every iteration a goal is set which is part of the entire goal of the project
- The prototype is only developed up to the point needed to achieve the goal of this iteration
- The tests for reaching the goal are executed
- The results of this iteration are evaluated and reflected against the overall goal of the project; in case the iteration has not worked towards the goal, or new insights have changed the goal, the overall goal or the goals of following iterations can be changed

This iterative approach leads to more "checkpoints", leaving more control over the way towards the end goal.

4.1 Preparations

Sections 4.2 and 4.3 describe two sets of tests. The first iteration is to find the characteristics of ORTE solely, the second iteration is to find the characteristics of ORTE in combination with a MySQL database, coming close to the idea of SQLbusRT.

4.1.1 Setup

First of all, a set of four machines is connected in a network by means of a hub. These machines are all setup with a linux distribution (Debian Sarge). A custom kernel with a special kernel patch is installed. This kernel patch provides high definition clocks, allowing more accurate time measurement. More importantly, the patch allows preemption of software interrupts for hardware components (in the case of this project the network adapter) by giving their interrupts real time scheduling priority.

All processes related to ORTE, being the ORTE manager and the test programs described in the following section receive real time scheduling priority. Also the interrupt handler for the network device as well as the incoming and outgoing network stack processes get real time scheduling priority.

In the second iteration the MySQL database server is used, which also receives real time scheduling priority. A FIFO scheduling algorithm is used, which means that the first process requesting processor time will have it, until all operations that were scheduled for this process are finished. After this, the second in queue gets its turn.

A step by step instruction on how to set up the custom patch and setting real time priority for processes can be found in appendix C.

4.1.2 Implementation of test software

Two programs are implemented, called *ping* and *pong*. Both programs are written against the C API of ORTE, and are functioning as a publisher as well as a subscriber on the ORTE bus.

Ping is the initiator of the test. It sends a publication on which pong has a subscription. As soon as pong receives the message, it will on its turn publish a message on which ping has a subscription. Ping will on arrival of the message publish a message for pong again, and this will continue in a loop until a predefined maximum number of messages has been sent back and forth, or until a problem occurs.

For the second iteration, the pong process is extended with a database interface, written against the mysql++ API. For all incoming messages, the pong process will do a database update. Immediately after the update, it does a read action, after which it sends the result of the read action back to the ping process.

In comparison to the ping pong in iteration 1, one link is added to the chain. Instead of a message traveling like: *ping - pong - ping - ...* as in iteration 1, it now travels like: *ping - pong - database - pong - ping - ....*

4.1.3 Time measurement

Ideally, one way travel time would be measured. This would mean that the ping records the time of sending a message, and pong records the time of retrieval of a message. The travel time would then be the difference between both time stamps. This is possible on a single machine, but becomes problematic when the travel time between two computers is to be measured, since both computers have their own clock which are not necessarily synchronized.

There are theoretical solutions to this problem, but they are not always feasible or useful:

Synchronizing clocks over the network The Network Time Protocol [14] lets computers synchronize their clocks over a network. For this protocol it is assumed there is at least a computer in the network which has an external clock device connected to it. It is also designed especially for networks with many computers, like the Internet. Because of the lack of an external clock device and the small scale of the network used in these measurements, the Network Time Protocol is not a suitable choice. In the setup as used for these experiments, accuracy problems will arise. The error can be as big as the roundtrip time.

[21] describes tests with synchronized clocks with a different algorithm, but problems are faced with the accuracy of the synchronization. The errors range from an average of 100 μs up to 1 ms. These errors are significant, making this algorithm unsuitable for our tests.

Synchronizing clocks using an external device As explained earlier, no external device is available for synchronizing the machines. In case a device would have been available, there's still the problem of accuracy. Also, the port which would be connected to the device needs preemption, though in this test it is desired to only preempt the network device.

Synchronizing clocks directly on the serial ports This method solves the problem of needing an external device. However, as with synchronization by means of an external device, there's still a problem with accuracy and preemption of the serial port.

The difficulties faced with measuring the one way travel time has lead to the decision to measure roundtrip times, which is decided to be the performance metric as decided in chapter 2.

In the first iteration Ping is the only process which is responsible for the time measurement and logging. It stores a timestamp on sending a message, and stores a new timestamp on arrival of the response to that message. The difference of these two time stamps is logged as the roundtrip time. It is important to know

that this time consists of several components. On sending a message, the following events occupy time:

Machine A (ping) Send thread in ORTE, outgoing network stack, physical transfer,

Machine B (pong) Incoming network stack, Receive thread in ORTE, Send thread in ORTE, outgoing network stack, physical transfer,

Machine A (ping) Incoming network stack, Receive thread in ORTE.

Logging all round trip times separately would make the log file become extremely bulky. Therefore, only the average time for every 1000 messages is logged. The number of deadlined messages is also counted per 1000 messages.

In the second iteration also the pong process makes a time log, but this time log is merely for the database response time. This way, the total roundtrip time can be compared with the response time of the database. Again, this is done per 1000 messages, just as in the ping process.

4.2 Iteration 1: Message distribution

The following section describes the first iteration. This iteration is to determine the characteristics of ORTE, which is the middleware used for data distribution in SQLbusRT.

4.2.1 Problem statement

For the data distribution in SQLbusRT, a readily available middleware component is used for data distribution. This middleware component, called ORTE, will tie all the publishers, subscribers and the database together.

At present there is no insight in how ORTE will perform in different scenario's. However, when SQLbusRT will be used as part of an application, it is most likely necessary to be able to predict the characteristics. Since ORTE is a major component of SQLbusRT, it is important to know how it will perform.

4.2.2 Goal

The goal of this first iteration is to determine the characteristics of ORTE. The characteristics of interest are the *performance* and the *reliability*. By changing the circumstances of the test, it will also be possible to determine ORTE's *scalability* regarding the amount of data to be distributed (message size), the network load and the number of publishers and subscribers.

Moreover, there is an interest in how constant the behavior of ORTE is. The more constant its behavior, the more predictable it is.

4.2.3 Approach

In different setups, messages are sent back and forth between two processes. The roundtrip times are measured. These values directly serve as the performance values as decided in chapter 2.

The deadline is set to 1 second in all setups. This is to make sure that messages will always return back in time under normal circumstances. The reliability is determined by referring back to the roundtrip times and counting the percentage of messages that would not have arrived with different deadlines. This percentage is calculated with equation 2.4 (see page 8).

The scalability is made visible in a number of graphs.

Test setup

All setups in this iteration use the ping and pong implementation. ORTE is used to make the processes communicate.

The configuration of the machines used in these setups is described in appendix B.

Unless stated otherwise, the message size used in these tests is 1 byte. The following setups will be used to measure the ORTE characteristics:

Single machine *ping pong* Both the processes ping and pong run on a single machine. No network delay is influencing the measurement, but the processes have to deal with resource sharing. The processor and memory will both be shared by both processes.

To determine hardware dependent performance and reliability, this test is executed on machine A, B, C and D.

Networked simple *ping pong* Two machines are used in this setup. One machine runs ping, the other machine runs pong. The network delay can be measured this way.

This test is run twice, once on machine A (ping) and B (pong), once on machine C (ping) and D (pong). This way, influence of hardware is determined.

Networked *ping pong* with varying message size The previous test is repeated on machine C and D, but with varying message sizes. The previous test is executed with a 1 byte message. This test will use message sizes of 128, 1024, 2048, 4096 and 8192 bytes.

Networked *ping pong* with network load The first networked ping pong test is repeated, but with an external source of varying network load. The measurement is done on machines C (ping) and D (pong). However, machines A and B will also run ping pong to generate extra network load.

Networked with multiple *pongs* This setup is similar to the networked simple ping pong, but instead of using one pong process, multiple pong processes are responding to the ping messages.

Machine A runs the ping process. Machine B, C and D all run pong processes. After sending a message, the ping process locks until all replies from machines B, C and D are retrieved (or when a deadline occurs). The ping process logs the average response times from machines B, C and D.

4.2.4 Test results

The test results are divided in four parts. The first part contains the round trip times which is the performance measurement. The second part contains calculations of the reliability using the roundtrip times. The third part discusses the scalability. The last part contains other presumably useful observations which are noted during the measurement.

Performance

Table 4.1 shows the minimum, maximum and average roundtrip time as well as the the standard deviation for all the measurements in iteration 1.

Please note that for every 1000 messages the average is calculated and logged. *Minimum* in this table means the lowest average of 1000, where *maximum* means the highest average of 1000. The absolute minimum and maximum are not logged.

	Min	Mean	Max	σ
Single machine, Machine A	91	117	122	1.269
Single machine, Machine B	2174	2203	2292	28.991
Single machine, Machine C	241	265	285	3.667
Single machine, Machine D	522	531	577	7.728
Networked (A and B)	1267	1289	1410	19.275
Networked (C and D)	500	504	509	1.884

Table 4.1: Iteration 1: Roundtrip times (μs)

As discussed in chapter 2 these roundtrip times serve as our performance quantification.

The single machine tests show a very strong machine dependency. The average roundtrip time on the slowest machine is more than eighteen times as much as the average on the fastest machine. Also the deviation is much higher on a slow machine. The difference between minimum and maximum time on the slowest and the fastest machine is respectively 118 μs and 31 μs. This makes the roundtrip time on a faster machine more predictable.

The strong hardware dependency is also visible in the simple networked tests. Eventhough the CPU's do not have a 100% usage anymore in these tests, there's

still a difference with a factor of more than two between the two setups. Again, the deviation in the slowest setup is higher than in the faster setup.

The graph in figure 4.1 shows the strong hardware dependent roundtrip times for both the single machine test and the simple networked test.

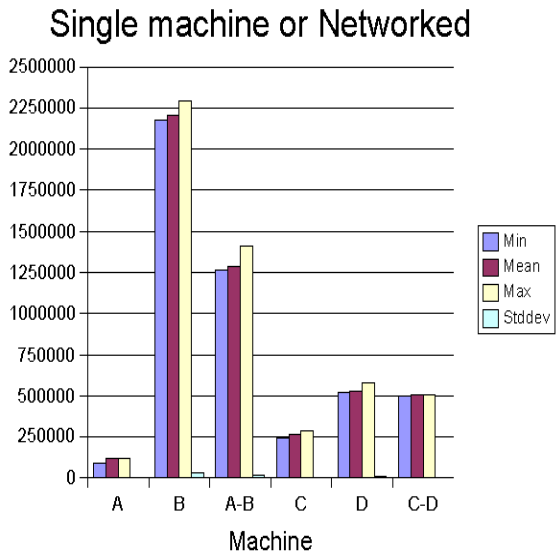


Figure 4.1: Performance: Single Machine versus Networked

Reliability

As discussed in chapter 2, the reliability is calculated with equation 2.4. This equation is as follows:

$$\mathcal{R} = \frac{N_{succeed}}{N_{miss} + N_{succeed}} \times 100\%$$

The reliability can be calculated for a set of hypothetical deadlines. To do this for all the executed tests and putting all these results in a table would make this table very bulky. Therefore, the results are represented in a graph. Figure 4.2 shows this graph, gathering the reliability for all performance tests.

The graph shows that the distance between 0% and 100% reliability for all measurements are very close to one another (since the lines are very vertical). This makes ORTE very predictable, which is useful in real time applications.

Scalability

To measure the scalability, some extra tests were executed. Somehow they are performance measurements as well, but they are mostly intended to see how the setup scales.

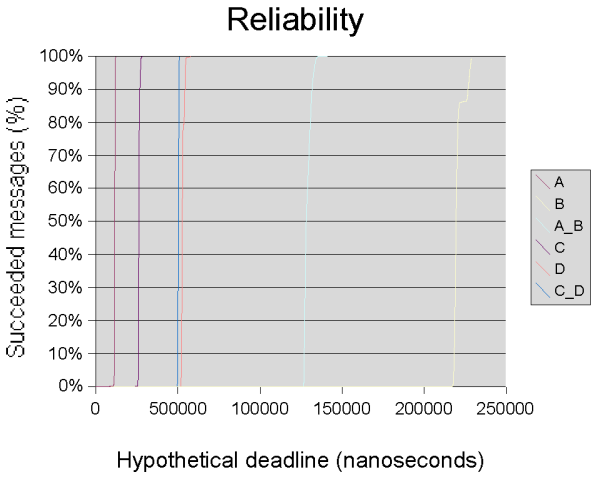


Figure 4.2: Reliability: Single Machine versus Networked

The results of the scalability tests are gathered in table 4.2.

	Min	Mean	Max	σ
1 byte	500	504	509	1.884
128 bytes	718	723	736	1.846
1024 bytes	2213	2217	2233	2.111
2048 bytes	4008	4014	4039	4.471
4096 bytes	7474	7486	7578	6.818
8192 bytes	14425	14544	14612	8.817
Networked w/ extra load	506	512	518	1.934
Networked w/ multiple pongs	843	859	891	5.207

Table 4.2: Iteration 1: Roundtrip times (μs) for scalability

The test with varying messages sizes shows a linear growth. Messages larger than 8192 bytes have not been tested. The linear growth is clearly visible in figure 4.3. Table 4.2 shows that also the standard deviation (σ) increases with a growing message size.

The measurement with extra network load shows just a small increase in time in comparison to the test without extra network load. The amount of extra network load might not have been significant.

The roundtrip times from the test with multiple pongs is difficult to compare with other measurements, since the machines all have different hardware. Some important observations have been done which can be found in “Other observations”.

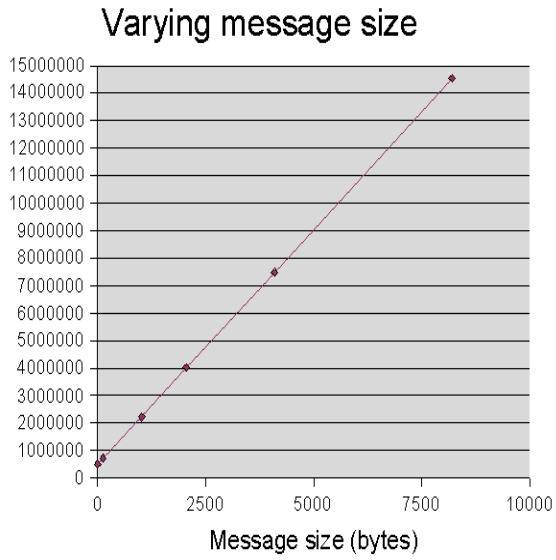


Figure 4.3: Varying message size

Other observations

During the test process, the CPU usage and network load were monitored for any unexpected behavior. The linux tool "top" was used for monitoring the CPU usage, and a led indicator on the hub was used for monitoring the network load. The scale of the led indicator on the hub (1, 2, 3, 6, 12, 25, 50 and 80%) is very inaccurate. Therefore, these values are merely indicative.

The CPU usage and network load for all tests are gathered in table 4.3.

	CPU A	CPU B	CPU C	CPU D	Network
Single machine, Machine A	100%	n/a	n/a	n/a	n/a
Single machine, Machine B	n/a	100%	n/a	n/a	n/a
Single machine, Machine C	n/a	n/a	100%	n/a	n/a
Single machine, Machine D	n/a	n/a	n/a	100%	n/a
Networked (A and B)	4%	97%	n/a	n/a	8-12%
Networked (C and D)	n/a	n/a	25%	55%	25%
Networked (C and D) 128 bytes	n/a	n/a	20%	40%	25%
Networked (C and D) 1024 bytes	n/a	n/a	8%	15%	50-80%
Networked (C and D) 2048 bytes	n/a	n/a	5%	9%	80%
Networked (C and D) 4096 bytes	n/a	n/a	4%	7%	80%
Networked (C and D) 8192 bytes	n/a	n/a	3%	8%	80%
Networked (C and D) w/ extra load	n/a	n/a	25%	55%	25%
Networked w/ multiple pongs	5%	50-100%	7%	12%	0-25%

Table 4.3: Iteration 1: Indication of the CPU usage and network load

The table shows that for all single machine tests, the CPU's are fully used. This is expected behavior, since both the ping and pong process are executed using one single CPU.

During testing, network traffic was monitored using a tool called Ethereal*. Ethereal showed UDP packet fragmentation for message sizes of 2048, 4096 and 8192 bytes. The frame fragments never extends 1500 bytes.

Interesting to note is that the test "Networked (C and D)" has the same load as the test "Networked w/ extra load", according to the hub's indicator. Presumably running two ping pong instances does not significantly increase the network load. This might explain why the roundtrip times for these two tests, as stated in table 4.2, do not differ much.

The test with multiple pongs had unexpected behavior. The pong process running on machine B occasionally caused 100% CPU usage, taking a lot of time to send back its response. This has caused many deadlines to occur in this test.

4.2.5 Conclusions and further work

The results found is these tests already give a hint of the characteristics of ORTE, but are not sufficient to make predictions about the performance in different untested scenario's.

Most tests show a strong hardware dependency. To find the source of this hardware dependency, more tests would have to be repeated on a larger set of machines, with carefully chosen hardware.

ORTE has proven to be stable in most setups, giving no deadlines and sending back all messages without corrupt content. There was however unexpected behavior in the "multiple pongs" setup. At present, no cause for this behavior has been found yet. Since this behavior was caused by only one machine in the setup and was not reproducible with other machines, it might be a hardware problem. More tests should be executed to fully eliminate the possibility of software problems.

To reach a better predictability of the ORTE performance, more tests are needed. Examples of more thorough tests would be the following:

Single machine *ping pong* The tests have now been executed on four machines with different hardware. The tests show a very strong machine dependency. It is therefore desirable to identify where the hardware dependency lies.

New tests could be executed on machines which only differ in one characteristic at a time. Varying the bus speed or clock speed can be done by just altering the BIOS settings, and therefore does not require any physical changes.

*See `http://www.ethereal.com/` for information and downloads

Changing the memory size will most likely not affect the results, since none of the computers was ever using the maximum available memory during the tests. However, this has not been verified.

Networked simple *ping pong* The network hardware might have a major influence on the round trip times. Not only the network interfaces, but also the hub, switch or any other connecting device can be of major influence.

Experimenting with different network hardware is therefore desirable. Computers were connected with a simple office hub in this experiment. Other tests could include the use of low and high end switches instead of a hub.

Networked *ping pong* with varying message size This experiment has probably been the most complete test of all, with the clearest result. There is however one thing missing in the result, which is the maximum possible message size.

This boundary could be determined with experiments. However, it might be more feasible to examine the source code to find a specification of the maximum message size. With boundary testing this value can then be verified.

Networked *ping pong* with network load With the network load taken in this test, only a slight change in roundtrip times was measured. Instead of taking another constant network load in a new test, varying the network load could help in finding the trend in which the round trip time increases with a linear growth of network load.

Networked with multiple *pongs* Only one test has been executed with multiple pongs. The results of this test are hard to compare with other tests, since all machines used in this test have different hardware.

To find the scalability regarding multiple publishers or subscribers, the test should be repeated with machines which are all equal, while varying the number of pongs.

4.3 Iteration 2: Database influence

The following section describes the second iteration. This iteration is to determine the characteristics of MySQL in combination with ORTE.

4.3.1 Problem statement

SQLbusRT will be put in practice for data distribution in systems with real time constraints. Relational databases are known to have varying response times under different circumstances. This varying query times make it impossible to

guarantee real time performance. However, under controlled circumstances, the query times might become more predictable.

4.3.2 Goal

The goal of this test is to find out if it is possible to gain a low deviation in query times under controlled circumstances, since a low deviation leads to a better predictability of query times.

Besides measuring the query times, also the deadlines and failed queries will be counted to determine the reliability in these setups.

4.3.3 Approach

Similar to the first iteration, in different setups messages are sent back and forth between two processes.

Different from the first iteration, the deadline is now set to 2 seconds in all setups, since the database forms a new link in the chain which causes delays. This does however not influence the reliability test, since it the reliability is still done on the hand of hypothetical deadlines.

Test setup

The ping and pong process which were used in the first iteration are used again in this iteration.

Time logging is implemented at both the ping and the pong process. The pong process calculates the total execution time for writing and reading to the database. The ping process logs the total roundtrip time, like in iteration 1.

In all setups described here, the ping process runs on machine C. The pong process runs on machine D, and connects to the MySQL database server trough a socket connection on localhost.

The tables in the MySQL database all use the *MEMORY* storage engine, which causes the table to be fully stored in RAM memory. This prevents the system to suffer from hard disk latencies on writing and reading data from the tables.

The following setups will be used to measure the characteristics of MySQL in combination with ORTE:

Table with single record In this test, a simple database is setup with just a single table containing one record. The record is updated every time the pong process receives a message from the ping process.

After writing the contents of the message to the database, it is immediately read by the pong process again. The result of the query is returned to the ping process.

Varying record size The previous test is repeated, but with varying message size. The previous test is executed with a 1 byte message. This test will use messages with the following sizes: 128, 1024, 2048, 4096 and 8192 bytes.

Note: During testing it became clear that the MEMORY engine does not support large texts or blobs. For this reason it became necessary to choose another engine. The default engine "MyISAM" has been used for this test.

As a result, the table is stored on hard disk, which can cause hard disk latencies. Hard disk latencies make access times to data in the table less reliable in comparison to an in memory table. However, the database still uses cache in RAM. The discussion on the actual influence of the MyISAM engine can be found in the last section of this chapter.

Table with growing record count This test determines the effect of growing tables. A table is indexed using a timestamp for every record. On insertion, the pong process will not update a record, but add a record to the table. On selection, the last added record will be retrieved.

Querying multiple records In this test, a database is used with one table containing respectively 2, 4, or 8 records. Only one of these records is updated every time pong receives a message. The selection query concatenates the contents of the records.

Querying multiple tables This test is similar to "querying multiple records", with one difference. Instead of having the records in one single table, the records are spread over different tables.

4.3.4 Test results

Just like in the first iteration, the results are split into four parts containing respectively the performance, reliability, scalability and other observations.

Performance

Table 4.4 shows the average roundtrip time for the first measurement. The other tests are done for scalability measurements, and can therefore be found under "Scalability".

The average and standard deviation in the leftmost columns are calculated from the time measurements which are done at the ping process. These values include all actions from sending the message, inserting it in the database, querying the database and sending back a message until the message has been received again by the ping process.

The average and standard deviation in the rightmost columns are calculated from the time measurements which are done in the pong process. These values only include the database related actions.

	Mean (total)	σ (total)	Mean (DB)	σ (DB)
Single record	1391	8.753	830	8.206

Table 4.4: Iteration 2: Roundtrip and query execution times (μs)

The performance measurement solely is not useful as such, but becomes more interesting when compared to other values. For this, see the "Scalability" section.

Reliability

Also in this second iteration the reliability is calculated with equation 2.4:

$$\mathcal{R} = \frac{N_{succeed}}{N_{miss} + N_{succeed}} \times 100\%$$

The results are represented in a graph, just as in the first iteration. This graph can be found in figure 4.4.

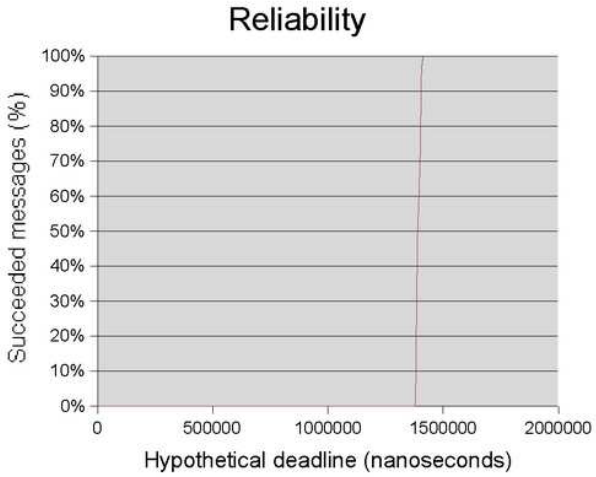


Figure 4.4: Iteration 2: Reliability

The database has not caused a high deviation. Just as in the first iteration, the difference in hypothetical deadlines between 0% and 100% succeeded messages is very little, which is visible by the very vertical incline.

The strong incline is very much desired for real time applications, since it makes the roundtrip time predictable.

Scalability

In the first iteration, a measurement with varying message size was executed. The roundtrip times in this test showed a linear growth against a linear growth

	Mean (total)	σ (total)	Mean (DB)	σ (DB)
128 bytes	1759	7.466	959	6.947
1024 bytes	3457	13.032	1147	12.535
2048 bytes	5447	25.209	1343	25.147
4096 bytes	9289	39.376	1698	39.581
8192 bytes	17158	44.310	2487	43.617
Growing table	107481	76023.824	105342	75091.817
2 records	1402	8.520	820	8.744
4 records	1420	8.106	835	7.670
8 records	1394	7.808	814	7.372
2 tables	1414	7.929	830	7.561
4 tables	1415	15.028	833	14.918
8 tables	1458	8.079	871	8.156

Table 4.5: Iteration 2: Roundtrip and query execution times (μs)

of the message size. In the second iteration, the presence of a database has not changed this linear growth.

Figure 4.5 shows both the total roundtrip time (top) and the query execution time (bottom) in one graph. The standard deviation can be read from table 4.5. This standard deviation shows an increase which is in proportion with the average roundtrip time for most values. The exception lies with the 128 bytes message, which has a lower deviation than the 1 and 1024 bytes messages.

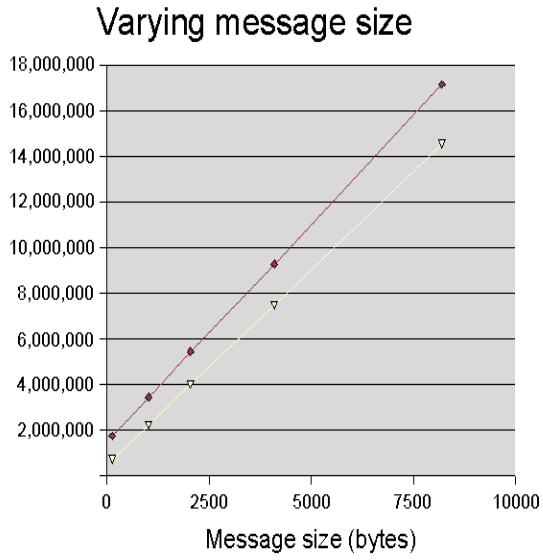


Figure 4.5: Iteration 2: Varying message size

The standard deviation for the test with increasing table size seems exception-

ally large, but this value might not be so meaningful. There's a constant increase in the roundtrip time while the record count grows. Therefore, the deviation would only increase when the test would run for a longer time. The increase of the roundtrip times for this test does not follow a straight line, though the increase is clearly visible. A graphical representation of this test is to be found in figure 4.6.

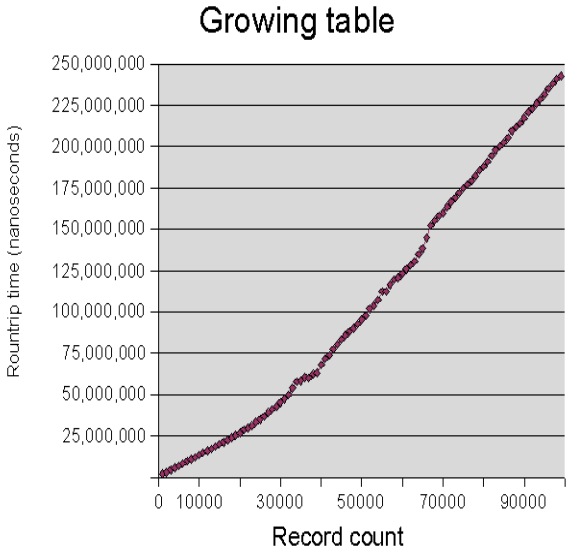


Figure 4.6: Iteration 2: Growing table

The results for the test with querying multiple records and querying multiple tables have been merged into one graph, (see figure 4.7), so they can be easily compared.

As the graph shows, the behavior of querying multiple records in one table, or querying records in separate tables, does not show a clear ascent with in increase of the records/tables. Important to note is that the y-axis does not start at 0. The differences in time between querying 1 up to 8 records is still very small. Querying multiple tables does show an increase, but querying multiple records within the same table shows a decrease again between 4 and 8 records. This might be because of optimizations, although this has not been verified.

Other observations

During testing the CPU usage and network load were monitored during the tests, just like in the first iteration. The results of these observations are gathered in table 4.4.

No exceptional values were found in these observations. However, if we compare the network load of the tests with a large message size in this iteration with

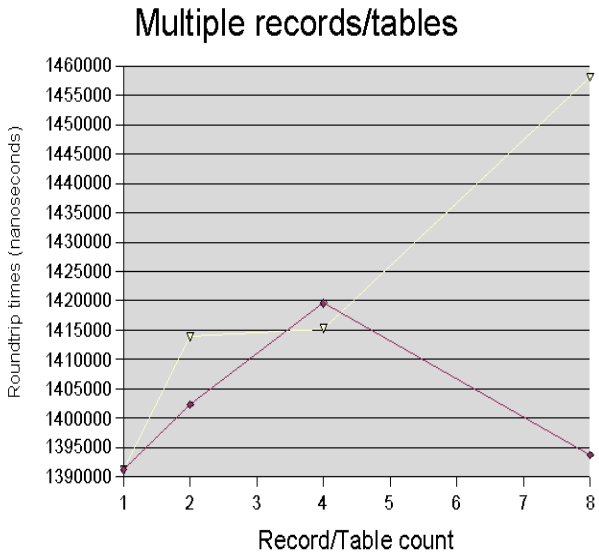


Figure 4.7: Iteration 2: Multiple records or tables

the ones in the first iteration, we can see a large difference. The processing time for the database is responsible for a decrease in network load.

Apart from the CPU usage and the network load, no significant observations were done. All processes functioned as expected.

	CPU C	CPU D	Network
1 byte	10%	90%	6%
128 bytes	8%	72%	12%
1024 bytes	5%	55%	50%
2048 bytes	4%	45%	50%
4096 bytes	3%	40%	50%
8192 bytes	2.5%	35%	50-80%
Growing table	1%	99%	1%
2 records	11%	90%	6%
4 records	10%	92%	6%
8 records	11%	92%	6%
2 tables	10%	92%	6%
4 tables	10%	92%	6%
8 tables	10%	92%	6%

Table 4.6: Iteration 2: Indication of the CPU usage and network load

4.3.5 Conclusions and further work

To start, it is important to notice that not all tests could be executed as they were planned. During testing, an error occurred while trying to write messages larger than 128 bytes to the database. An error message indicated that this was caused by the chosen database engine. This is confirmed by the MySQL manual page about the MEMORY storage engine. It states *"MEMORY tables cannot contain BLOB or TEXT columns."* [1].

To still be able to test the effect of growing record sizes, another storage engine was chosen, being "MyISAM". This might have its consequences during use in practice, since disk latencies might result in unpredictable response times.

The test with varying message size was successful, and resulted in very predictable behavior. It did not show any disturbances due to disk latencies. Perhaps when the test is executed for a longer period of time, these disturbances might still show up.

As in iteration 1, the roundtrip times show a linear growth with an increase of the message size. The database is responsible for the highest relative increase. This is not necessarily a problem, since it is unlikely that very large records (up to 8kB) will have to be stored in and retrieved from a database in a real time system.

There is no need to execute more tests with message sizes which reach the boundaries of the MySQL BLOB size, since the maximum size of ORTE messages is much lower than that of a BLOB in MySQL.

The increase in roundtrip times for growing tables is very strong. It should however be noted that on purpose a "dumb" query was used, which forces the entire table to be processed:

```
SELECT value
FROM table
ORDER BY id DESC
LIMIT 1
```

When the system is put to practice, most of the effects of this increase in record count can be reduced by good database and query design.

As discussed earlier, the results for the test do not show a clear ascent with in increase of the records/tables. Therefore, it might be interesting to execute more tests of this form, but with a larger amount of records and tables. Perhaps when there is a larger number of records to be queried, a clearer ascent can be determined.

Chapter 5

Conclusions and Recommendations

5.1 Conclusions

The goal of this project was twofold. The following questions were to be answered:

1. Is it possible to create a middleware solution offering storage and real time distribution of data, with the flexibility of easily adding and removing components to and from the system?

2. Can we predict *performance*, *reliability* and *scalability* of a system based on this middleware solution, before full implementation of that system?

The prototype that was written during this project has shown that ORTE, which was chosen as the RTPS implementation, and MySQL, which was chosen as the database engine, work well together. These two parts, together with the creation of interfaces to connect these two parts with each other and with external components have served as a middleware solution offering storage and real time distribution. Thanks to ORTE, the flexibility of adding and removing components was given. With a simple data producer and data consumer, the system was put to the test.

During research as well as implementation some fields for future research were identified (see section 5.2). Without this research SQLbusRT will not yet be complete, but the first tests show promising results.

To decide on the usability of SQLbusRT, it will have to be tested against more realistic scenarios. So far, only very simple scenarios have been tested. Currently SQLbusRT still lacks real life applications with a full set of requirements. Without

these requirements, it is still difficult to decide whether SQLbusRT going to fulfill the needs of these applications.

The requirements are also necessary to verify the real time characteristics of SQLbusRT. The question whether SQLbusRT can serve as a real time storage and distribution middleware solution highly depends on the demands of the system in which it will be used, as well as on how the total system is implemented. So far, no mechanisms have been implemented in SQLbusRT which guarantee real time performance.

To answer the second question, a set of tests has been executed, which have been fully documented in chapter 4. These tests are the beginning of making predictions about the performance, reliability and scalability of systems based on SQLbusRT. Since SQLbusRT is not yet in a mature state, the results of new performance, reliability and scalability measurements in the future might give different results than the ones found in this project so far. But likely, when SQLbusRT matures, these changes will only be positive.

Generally, the results of the tests which have been executed show one good and one bad thing. The good thing is, when changing the characteristics of the messages to be sent or the database to be queried, the deviation in response times is low, which adds up to the predictability. The bad thing is that the response times are very hardware dependent. This has a negative influence on the predictability, and proves the absolute values found in the tests less useful.

With the results from the tests that have been executed, it will become very difficult to make predictions about the performance, reliability and scalability of ORTE. The absence of requirements for SQLbusRT makes it very hard to design the right tests for finding these values in the current phase of the project.

To sum things up, SQLbusRT has shown to be an interesting research project, but there's much to be done before it is a mature middleware solution. The information that has been found in this Master project can serve as the base for further research. Section 5.2 already gives some thoughts for future research areas.

5.2 Recommendations

This research project has been the prework for a project that will most likely be continued in the near future. The results have shown that SQLbusRT might be a highly interesting middleware solution to build real time systems, but much work is still to be done.

5.2.1 Testing

First of all, many more tests need to be designed and executed to come to a better predictability of the behavior of SQLbusRT. Recommended tests have been

summed up in chapter 4. When SQLbusRT matures, even more sophisticated tests can be designed working towards more realistic scenarios.

Besides doing more testing, another approach towards data collection might be desirable. In all tests that have been performed during this research project, the average roundtrip time for every 1000 messages was logged. This way, some useful data might get lost. High peaks get flattened out by all the other time measurement in the same log record. There are several ways to prevent this from happening in the future, all with their pros and cons:

- The first way is to log all messages roundtrip times. No information gets lost this way. All calculations that need to be done on the data can be done afterwards. The drawback of this method is that data files become extremely bulky. Also, writing so many log entries during a measurement will have a bigger influence on the outcomes, since writing to a log file is time consuming.

- Another approach would be to sample one roundtrip time every 1000 messages. The benefit of this approach in comparison to the previous approach is that log files do not become that bulky. In fact, the log files will become just as large as the log files that have been produced in this research project. A benefit in comparison to the taken approach is that *real* results are logged, not average values. A drawback is that high peaks, instead of being flattened out, might be totally missed.

- Perhaps the best approach is to log the average value for every 1000 messages, but completed with the minimum and maximum value in that particular set of values. If needed, the standard deviation can even be logged. Drawback of this approach is that log files become a bit larger, but still they do not reach the size of the first mentioned approach.

5.2.2 Future research and development

Besides testing, there is still a lot of research to be done. During this research project, some of these future fields of research have already been identified:

API At present, a very simple API was designed on top of the ORTE API which was suitable for testing. It lets a simple data producer and a simple data consumer communicate.

All communication passes through the database, since no algorithm is yet designed to choose between the database or a data producer directly (see "Request handling"). A complete API should be designed which lets the application programmer use all the features SQLbusRT provides, or will provide in the future.

Request handling The prototype that was developed for this research project uses a mechanism for handling requests which is flexible to be extended. In the current state, it can only handle standard SQL requests, which was suitable for testing, but it does not suffice when SQLbusRT will be put to practice, since real time constraints and prioritizing information should be passed in the same request by means of an extended SQL form.

Not only the syntax for querying, but also the method of handling requests might still need a lot of improvements. At present, the way queries are handled (see section 3.2.3 or figure 3.4) does not yet follow the philosophy of the publish subscribe model, since publish subscribe in that case the query would be the subscription topic.

It would be highly interesting to research whether ORTE can be extended with the possibility to recognize similar queries. For instance, the two queries request the same data, but they are not literally the same string:

```
SELECT value AS a
FROM table
```

or:

```
SELECT value AS b
FROM table
```

Ultimately, this data would have to be requested from the database only once, to be distributed by SQLbusRT under different names afterwards.

This problem becomes even more sophisticated when the query language becomes time aware, and one client requests data at a 1 second interval, and a second client requests the same data at a 2 second interval.

Research towards, comparing, prioritizing and scheduling requests can help to find solutions to fully optimize SQLbusRT.

Another issue to research is how to recognize whether an SQL request is to be directed to the database, or when it will have to be translated to fetch information from a data producer directly.

ODBC connection The choice for MySQL was made since it is an open source database engine which makes adjustments to the code possible. However, so far it is not clear if adjustments at the database side are necessary, since some form of preprocessing of the extended queries might be performed by SQLbusRT.

In case adjustments at the database side are not necessary, there is no benefit in obliging the application programmer to use the MySQL database

engine. With an ODBC connection the application programmer could use a database engine of choice. Important to note is that this might create the need for an abstraction layer because of the preprocessing of extended queries by SQLbusRT.

Active databases When "on issue" requests need to be handled, an active database is needed. For instance with a query like: "Give current temperature when last three measurements were below zero", a trigger is needed from the database which notifies SQLbusRT in case the condition "last three measurements were below zero" becomes valid. MySQL supports triggers, but these are only to be used within the database.

A solution has to be found to trigger events outside the database.

Temporal databases Temporal databases offer some features from which SQLbusRT could benefit. A special query language for temporal databases, called TSQL2 [22] exists.

Interesting would be to research whether extending SQLbusRT with a temporal database is feasible. TSQL2 could become the language of choice, or could serve as a base for designing a SQLbusRT specific language.

Currently the University of Arizona has a research project which is focussed on adding features to manage time-oriented data in existing databases. One of the databases to be supported is MySQL. This subproject carries the name τMySQL (tauMySQL). The tau project can be found at `http://www.cs.arizona.edu/projects/tau/`.

Bibliography

[1] The memory (heap) storage engine. http://dev.mysql.com/doc/refman/5.1/en/memory-storage-engine.html. [cited at p. 42]

[2] Jean Bacon and Ken Moody. Edsac(21): Event-driven, secure application control for the twenty-first century. April 2004. [cited at p. 21]

[3] André B. Bondi. Characteristics of scalability and their impact on performance. [cited at p. 8]

[4] Frank Bushmann, Regine Meunier, Hans Rhonert, Peter Sommerlad, and Michael Stal. *A system of patterns*, pages 71–95. Wiley, 1996. [cited at p. 13]

[5] Ondrej Dolejs, Petr Smolik, and Zdenek Hanzalek. On the ethernet use for real-time publish-subscribe based applications. [cited at p. 7, 16]

[6] Patrick Th. Eugster and Rachid Guerraoui. Contect based publish/subscribe with structural reflection. In *6th Usenix Conference on Object-Oriented Technologies and Systems*, January 2001. [cited at p. 15]

[7] Tom Gilb. *Competitive Engineering: A Handbook for Systems and Software Engineering Management using Planguage*, chapter 5. June 2003. [cited at p. 6]

[8] Mark D. Hill. What is scalability? [cited at p. 8]

[9] Pankay Jalote, Brendan Murphy, Mario Garzia, and Ben Errez. Measuring reliability of software products. [cited at p. 7]

[10] S. Kim, S. Son, and J. Stankovic. Performance evaluation on a real-time database, 2002. [cited at p. 8]

[11] Jan Krakora, Pavel Pisa, Frantisek Vacek, Zdenek Sebek, Petr Smolik, and Zdenek Hanzalek. *Communication Components*, February 2004. [cited at p. 16]

[12] M.R. Lyu. *Handbook of Software Reliability Engineering.* IEEE Computer Society Press, 1996. [cited at p. 7]

[13] John D. McGregor, Judith A. Stafford, and Il-Hyung Cho. Measuring component reliability. [cited at p. 7]

[14] D. Mills. Precision synchronization of computer network clocks. *ACM Computer Communication Review*, 24(2):28–43, April 1994. [cited at p. 27]

[15] Gerardo Pardo-Castellote, Bert Farabaugh, and Rick Warren. An introduction to dds and data-centric communications, April 2004. [cited at p. 21]

[16] Peter Robert Pietzuch. Hermes: A scalable event-based middleware. Technical Report 590, University of Cambridge, Computer Laboratory, June 2004. [cited at p. 21]

[17] Krithi Ramamritham. Real-time databases. *Distributed and Parallel Databases*, 1(2):199–226, 1993. [cited at p. 5]

[18] RTI. Real-time publish-subscribe wire protocol specification. [cited at p. 16, 21]

[19] RTI. Data management for real-time distributed systems, 2006. [cited at p. 21]

[20] Stan Schneider, Gerardo Pardo-Castellote, and Mark Hamilton. Can ethernet be real time?, November 1998. [cited at p. 16]

[21] Seppo Sierla, Jukka Peltola, and Kari Koskinen. Evaluation of a real-time distribution service. `http://www.rti.com/docs/RTPS_Overview_HUT.pdf`. [cited at p. vi, 27]

[22] Richard T. Snodgrass, Ilsoo Ahn, Gad Ariav, Don S. Batory, James Clifford, Curtis E. Dyreson, Ramez Elmasri, Fabio Grandi, Christian S. Jensen, Wolfgang Kafer, Nick Kline, Krishna G. Kulkarni, T. Y. Cliff Leung, Nikos A. Lorentzos, John F. Roddick, Arie Segev, Michael D. Soo, and Suryanarayana M. Sripada. Tsql2 language specification. *SIGMOD Record*, 23(1):65–86, 1994. [cited at p. 47]

[23] Hans van 't Hag. Opensplice overview white paper. [cited at p. 19]

[24] Gertjan Oude Vrielink. Real-time databases, theorie versus praktijk. Master's thesis, University of Twente, 1999. [cited at p. 6]

Appendices

Appendix A

Assignment description

A.1 Original (Dutch)

In machinebouw, meetinstituten en (wetenschappelijke) onderzoekscentra wordt steeds meer data gegenereerd. Deze data bevat performancegegevens van het onderhavige proces.

Deze data wordt opgeslagen in databases, van waaruit on-line en off-line gegevens kunnen worden opgevraagd. De opgevraagde gegevens worden gebruikt voor directe bijsturing van het onderhavige proces, en voor latere analyse van de performance resultaten.

Nu wordt voor bovenstaande problematiek gebruik gemaakt van standaard SQL databases en standaard ODBC databasekoppelingen, die echter de volgende noodzakelijke elementen missen:

Real Time SQL Het genereren van een query binnen gestelde deadlines. In situaties zoals hierboven beschreven is het noodzakelijk om resultaten uit de database te verkrijgen binnen bepaalde tijd, om zodoende het onderhavige proces direct en online bij te sturen.

Gedistribueerd Optimalisatie als een resultaat van een query meerdere genteresseerden kent. Als diverse onderdelen van de besturing genteresseerd zijn in een bepaalde query dan zal deze query met de huidige bekende mechanismen meerdere malen uitgevoerd moeten worden. Wat hier nodig is, is de mogelijkheid om een resultaat van een query aan meerdere genteresseerden is aan te bieden, zonder de query meerdere malen uit te voeren.

Real Time Publish Subscribe mechanisme om gegevens tussen enteresseerden van een query en database engine(s) uit te wisselen, binnen gestelde deadlines in het tijddomein. Met de huidige bekende mechanismen is een

ODBC of embedded SQL koppeling mogelijk, echter er kan altijd maar een afnemer van een query resultaat zijn. Wat hier nodig is een Real Time Publish Subscribe mechanisme uitgebreid met een methodiek om query vragen en resultaten uit te wisselen met 1 of meerdere database engines.

Real Time fallback mechanisme Bij uitval van een database engine terugvallen op een duplicaat. Dit is al mogelijk, maar afnemers hebben een tijdrovende reconnect nodig. Wat hier nodig is, is een mechanisme om zonder tijdverlies te kunnen terugvallen op een duplicaat database engine.

A.1.1 Opdracht

Haalbaarheidsonderzoek, ontwerp en realisatie van een prototype van SQLbusRT op Linux, gebruikmakend van MySQL en een Real Time Publish Suscribe protocol.

Doel is het uitbreiden van de SQL syntax en de open source database SQL voor gebruik in een real-time omgeving voor bijvoorbeeld machinebouw, meetsystemen in windtunnels en medische systemen.

De syntax van SQL moet worden uitgebreid met primitieven voor real time constraints en loosely coupling.

A.1.2 Gekozen oplossingsrichting

In het project wordt een common off the shelf (open source) database engine gebruikt. Deze wordt gekoppeld aan een beschikbaar (open source) Real Time Publish Subscribe middleware, en aangepast voor real time gedrag (priotisering van queries en bewaken van deadlines in het tijddomein). Twee projecten uit de open source community worden dus gentegreerd en uitgebreid met mogelijkheden voor real time en gedistribueerd SQL.

A.1.3 Zelf te ontwikkelen methoden en/of technieken

Nieuwe technieken ontwikkelen op het gebied van real time dataprocessing met:

- honoreren van SQL aanvragen
- priotiseren van SQL aanvragen
- bewaking van deadlines op SQL queries
- Real Time Distribueren van resultaten van de SQL queries over meerdere afnemers

A.1.4 Nieuwe principes op het gebied van informatietechnologie

- Priotiseringsmechanisme voor SQL queries. Hiervoor wordt mogelijk de SQL syntaxis uitgebreid
- Real Time Distributie en optimalisatiemechanisme voor SQL queries en het verspreiden van de resultaten van de queries
- Bewaking van query deadlines op de queries
- Bewaking van transport deadlines van de query resultaten naar de afnemers

A.1.5 Toepassing

Machinebouw, procesinstallaties, meetinstrumenten, real time data acquisitie en real time trading

A.1.6 Doelgroep

Machinebouwers, ontwikkelingsbedrijven, (wetenschappelijke) onderzoeksinstellingen, ingenieursbureaus, SCADA en procesindustrie.

A.2 Translated to English

In machine construction, laboratories and research centers more and more data is being generated. These data contain performance information about the proces in hand.

These data are being stored in databases which allow online and offline data requests. The requested data is used for controlling the proces in hand, and for analysis of the performance results afterwards.

Currently standard SQL databases and ODBC database connections are used. However, these lack the presence of the following components:

Real Time SQL Generating query results within predefined deadlines. In situations as described here, it is necessary to retrieve data from the database within a certain time frame, so the proces in hand can be controlled directly and online.

Ditribution Optimization in case the result of a query knows multiple interested components. If several components of the control system are interested in a certain query result, then the query has to be executed several times. It is desirable to distribute the query results to different components without having to execute the query more than once.

Real Time Publish Subscribe Mechanism to exchange data among system components and database engines, within the predefined deadlines. Currently known mechanisms allow an ODBC or embedded SQL connection, though the amount of requesters for a query result is limited to just one. What is desired is a Real Time Publish Subscribe mechanism extended with a method to exchange query requests and results with one or more database engines.

Real Time fallback mechanism Fall back on a duplicate on failure of a database engine. This is already possible, though requesters take significant time to reconnect to a duplicate. A mechanism to fall back on a duplicate database engine without delay is desired.

A.2.1 Assignment

Feasibility study, design and realization of an SQLbusRT prototype on Linux, using MySQL and a Real Time Publish Subscribe protocol.

Goal is to extend the SQL syntax and using an open source database to be used in a real-time environment for e.g. machine construction, measurement systems in wind tunnels and medical systems.

The SQL syntax has to be extended with primitives for real time constraints and loosely coupling.

A.2.2 Chosen direction

In the project a common of the shelf (open source) database engine is used. This will be connected to a readily available (open source) Real Time Publish Subscribe middleware, and will be adjusted for real time behavior (prioritizing queries, guarding real time constraints). So, Two open source projects will be integrated and extended for real time and distributed SQL.

A.2.3 Methods and techniques to be designed

New techniques in the field of real time data processing with:

- Respondinng to SQL requests
- Prioritizing SQL requests
- Guarding deadlines on SQL requests
- Real Time distribution of SQL requests to multiple interested components

A.2.4 New principles in the field of information technology

- Prioritizing mechanism for SQL queries. The SQL syntax might be extended for this.
- Real Time distribution and optimization mechanism for SQL queries and distributing query results
- Guarding real time constraints on queries
- Guarding real time constraints on query result transportation

A.2.5 Usage

Machine construction, process installations, measurement instruments, real time data acquisition and real time trading.

A.2.6 Target groups

Machine constructors, development enterprises, (academic) research institutes, engineering companies, SCADA en and process industry.

Appendix B

Test setups

B.1 Hardware

	System A	System B	System C	System D
Dell Optiplex series	GX270	Gs+5166L	GS200	GX110
CPU family	Intel Pentium IV	Intel Pentium	Intel Pentium III	Intel Pentium III
L2 cache	512 kB	256 kB	256 kB	512 kB
Clock speed	2.80 GHz	133 MHz	733 MHz	500 MHz
Bus speed	2x400MHz		133 MHz	
Memory size	512MB	80MB	128MB	128MB
Memory type	DDR RAM	EDO RAM	RDRAM	SDRAM

The machines are connected through a the following hub:

3Com OfficeConnect TPC (3C16701)

B.2 Operating system and software

	Product	version
Operating system	Linux	
Distribution	Debian	3.1 (Sarge) testing
Kernel	Linux kernel	2.6.16
Kernel patch	RT Preemption Patch	2.6.16-rt29
DBMS	MySQL	5.1
DDS	ORTE	0.3.1

Appendix C

Setting up preemption

The following is a step by step guide on how to setup the real time preemption patch for the linux kernel, as it was used in this project.

This guide is inspired by this web site:
`http://tapas.affenbande.org/wordpress/?page_id=6`

C.1 Patching and compiling the kernel

1. In this test we used the following kernel and patch: `linux-2.6.16`, which you can get at `http://www.kernel.org/` and `patch-2.6.16-rt29` by Ingo Molnar, available at `http://people.redhat.com/mingo/realtime-preempt/`
2. Download the kernel and the patch
3. Unzip the kernel with: `tar xjf /path/to/linux-2.6.16.tar.bz2`
4. Go into the kernel directory: `cd linux-2.6.16`
5. Do a dry run on the patch (Dry run means the patch will not actually be executed, it's just to check whether it would execute without errors): `patch --dry-run -p1 < /path/to/patch-2.6.16-rt29`
6. When no errors are reported, run the real patch: `patch -p1 < /path/to/patch-2.6.16-rt29`
7. Copy your previous kernel `.config` to the current directory. If you enabled `/proc/config.gz` support you can just take the one from the currently running kernel: `zcat /proc/config.gz > .config`
8. Run `make oldconfig`

9. Run `make menuconfig`. See the next section to find out what to change in this menu configuration of the kernel.

C.2 Configuring the new kernel

The realtime preemption patches introduce new preemption models that allow for reliable low latency operation. When executing the `make menuconfig` as stated in the last step in the previous section, there should now be a addition to the menu `Processor type and features`. This addition is called `Preemption Mode`. It offers four choices:

```
No Forced Preemption (Server)
Voluntary Kernel Preemption (Desktop)
Preemptible Kernel (Low-Latency Desktop)
Complete Preemption (Real-Time)
```

For our tests, we choose to have real time preemption, so we choose:
`Complete Preemption (Real-Time)`

In addition to this, it is advisable to switch on:
`High Resolution Timers`

List of Figures

List of Tables