Helsinki University of Technology
Information and Computer Systems in Automation
Espoo 2003 Report 9

MIDDLEWARE SOLUTIONS FOR AUTOMATION APPLICATIONS – CASE RTPS

Seppo Sierla

TEKNILLINEN KORKEAKOULU
TEKNISKA HÖGSKOLAN
HELSINKI UNIVERSITY OF TECHNOLOGY
TECHNISCHE UNIVERSITÄT HELSINKI
UNIVERSITE DE TECHNOLOGIE D'HELSINKI

Helsinki University of Technology
Information and Computer Systems in Automation
Espoo 2003 Report 9

MIDDLEWARE SOLUTIONS FOR AUTOMATION APPLICATIONS – CASE RTPS[1]

Seppo Sierla

Abstract: This thesis has been written as a part of a research project, whose goal is to define the architecture and communication requirements of next-generation process automation systems. We focus on defining appropriate communication mechanisms for components that communicate with each other using Ethernet.

The theoretical part starts by summarizing the communication requirements that have been defined in the research project (OHJAAVA-2). Two middleware standards, the CORBA Notification Service and RTPS, are then described and their usefulness for our purposes is evaluated.

The practical part contains a description of a testing environment for evaluating a RTPS implementation. The test cases are based on communication scenarios that are typically encountered in process automation systems. The results are then presented and the impact of all relevant factors is analyzed.

We conclude that the NDDS implementation of RTPS is a very promising middleware solution for process automation systems. There is no perfect product that satisfies all of our requirements, but good results can be expected from using RTPS, if the system designers appreciate the strengths and limitations of the middleware standard.

Keywords: middleware, distributed automation system, DCS, communication requirements, RTPS, NDDS

Helsinki University of Technology
Department of Automation and Systems Technology
Information and Computer Systems in Automation

[1] Republishing of the Master's Thesis

Distribution:
Helsinki University of Technology
Department of Automation and Systems Technology
Information and Computer Systems in Automation
P.O.Box 5400
FIN-02015 HUT
FINLAND
Tel. +358-9-451 5462
Fax. +358-9-451 5394

ISBN 951-22-6604-0
ISSN 1456-0887

Picaset Oy
Helsinki 2003

Foreword

Forewords are rarely read, although it is customary to write them. The best foreword that I have seen is in The Karamazov Brothers [Dostoyevski 1988]. The author points out that there are some very considerate readers who finish a book before forming any opinion about it. He then says that it will be quite enough for the reader to work his way through the first part of The Karamazov Brothers (> 400 pages.) If this will fail to interest him, there will be no reason to start the second part. For my part, I will be very satisfied if the reader will scrutinize the first 5 sections of this thesis. He will then be in a good position to judge whether or not it will be profitable to proceed to the testing sections.

Seppo Sierla

Espoo, 22.5.2003

Contents

Terms and Abbreviations

API	*Application Programming Interface*
COM	*Component Object Model*
CORBA	*Common Object Request Broker Architecture*
DDS	*Data Distribution Service*
NDDS	*Network Data Delivery Service*
QoS	*Quality of Service*
RMI	*Remote Method Invocation*
RTI	*Real Time Innovations*
RTT	*Round-Trip Time*
RTPS	*Real-Time Publish-Subscribe*
TC	*Test Case*
TS	*Test Suite*

1 Introduction

The name of this thesis is "Middleware Solutions for Automation Systems – Case RTPS". The work has been done for the Laboratory for Information and Computer Systems in Automation at the Helsinki University of Technology as a part of the "OHJAAVA-2, Modern Distribution Solutions in Open Control Systems" research project. OHJAAVA-2 belongs to Tekes' "Intelligent Automation Systems" technology program [ÄLY]. The other participants in the project were VTT Industrial Systems (Espoo) [VTT] as a research partner and the companies Metso Automation Networks (Tampere) [Metso] and Raute Precision (Lahti) [Raute].

1.1 Background and Starting Point

1.1.1 The Architecture of a Single Node

In a master's thesis done for the preceding OHJAAVA project [OHJAAVA], the component-based architecture of a single node has been described in detail [Peltola 2002]. In particular, an interface between application components and the middleware has been defined. This is an appropriate starting point for this thesis, which focuses on the communication of components that have been distributed onto several nodes.

The architectural model of a single node will serve as the background of this thesis, and we will summarize its main elements here [Peltola 2002]. The model is not based on any existing middleware standard; rather, it has been derived from the requirements of the application (i.e. the automation system.) Some of the model's characteristics might have to be adjusted when a real system is implemented.

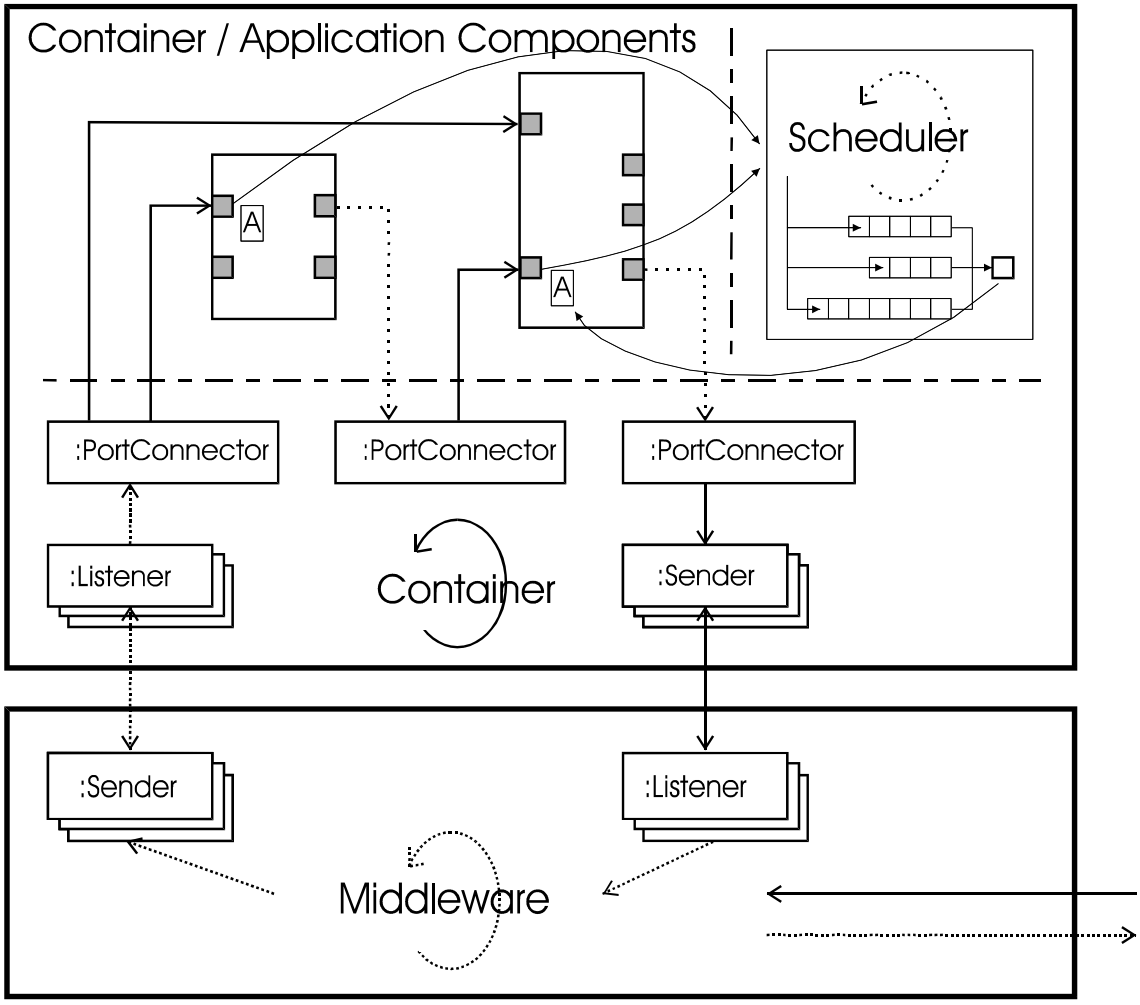

Figure 1 The container architecture [Peltola 2002]

Figure 1 shows the four main elements on each node: the middleware services, the container, the application components and the scheduler.

1.1.2 The Publish-Subscribe Model

The exchange of information is based on the publish-subscribe model, which is described at length in section 3 of this thesis. In short, all data transmission in based on the following 2 types of interfaces:

The **listener** subscribes to some data and forwards it to all of the registered recipients. These might be objects in the same application container, or a proxy in the middleware services if the receivers are in a different component or on a different node.

Senders receive data from the application component or the middleware and forward it to the connected listener.

The main reason for using senders and listeners is to decouple communicating application components, so that technical problems related to distribution are encapsulated and reconfiguration will be easier.

1.1.3 The Container and Application Components

The figure shows how application components use port connectors to join to the sender and listener interfaces as well as to each other. The reason for this is that the communication of automation components is typically modeled by connecting their ports. These are design-level concepts and they might not have any corresponding element in the actual implementation. For example, in the case of the middleware standards that are described in this thesis, all of the connections are made by calling the member functions of appropriate objects. It might be useful to define port and port connector classes.

Whatever technique is used to establish the connections, the division of objects among a *container* and application components is conceptually meaningful. The application objects manage the algorithms and data flows that control the process. Higher-level application components can encapsulate other components, so that every high-level component is responsible for managing some part of the process. From these, executable components can be created. The container objects are responsible for executing these application components using a thread with an appropriate cycle time and priority. The container also provides communication objects that are used by the application to access the services of the middleware.

Typically, the middleware will provide class libraries from which the appropriate communication objects can be created. These libraries might be linked to the application, so the container and application objects would be running in the context of the same process. The container objects are in turn responsible for accessing the middleware, which might have its own objects for handling UDP/TCP traffic. All of these details are encapsulated by the communication objects in the container.

In the sections about RTPS and CORBA, we will elaborate on how well this architecture can be implemented with existing technologies.

1.1.4 Scheduling

A multithreaded design is a natural way to implement this model, where application components and middleware entities have been decoupled. This is also a powerful tool for controlling the applications real-time behavior. The threads can be grouped into the following categories:

- *Application level* threads transfer data between port connectors and the ports of the application components. They will also be used to execute whatever algorithms have been coded into these components. A scheduler is responsible for prioritizing these tasks, so a thread pool containing threads with different priorities can be used.
- *Container level* threads are responsible for transmitting data and messages between the application components and the middleware.
- *Middleware level* threads route data from senders to listeners. These objects might well be located on different nodes, so some mechanism such as UDP sockets must be used.

Splitting the threads into 3 categories is conceptually helpful, but it must be remembered that ultimately all of these threads will compete for the same CPU. The desired real-time behavior can only

be achieved by giving each thread an appropriate priority. For example, if the container and middleware have too great a priority, the application level threads might not be executed in time. However, if all application level threads are given a high priority, a critical task might not be able to execute because the middleware couldn't supply its input data on time.
In practice, there is no best solution to this and many valid approaches exist. Elaborate prioritization schemes try to guarantee that all deadlines are met whenever possible, and that the most critical tasks are given priority. However, these algorithms will be competing for the precious CPU time. To minimize the middleware's load on the processor, some designs simply use the first-come first-serve principle. Finally, it must be remembered that the nature and reliability of the scheduling is ultimately dependent on the underlying operating system.

1.2 Goals

1.2.1 The Goals of OHJAAVA-2

" *The goal of the OHJAAVA-2 project, Modern Distribution Solutions in Open Control Systems, is to define the technical requirements of a communications layer (middleware) in a distributed control system. In addition we pursue to gain knowledge about ongoing standardization activities and commercial and open source product offering in the area.*
From the point of view of the requirements discovered in the project, we evaluate the middleware standards and technologies available and pick some of them to a more elaborate analysis. The most promising ones will be tested in a test bench built to our and VTT's lab environments. Potential of the evaluated technologies for commercial product development will be estimated. " [OHJAAVA-2]

1.2.2 The Goals of the Thesis

The goals of the thesis are a subset of the goals of the project. One of the first tasks of the project is to define requirements for the various application-level communication mechanisms that are used by the components of an automation system. My contribution to this work is very minor. The next task is to gain knowledge about the technologies that can be used for this purpose and to write technology description documents for the most promising ones. I am responsible for the RTPS (Real-Time Publish-Subscribe) document [OHJAAVA-217] and for a significant part of the Real-Time CORBA document [OHJAAVA-220]. My main responsibility in the project is related to the testing of the most promising products. The tests are based on a technology-independent test specification, which describes test arrangements and the test suites [OHJAAVA-216]. The suites in turn are based on the requirements that are captured in the first part of the project. The NDDS implementation of RTPS has been selected as the most interesting product that is available for testing, and I am fully responsible for testing it. The NDDS test specification is derived from the technology independent specification [OHJAAVA-227]. These specifications have provided the requirements for the tests components. My task is to design and implement these components and use them to run the specified tests.

1.3 Scope

The focus of the preceding thesis [Peltola 2002] that was done in the OHJAAVA-project was on a single node and the various components running on it. This work concentrates on the mechanisms that the components can use to communicate with each other. In the OHJAAVA-2 project, the necessary mechanisms and their quality of service (QoS) requirements are analyzed. This results in a logical description of the design-level mechanisms. These must be mapped onto some concrete mechanisms that are provided by some existing standard or product. The scope of this thesis is focused on the following tasks:

- Write descriptions of some of the most promising technologies (RTPS and RT CORBA)
- Define a technology independent test specification, with traceability to the logical communication requirements
- Implement product specific test specifications and test software
- Run the tests and analyze the results

1.4 The Structure of the Thesis

The thesis consists of a theoretical and a practical part. The theoretical part (sections 2-5) discusses the communication requirements for modern automation systems and describes 2 existing standards. The practical part (sections 6-8) has a description for a testing environment that is based on these requirements and an analysis of the results that were obtained in the tests.

Section 2, (Analysis of Communication Requirements) describes the communication requirements for modern automation systems and our architectural vision.

Section 3, (CORBA Notification Service) describes one promising middleware solution whose background is in the IT industry. We describe how well the requirements in section 2 can be satisfied by the notification service.

Section 4, (RTPS) describes the RTPS standard as an alternative to the more familiar CORBA.

Section 5, (The Container Design Pattern) discusses some of the shortcomings of RTPS and CORBA and proposes an application level solution to them, which is based on our architectural vision.

Section 6, (Test Arrangements) describes a testing environment which will be used to test RTPS in a number of communication scenarios that are typically encountered in automation systems. There are 2 test suites for functional tests, which exercise the key communication mechanisms. After these, there is a scalability test suite, which is used to observe how the performance deteriorates as the number of communicating entities and the volume of traffic is increased.

Section 7, (Functional Test Results) Presents the results for the functional test suites. The impact of the various factors that affect the results is analyzed.

Section 8, (Scalability Test Results) presents the results for the scalability test suite. Any differences to the functional test results are pointed out and the causes are analyzed.

Section 9, (Conclusions) has a discussion about the strengths and weaknesses of RTPS as a general-purpose middleware. Then, some special requirements for process automation systems are reviewed and the strengths and limitations of RTPS with respect to these are described.

1.5 The Methodology

1.5.1 The Nature of the Task

The evaluation of middleware solutions is somewhat different from most software engineering tasks. Typical methodologies are aimed at producing an application that satisfies the requirements of some user group. Testing projects usually try to verify that an implementation conforms to a specification. Here we are evaluating the suitability of existing middleware products, given the communication requirements of process automation systems. Therefore, we do not systematically test whether NDDS satisfies the RTPS specification, since RTI must have done that much more thoroughly than is possible within the timetable and goals of OHJAAVA-2. Rather, our task is to design and implement a test environment, which will exercise the middleware's capabilities in the same way as a real automation system would do.

Whatever methodology is chosen, it must be adapted to the circumstances. Before describing the chosen solution, we point out the following circumstances:

- Much of the work (e.g. all the design, implementation and running of the test components) was done by one person, so the methodology here should be lightweight.
- The specification, design and result documents are used to communicate the progress of the work to others.
- The project already has some conventions about documentation. These have in turn been derived from VTT's quality system. Each task has a corresponding document, and the project has guidelines for the structure, content and format of these documents.

1.5.2 The Workflows

Nevertheless, the methodology that is used here has much in common with the usual ones. The work can be divided among workflows that consist of activities. An iterative approach was used, so that in each iteration, work was done in one or more tasks (workflows). We now describe each workflow; this has been written after the results were obtained, when we know all the tasks that were actually carried

out. As we mentioned in the previous section, each task (apart from learning) has a corresponding document, whose purpose is to

- communicate with other participants
- document the results
- serve as input for other tasks
- document any modifications that were made afterwards

Requirements analysis: The requirements for a middleware standard are based on the logical communication requirements of a process automation system. Based on the many different scenarios that we have identified, we describe suitable communication mechanisms for each of them. The appropriate reliability and QoS requirements are also recorded. This work is done together with other researchers and industrial contacts.

Technology analysis: The requirements have been defined from the point of view of the application, but some existing standard and product must be eventually used. The main goals of the analysis task were to:

- learn the background and intended use of the technology
- describe the available communication mechanisms and QoS properties
- evaluate standardization, interoperability and commercial aspects

Learning to use the technology that is to be tested: Based on the initial results of technology analysis, NDDS and the TAO implementation of CORBA were selected for practical testing. At this point only simple demonstrations were built, since the goal here was to become familiar with the possibilities of the products before planning any tests in detail. At this point it was noticed that clock synchronization must be dealt with before any accurate measurements can be obtained. We also identified the OS as a significant cause of unpredictable behavior. Our 100Mbps Ethernet performed reliably, predictably and efficiently relative to the processing that was done under the Windows and Linux operating systems. This knowledge was useful later in focusing the tests.

Test specification: Based on the requirements analysis, a general test specification was developed. Its goal was to define tests for the key communication mechanisms, bearing in mind the QoS requirements. Technology dependent specifications were derived from this, so that it would be straightforward to implement the tests. The test specifications can be considered successful, if they exercise the middleware in realistic situations that cover the most important scenarios that occur in an automation system.

Designing and implementing the test components: The tests were defined in terms of components that generate traffic, which corresponds to the traffic in an automation system. The inputs to this phase are the test specifications, from which the requirements for the test components can be identified. Success in this phase is not simply achieved by building components with the desired functionality. There are

many ways to achieve this, and they can be quite different in their real-time behavior. These differences are not always obvious even when good documentation is available, so some experimentation with the alternatives was necessary.

Running the tests and analyzing the results: Before the final tests could be run, the participating nodes (2 Windows and 4 Mandrake Linux) had to be prepared and equipped with the test software. When tests were run, many MB of raw data was created, so this had to be processed and analyzed. Although some preprocessing was done in the test components, Excel was used for further processing and visualization.

1.5.3 The Timing of the Tasks

Figure 2 shows how much effort I spent on the different tasks (workflows) during the 5 iterations. We did not have formal iterations with starting and ending dates, rather we use iterations to refer to different stages of the project. In each iteration, work was done for some of the tasks, and each task depends on the work done on several other tasks during the previous iterations. Since the workflows are dependent on each other, the success of the whole thesis depends on timing the progress of each workflow properly. For example, test specifications are unlikely to be successful if requirements and technology analysis have not been carried out in sufficient detail. However, there are also circular dependencies: experience from implementing and running test components might force us to adjust the test specifications and to update some technology description documents.

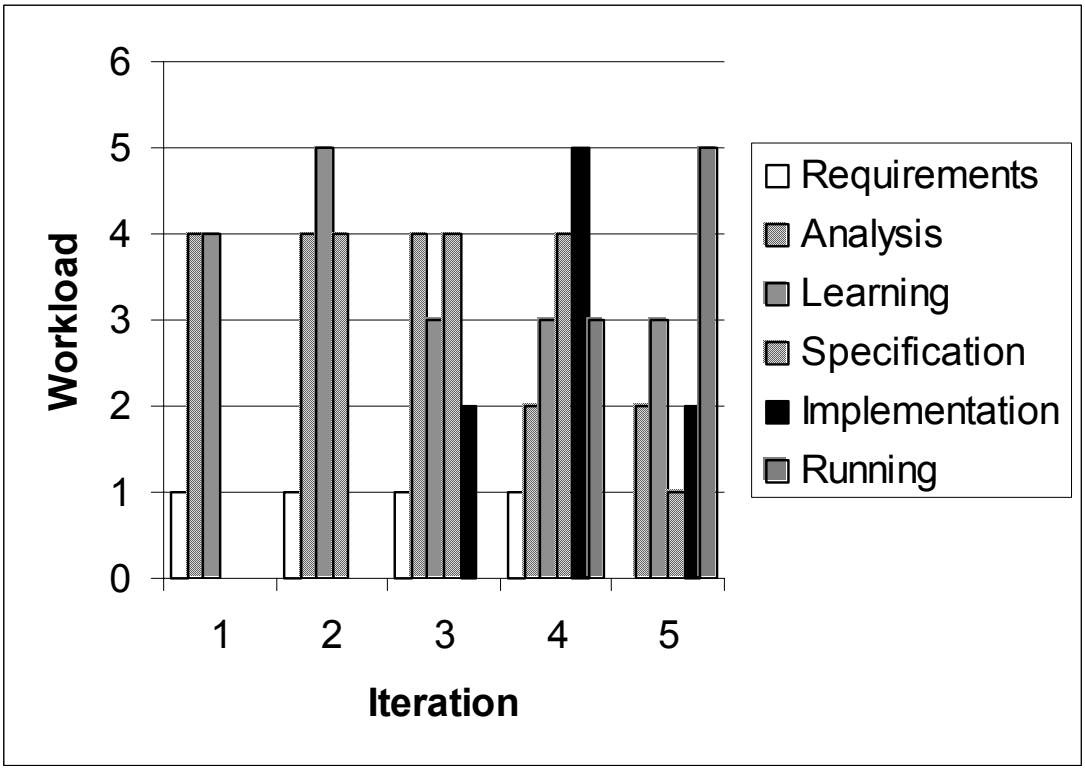

Figure 2 The workload

On the whole, the chart does not have any surprising information. Analysis and learning are mainly carried out in the beginning of the project, specification activity is heavy during the middle part and implementation and running are left to the later stages of the project. However, it is evident that no waterfall model has been applied, since there is significant overlap in the tasks. This was intentional; we did not have much experience from similar work, so an iterative approach was required: the experience from any task could be used by other tasks in subsequent iterations *of the same project.*

The low effort for the requirements task is justified, since others in the project were responsible for that. However, the chart indicates one problem in the project: not enough effort was put into requirements analysis early on. This made it more difficult to carry out the other tasks; the problem is elaborated in the risks section.

Now that the project is over, we can say that the methodology was well chosen. Each of the 6 tasks described above were important, and a problem in any of them could have prevented us from reaching the goals that were set for this thesis.

We finally point out that since the chart is based on my working hours, most of the effort has been spent on NDDS and some was spent on CORBA. CORBA was chosen for testing only in the 4th iteration, so there is a significant amount of effort that was spent on analysis, learning and specification towards the very end of the project. Normally this would be undesirable, since the main work of these tasks should be done well before the end of the project.

2 Analysis of Communication Requirements

The starting point for our evaluation of middleware solutions is our architectural vision for next generation automation systems. The architecture and communication requirements are described from the perspective of the application designer; these requirements have been obtained from discussions with researchers and Finnish companies in the process automation industry. We will discuss the role of the middleware platform over the lifecycle of a product, the communication requirements of the application components and the architecture of the middleware services. We emphasize the communication requirements and will develop tests for these. The focus of this thesis is on testing, so these requirements are summarized briefly [Peltola 2003]; more detailed information is available in [OHJAAVA-212].

2.1 The Role of Middleware over the Lifecycle

The benefits that middleware vendors typically claim for their products usually cover many phases of the lifecycle. Indeed, in order to reap these benefits, the deliverables of one phase should be the expected inputs of the next one. Otherwise, if conceptual and structural changes are needed, the original design principles will be violated and the benefit of quick development cycles will be lost. For example, it should be possible to implement the design-phase entities and communication links in a straightforward way (or with automatic code generation). The programmer should not be engaged in coding workarounds for technical obstacles, and neither should he need to adapt the design according to the constraints of his tools. It should be possible to define an architecture that can be modified or extended later without compromising the original design principles or tampering with the implementation of existing components.

We must distinguish between the lifecycles of automation systems and automation applications. With the latter we mean the result of a project that solves a client's problem. This involves the control algorithms for the instruments in that particular process and the data flows between application components. The automation system is responsible for executing the application and using the middleware services to establish the communication paths. Therefore, one version of the automation system can be used in several deliveries of automation applications. The application components and large parts of applications are also reused in later projects whenever possible.

We now describe the role of middleware in some key phases of the lifecycle. Only the automation system directly uses middleware services. However, the demands of the application's lifecycle place requirements on the automation system, and it should try to meet these by making good use of the middleware. The following phases belong to the application, i.e. a client project. We describe the ideal scenario, where developers do not encounter technical details and complications. This is possible if the automation system provides appropriate services, and the system will in turn rely on the middleware to implement these.

Specification: The specification of the application's functionality should be done in terms of design-level communication mechanisms. Quality-of-Service (QoS) requirements are also listed. The assumption here is that the automation system will provide corresponding mechanisms. These in turn rely on the implementation-level mechanism of the middleware product. Therefore, the specifications should already use constructs that the automation system provides, and the system should in turn provide constructs that application designers need.

Design: The design of the application should correspond to the structure of the process that is being automated [IDA 2001]. The application will then be built from components that correspond to certain parts of the process. The communication among components should therefore be based on names that correspond to entities in the process, such as measurements and control signals. Interaction among components is defined solely by these names using some of the design-level mechanisms. The automation system can easily provide such services if the underlying middleware supports a similar naming model.

Implementation: Implementing an application is easy, after it has been designed with the appropriate tools. The automation system's components are used to execute the application and establish the communication paths.

The usefulness of a middleware product is really tested when the time comes to implement a version of the automation system. Some vendors even provide tools with automatic code generation for this purpose. The cherished design-level concepts must be implemented in software, e.g. with components and objects. This should be straightforward if the design has been specified with concepts that the middleware supports. However, since middleware standards' have different backgrounds, they also assume somewhat different ways of modeling the application. If developers use middleware that does not support their model, the implementers might be forced to 'break' their paradigm. This is always error-prone and will complicate issues in later phases of the lifecycle. If appropriate middleware is available, it will be easy to build the components and define their interactions and real-time behavior with the implementation-level communication mechanisms.

Modifications and reconfiguration: It is not uncommon that an application must be modified to satisfy newly discovered requirements. In the case of process automation applications, a successful vendor will sell the same components to many customers. However, this requires much tailoring in each case, and modifying either the interface or implementation of existing components is never easy. A good automation system will encapsulate all technical details, so application components can be modified or moved with no or minimal reconfiguration work. Any communication links are based on logical names, so the application structure is decoupled from technical details. Because of this decoupling, it is possible to move existing components to different locations without making modifications. Further, since communications are defined in terms of process-related names, the same components can be configured for a different process by supplying the appropriate names as parameters. However, the system components can only support this level of flexibility if the underlying middleware has these capabilities.

2.2 The Communication Mechanisms

Figure 3 illustrates the most important mechanisms that will give the best support for designing process automation applications. No existing product satisfies these requirements in full; we will discuss some of the restrictions in sections 3 and 4. In section 6 there are test suites for exercising these mechanisms. At the beginning of each suite, we refer to requirements in this section and also summarize the main QoS requirements from [OHJAAVA-212].

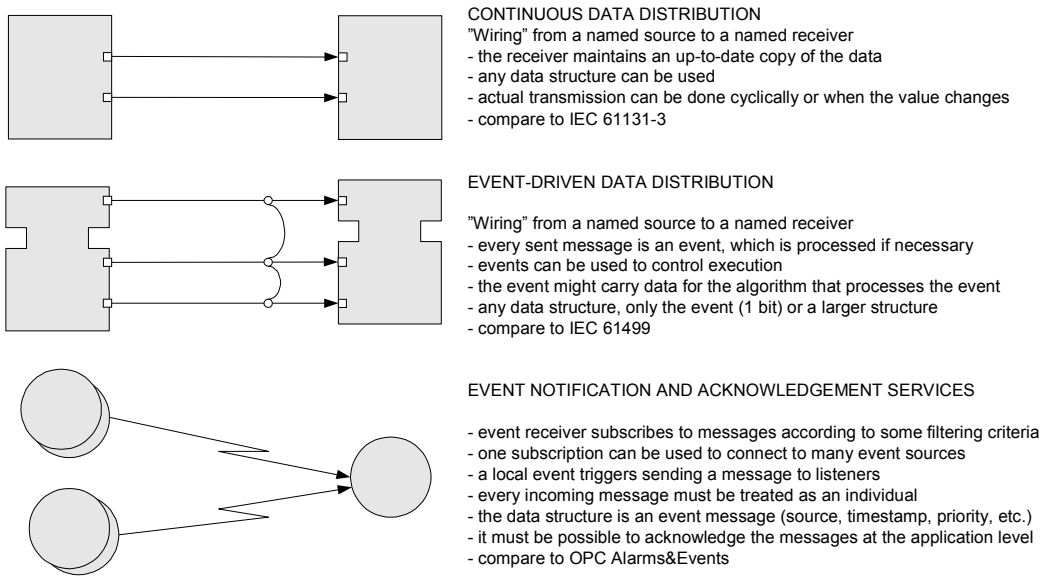

Figure 3 The key communication mechanisms [OHJAAVA-212]

2.2.1 Continuous Data Distribution

This mechanism is familiar to anyone who has programmed PLCs using function blocks or designed IC circuits. A connection is made between the output and input ports of two blocks by "wiring" them. The signal value on both ends of the wire should always be the same. When computer technology is used, an illusion of wiring is created, by having the producer transmit a fresh data value frequently. The required frequency depends on the dynamics of the process, and the shortest cycles in modern systems are usually around 5-10ms. However, in automation systems there are many tasks whose period is hundreds of ms, several seconds or even minutes.

In the basic case, the wire is connected to one receiver, but the continuous data distribution mechanism should be able to handle any number of receivers.

This mechanism is typically used for the real-time control of a process. The control algorithms have been designed to operate with a certain period, so the untimely delivery of data can result in unstable behavior. Therefore, continuous data distribution places the most stringent requirements on latency

and jitter. However, a slightly (e.g. several ms) longer latency is usually acceptable in order to minimize the jitter. The delivery should be reliable, but missing a delivery is typically better than delaying everything with retransmissions.

The actual transmission can be done cyclically or when the value changes significantly. The latter is preferable if the data structures are large (e.g. 100kB), but this requires guaranteed delivery.

2.2.2 Event-Driven Data Distribution

The event-driven programming paradigm is a powerful approach for developing automation applications, but it requires suitable communication mechanisms. Data ports are wired as in continuous data distribution, but the transmitted message is an event instead of some signal value. An incoming event will trigger the execution of an algorithm that is associated with the input data port.

This algorithm will usually process some input data; this might be carried by the event message, or it might be wired to other input ports. It is important that all of the input data is coherent, e.g. from the same sweep of an earlier algorithm. Whichever way the event-driven data distribution mechanism is implemented, it should guarantee the coherence of the data.

Since the execution of the application is controlled by events, missing even one of them might have very serious consequences (in the worst case, some critical task will never be executed since it will wait forever for an event that was lost.) The mechanism should therefore guarantee the delivery of every event to all receivers. Even so, application designers would do well to give some thought to the consequences of missing an event, because the network might be down for a long time.

2.2.3 Event Notification and Acknowledgement Services

Although this mechanism is also used to transmit events, its nature is different from event-driven data distribution. The latter is used to control the normal execution of the application. In the case of event notifications, we are sending alarm and notification messages to all interested receivers. These events are generated when the state of a component changes in some significant way. Sometimes, the messages are only sent to inform an operator, or they might be stored in a history database. An alarm might force the system to stop its normal operation and embark on a special course of action (such as open an emergency valve or stop the process safely.) The reliable delivery of messages is again required, since missing even one alarm can have very serious consequences. We can see that there is a separate domain of events that are not involved in controlling the application's normal execution, but can nevertheless interfere with it. This design approach has proven its usefulness with automation systems, since they are large and complex entities, which must be prepared to cope with a great number of potential problems.

Another feature of event notifications is that they must be acknowledged from the application level. This will become complicated when there are several receivers. Many acknowledgement models exist; for example, it might be enough for one operator to acknowledge the event, or then a response from some or all of the receivers might be required.

The designer needs a mechanism for event notifications, because the event-driven data distribution mechanism is lacking in two ways. First of all, there must be a flexible mechanism for subscribing to a large number of events from many different components. For example, we might ask for all temperature related alarms from a certain process area that have priority medium or greater. Subscribers to the event-driven mechanism will specifically name the desired event. Secondly, if there is inadequate support for application level acknowledgements, the system developer will have to handle these by adding much functionality into his design of the automation system.

2.2.4 Other Mechanisms

We have identified a need for 3 other mechanisms, but these are simpler and the requirements related to them are easier to fulfill. Therefore we have not developed test for them, and will only briefly describe them here.

Request/reply can be used to obtain some data or to start some service. A typical use of this would be to read or modify a parameter of a controller. This kind of a mechanism is easier to implement because of its simplicity and the looser real-time requirements. Reliability is desirable, though. The name or location of the service might not be known at compile time, so some of the server components might advertise their interface in some naming or directory service.

The reply might be asynchronous, so a callback routine is invoked when the reply is received.

Remote read/write is a simpler version of request/reply, since no other functionality is performed beyond writing or reading a parameter. The designers might make the assumption that the operation is always successful. In the future, an exception handling mechanism should be required, and many implementations of request/reply (e.g. CORBA) already provide this.

Message-oriented communication decouples senders and receivers. Messages are sent to a mailbox or channel, so the sender does not need to know anything about the receivers. Receivers will in turn retrieve the messages from a channel, or they will filter them with some criteria.

2.3 The Structure of the Middleware Services

There are two common middleware architectures: peer-to-peer and centralized. In the centralized architecture, a server keeps track of all the communication links and routes the data and messages to the correct receivers. This approach is neither scalable nor efficient, because the server will easily become a bottleneck and there is no direct communication between the senders and receivers. Fault-tolerance is also a problem, because a failure of the server will bring down the whole system.

The peer-to-peer solution is based on having middleware services on every node. These maintain up-to-date information of communicating parties on other nodes and provide direct routing of messages to the receivers. A failure of one node will only affect communication links with an endpoint on that node.

The pure centralized solution is considered unacceptable for automation systems. The peer-to-peer architecture is in many ways ideal, but partially centralized solutions can also be considered.

In either case, an API for using the communication mechanisms must be provided. This will naturally encapsulate all of the lower level details related to the OS and network stack. Therefore, it is not important how the mechanisms are implemented. The TCP retransmission mechanism and flow control can be used, or an implementation could provide its own on top of UDP. Some products will rely on IP and Ethernet, but there is support for other transports as well.

Some services are also required. It is often necessary to dynamically discover the location and sometimes even the interface of a service. Support for redundancy is also useful, although there are many redundancy models, so the middleware might not provide a suitable one. Security issues must be dealt with somehow. When the process control level is integrated vertically to higher-level management systems, security risks increase dramatically. A security architecture with all the necessary mechanisms is required from a product that will be used for vertical integration.

Middleware is usually designed to be used with other common Internet technology. HTTP-based methods enable some data or functionality to be accessed with web browsers. FTP can be used for file transfer. Common Internet security techniques can be used, but this will be much easier if the middleware is designed to work with them.

Most middleware standards aim at accomplishing some tasks as well as possible. Indeed, middleware that could be used under any circumstances would not perform optimally in all situations. The architecture of the middleware, along with the available communication mechanisms, will constrain the scope where the middleware should be applied. For enterprise-wide solutions, a well chosen mixture of one or more middleware products and other Internet techniques will be necessary.

3 CORBA Notification Service

3.1 Introduction

Some existing technology must be used to implement the communication mechanisms that were described in section 2. Currently, no product offers enough functionality to fully implement all of the desired design-level mechanisms. The mechanisms that are available depend on the background of the technology and its intended user groups. The choice of any middleware product will constrain application developers significantly, so it is necessary to have a clear understanding of the potential and limitations of the available alternatives. These constraints must be well understood by the designers, because otherwise implementers will be forced to break the design.

In this section, we describe CORBA notification service. The technology is easily understood by anyone with a background in computer-science, since it is based on the RMI (Remote Method Invocation) mechanism. Being well known and having numerous successful applications in many different fields, CORBA will serve as an introduction and comparison to RTPS, which is the main focus of this thesis.

We assume that readers are familiar with basic CORBA; good introductory material and tutorials are available at [OMG]. The main idea behind RMI is that a distributed application consists of objects that either provide services or use them. The services are described with an interface that is similar to a class definition, but it is written in IDL and is therefore independent of any programming language. Servant objects that implement these interfaces are run on some server node, and a globally unique object reference is assigned to each reference. Clients can use these references to invoke methods on the servants. This is done just like an ordinary method call using a proxy in the client's address space. The call and response are routed to the correct recipient by the ORBs (Object Resource Broker) on each node.

The IT industry has noticed that this approach is very helpful in a variety of application areas. However, the basic mechanism is unsuited for process automation for 3 reasons. First of all, automation applications are modeled by wiring function blocks in order to describe the data flows and the algorithms that process the data. Switching to a UML based modeling method would require considerable retraining from the automation engineers. We do not see any inherent limitation why object modeling could not be used as well as function blocks, but suitable frameworks are needed for this industry. These can only be developed by engineers with a solid understanding of object modeling as well as automation systems. The second problem is efficiency. When algorithms are executed periodically, thousands of inputs and outputs must be read and written every second. Using RMI for each of these is very time-consuming. Thirdly, RMI will couple clients and servers tightly together, limiting the flexibility that is desired from middleware.

The notification service specification [OMG 2002] is aimed at solving these problems. It is a good attempt to implement communication mechanisms such as those described in section 2 with object

technology. It is unlikely that automation engineers can ignore the advances in the IT-field indefinitely; indeed, many of the most successful inventions, such as higher level programming languages, have already been adopted. As the demands for vertical integration grow, the benefits of using object technology at the lower levels of plant automation should be obvious.

3.2 The Notification Service Architecture

With the event service, OMG has used its basic RMI mechanism to build higher-level communication services. The notification service [OMG 2002] has extended this work and provides QoS support using the Real-Time CORBA specification.

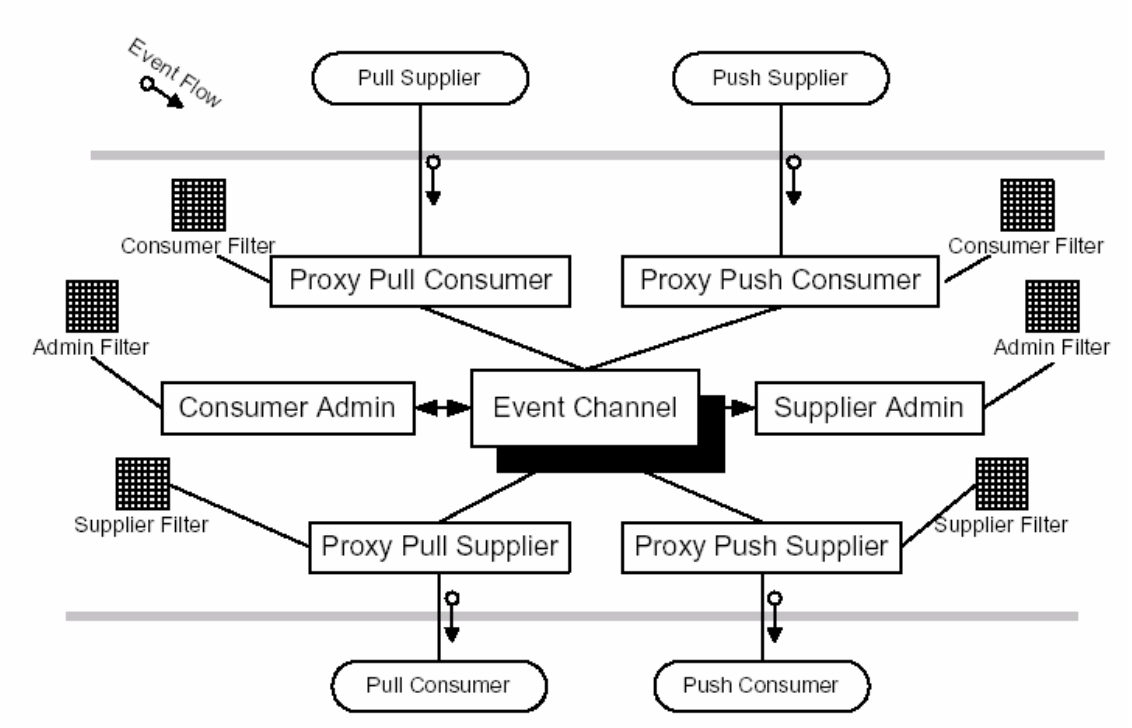

Figure 4 The Notification Service architecture [OMG 2002]

Figure 4 shows the notification service architecture; the horizontal lines separate the middleware services from the container objects. The key element is the event channel, which decouples suppliers and consumers [Peltola 2003]. Suppliers can connect to the channel and provide data to it without having any knowledge of the location, number or nature of the consumers. The consumers in turn connect to a channel and can receive all of the messages that have been supplied to the channel. A Structured event-data type can be used for messages, and much useful information can be stored in its fields. For example, signal name, id, priority, severity and process area are examples of fields that could be useful for application designers. Unwanted traffic can be filtered according to the values of certain fields; filters can be set at many levels in the notification architecture. The event channel

objects also have QoS properties that can be used to prioritize the traffic and use the resources in a way that is optimal for the application.

The notification architecture is typically centralized, and we pointed out the problems with this in section 2. However, it is possible to start notification services on several nodes or even on each node. Clients will still have to resolve the name of the appropriate channel(s) in order to receive the desired messages, so this solution is not as transparent and easy to reconfigure as we would like. A good application-level framework could provide an acceptable workaround to these problems.

We will finally evaluate the architecture with respect to the architectural vision described in section 1.1. The event channel and the supplier and consumer proxies belong to the middleware services. These are run in the notification service's process, which might be on another node. The supplier and consumer objects belong to the container and occupy the same address space as the application objects. The link between the container and the notification service is established by connecting the supplier and consumer objects to suitable proxies. This can be done when the container has obtained the proxy references from the channel. A reference to the channel is found using the naming service when the name of the channel is known. Now the container has a consumer and supplier that correspond to the listener and sender in section 1.1. These have methods for sending messages or handling received messages, so a simple interface is available to application objects.

3.3 The Communication Mechanisms

We will now evaluate the suitability of the notification service for implementing the key design-level mechanisms in section 2 [Peltola 2003].

3.3.1 Continuous data distribution

Conceptually, the event channel can be used for this quite naturally. The suppliers can push data to the channel at the rate that they produce it. Consumers can receive any subset of this data by setting the appropriate filters. Therefore, we have a publish-subscribe mechanism that was introduced in 1.1.2 and is explained in more detailed in the section for RTPS. As we said, OMG has used its RMI mechanism to communicate with the event channel, so distributing large quantities of data in this way is inefficient. OMG engineers have admitted this and are working on a DDS (Data Distribution Service) specification, which should be published in the summer of 2003 [Wang 2002]. This is done together with the developers of RTPS, so DDS should combine the strengths of the Notification Service and RTPS.

3.3.2 Event-Driven Data Distribution

The Event channel is naturally best suited for transmitting events. The intended receiver can be indicated by a value in one or more fields of the structured event, and consumers can filter messages with this value.

The algorithm that processes the event will usually require the latest values of some input signal, and these values must always be from the same sweep of the algorithm that computed them. One solution is to include the signals in the structured event, but this will lead to complications, when there are other consumers that need these signals. When new versions of the automation application are developed for new clients, the signals that are needed to process the event might be different. The structured event is flexible enough to support the dynamic addition of fields for the required signals, so maintenance problems can be avoided by sophisticated coding.

The heavy algorithms in the notification service increase the latencies in event transmission. The filtering times are especially unpredictable. The OMG is working on a Real-Time Notification Service specification to solve this problem.

3.3.3 Event Notification and Acknowledgement Services

There is no support for acknowledgements, but otherwise the notification service is better suited for this mechanism than the previous two. The flexibility of the channel can only be appreciated when consumers try to receive events from many suppliers on different nodes. The structured event provides good filtering possibilities, since an arbitrary number of filterable name-value pairs can be used. The events and alarms in an automation system can have many fields, so that the value of each field is a leaf in a tree-like hierarchy. The only shortcoming of the structured event is the lack of support for multiple hierarchies.

4 RTPS

4.1 Introduction

The distinguishing characteristic of RTPS is that it provides a practical, flexible and efficient publish-subscribe communication mechanism. Signals, alarms, notifications and events are identified with unique topic names, such as 'pressure' or 'temp' in Figure 5. Every time that a producing component generates a new value for a topic, it publishes an issue for that topic. The RTPS middleware will then route the issues to all subscribers, who have expressed their interest in the topic.

Publishing has been made as easy as possible. A publication is created with a topic name and an object with a suitable data type is associated with it. Optional QoS properties can also be specified. A new issue is sent simply by calling the send method of the publication; this will send the current value in the associated data object.

At the receiving end a subscription is created for the topic. A listener class must also be defined, since it will have the routine for processing the received issues.

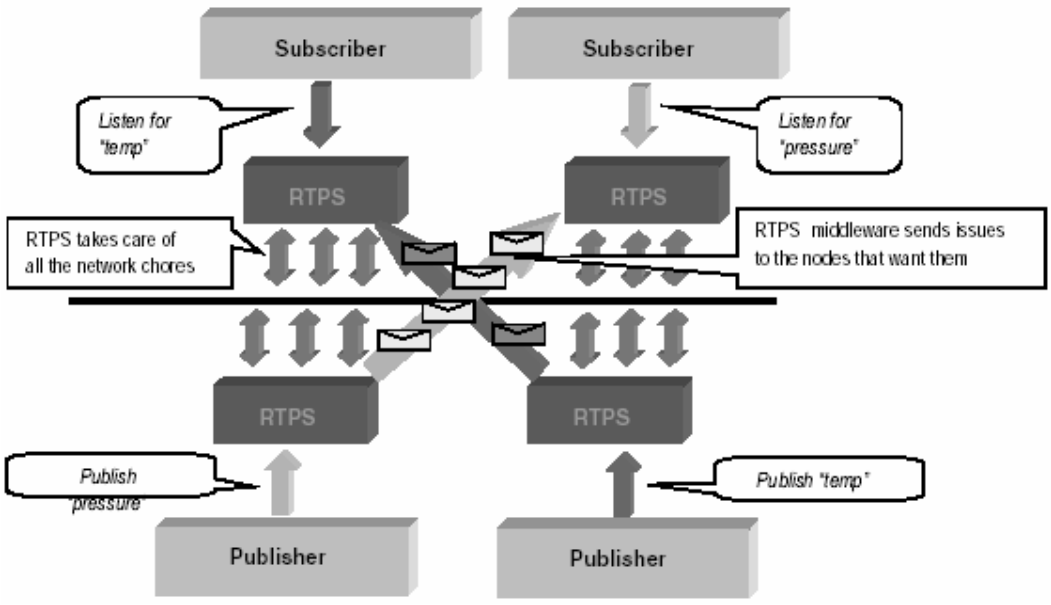

Figure 5 The publish-subscribe principle [RTI 2002]

4.2 Architecture

RTPS is a protocol specification, so it leaves implementers a certain amount of freedom as far as the architecture is concerned [Wang 2002]. In these cases we describe NDDS, the only implementation.

RTPS uses a pure peer-to-peer architecture; each node has a manager that keeps track of the topics that are published and subscribed to on the different nodes. The managers communicate with each other in order to keep this information up-to-date as publications and subscriptions appear and disappear. The manager is run in its own process and it communicates its knowledge to all of the NDDS components that are running on that node. Linked into these is all of the functionality needed to send and receive issues with UDP [Wang 2002].

RTPS has been designed to coexist with other standard Internet technology [IDA 2001]. Its purpose is to provide high-level communication mechanisms for real-time traffic. Figure 6 shows the place that RTPS occupies in the OSI model. HTTP, FTP, SNMP and other protocols can be used for the tasks that they are well suited for. An API provides access to the communication objects, such as publications, subscriptions, listeners and QoS property objects.

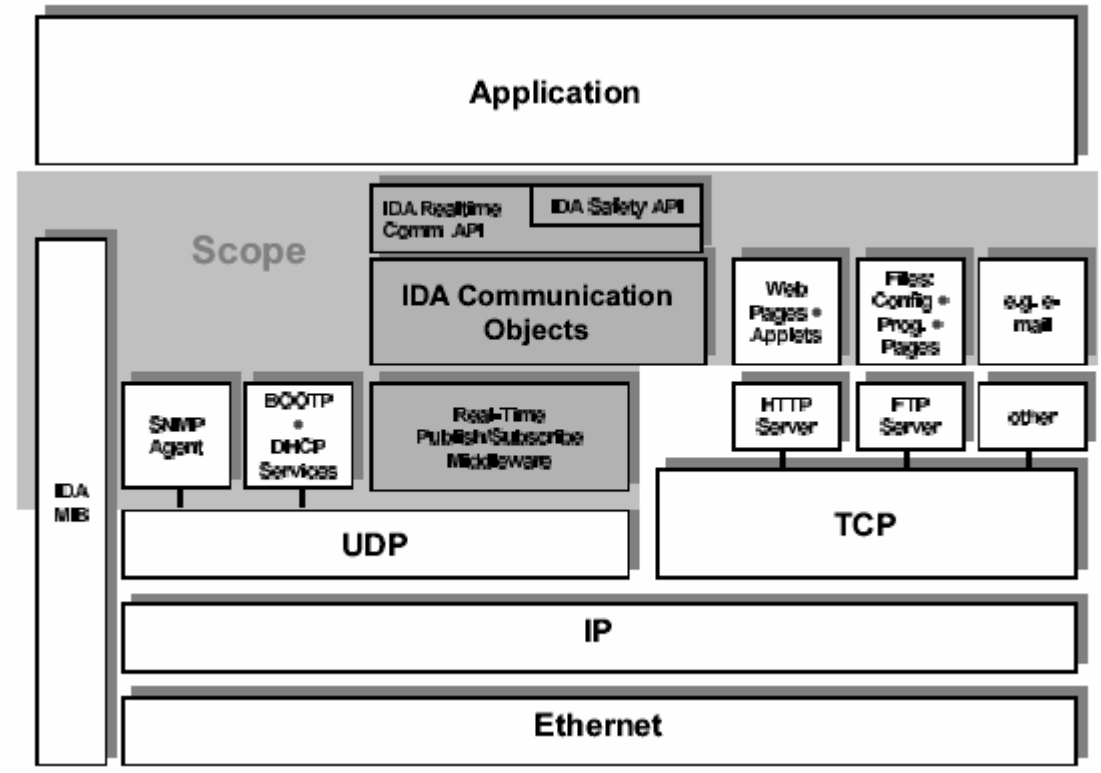

Figure 6 The IDA architecture [IDA 2001]

The RTPS architecture is similar to our model in section 1.1. The communication objects correspond to the listener and sender in the container. Above these is the application, which uses these to send and receive the data using only logical names. The middleware services are provided by the manager as well as the libraries that are linked into the application. The only difference is in the use of threads. NDDS has receiver and sender threads that handle the UDP communication (the middleware services level). However, the receiver thread also executes the listener's method in the container and can perform application-level issue processing. The sender thread will send the issues that are given to it at

the container level, as expected. However, in the synchronous mode, the caller of send (a container thread) will execute the UDP sending routine. These details should be remembered by the designer of the container, since they can have an effect on real-time behavior.

4.3 The Communication Mechanisms

4.3.1 Continuous Data Distribution

The publish-subscribe mechanism is well suited for continuous data distribution. The use of logical topic names makes it easy to deliver data to receivers at different physical locations. For example, a measurement might be needed by the controller, control room and history database, but it is enough to publish it once without knowing anything about receivers. Subscriptions can be created for each of the 3 client applications. If some new functionality is added later, for example a diagnostics service that needs the measurement, it is enough to create a new subscription to this topic without modifying any existing functionality. Moving the components to different nodes will not require any code changes either.

Publish-subscribe satisfies the real-time requirements for continuous data distribution better than any other mechanism that we have studied. The design uses UDP to send the issues directly to the subscribing applications, and the overhead from any RTPS-related processing has been minimized. Latencies are typically under 1ms with moderate CPU load and a 10-100Mbps Ethernet. No sophisticated algorithms such as sending issues based on some priority or deadline values are used here, because they would increase the latencies and CPU load [Wang 2003].

Figure 7 illustates some of the main QoS properties that can be set using the publish-subscribe API [RTI 2002]. The leftmost line on both timelines indicates the time-point when the last issue was received.

The upper timeline illustrates the behavior of an ordinary subscription. The minimum separation property specifies an interval after receiving an issue when new issues will be discarded. This can be used on a subscription that does not want a measurement value every 100ms, e.g. a history database. However, a new issue is expected to arrive before the deadline has expired. Otherwise, the application is notified. This is the only determinism that RTPS guarantees.

The lower timeline shows a subscription for a topic with several redundant publications. Each publication has a strength topic, so that the subscription can choose issues from the strongest one. The persistence of an issue indicates how long the data is valid, so during this time period only a new issue with a greater strength will be delivered to the application. After the persistence of the last issue has expired, any incoming issue will be accepted, regardless of its strength.

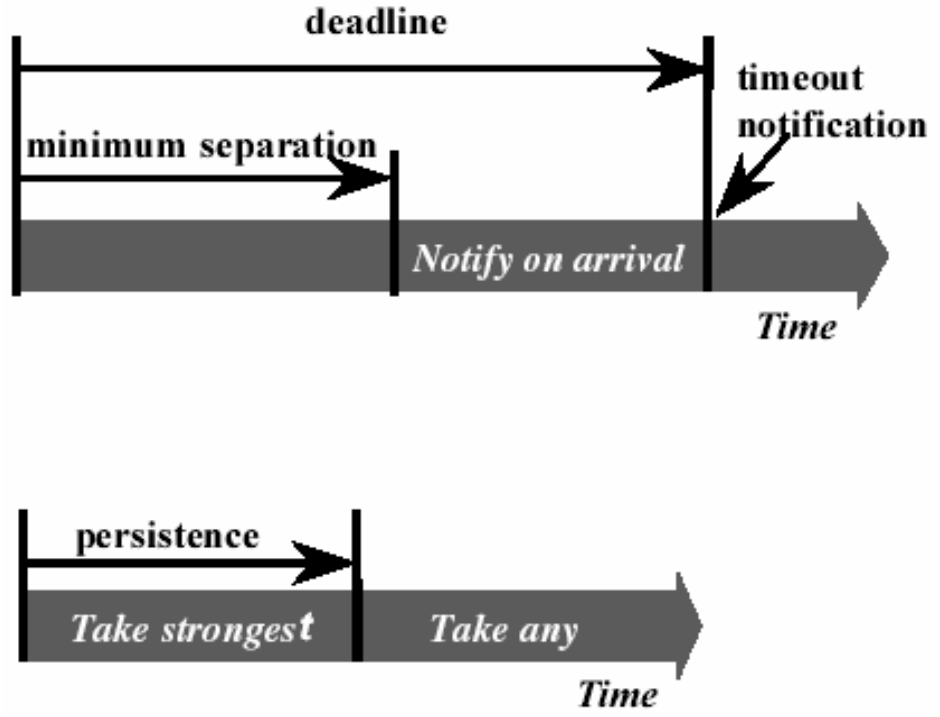

Figure 7 RTPS QoS properties [RTI 2002]

4.3.2 Event-Driven Data Distribution

RTPS also provides a reliable version of publish-subscribe. This uses send and receive queues, acknowledgements and retransmission to guarantee that every issue is delivered to every subscription in the same order as they were sent. The mechanism is very similar to how TCP delivers packets, but it has been implemented on top of UDP and lets the application programmer set many parameters that control reliability, determinism and memory usage.

Reliable publish-subscribe can be used to implement the event-driven data distribution mechanism. The 'wiring' from an event output port to one or more input blocks can be made conveniently by defining a topic name for this purpose. The listener for this topic can trigger any algorithms that process the event.

The only difficulty here is the integrity of the input signals that the algorithm will use. The situation is similar to CORBA: the event message can be a data structure that also contains the signals, but this leads to maintenance problems. An application level solution to the integrity problem is suggested in section 5.

The QoS support for event-driven data distribution is very adequate. On an Ethernet, lost issues are very rare, so the latencies are similar to ordinary publish-subscribe. By setting the appropriate publication properties, we can enforce strict reliability, which guarantees that every single event will be delivered.

4.3.3 Event Notification and Acknowledgement Services

Reliable publish-subscribe can also be used for event notifications. The relevant features here are the hierarchical structuring of topic names and the filtering of topics. A topic name can consist of several fields, such as 'heating/pressure/too_high/sensor3/13-2D'. Subscriptions can be made by specifying filtering criteria on some or all of the fields. The criteria are based on regular expressions. A pattern subscription will create subscriptions to all topics that satisfy the filtering criteria. The difference between this method and CORBA's structured event is that the latter has a list of filterable name-value pairs, where filter expressions are based on the value fields. NDDS only has a list of filterable names, so it is not possible to, for example, filter issues for a topic according to their priority or severity. Either all of the issues for a topic are delivered or then none are. Otherwise, this mechanism has the same shortcomings as have been discussed for its CORBA counterpart.

There is no support for application-level (operator) acknowledgements.

4.3.4 Request/Reply

Figure 8 illustrates RTPS's Client-server mechanism, which is a good implementation for request/reply. Services are identified with service names that correspond to the topic names of publish subscribe. It is possible to have several redundant servers with the same service name as illustrated in the figure. The application programmer specifies a minimum wait, and all responses that are received during this time from any of the servers are kept by the middleware. After the minimum wait has expired, the middleware delivers the reply with the greatest strength property to the application. If no replies were received, the first reply that comes after the minimum wait will be delivered immediately. If the deadline expires before any replies are received, the application is notified. In the current version 3.0 of NDDS, request and reply messages may get lost, but the next version will deliver them reliably [Wang 2002]. It is therefore possible to build a certain level of determinism, reliability and fault-tolerance into the application.

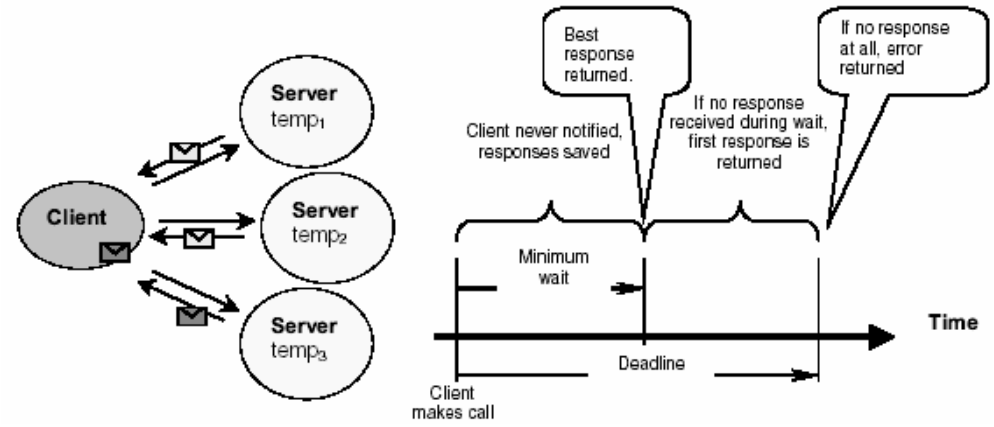

Figure 8 The Request/Reply implementation

5 The Container Design Pattern

5.1 Goals

The reader might not yet be convinced of the usefulness of the container in our architecture. However, none of the middleware standards that we have studied satisfy all of the requirements that we have mentioned above, so an application-level solution is required. The container design pattern aims to solve the following problems:

1. The integrity of data means that an algorithm will always get all of its inputs from the same sweep (iteration) of the algorithm that produced them. The only way to guarantee this with existing middleware is to group all of the signals into one data structure that is always sent as one message. This approach is unacceptable, because it leads to serious maintenance problems. This is perhaps the greatest shortcoming in the products that we have evaluated.
2. In our test results, we will have to point out frequently that any middleware has to rely on the operating system to schedule its threads. Deterministic behavior can therefore only be achieved if it is possible to control the scheduling. It should therefore be possible to execute the time-critical tasks (e.g. cyclic tasks with high frequency) with high-priority threads.
3. The IT-community has discovered that developing complex systems is much easier if the architecture is composed of parts that have well-defined responsibilities and interaction mechanisms. These parts encapsulate any details about how this functionality is implemented. We will use the layered approach here, since it is natural to separate the application logic into its own layer. The execution logic and middleware interactions are handled by the container layer, which is below the application and above the middleware.
4. Data should not go via the middleware layer needlessly, because this is inefficient.

5.2 The Solution

Figure 9 shows the container and application layers modeled as packages. I am elaborating the design in [Peltola 2002].

The application is designed using application components that are similar to function blocks. They read data from their input ports, execute some algorithms on the data and write the results to output ports. This is done cyclically with some period. All of the components with the same period can be run in the same thread and can therefore communicate directly with each other. Components with different cycle times must always communicate using the middleware. However, a component is only concerned with accessing its input and output ports and should not worry about how the data gets there.

The PortConnector in the container layer is responsible for supplying data to input ports and transmitting data from output ports. The PortConnector is fed either from an output port of an

application component (using the Read() method), or then it receives data from a Listener object (using ReadListener). The listener is the interface to the middleware services. The data that has been most recently read can be written, using Write(), to one or more input ports. If the receiver is not running in the same executable component, the data is given to the middleware layer using a Sender object and the WriteSender() method.

The container is also responsible for executing the application components. As we mentioned in section 1.1.3, application components can form a hierarchy, where a higher level component can encapsulate other components. Suitable higher level components are selected as executable components, and these correspond to the ApplicationComponent in the class diagram. They are assigned to an ExecutionThread with the appropriate period. This thread performs the I/O and algorithm execution for all of these components at the required frequency.

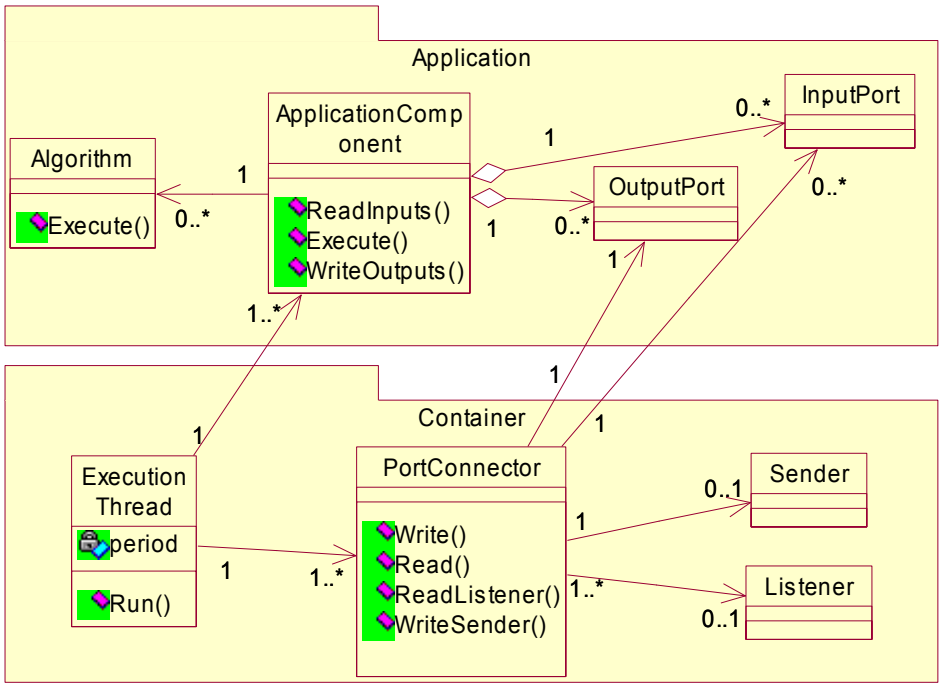

Figure 9 Container class diagram

To implement this solution, the necessary objects must be created and their associations must be established, as shown in Figure 9. Then the programmer must create an operating system thread for each ExecutionThread object and give it a suitable priority. The application is started, when these threads are forked into the Run() methods of the appropriate ExecutionThread objects. Run() has an infinite loop, and Figure 10 shows one iteration of the loop:

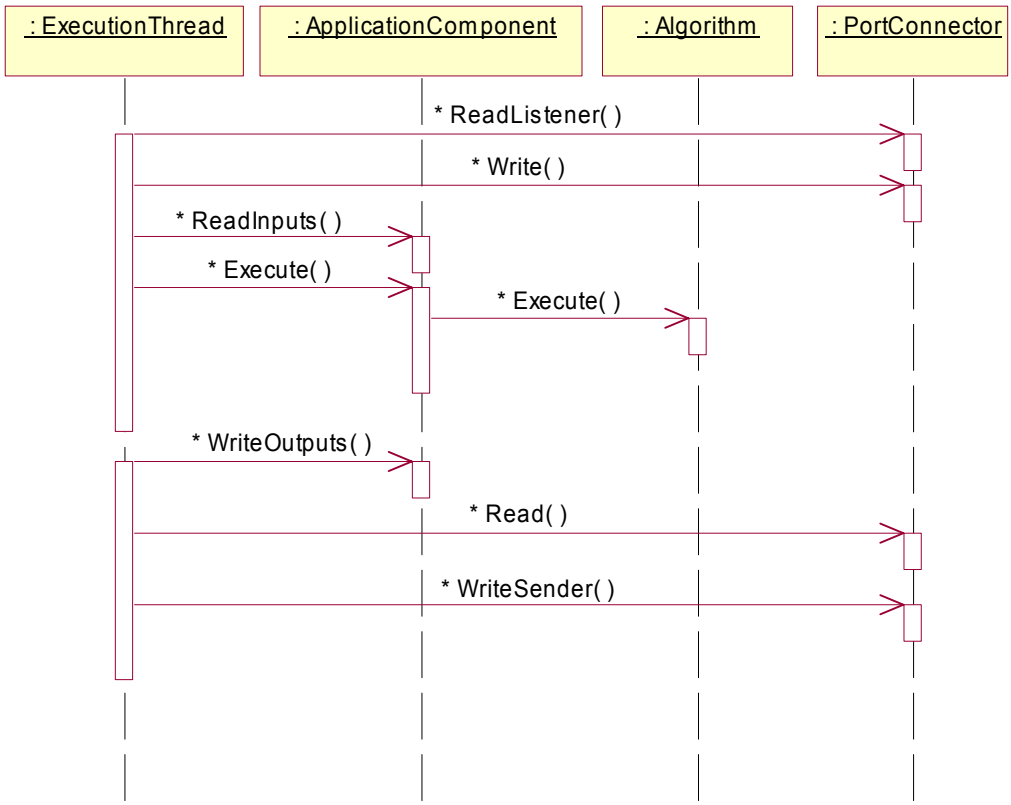

Figure 10 One Iteration in ExecutionThread::Run()

First, for those PortConnectors that are associated to Listeners, any new data from the middleware is read into the connectors' buffers with ReadListener(). Then the buffers' current values are written from the port connectors to the input ports. Then, the application components' ReadInput() is called, which reads the data from the InputPort objects. Now each component's Execute() can be called, and this in turn executes all of the associated algorithms, using the recently obtained input data. The output ports are then updated. Finally, PortConnector::Read is called, which reads the OutputPort objects. If the output port was wired to a component in the same execution thread, the appropriate input port will be updated in the next iteration by PortConnector::Write. Otherwise, the data will be handed over to the middleware with WriteSender(). The iteration markers (*) indicate that the operation is performed over a collection of objects. Nested application components will be executed recursively, until an algorithm of a basic application component is reached.

For components that are running in the same execution thread, it is clear from the sequence diagram that all inputs will be from the same sweep of an algorithm. For inputs that are received via Listeners, we need some way to guarantee that all signals from the same application component will be from the same sweep of the algorithm. This is only possible if the Listener knows what component created each signal. The solution to this is naming the signals appropriately. With NDDS, we can add a code for the application component to the topic name, If the notification service is used, a filterable field can be added for this purpose. Since the application components should correspond to some part of the

process that is being automated, it is appropriate to use the component id in the names of the signals that they create. Now we can create a Listener for each application component from which we want data. The filter or pattern subscription mechanism will be used to receive some or all of its signals. If the signals have a sequence number, it will be very easy for the Listener code to enforce the integrity requirement. ReadListener() will then return signal values from the most recent iteration for which all signals have been received up to that point.

5.3 Benefits and Constraints

With respect to our goals, the following benefits have been achieved:

1. Data integrity is enforced with simple coding in the listener classes. The scheme works if the signals are named as described above, and this naming is consistent with the structure of the process.
2. If a RTOS is used, deterministic scheduling of the algorithms is achieved by running each ExecutionThread object in a thread with a suitable priority. Typically, we would use higher priorities for those threads with shorter cycle times.
3. The container functionality is independent of the application logic, so it can be reused by application designers when building solutions for different clients. The application design will not be encumbered by any details relating to the execution of the algorithms or implementing the connections for the data flows.
4. Components with the same cycle times can be run in the same thread. Therefore, they can exchange data directly via a port connector, so middleware services are not used needlessly. For safety and robustness, the design forces all communications among components with different cycle times to go via the middleware.

The design constrains application developers to apply a certain framework. However, this framework is based on the function-block standards, so it the natural paradigm. Most application designers will be content if they do not need to change their thought models to those that dominate the IT industry. Nevertheless, their applications can communicate using established IT-technology, such as CORBA.

The framework has been developed for cyclic data distribution. If we are to use some of the other communication mechanisms that we have described, the model can be extended. For example, if an algorithm generates an alarm or notification message, a Sender object can be used to dispatch this as well as any cyclic data.

Accommodating event-driven data distribution into the model will require some more thought. In this case, an input port can also be an event port, that does not receive new data in every iteration. PortConnectors will have to distinguish between events and cyclic data, so they will write event input ports only when an event is received. An incoming signal in these will trigger some algorithm, but

these are executed by a thread whose priority is different from the cyclic execution thread. If necessary, there can be several threads with different priorities for processing events.

Above we have assumed that cyclic processing continues regardless of events. Pure event-driven processing would mean that every task (even a cyclic one) must be triggered by an event. In this case our model needs fundamental modifications, since we will not be using ExecutionThread objects for periodic tasks. However, we feel that this design pattern is well suited for realistic applications.

6 Test Arrangements

6.1 Introduction

The purpose of these tests is to determine if a middleware product satisfies the communication requirements of a process automation system. We will focus on evaluating the features of the middleware solution, so no process will be simulated. Consequently, the data that is sent does not need to contain meaningful process data; we can send information that is useful for testing purposes such as sequence numbers and timestamps. This assumption reduces the complexity considerably, since there are less dependencies among the communicating entities. For example, we can just send an alarm when we want to test a feature without first observing the process state (since there is no process.)

We will be testing functionality, performance, reliability and scalability. In the first phase we will concentrate on functionality; we will also do simple performance testing that is limited to 2 or 3 nodes. Later, we will test for scalability by running the components on 6 nodes. The plans for the first phase should be such that the transition to the scalability test phase can be done by increasing the number of existing components that were created in the first phase. The components should be configurable, so the number of communicating entities and their properties can be given as parameters.

The communication scenarios and traffic patterns are similar to what might be encountered in a small process automation system.

We are testing the NDDS implementation of RTPS.

6.2 Goals

Tests will be planned to evaluate the following features of the middleware product:

- Test the key communication mechanisms, i.e. the mechanisms that are most suited for implementing the design-level communication mechanisms that we have described in section 2.
- System startup and configuration changes at run-time.
- Name/Directory services. These allow a component to find out at run time what data and services are available and how they are accessed (a true middleware product will hide the actual location from the user). In the case of NDDS, this functionality should be handled automatically by the managers. It will be tested with the pattern subscriptions.
- Scalability (number of nodes, number of communicating components, size of payload, frequency of transmission)
- Reliability (acknowledgements and retransmission, redundancy)

- Performance (latencies, jitter, throughput, determinism)

6.3 Application Structure

In these tests we will have no real process to control and neither will we simulate any process with some kind of a model. The purpose is to focus on the features of the middleware product. Tests will be modelled after the various kinds of *communication cases* that are found in process automation. A communication case involves the transfer of data for a specific purpose, such as the continuous transfer of measurement data or the ordering some kind of a report. The requirements imposed on the communication mechanism can therefore vary quite a lot among the different communication cases, so each case will test some set of features of the middleware. (Detailed examples communication cases will be found at the end of section 6.)

The structure of the testing application is a collection of *components* that generate or receive the data that is exchanged in the various communication cases. In this section, we describe some components that are found in most automation systems, but we are only concerned with their communicational behaviour.

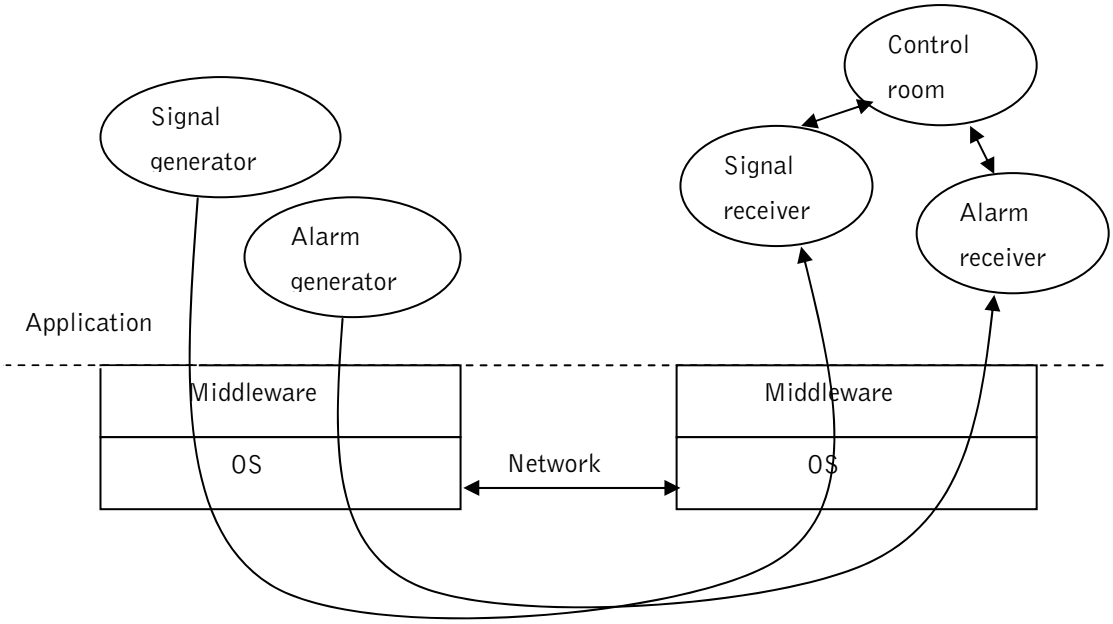

Figure 11 An example communication case scenario

Figure 11 shows an example scenario where a few of the components in this section are communicating. Since we want to test the functionality of the middleware, we are interested in the data that is transferred between the components and not process dynamics. (A unidirectional arrow would imply a one-way communication mechanism such as publish-subscribe.)

The following is a list of possible component types. Some components such as the database component are included for completeness; it will not be necessary to implement all of these for our tests.

Cyclic signal generator: A signal such as a measurement is generated and transmitted at a fixed rate. Some parameters would be the rate and the size and type of the data (ranging from binary signals to complex data structures). Timestamps and other accompanying information might be included. The component can also be used to generate several signals, depending on what parameters are given.

Signal generator: A signal is transmitted at unpredictable intervals, whenever the difference between the current value and previously transmitted value exceeds a certain threshold. Depending on the structure of the middleware, it can be either the application's or middleware's responsibility to detect the change and transmit the value.

Alarm generator: This sends an alarm with a certain type/name and other accompanying data, such as severity, priority, the value of some signals, whether or not an acknowledgement is expected, etc. The same component can be configured to generate many alarms.

File server: When the system is started and even when it is running, files need to be downloaded from a file server. These might be executable components, configuration files, recipes etc. A request must specify the correct file name and version number. A large volume of data must be sent over the network, so this should go at a low priority in order to avoid delaying more time-critical data.

The clients of the file server will usually be system components responsible for running the application components and retrieving their configuration files.

Signal receiver: This component expects to receive a signal with a certain data type. The receiver might expect values to arrive at a fixed rate or before some deadline has expired.

Alarm receiver: This component has subscribed to certain alarms (for example all of the alarms from some machine whose priorities exceed a certain value.) A certain kind of data structure containing the alarm and the accompanying data is expected. The receiver might send an acknowledgement back to the source of the alarm.

Database component: This component works in conjunction with alarm and signal receivers to store great volumes of process data. It also accepts all kinds of requests such as the values of a signal during some time period or all of the alarms with certain properties.

Process controller: This component manages the parameters for a process controller (consisting of parts such as PID blocks), which can be read or modified by other components (e.g. the control room).

Control room: This component handles all kinds of functions that are required from control room software, such as displaying process signals and alarms, ordering reports and querying and setting controller parameters. The control room component will make use of the services of many of the components mentioned above, such as the signal and alarm receivers, so it could be modelled as an aggregation of these.

6.4 Clock Synchronization

Clock synchronization between two machines is dealt with in software by obtaining an offset that expresses the difference between the timers on the 2 machines. Figure 12 shows machines A and B. To start the synchronization procedure, machine A sends a message containing its timer value A1. Machine B receives it and records its own timer value B1. B responds immediately and A receives the response at time A2. Now A can approximately calculate that time B1 corresponds to the average of A1 and A2 on A's timer. The offset between the timers is then (A2+A1)/2 – B1. This value tells us how much A's timer is in front of B's. Since all messages from A include a timestamp, we simply subtract the offset value from it in order to obtain a timer value that is synchronized with B's timer.

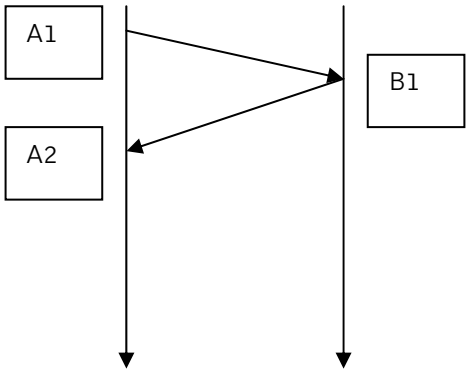

Figure 12 The synchronization algorithm

A realistic scenario will often have more than 2 nodes, so we choose one of them as the master clock node, and every other node will have a synchronization client that calculates the offset between its timer and the master's timer. Machine B above would therefore be the master node and A would be any node that hosts a synchronization client. Now all of the nodes can subtract their offset from the local timestamps, so the resulting timestamps will all be synchronized according to the master timer. Latencies can then be computed simply as the difference of two timestamps.

If the latencies for the request and response messages are equal, there will be no error. Otherwise, the round trip times (RTT) can be used to calculate an upper bound for the error. If the synchronization procedure is repeated thousands of times, we presume that the error for the minimum RTT is close to 0 for our purposes. In the worst case, when the latency for one direction is $RTT_{min}/2$ and we have the most asymmetric scenario, the difference between the timestamp B1 and the average of A1 and A2 (i.e. the error) is $(RTT-RTT_{min})/2$. We can say that errors in the latencies that we report are typically on the order of 0.1ms and nearly always under 0.2ms. However, under peak load, some errors might have approached 1ms. Although this is not ideal for a general-purpose evaluation of the middleware, the accuracy is very satisfactory considering the performance requirements of process automation systems.

We must also account for clocks drift, which can be several ms over 10s between two of our nodes. Therefore, taking averages of many values will easily cause our offset to lag behind the real value. We eventually decided to perform the synchronization procedure 3 times every 200ms and took the offset

with the smallest error (i.e. the smallest RTT). If none of the 3 offsets were satisfactory, we used the old one, since the value will not become obsolete in 200ms. Therefore, the algorithm (which was run at real-time priority) performed well unless the CPU was under a peak load for a longer period of time.

Timestamps were taken using a function that was included in the NDDS libraries; this uses the queryPerformanceCounter() on Windows and getTimeOfDay() on Linux. Since we take clock drift into account as explained above, the accuracy of these functions is very adequate for our purposes. (There are greater sources of error.)

6.5 Structure of the Tests

In the functional testing phase, we want to test the key communication mechanisms that we described in section 2, so we have to choose the most appropriate communication mechanisms that the product offers. In section 4, we described the mechanisms of NDDS. Continuous data distribution will be tested with best-effort publish-subscribe, and the test cases for this are grouped under test suite 1. Event-driven data distribution and event notification and acknowledgement services should be implemented with reliable publish-subscribe, so we test this in suite 2. Due to a lack of time, we are not testing NDDS's request/reply mechanism. In informal tests we have noticed that it worked as expected, and the performance requirements for this mechanism are not very high in automation systems. Finally, we have a scalability test suite, where we use the components from the functional testing phase. We will configure these differently and run more components on a greater number of nodes to observe how they interfere with each other.

We will only need to implement some of the components that were described in section 6.3. In the test cases, we don't specify any exact values for parameters such as sending rate or packet size. These are considered to be parameters of the components, and exact values will be given in the test reports. The test components have been implemented in such a way that output will be written to a file. Any write operation either to the screen or to a file can take a considerable amount of time, so any output is stored in memory and written out only after the test has been executed.

6.6 Continuous Data Distribution (TS-1)

Test suite id: 1

Requirements to be tested: We test a communication mechanism suitable for continuous transmission (wiring) of data. We have tests for multiple consumers and redundant producers. We examine the effect of starting or shutting down some of these on the run. We observe what happens if consumers do not get new data by the time they expect it. We also test system startup: will the communication be established automatically and will any of the initial data transmissions be lost. We also measure the time it takes to start up the system.

A measurement is generated frequently, either cyclically or when the value changes significantly. Consumers are interested in the latest value. Low delay is more important than reliability. (Best-effort delivery.)

However, there are some situations in which reliable in-order delivery of cyclic data is required, so we will have a separate test for that in suite 2, where a reliable mechanism is tested.

Content: The value that is sent is not important, since process dynamics are not simulated. In these tests, we send a double timestamp and integer sequence number. In suite 2 we will have a test case with a larger amount of data in a single message.

In general, most messages only have a few bytes of data, but there might be a need for sending data structures of up to 200kB.

Communicating parties: There can be one or more subscriptions; the publication should not need to know how many there are or where they are located. If redundancy is desired, there can be several publications for the same topic.

Communication mechanism: We use ordinary publish-subscribe.

Performance: The main performance issue is the rate at which the data is transmitted. For the subscription, we will also specify the deadline (from the arrival of the previous issue). Each subscription will measure the latency of every issue that was received.

The strictest performance requirements for this mechanism come from process control. Measurement and control signals have to be transmitted frequently. Cycle times between 5 and 1000ms are common in process automation systems. Determinism is also important, since jitter will affect the behaviour of the control algorithms and can decrease the quality of the product. In some cases, it does not matter if the latencies are even 100ms as long as jitter is kept to a minimum.

There can easily be hundreds of signals that must be transmitted with this mechanism, so the performance requirements should be met even if the volume of traffic is scaled up.

Apart from process control, the continuous mechanism is needed for such tasks as monitoring and diagnostics. The performance requirements for these are considerably looser, so it would be useful if the middleware could prioritize the more critical data.

Reliability: If the subscription does not receive new data before its deadline expires, an error has occurred. With this communication mechanism we do not attempt to guarantee that every transfer is successful. Redundant publications for the same topic can be used in the event that the primary publication fails.

When new values are sent periodically, losing an issue occasionally is acceptable, if the mechanism is optimized for efficiency. If new values are transmitted only when the signal changes significantly (i.e. at irregular intervals) delivery should be reliable.

In these tests, we send values periodically.

6.6.1 Basic flow with one subscription and one publication

Purpose: A topic is defined and issues for it are published cyclically. An integer sequence number and a double timestamp are the values that are sent with each issue. There might be several subscriptions to this topic, so that same value must be transmitted to all of them. The frequency of transmission can be given as a parameter to the test.

Method: The publication and subscription are started. The publication sends issues with sequence numbers 1 to N and shuts down. The subscription should record what issues are received and take the latencies as the differences of the sender's and receiver's timestamps. It will then print a summary of the received data to a file. This contains the minimum, maximum, average and standard deviation of the latencies. The smallest and greatest sequence numbers that were received will be recorded. We expect to receive messages for all sequence numbers in this range, so if this does not happen the sequence numbers that were missed should be recorded. Each subscription should also write a special alert if a deadline is missed.

Expected outcome: The output of each subscription might indicate that not all of the issues were received, since we are using a best effort mechanism. This is acceptable. However, if deadlines were missed, we have an error situation that should be reported.

Components used: The publication is created with a cyclic signal generator component, which is given the production rate and N, the number of issues to send, as parameters. The subscription is implemented with a signal receiver, which is given a deadline as a parameter.

The test cases that follow are variations of this basic flow.

6.6.2 Basic situation with one publication and many subscriptions (TC-1.1)

Test case id: 1.1

This is the basic flow scenario with 3 subscriptions.

6.6.3 Starting a new subscription when the test is running (TC-1.2)

Test-case id: 1.2

Purpose: To check that the middleware notices at run-time the appearance of a new subscription to some data for which a subscription already exists.

Method: One publication and subscription are started as in the basic flow scenario. After a delay of length D, a new subscription is started on a different node. The subscription should record the time when it notified the middleware that it is interested in receiving values as well as the time when it got the first value.

Expected outcome: Just as in the basic flow, but the second subscription should not have the initial values. Deadlines should not be missed after the first issue has been received.

Components used: The signal receiver should take a parameter D, which is the delay for which the component waits before creating the subscription. All signal receivers should record the time when the subscription was made and the time when the first issue arrived.

6.6.4 Publication or network failure (TC-1.3)

Test-case id: 1.3

Purpose: Sometimes subscriptions might not get fresh data because there is a failure in the publication or the network connecting the publication and subscription. The middleware should have functionality that enables the subscription to detect this kind of a failure (i.e. alert when fresh values are not received soon enough).

Method: One publication and subscription are started. A failure is simulated by having the publication first send the values 1 - N1-1 and then just wait instead of sending the values N1 - N2 and then send the final values N2+1 – N.

Expected outcome: As in the basic flow, but instead of having values N1-N2 there should be appropriate error messages. Values should be received from N2 onwards with no deadlines missed.

Components used: The cyclic signal generator should take parameters N1 and N2 which is the range of sequence numbers for which no data is sent. If these are both zero, the feature is not used.

6.6.5 Publications with different properties (TC-1.5)

Test-case id: 1.5

Purpose: Having more than one publication for the same measurement means that some of them are redundant. Different technologies have different concepts for redundancy and provide different properties for the designer to configure the main and redundant producers. With NDDS, publications with the same topic can be used to provide redundancy. Each publication has an individual strength parameter, which is described in section 4.3.1.

Method: We have one publication with strength 2 that publishes issues 1-N, with the exception of issues N1-N2. The redundant publication with strength 1 publishes values 1-N3. N1<N2<N3<N. There is one ordinary subscription.

Expected outcome: The two publications might not start at the same time, so we cannot expect the subscription to get issues with sequence numbers always increasing by one. Rather, we are interested in whether deadlines were missed and if the highest sequence number that was received was N.

Components used: The signal generator should also have a strength parameter.

6.7 Event Notification and Acknowledgement Services (TS-2)

Test Suite id: 2

Requirements to be tested: Two application-level communication mechanisms are commonly associated with alarms and events: event-driven data distribution and event notification and acknowledgement services. We have described these in section 2. Both of these will be implemented with the reliable publish-subscribe mechanism, so we will only test one of them. We have chosen the latter, because we want to test NDDS's ability to handle filter-based subscriptions. No new functionality would be tested if we would design a suite for event-driven data distribution.

A filter subscription is made by requesting all of the notification messages that satisfy the filtering criteria. Therefore the name or header of the messages will have to have some filterable fields. We describe the scheme that is used in these tests in the basic flow.

The event messages are generated in bursts. The interval between bursts is fixed for each event generator, so the tests are repeatable. Every event message should be transferred reliably and with reasonably low latency to all interested parties. The middleware should guarantee that all events are delivered.

A special case of the event transfer is when the events are generated at fixed intervals. This will be used to test reliable cyclic transfer of data.

With acknowledgement services we do not mean protocol-level acknowledgements, but those that are generated by a human operator. Since NDDS has no special functionality to handle this, we will not give acknowledgement services any further consideration here.

Content: The data type is a structure with the following fields: (double) timestamp, (int) sequence number and (char array) table. The latter is a variable length array that can be used to adjust the size of the data.

Typical alarm messages might have a number of other data fields attached to them, but their size is usually under 1kB. Some are only a few bytes.

Communicating parties: There can be any number of subscribers, each with their unique pattern for subscribing to event topics; the publications should not need to know how many there are or where they are located.

Communication mechanism: A mechanism that is optimised for reliability rather than very low latency. Delivery should be guaranteed and in-order. The reliable publish-subscribe mechanism will be used, since it claims to satisfy these requirements.

Performance: Events are not generated at a steady rate, since we send them in bursts with no delay between issues in a burst. The momentarily large volume of data is useful for testing the scalability of the system, but it should not prevent reliable delivery.

It is usually satisfactory that alarm and notification messages are delivered in < 1s. However, in problem situations there might be great bursts of these, so all of them should be handled reliably in this time.

The resolution of the timestamps should be around 1ms.

Reliability: There should be some mechanism such as one using acknowledgements and retransmissions to ensure that all messages reach all subscribers even if the network loses some packets.

If some data cannot be delivered (perhaps the network is down for a long time) the sender should be informed.

6.7.1 Basic flow

Purpose: We create an event source that produces several unique events. These should have a name and some kind of mechanism to organize them into categories. With NDDS, we use the topic name with fields separated by '/' characters. The name has the following fields:

<process area>/<category>/<type>/<measurement>/<id>

e.g:

"mixing/unexpected/too_large/temp/13-1"

Subscribers should be able to say that they are interested in the events that come from a certain process area, or they want all temperature related events, or they might want to receive all of the alarms from some machine. Consider the following subscription pattern:

"mixing/unexpected/*/temp/*"

This will request all unexpected temperature-related events from the mixing process.

We will also create subscribers, so that each subscriber will request some subset of the sources' events.

Method: The producer is initialised with a file containing the topic names. It will then send bursts of event messages (issues) at regular intervals for each topic. The messages will have sequence numbers from 1-N. Consumers will subscribe to a subset of the events and they will check the sequence numbers of the received messages. The logs will only list summary data for the latencies. If some sequence number was not received or if it came out of order an error statement will be logged.

If the producer has problems sending its buffers because the old data is not acknowledged soon enough, some statement will be recorded in the producer's log. If the buffer is full, the send should block for one sending period. If no space becomes available in the queue during this time, the problem should be logged and the same issue sent again. If the producer detects that one subscriber is down, it should log this as well. The sender may only discard unacknowledged messages after detecting that the subscriber in question is down. Otherwise it should keep trying to send the same message again and again.

Expected outcome: The log files of the producer and subscribers should be compared. The subscriber's log should contain all messages for those topics that were subscribed to, unless the producer dropped a subscriber. Some of the first messages might be lost, id the connection between the publisher and subscriber was not established before sending began. If messages were not received in order, this should be reported.

Components used: Event generator and event receiver.

The generator should have a property burst level B (an integer >= 1). This should be implemented so that the component sleeps for a time P (where P is the period of the component) and then sends a burst of B messages for each event that has been created without sleeping in between. With B=1, we have a reliable cyclic signal generator.

Unless otherwise mentioned, the sending and receiving buffers should have space for 5 messages. If the sender's queue is full, the send() will block for one period P.

The table array in the data type has length 0, if we do not specify some other value.

6.7.2 Reliable cyclic data transfer (TC-2.1)

Test case id: 2.1:

Purpose: To test the mechanism with 1 generator and one receiver component. We use burst level 1, so this will test the reliable delivery of cyclic data.

Method: A generator is started with 10 publications with different topics. A receiver is started with such subscription criteria that it will receive all 10 events. The generator will send N event messages with a period P (B=1) and then it shuts down. The receiver will wait for an end signal and then write the results to a log file.

Expected outcome: After receiving the first issue, there should be no misses. If the network is loaded, packet losses can occasionally cause higher latencies. If the sending buffers get full, so that sending a new message after the required interval is not possible, we have a problem with cyclic delivery. This can be detected from the log of the generator.

The event generator will read its list of events from a file elist1.txt with the following contents:

mixing/expected/value/temp/1-5
mixing/expected/value/temp/1-3
mixing/expected/value/pressure/1-4
mixing/unexpected/fault/temp/1-3
mixing/unexpected/too_large/temp/1-3
furnace/expected/value/temp/2-5
furnace/expected/value/temp/2-3
furnace/expected/value/pressure/2-4
furnace/unexpected/fault/temp/2-3

furnace/unexpected/too_large/temp/2-3

The receiver will subscribe with the pattern ''*/*/*/*/*''.

6.7.3 Several event generators and receivers (TC-2.2)

Test case id: 2.2

Purpose: To test the behaviour of several event generators and receivers.

Method: We start 2 event generators and 3 receivers. Each generator has 10 unique events and each receiver will subscribe to some subset of the events. A moderate burst level of 3 is used.

One generator uses elist1.txt and the other uses elist2.txt, which has the following contents

cooling/expected/value/temp/3-5
cooling/expected/value/temp/3-3
cooling/expected/value/pressure/3-4
cooling/unexpected/fault/temp/3-3
cooling/unexpected/too_large/temp/3-3
furnace/expected/value/temp/2-8
furnace/expected/value/temp/2-9
furnace/expected/value/pressure/2-14
furnace/unexpected/fault/temp/2-13
furnace/unexpected/too_large/temp/2-13

The first subscriber uses these criteria:

''cooling/*/*/*/*''

The second subscriber:

''*/*/*/pressure/*''

The third subscriber:

''*/*/*/*/3-5''

Expected outcome: The first subscriber should receive the 5 events from the 'cooling' process area. The second should receive one pressure-related event from cooling and mixing and 2 from the furnace.

6.7.4 Only one subscriber goes down (TC-2.3)

Test case id: 2.3

Purpose: To test how well the middleware can detect that a subscriber has gone down. Missing acknowledgements from such a subscriber should not prevent delivery to other subscribers.

Method: One signal generator with 10 distinct events (from elist1.txt) is started. 2 receivers are started on different nodes, and both of them should subscribe to all of the events. 10s after one of the subscribers has been started, we unplug its network connection, and after another 10s we will plug it back in (we use a wrist watch). The generator parameters N and P should be such that it will continue sending for some time after this. Burst level 3 is used.

Unplugging the Ethernet cable will usually cause the computer to disconnect its network interface, so reconnecting the interface might take some time after the cable has been plugged in. Therefore, we should connect the generator node to one hub/switch and the receiver node that will be unplugged to another, so that the two hubs are linked by a cable. Pulling out this cable should not cause any computer to disconnect its network interface. If this can not be done (e.g. due to lack of resources) the arrangements should be documented.

Expected outcome: After the subscriber computer has been disconnected, the middleware should use the heartbeat mechanism to detect that a subscription is down. Depending on the parameters to this mechanism, this will take some time T. During this time the sender's queue will fill up and after that new messages cannot be sent. The connected subscriber will not receive data after the send queue is full. After the time T has passed, the sender's queue can be emptied and the connected subscriber will get messages, so that it will not miss any of them. The disconnected subscriber should start getting messages after it has been plugged back in.

We can measure (with a wrist watch) the time for which the subscriber was disconnected. Since we know P and B, we can calculate approximately how many messages were sent during this time. The disconnected subscriber should not have missed much more than these.

6.7.5 Increasing the burst level (TC-2.4)

Test case id: 2.4

Purpose: In the previous tests, B <= 3. Since the send and receive queues have had space for 5 messages, the middleware should have been able to handle the bursts easily. Now we set B >> the queue size and observe how often the send gets blocked when sending a burst.

Method: Just like test case 2.1, but with a higher burst level B=30.

Expected outcome: The components log enough information that tells how often the sender gets blocked. The bursts that do not fit in the queue should not prevent the delivery of any messages.

6.7.6 Network goes down (TC-2.5)

Test case id: 2.5

Purpose: As opposed to test case 2.3, we now have only one generator and one receiver. We then disrupt the network connection between them for some time. The middleware will eventually notice that there is no subscriber, so it will not expect acknowledgements and all sends will be successful. (When the sending buffers get full, some data that the subscriber has not received will be lost.) In this

case it is best to have a greater sending buffer, so that temporary network problems will not prevent receivers from getting the messages when the connection is re-established.

Method: Just like 2.3, but with only one receiver component. The sending buffer should have enough space to accommodate all of the messages that are generated during the 10s downtime.

Expected outcome: All messages should be received, although the latencies for some will be approximately equal to the downtime.

6.7.7 Large event messages (TC-2.6)

Test-case id: 2.6

Purpose: To observe how well the middleware handles large event messages (does the performance suffer and is reliability compromised?)

Method: Just like test case 2.1, but we use burst level 3 and large table lengths, so that the message size will be close to the 63kB limit permitted by RTPS.

Expected outcome: No messages should be lost, but the latencies might well be longer. The sending queue might also fill up.

6.8 Scalability Tests (TS-S)

Scalability tests will be run with the same components as the functional tests. However, the number of nodes and components as well as the number of publications and subscriptions in each component will be greater.

Test case S.1 describes the basic scalability test configuration. The other ones are variations of this; each variation tries to simulate some situation that might occur in an automation system.

Even though the volume of traffic has been increased, the same QoS requirements apply as in the functional tests. Please refer to the requirements listed for test suites 1 and 2.

6.8.1 Multirate cyclic transfer of measurement data (TC-S.1)

Test-case id: S.1

Purpose: In functional tests, we examined the behaviour of transferring data with a fixed period. Although we moved to short periods, the volume of traffic and number of participating nodes was small. Here we will have three nodes that publish many topics with different periods. We want to see how much the performance deteriorates compared to the functional tests. Ideally, the middleware should prioritise the topics with shorter periods, but RTPS does not support this. When the load becomes heavy, the performance should degrade gracefully. It is acceptable if some measurements are missed and latencies increase. If this happens during a peak load, the production process will still

continue with some temporary drop in quality. However, if the system behaves wildly or crashes, we will get serious problems with product quality and the chance for safety problems is greatly increased.

We also want to see how long it takes for the system to start up.

Method: We start the following components on the nodes A-F.

Node A: Signal generator publishes 100 topics 'temp1' to 'temp100' with a period of 100ms.

Node B: Signal generator publishes 100 topics 'pressure1' to 'pressure100' with a period of 50ms.

Signal receiver subscribes to all the temp topics with a deadline of 200ms.

Node C: Signal generator publishes 100 topics 'current1' to 'current100' with a period of 20ms.

Signal receiver subscribes to all the pressure topics with a deadline of 100ms.

Node D: Signal receiver subscribes to all the temp, pressure and current topics with a deadline of 200ms.

Node E: Signal receiver subscribes to all the current topics with deadline 40.

Node F: Signal receiver subscribes to all the pressure and current topics with a deadline of 100ms.

Expected outcome: If the nodes have decent hardware, it should be possible to handle all of the data. However, the data flows are not even, but at 20, 50 and 100ms intervals 100 signal values are transmitted. These must be processed individually, and processing the data takes a longer time than generating it. For this reason, some of the latencies will be longer than in the functional tests. The greater volume of traffic on the network will also increase the probability of collisions and lost data.

6.8.2 Alarm bursts (TC-S.2)

Test-case id: S.2

Purpose: We generate bursts of alarms as in the functional tests in communication case 2. The purpose here is twofold:

- to observe the effect of the other real-time traffic on transmitting the alarms, and
- to observe the effect of a burst of alarms on the cyclic transfer of measurement data.

6.8.3 Fast rate control (TC-S.3)

Test-case id: S.3

Purpose: There might be some control loop or other process that needs to communicate faster than the 20ms cycle in S.1. In the functional tests we have observed that it certainly is possible to go below 20ms, but we had a light system load. Here we have the same arrangements as in S.1, but we also include a signal that is transmitted at a clearly faster rate.

6.8.4 Dynamic startup time (TC-S.4)

Test-case id: S.4

Purpose: Consider the situation where an operator in the control room opens a screen that displays various measurement data. We want to know how long it will take to establish the data flow from the sources to the control room application, i.e. how long will it take for us to get the first data for each signal.

Method: It is realistic to start up the other components and then at run-time subscribe to the data that the control room needs. We modify S.1, so that node D would be the control room workstation. It will have a delay of 10s, so that it will start up when the application is running. From this point, we will measure how long it takes to get the first data for each of the signals.

6.8.5 Increasing the load with large data structures (TC-S.5)

Test-case id: S.5

Purpose: The event generator has a variable length array field which can be used to create large packets (tens or hundreds of kB). We will send a few of these and observe the effect that this has on the transmission of the signal values.

Method: We will add the following components to the configuration in S.1:

An event generator is added onto Node B and an event receiver on Node D. With some period, these will send large data packets from B to D.

7 Functional Test Results

7.1 Introduction

7.1.1 Purpose of this Section

In this section, we describe the exact test arrangements and parameters used to run the tests. We do not include all of the results, since the log files for this test suite alone have a total of over 100000 lines. Here we show summary data as well as any information that was unexpected or otherwise interesting.

7.1.2 Evaluating the Performance of the Middleware

After presenting the results for the various test cases, we would like to make some conclusions about how well the middleware performed. Unfortunately, it is not possible to answer this simply with some number, as we can do in measuring the top speed of a car or its fuel consumption over 100km. The reason for this is that the test application, middleware, operating system, network and computer hardware are all involved in the tests, and it is not possible to observe the effect of any one of these alone. Our goal in testing is to gain insight into the behaviour of each of these 5 vital elements of any distributed system. Then we can make some conclusions about each of them individually, and we can identify issues that should be taken into consideration by application developers. Only then will it be possible to use the middleware to its best advantage and to satisfactorily deal with any possible limitations.

In order to gain this kind of insight, we have prepared a variety of scenarios. By comparing the results of different tests, and remembering any differences in the test arrangements, we can suggest an explanation as to why changing one thing in the arrangements produced different results. For example, we can have a generator and receiver component on one node and another identical receiver on another node. By comparing the logs of the 2 receivers, we can observe how much the network contributed to latency and jitter.

7.1.3 Understanding the Load

Finally, in order to accurately evaluate performance, we need an understanding of the total load on the system. We use a switched Ethernet and run no other applications, so there is no significant amount of background traffic. However, it should be remembered that each node synchronizes its clock by sending and receiving 3 UDP packets (containing a double timestamp) every 200ms. Also, the application does some processing in taking timestamps and storing results in main memory. Very roughly, we have observed a 20-30% performance improvement when we stripped all of this away.

7.1.4 Using Timestamps to measure Latencies

We take the sending timestamp just at the last point before the data is given to the middleware on a per-issue basis (and not before calling the publisher send.) We take the receiving timestamp as soon as the issue processing routine is invoked.

7.1.5 Bugs

In general, the NDDS middleware kept its promises of providing a convenient and portable data distribution service for an application with components running on different platforms. We only discovered 2 bugs in these tests, and neither of them will cause significant problems to a developer who is aware of them.

Section 7.2.3.4 describes some spurious deadline alerts that were received by subscriptions on Windows nodes even though the deadline had not yet expired. I also provide a suggestion on how to filter out these alerts. RTI suspected that this might have been caused by the various tasks being scheduled in such a way that they interfere with each other [Kindel 2002].

Section 7.3.4.2 describes a bug in the API for reliable publishing. This was known to RTI, and the desired behaviour can be achieved by using other parts of the API [Wang 2002].

7.2 Cyclic Transfer of Measurement Data

7.2.1 Nodes and network

We refer to each node by the last part of its IP address. The following nodes are used:

Node 45 (667MHz, 256MB RAM, Windows XP)

Node 72 (400MHz, 128MB RAM, Mandrake 8.2 Linux)

Node 199 (2GHz, 512MB RAM, Windows XP)

Nodes 45 and 72 are connected to the same switching router using 100Mbps Ethernet. However, there was another 100Mbps switch between node 72 and switching router. 199 is connected to another switching router at 100Mbps and these routers (which belong to the backbone) operate at 1Gbps.

Time synchronization components run at real-time priority. Test components on Windows use high priority. Components on Linux use default (0) priority. After some experiments, the test components were given these priorities to prevent problems with system instability and interfering with the clock synchronization. In our experience, raising the test component priorities does not significantly improve performance (mean latencies or jitter) when there are no other user apps and the UI is not used.

7.2.2 Results of test case 1.1

7.2.2.1 Parameters

We have one signal generator on node 199 and receivers on nodes 199, 45 and 72.

Generator publishes 10000 issues for topic 'temp1' with period 20ms. (Total time is 200s.)

All receivers have deadline of 40ms.

All components have 0 initial delay.

Receiver's log files have names L1_1_IP.txt, where IP is 199, 45 or 72.

7.2.2.2 Minimum, maximum and mean latencies

Figure 13 shows the minimum maximum and mean latencies for the receivers on each node:

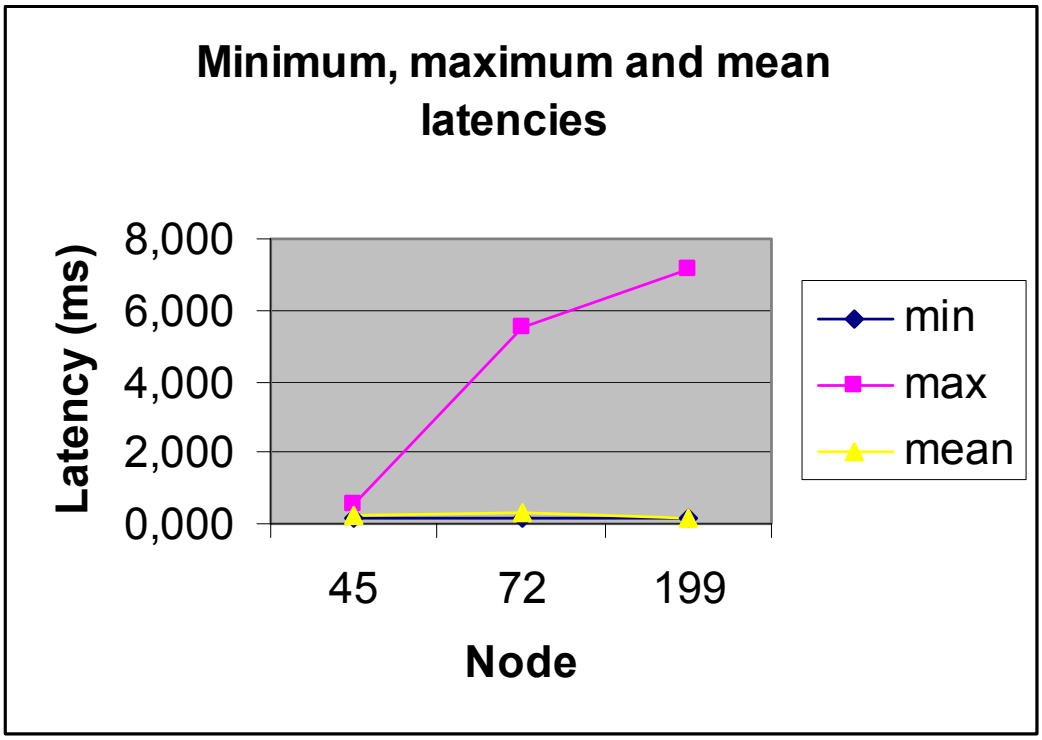

Figure 13 Min, max and mean

We notice that when the sample size is 10000, a maximum of several ms is possible. The maximums don't correlate with the speed of the processor or the network connection, since the receiver on 199 is on the fastest machine and the data does not pass over the network. However, 199 is the only node which has both a signal generator and receiver, so these two could have competed for processor resources. Being of equal priority, the generator could have prevented the receiver from getting the CPU immediately after an issue arrived.

The other values are better understood from the next chart.

7.2.2.3 Minimum, mean and standard deviation

Figure 14 shows the minimum, mean and stdev curves. The data points on the latter are computed by adding the standard deviation to the mean. What this means depends on the statistical distribution of the data. In any case, the distribution of values around the mean is not symmetric, since latencies are >0ms. If we have a normal distribution, 67% of the values are within one standard deviation from the mean. Looking at the logs, we observe that at least 90% of the points are within one standard deviation. These two charts and the logs indicate that there are a few latencies that are much larger than the typical values. We can only explain this by the action of the scheduler: one of the numerous operating system threads is scheduled in such a way that the subscription is not notified immediately when the data arrives. The behaviour of the scheduler and all of the system and user processes follows some intricate and complicated patterns that can hardly be modelled and predicted accurately (at least since these are not RTOSes.) However, these results support our belief that latencies cannot be accurately modelled by the same statistical tools as most phenomena.

We conclude that the application should be designed with 'firm' deadlines (i.e. the deadline should be met nearly always, but a failure to do so occasionally will not cause serious damage.) If this is not acceptable, a RTOS which provides guarantees about its scheduling is required.

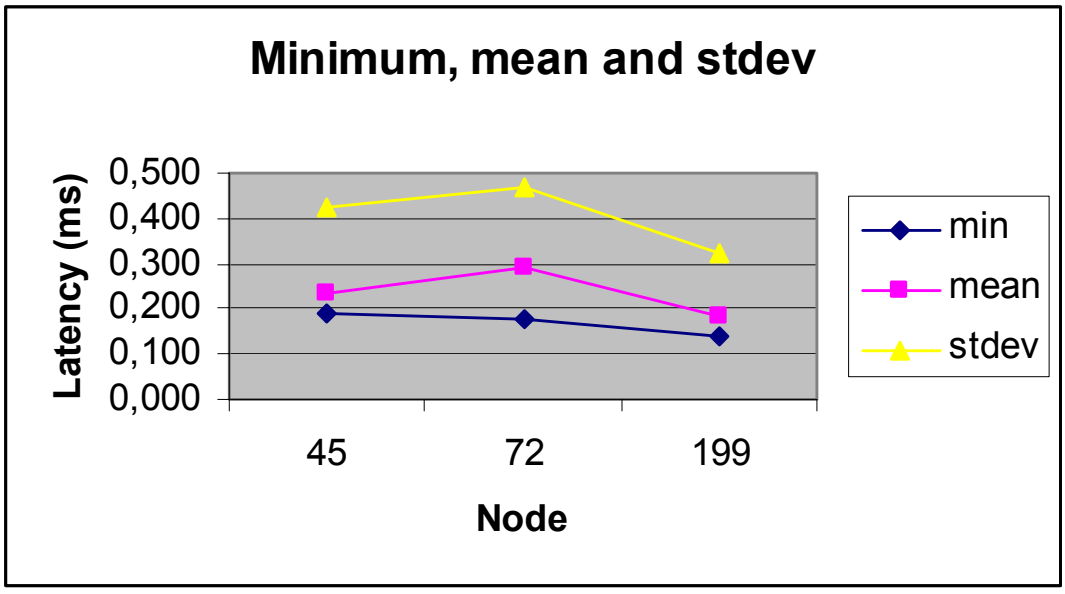

Figure 14 Min, mean and stdev

7.2.2.4 Other observations

The following table shows how long it took after getting the start signal to get the first issue. (All test components wait for the start signal and then create the publications and subscriptions.) It looks like all components got this done at the same time. The publisher sent the first 29 issues although no subscription was ready, and this is the expected behaviour, since we didn't use a subscription wait.

Node	Delay in getting 1st issue	Sequence number of 1st issue
45	607,6	30
72	559,7	30
199	601,5	30

No deadlines were missed on any node after the first issue was received.

7.2.3 Results of test case 1.2

7.2.3.1 Parameters

We have one signal generator on node 72 and receivers on nodes 45 and 72.
Generator publishes 10000 issues for topic 'temp1' with period Pms. (Total time is 200s.)
The test is repeated for values P=20, 10, 5, 3, 2, 1
All receivers have deadline of 2*Pms.
Receiver on 72 has 0 initial delay.
Receiver on node 45 has P seconds delay.
Receiver's log files have names L1_2_IP_P.txt, where IP is 45 or 72.

7.2.3.2 Latency statistics

We first compare the results for the receivers on the 2 nodes in Figure 15. We can see that mean values are very similar. On one hand, we could expect lower latencies on node 72, since the receiver and publisher are on the same node. On the other hand, node 72 hosts 2 test components that must share the CPU, and the hardware on that node is slower. From the results, we can conclude that these effects cancel each other, resulting in similar mean performance.

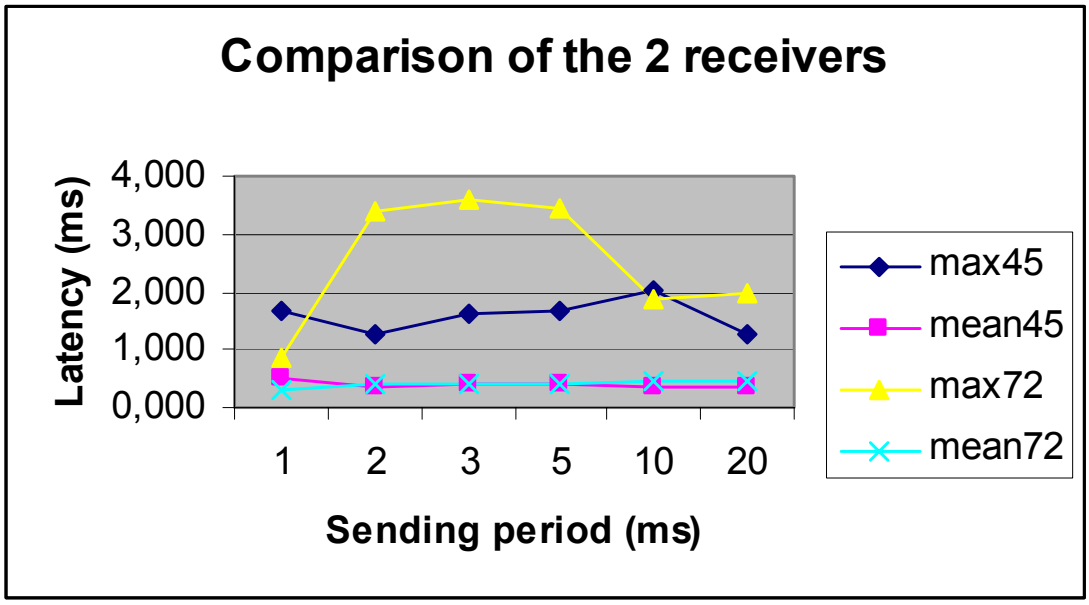

Figure 15

We can also speculate why node 72 (running Linux) has greater maximum latencies. The test component on Windows (node 45) does have somewhat higher priority, but looking at test case 1.1, so does the subscription on node 199, and yet it has the greatest maximum. The explanation that fits with the results of test cases 1.1 and 1.2 is that a machine running publishing and receiving components has greater contention for the CPU, which results in poorer response for some component.

Figure 16 gives a detailed look at the results for the subscription on the Windows node. The minimum value gives a good idea of the best case performance with ideal scheduling. The standard deviation curves show that the jitter is very satisfactory for process automation applications. i.e. a control loop usually works well even with longer latencies as long as new data arrives at a steady rate. The jitter is much under 1ms, and typical control loops operate at a much longer period, so this is unlikely to cause unstable behaviour in the control algorithm.

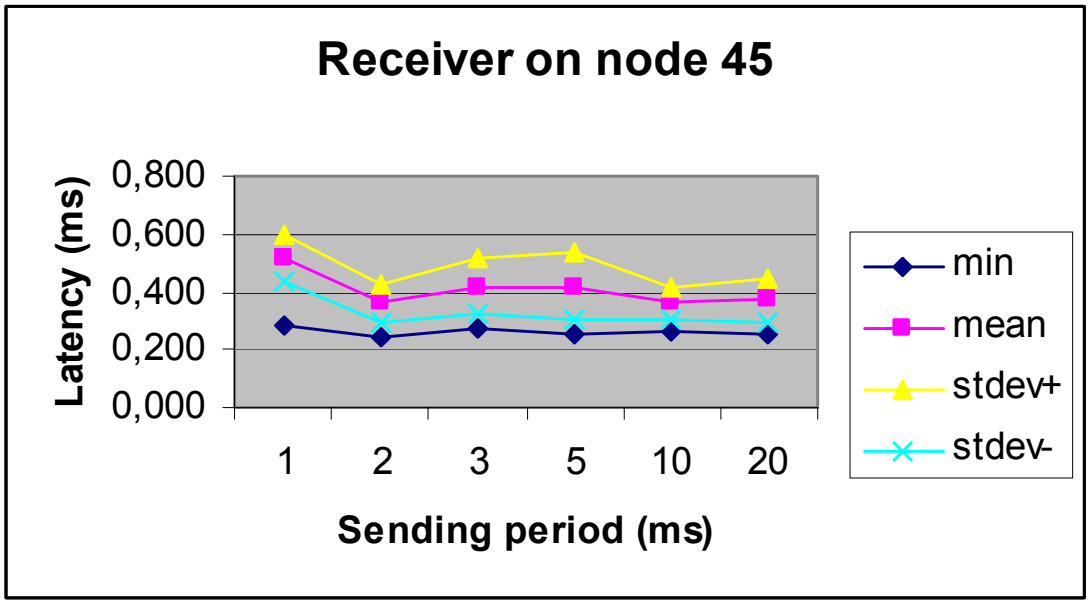

Figure 16

The corresponding results for node 72 in Figure 17 are very similar. We can conclude that the network only adds a small delay, but it is not a primary cause of jitter. These results support our view that to get good performance, it is important to focus on the nodes themselves. This involves the hardware, the OS (in particular the scheduler) and keeping the CPU load and number of interrupts to a minimum. (An OS that handles interrupts well is recommended; the ordinary Windows and Linux that we use here are not very good in this respect.) As far as application design is concerned, all unnecessary issue processing should be avoided. For example, transmitting issues also to uninterested parties with multicast does not give additional load to the network, but the receiver threads on the receiving nodes must process a great number of interrupts. These results warn us that performance might deteriorate rapidly because of this.

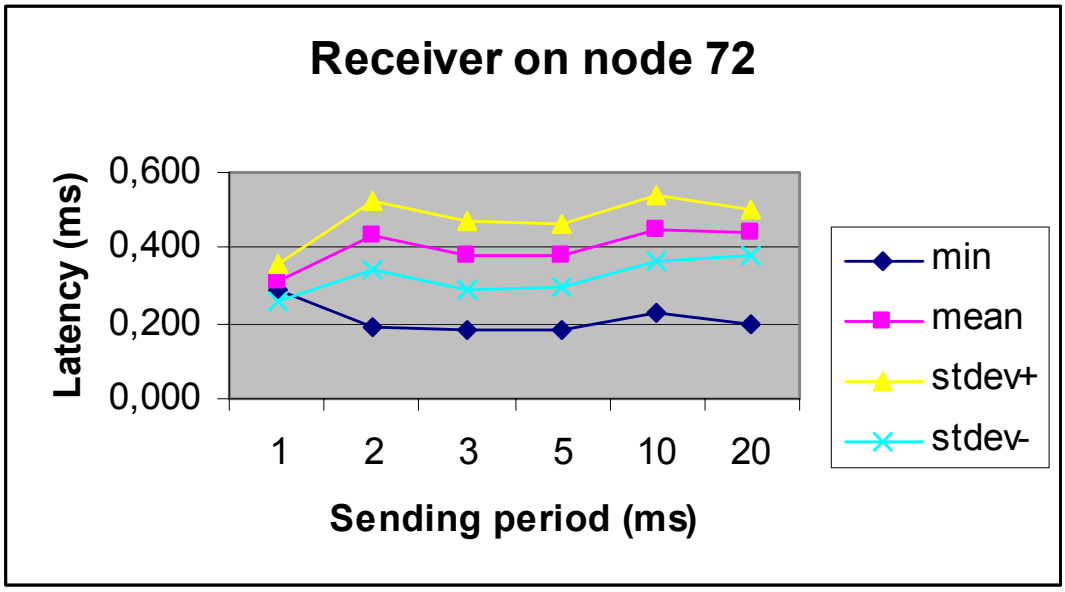

Figure 17

Finally, we note that there are no clear trends to be seen when the sending period is decreased. The performance did not deteriorate as the period was lowered down to 1ms. However, with only one topic even this does not cause serious CPU load, so we have to perform some scalability tests to get a better understanding of the performance with short cycle times.

7.2.3.3 Other observations

The following table lists the delays in getting the first issues, measured from the point where we made the API call to create the subscription. We also have the sequence number of the first issue, and we can confirm that this number multiplied by the sending period is close to the delay in getting the first issue. (The Linux that ran the publisher had 1ms timer resolution, so the period was in practice nearly 1ms greater than reported here.) This looks like the publication is ready to send very quickly, but it takes some time for the subscription to be noticed and a data flow to be established. The subscription on node 45 appeared at run time, and the initial delays for it are lower. We presume that much setup work had already been done when the publication and subscription were running on node 72. From the values in the fourth column, we conclude that the communication between managers on different nodes is rapid.

	Subscription on 72		Subscription on 45	
period(ms)	1st delay(ms)	1st seq#	1st delay(ms)	1st seq#
1	320	146	92	541
2	310	100	326	766
3	598	151	4	752
5	18	3	6	835
10	82	8	84	919
20	426	21	261	966

From the figures in this report, we see no indication that dynamically adding a subscription would cause peak latencies. Also, the logs do not show any latency peaks at the point where the new subscription appeared.

7.2.3.4 Deadlines

The Linux node missed one deadline in the test with period 2ms and deadline 4ms. The alert came 4.8ms after receiving the issue with sequence number 116. In the test with a period of 1ms and deadline 2ms, 6 deadlines were missed. The alerts came between 1.9 and 4.9ms after getting a fresh issue. No other deadlines were missed on Linux.

On Windows, we only missed one deadline in the tests with sending periods 1ms and 2ms. However, there were a large number of spurious deadline messages. These came *before* the actual deadline had occurred. Here is a small excerpt from the log for the case with a 4ms deadline, which illustrates typical results for all of the receivers on Windows for the tests with periods 1-10ms (no problems were observed when the sending period was 20ms.)

Deadline missed 1.714743ms after receiving issue with seq# 777
Deadline missed 1.858895ms after receiving issue with seq# 807
Deadline missed 2.014781ms after receiving issue with seq# 837
Deadline missed 2.160609ms after receiving issue with seq# 867
Deadline missed 2.445562ms after receiving issue with seq# 897
Deadline missed 2.496685ms after receiving issue with seq# 927
Deadline missed 2.647542ms after receiving issue with seq# 957
Deadline missed 2.840304ms after receiving issue with seq# 987

An application might behave erratically, if it gets deadline messages prematurely. Fortunately, it is quite straightforward for the application programmer to filter out the extra deadline alerts in OnIssueReceived(). (This is the routine for handling incoming issues and deadline alerts.) I would do this as follows:

The Listener class is given a private member latestFreshData. Whenever a fresh issue is received, the value of latestFreshData is set to the current time. Whenever there is a deadline alert, we compare the difference between the current time and latestFreshData. If this is less than the deadline property of the subscription, we can ignore the alert by returning from OnIssueReceived() immediately. (Timestamps are taken with an accurate function.)

7.2.4 Results of test case 1.3

7.2.4.1 Parameters

We have one signal generator on node 72 and a receiver on node 45.
Generator publishes 10000 issues for topic 'temp1' with period 20ms. (Total time is 200s.)
We use values N1 = 2000 N2 = 4000 (i.e. issues in this range are not sent.)
Receiver has deadline of 40ms and 0 initial delay.
Receiver's log file has name L1_3.txt

7.2.4.2 Results

Here are some interesting parts from the log:

```
First issue received 379.962152ms after starting up the component
...
Seq#: 1997, Latency: 0.228332ms
Seq#: 1998, Latency: 0.254062ms
Seq#: 1999, Latency: 0.197872ms
Seq# 2000 missed.
...
Seq# 4000 missed.
Seq#: 4001, Latency: 0.357722ms
Seq#: 4002, Latency: 0.348140ms
Seq#: 4003, Latency: 0.236585ms
Seq#: 4004, Latency: 0.214851ms
Seq#: 4005, Latency: 0.213340ms
Seq#: 4006, Latency: 0.206225ms
...

Sequence number of first received issue: 20.
In ms: Min: 0.130976, Max: 0.553934, Mean: 0.257598 Stdev: 0.044771
Deadline missed 124.457124ms after receiving issue with seq# 1999
Deadline missed 254.537866ms after receiving issue with seq# 1999
Deadline missed 384.839865ms after receiving issue with seq# 1999
Deadline missed 514.921445ms after receiving issue with seq# 1999
```

```
Deadline missed 615.161811ms after receiving issue with seq# 1999
Deadline missed 745.433639ms after receiving issue with seq# 1999
Deadline missed 875.511448ms after receiving issue with seq# 1999
...
```

We see that only the issues 2000-4000 are missed. Having the publication stop sending for some time (2000*20ms = 40s) did not cause any confusion when sending was resumed. The best-effort mode did not lose any issues, nor were the latencies conspicuously high at this point.

After sending the 1999th issue, the subscription will naturally miss a number of deadlines; I have copied here the first few alerts from the log. The receiver's deadline is 40ms, and we get alerts around every 100ms. The RTPS specification says that a deadline notification should come within 2 deadline periods.

7.2.5 Results of test case 1.5

7.2.5.1 Parameters

We have signal generators on nodes 72 and 199 and a receiver on node 45.
Generator on 199 has strength 2. It publishes 10000 issues for topic 'temp1' with period 20ms. (Total time is 200s.) We use values N1 = 2000 N2 = 4000.
Generator on 72 has strength 1 and it publishes 8000 issues for 'temp1' with period 20ms.
Receiver has deadline of 40ms.
All components have 0 initial delay.
Receiver's log file has name L1_5.txt

7.2.5.2 Results

The log has over 10000 lines, so here are the interesting parts:

```
First issue received 401.100586ms after starting up the component.
...
Seq#: 1997, Latency: 0.163941ms
Seq#: 1998, Latency: 0.187682ms
Seq#: 1999, Latency: 0.159181ms
Seq# 2000 missed.
...
Seq# 2619 missed.
Seq#: 2620, Latency: 0.173555ms
Seq#: 2621, Latency: 0.215700ms
Seq#: 2622, Latency: 0.214459ms
Seq#: 2623, Latency: 0.211578ms
Seq#: 2624, Latency: 0.192424ms
...
```

```
Seq#: 3810, Latency: 0.201212ms
Seq#: 3811, Latency: 0.199563ms
Seq#: 3812, Latency: 0.216544ms
Seq# 3813 missed.
...
Seq# 4000 missed.
Seq#: 4001, Latency: 0.190609ms
Seq#: 4002, Latency: 0.178032ms
Seq#: 4003, Latency: 0.162662ms
Seq#: 4004, Latency: 0.154275ms
Seq#: 4005, Latency: 0.146727ms
...
Sequence number of first received issue: 20.
In ms: Min: 0.114760, Max: 0.450669, Mean: 0.167832 Stdev: 0.027816
```

No other issues were missed than indicated above. Deadline behavior was as in test case 1.3 (deadlines were only missed after issue 1999).

When the primary publication stops sending at issue 2000, it is not until issue 2620 than we start getting data from the secondary publication (that has been running all the time.) After that we never miss any deadlines. Curiously, issues 3813 – 4000 are missed. This can be explained by the fact that the secondary publication happened to have a ca. 5% longer period in practice (due to the 1ms timer resolution.) This means that in the time that the primary publication's loop had gone through 4000 iterations, the secondary one had only done 3812. The steady flow of fresh data was not interrupted at this point, which is confirmed by the fact that we got no deadline notifications after any issue other than 1999. The middleware has behaved according to its specification, and this is no problem for cyclic transfer of measurement data. If there is some application where it is important to get issues with consecutive sequence numbers, the reliable publish-subscribe mode should be used. Having redundant publications with slightly different cycle times must be taken into consideration.

7.3 Event Notification and Acknowledgement Services

7.3.1 Nodes and Network

We refer to each node by the last part of its IP address. The following nodes are used:

Node 45 (667MHz, 256MB RAM, Windows XP)

Node 72 (400MHz, 128MB RAM, Mandrake 8.2 Linux)

Node 92 (533MHz, 64MB RAM, Mandrake 8.2 Linux)

Nodes 92 and 72 are connected to the same 100Mbps switch. This switch and node 45 are connected to another switch using 100Mbps Ethernet.

Time synchronization components run at real-time priority. Test components on Windows use high priority. The environment for running components on Linux is slightly different from the previous section, since we have solved some problems. The test components are run at a priority of -10 and the code optimization switch –O2 has been used at compile time for better performance.

In our experience, raising the test component priorities does not significantly improve performance (mean latencies or jitter) when there are no other user apps and the UI is not used. Comparing these results to those in the previous test suite will show that there are no major performance differences. This can be explained by the fact that performance will depend mainly on how well the OS schedules processes and handles interrupts. Since we have no other user applications running, there will not be much difference in this behaviour.

7.3.2 Results of test case 2.1

7.3.2.1 Parameters

We have one event generator on node 72 and receivers on nodes 45 and 72.
Generator publishes 10000 issues for topics in elist1.txt (see OHJAAVA-227) with period 20ms. (Total time is 200s.)
Burst level 1 is used, so we test reliable cyclic transfer.
As stated in OHJAAVA-227, send and receive queue lengths are 5 and table array size (in the data type) is 0, unless otherwise mentioned.
Subscription pattern is */*/*/*/*, so all topics should have been subscribed to.
Receiver's log files have names L2_1_IP.txt, where IP is 45 or 72.
Publisher's log file is PL2_1.txt

7.3.2.2 Mean, min, max and stdev

Figure 18 shows the statistics for the first, fifth and tenth topics in elist.txt.

The topics are labelled with the number 1, 5 or 10 and W or L, depending on whether the receiver was on the Windows or the Linux node.

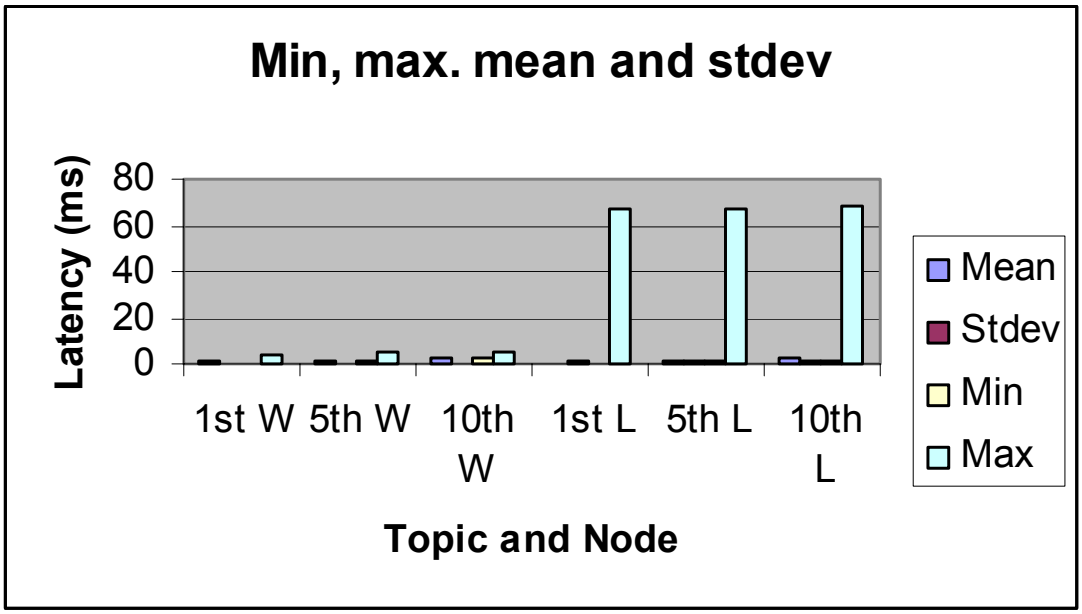

Figure 18

The most important observation that can be made from the chart above is that there were some surprisingly long maximum latencies on the Linux node. Looking at the log, we can see that the maximum occurs at issue number 4752 for each topic. The following excerpt is for the first topic, but all topics exhibited similar behaviour. If network or clock synchronization problems would be the cause of this, we could expect similar results on Windows. We can conclude that the event receiver on Linux was not scheduled promptly at this point. This is not because there was not enough processor capacity, because this test ran for several minutes, publishing at the same rate, and otherwise it performed decently. We also see that although these components were now run at high priority (minus 10), this does not guarantee good scheduling at all times. Since the NDDS middleware operates in best effort mode, it is hard to believe that it would be responsible for such anomalous behaviour. We again make the conclusion that a RTOS with appropriate scheduling is required, if the application has hard deadlines.

```
Seq#: 4752, Latency: 66.787004ms
Seq#: 4753, Latency: 47.472954ms
Seq#: 4754, Latency: 40.187001ms
Seq#: 4755, Latency: 20.841956ms
Seq#: 4756, Latency: 3.614068ms
Seq#: 4757, Latency: 0.448942ms
Seq#: 4758, Latency: 3.880978ms
Seq#: 4759, Latency: 0.482082ms
```

We now plot the data without the maximums in Figure 19 in order to get a better look at the other statistics:

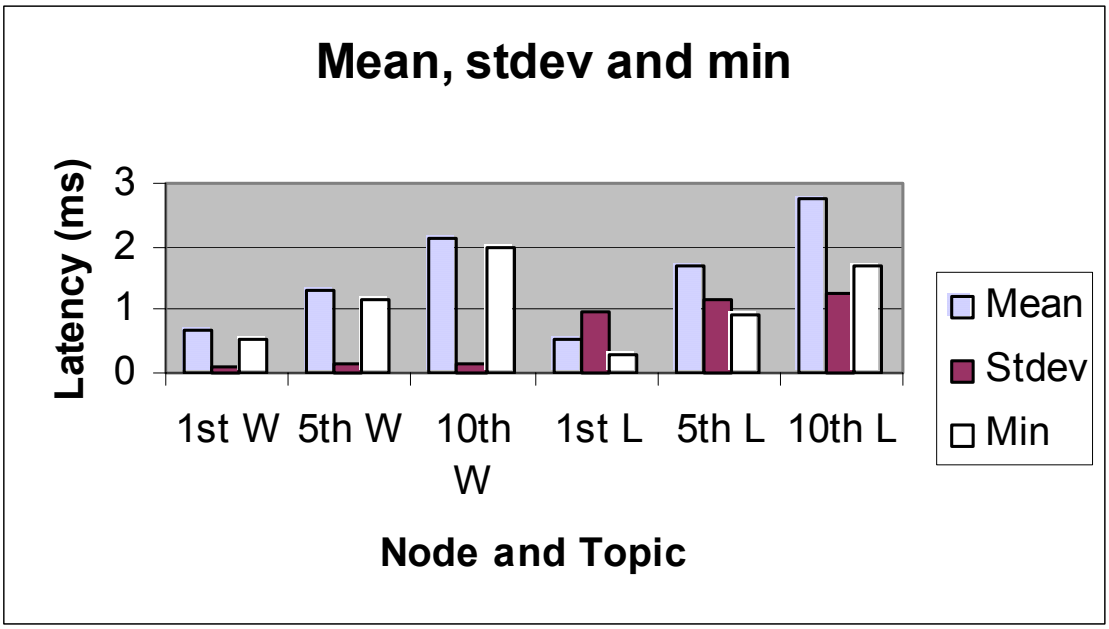

Figure 19

It is clear that the problem explained above has affected the mean and standard deviation values for the subscriptions on the Linux node. Only the minimums are unaffected by this, and they are better for the Linux receiver (since the receiver and generator were on the same node.) It is interesting to observe that the latencies increase quite linearly as the topic number increases. We will offer an explanation in the next subsection.

7.3.2.3 Latency distribution

Figure 20 has a histogram for the first topic on the Windows receiver. (On the horizontal axis we have a range of latencies and the column height is the number of issues whose latency was in that range.) Since the mean for these was 0.66ms and standard deviation 0.1ms, there is nothing surprising in the histogram. The distribution is fairly regular. What cannot be seen from the chart is that the last category (>=0.85ms) has a size of 24 and that 13 issues had a latency of over 3ms.

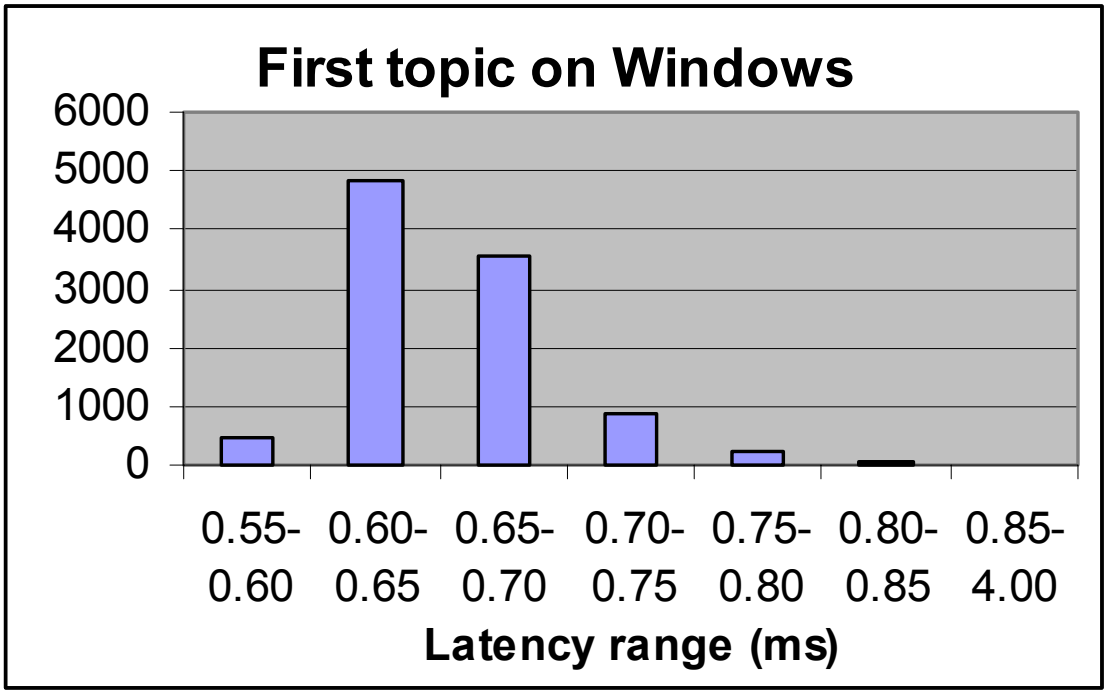

Figure 20

The latency distribution in Figure 21 has been derived from the latencies of all 10000 issues, after they were sorted in ascending order. The first column has the mean of the 1000 smallest latencies, the second has the mean for the next 1000 and the last has the mean for the 1000 longest latencies. We have displayed values for the 1st, 5th and 10th topics on the Windows receiver.

We see that the distribution is very even. Even the column that shows the mean of the final 10% is less than 0.3ms greater than the mean. This illustrates an observation that is also made by reading through the logs: there are a few rare latencies that are much greater than the mean. Since these are scattered around the logs, and the processor load is very much constant during the tests, we attribute these peaks to undeterministic scheduling.

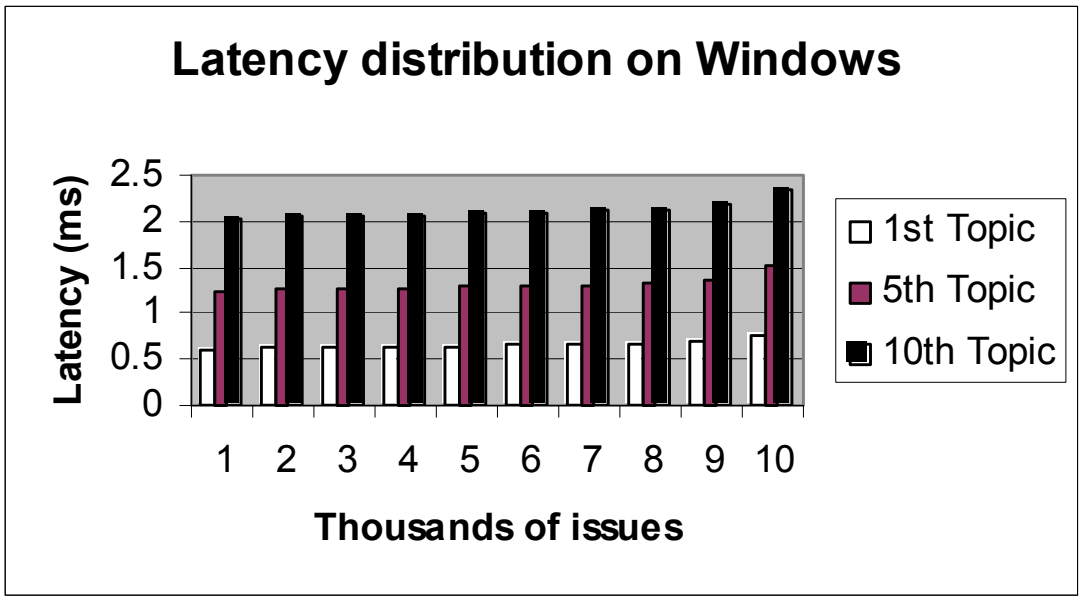

Figure 21

We can clearly see a pattern that we pointed out in the previous chart: the latencies increase linearly as the topic number increases. We can explain this by observing that the middleware sends issues for the topics in the order that they were read in from the topic name file. (I.e. when the publisher's send method is invoked, it first send an issue for the publication for topic one, then for topic 2 and so on.) We have also observed that in practice issues nearly always also arrive in this order. Since issues for all topics (10 in this case) are sent by one publisher send operation, it is not possible to process them at once at the sending and receiving ends. The ones that are sent later have to wait. By measuring the publisher's total send delay, and remembering that sending was done synchronously in the calling thread, we can conclude that the bottleneck is the NDDS receiver thread. This thread is responsible for passing the incoming issues from UDP to the application and executing the OnIssueReceived() routine in the application that processes the data. In order to confirm this explanation, we tried adding some extra processing into the OnIssueReceived() routine and observed that the latencies for the later topics again increased linearly.

7.3.2.4 Other observations

The publishers log was empty (i.e. the publisher did not experience difficulties in sending at this rate.)

The delay in getting the first issue (after staring up the components) was between 2027 and 2031ms for all the topics on the Linux receiver. The Windows receiver recorded values between 2003 and 2006ms.

7.3.3 Results of test case 2.2

7.3.3.1 Parameters

We have event generators on nodes 72 and 92 and receivers on nodes 92, 45 and 72.
Burst level 3 is used.
Generator on 72 publishes 10000 bursts (each containing 3 issues) for topics in elist1.txt with period 20ms. (Total time is 200s.)
Generator on 92 publishes 2000 bursts for topics in elist2.txt with period 100ms. (Total time is 200s.)
The first subscriber on 45 uses this pattern:
"cooling/*/*/*/*"
The second subscriber on 72:
"*/*/*/pressure/*"
The third subscriber on 92:
"*/*/*/*/3-5"
Receiver's log files have names L2_2_IP.txt, where IP is 92 or 72.
Publisher's log file is PL2_2_IP.txt

7.3.3.2 Understanding the effect of bursts

In all previous tests we have sent one issue for each topic at regular intervals. Now we are sending 3 issues for each topic without pausing in between. The following charts show statistics based on dividing the sequence number by the burst size and taking the remainder (modulo division). If the modulo is 0, the issue was the first in a burst, if the modulo is one, the issue was the second of a burst and if the modulo is 2, we have an issue that was sent last in a burst.

Figure 22 shows the latencies for the first topic on the Windows receiver. (This is the 6th topic sent by the generator.) Presenting the data in this way reveals a very linear pattern: the later issues in a burst have to wait before the first ones are processed. From the slope of the curve we notice that the delay for processing each new issue was around 0.5ms in this case.

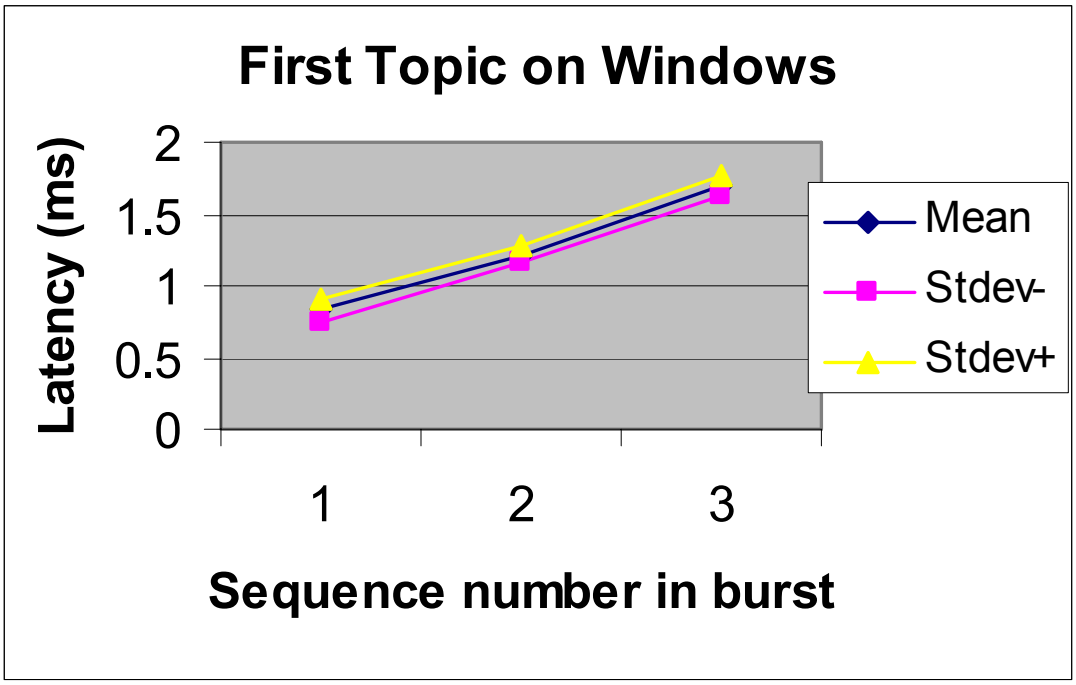

Figure 22

Figure 23 shows the same data for the last topic on the Windows receiver; the results do not challenge any conclusions that were made before. The only difference is that the latencies are systematically higher, as is to be expected for topics whose issues are sent last.

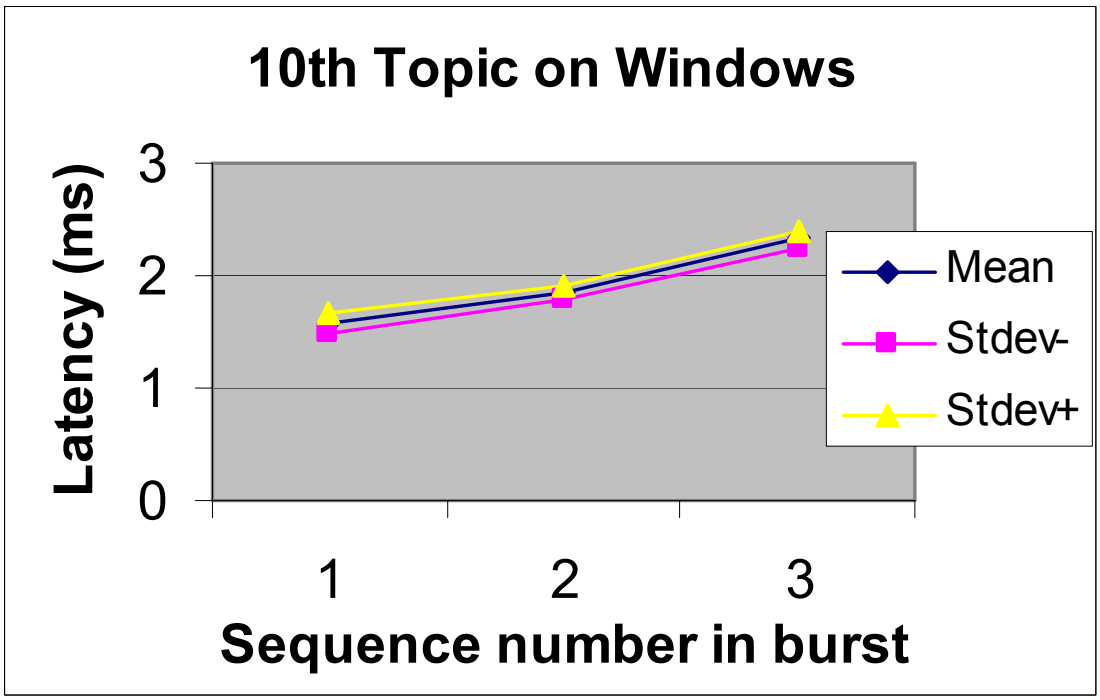

Figure 23

For the receivers on the Linux nodes, we observed similar results, but with a few exceptions. These can be illustrated by observing the log of the receiver on 72. Remember that it subscribed to all pressure related events, and that some of them were generated on the same node (72) and some on the node 92.

The following excerpt is from the log for `cooling/expected/value/pressure/3-4`, a topic published by a generator on node 92:

```
Seq#: 10, Latency: 0.815034ms
Seq#: 11, Latency: 0.633955ms
Seq#: 12, Latency: 0.658035ms
Seq#: 13, Latency: 0.824928ms
Seq#: 14, Latency: 0.611901ms
Seq#: 15, Latency: 1.331925ms
Seq#: 16, Latency: 0.648975ms
Seq#: 17, Latency: 0.632882ms
Seq#: 18, Latency: 0.676870ms
Seq#: 19, Latency: 0.904083ms
Seq#: 20, Latency: 0.644088ms
Seq#: 21, Latency: 0.664115ms
```

We see that the issues that were first in a burst (sequence numbers 10, 13, 16, 19) exhibit a latency that is greater than that for the second issue in the burst. The third issue has again a larger issue than the second. This behaviour is consistent throughout the log.

We believe that this is caused by having a receiver and generator component on the same node (on both 72 and 92), since we have not observed this otherwise. Whatever intricate pattern the scheduler uses to run the components and handle the interrupts will affect the latencies.

7.3.4 Results of test case 2.3

7.3.4.1 Parameters

We have an event generator on node 72 and receivers on nodes 45 and 72.
Burst level 3 is used.
Generator publishes 5000 bursts (each containing 3 issues) for topics in elist1.txt with period 20ms. (Total time is 100s.)
Both subscribers use this pattern:
"*/*/*/*/*"
Receiver's log files have names L2_3_IP.txt, where IP is 45 or 72.
Publisher's log file is PL2_3.txt

7.3.4.2 Difficulties

In this test case, we had a situation where issues were lost, so the reliability mechanism should have handled the necessary retransmissions. In the code, we check the return value of the publisher object's Send() method. If this is false, the issue could not be sent, because the send queue was full. Therefore, the application will keep retrying with the same issue, in order to achieve successful delivery.

However, the logs for this test showed that issues were lost even by the local subscription, although the publisher had never returned false. I asked RTI for an explanation, and they admitted that there is a known bug in the publisher [Wang 2002]. The Send() will always return true.

If this problem is known to the developers, it is possible to use other parts of the API to control reliable delivery. Namely, a publication listener should be registered. This receives events when the queue gets full as well as when the number of issues exceeds a certain level. These can be used to control the behaviour of the publications.

While this bug can be dealt with if it is known, coding a workaround at this point would require redesigning the event generator, starting from its architecture. At this point in the project there is simply no time for this. The schedule for the testing effort was already compressed when, halfway through the project, we were required to make Linux the main testing environment.

As far as the results here are concerned, we can say that the bug has not affected any other tests than those where the cable was pulled out. (This can be seen from the logs: there were no dropped issues.) This is because the queue size was long enough for the reliable delivery to do its job without running out of queue space. However, this test and test case 2.5 will not perform as expected. Nevertheless, in the next section we have the results for this test. We believe that they do increase our insight of what is happening, even though we could not confirm the expected outcome.

7.3.4.3 Effect of pulling out the cable

The main difference to the previous test case is that the cable was pulled out. The effect of this is not obvious from statistics and charts, so we will show relevant parts of the logs. These excerpts are from the first topic in the log from node 45; the results for the other topics were similar.

```
Statistics for topic 'mixing/expected/value/temp/1-5':
First issue received 1991.709982ms after starting up the component.
Seq#: 1, Latency: 3.842345ms
Seq#: 2, Latency: 4.870260ms
Seq#: 3, Latency: 4.746700ms
Seq#: 4, Latency: 3.683004ms
Seq#: 5, Latency: 4.469649ms
Seq# 6 missed.
Seq# 7 missed.
Seq# 8 missed.
Seq# 9 missed.
```

```
Seq# 10 missed.
Seq# 11 missed.
Seq# 12 missed.
Seq# 13 missed.
Seq# 14 missed.
Seq# 15 missed.
Seq# 16 missed.
Seq#: 17, Latency: 4.180390ms
Seq#: 18, Latency: 4.308903ms
Seq#: 19, Latency: 2.081569ms
Seq#: 20, Latency: 3.299750ms
Seq#: 21, Latency: 8.863063ms
Seq#: 22, Latency: 0.729161ms
…
Seq#: 943, Latency: 0.803651ms
Seq#: 944, Latency: 2.178448ms
Seq#: 945, Latency: 3.398734ms
Seq# 946 missed.
…
Seq# 2235 missed.
Seq#: 2236, Latency: -1046875188495.188100ms
Seq#: 2237, Latency: -1046875188493.828700ms
Seq#: 2238, Latency: -1046875188492.561400ms
Seq#: 2239, Latency: -1046875188502.334100ms
Seq#: 2240, Latency: -1046875188500.988800ms
Seq#: 2241, Latency: -1046875188499.764500ms
Seq#: 2242, Latency: -1046875188502.303500ms
Seq#: 2243, Latency: -1046875188501.043000ms
Seq#: 2244, Latency: -1046875188499.859300ms
Seq#: 2245, Latency: 0.795510ms
Seq#: 2246, Latency: 2.074873ms
Seq#: 2247, Latency: 3.267846ms
```

The following lines are from the log on 72 for the same topic:

```
Seq#: 946, Latency: 1.453042ms
Seq#: 947, Latency: 3.772020ms
Seq#: 948, Latency: 6.101012ms
Seq#: 949, Latency: 43.009996ms
```

```
Seq#: 950, Latency: 52.541018ms
Seq# 951 missed.
...
Seq# 961 missed.
Seq#: 962, Latency: 4.223943ms
Seq#: 963, Latency: 3.785968ms
Seq#: 964, Latency: 5.051017ms
Seq#: 965, Latency: 6.280065ms
Seq#: 966, Latency: 7.546067ms
Seq#: 967, Latency: 1.468062ms
Seq#: 968, Latency: 3.823042ms
...
Seq#: 1181, Latency: 3.816009ms
Seq#: 1182, Latency: 6.189942ms
Seq#: 1183, Latency: 44.186950ms
Seq#: 1184, Latency: 47.713041ms
Seq#: 1185, Latency: 53.623915ms
Seq#: 1186, Latency: 37.333965ms
Seq#: 1187, Latency: 39.551020ms
Seq# 1188 missed.
Seq#: 1189, Latency: 22.589922ms
Seq#: 1190, Latency: 26.756048ms
Seq#: 1191, Latency: 34.646988ms
Seq#: 1192, Latency: 26.845098ms
Seq#: 1193, Latency: 30.179024ms
Seq#: 1194, Latency: 43.353081ms
Seq#: 1195, Latency: 27.096987ms
Seq#: 1196, Latency: 29.036045ms
Seq#: 1197, Latency: 27.978063ms
...
Seq#: 2233, Latency: 1.487970ms
Seq#: 2234, Latency: 3.826022ms
Seq#: 2235, Latency: 6.171942ms
Seq#: 2236, Latency: 13.749957ms
Seq#: 2237, Latency: 16.118050ms
Seq#: 2238, Latency: 18.463969ms
Seq#: 2239, Latency: 6.226063ms
Seq#: 2240, Latency: 8.553028ms
Seq#: 2241, Latency: 11.216998ms
```

```
Seq#: 2242, Latency: -1046875188487.042480ms
Seq#: 2243, Latency: -1046875188484.765381ms
Seq#: 2244, Latency: -1046875188478.174438ms
Seq#: 2245, Latency: 8.507967ms
Seq#: 2246, Latency: 10.813951ms
Seq#: 2247, Latency: 13.261080ms
Seq#: 2248, Latency: 2.462029ms
Seq#: 2249, Latency: 7.305026ms
Seq#: 2250, Latency: 9.616971ms
Seq#: 2251, Latency: 2.388954ms
...
Missed seq#: 6.
Missed seq#: 7.
Missed seq#: 8.
Missed seq#: 9.
Missed seq#: 10.
Missed seq#: 11.
Missed seq#: 12.
Missed seq#: 13.
Missed seq#: 14.
Missed seq#: 15.
Missed seq#: 16.
Missed seq#: 951.
Missed seq#: 952.
Missed seq#: 953.
Missed seq#: 954.
Missed seq#: 955.
Missed seq#: 956.
Missed seq#: 957.
Missed seq#: 958.
Missed seq#: 959.
Missed seq#: 960.
Missed seq#: 961.
Missed seq#: 1188.
Missed seq#: 1248.
Missed seq#: 1305.
Missed seq#: 1359.
Missed seq#: 1530.
Missed seq#: 1584.
```

```
Missed seq#: 1752.
Missed seq#: 1803.
Missed seq#: 1860.
Missed seq#: 1917.
Missed seq#: 1971.
Missed seq#: 2022.
Missed seq#: 2079.
Missed seq#: 2187.
```

We see some strange behaviour in both logs. The receiver on 45 has missed issues 946-2235, because the network connection was broken at this time. However, both receivers also missed issues 6-16, which shouldn't happen with the reliable mode. Also, during the time the network was down, the receiver on 72 missed a few issues. This was accompanied by some rather long latencies, e.g. issue 1188 is missed at a time when latencies between 20 and 50ms are common.

We now understand that the sender's queue overflowed and that old issues were discarded, because the Send() return value provided misleading information to the application. High latencies on node 72 were observed when the network was down. This can be caused by the high priority synchronization algorithm bombarding its UDP socket in a futile attempt to contact the clock server.

The negative latencies after issue 2236 are caused by the clock synchronization algorithm, which took nearly a second to recuperate after its connection to the master clock was restored.

7.3.5 Results of test case 2.4

7.3.5.1 Parameters

We have one event generator on node 72 and a receiver on node 45.
Generator publishes 1000 issues for topics in elist1.txt (see OHJAAVA-227) with period 100ms. (Total time is 100s.)
Burst level 30 is used.
As mentioned before, send and receive queue lengths are 5.
Subscription pattern is mixing/*/*/*/*, so the first 5 topics should have been subscribed to.
Receiver's log files have names L2_4.txt.
Publisher's log file is PL2_4.txt

7.3.5.2 Analyzing the behaviour of a burst

We suspect that that a similar pattern with latencies will occur with each burst, so we have calculated statistics based on the sequence numbers of issues within a burst (For the first topic in elist1, this data has been plotted Figure 24; Figure 25 has the corresponding results for the 5th topic.).

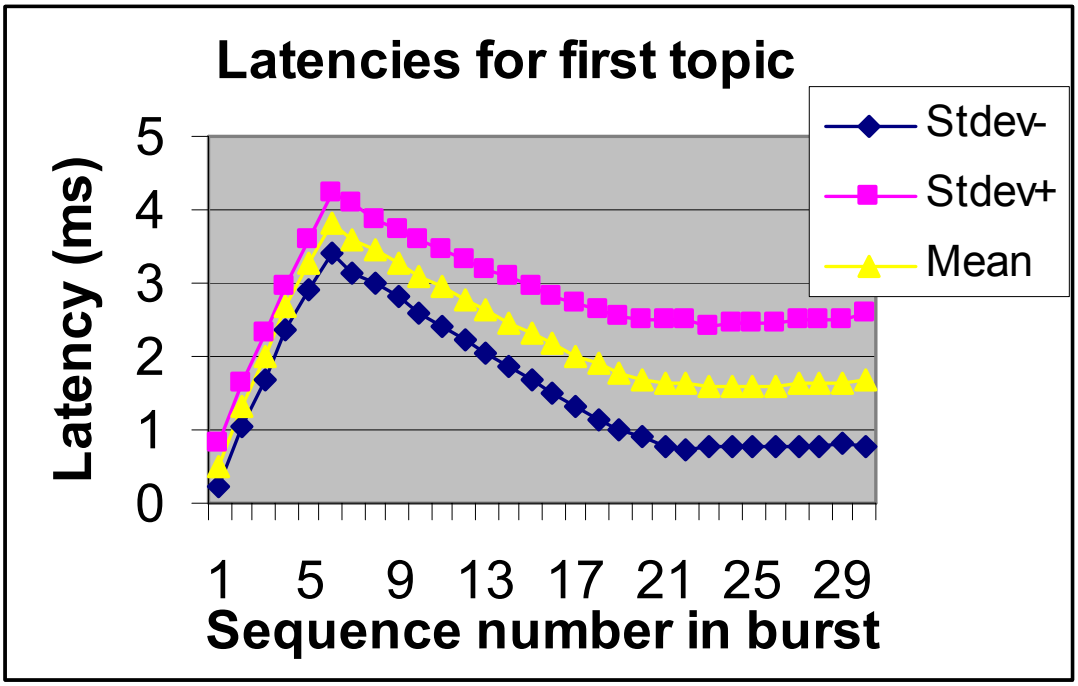

Figure 24

At first we observe the same behaviour as when sending with a burst rate of 3: the latencies of subsequent issues in a burst increase almost linearly. We explained this with the issues being processed by the receiver more slowly than they could arrive, so some of them had to wait in the receive queue. Now, looking at the results, it is important to remember that the length of the sending and receiving queues was 5. When the sender's queue fills up, the send will block for 100ms or until there is space in the queue. The receive queue also gets full at around the same time, since it has the same size. This explains the peaks in the 2 charts at the 5th and 6th issues sent, because that is the point where the traffic jams. It is just like driving in a traffic jam: at one point the traffic comes to a halt and then it gradually starts moving again. The exact pattern depends on the inner workings of the scheduler and interrupt handler. (Remember that the sending timestamp is taken just before the issue is actually sent off. This explains why the latencies of issues later in the burst are sometimes lower than the previous ones.)

At this point we point out that the publisher's log was again empty, i.e. sends were always successful. This is unsurprising, since the send would have blocked for 100ms before returning an error.

Now compare the charts for the first and last topics. As we have observed before, the latencies for the first topics are systematically smaller, since their issues are sent first. In this test case we observe the same behaviour until the queues fill up. After that this effect is not so pronounced, since the scheduling is different. This is perfectly acceptable: the middleware makes no promises about treating one topic in a publication any differently than another. We have just made observations about the order in which jobs are processed in practice.

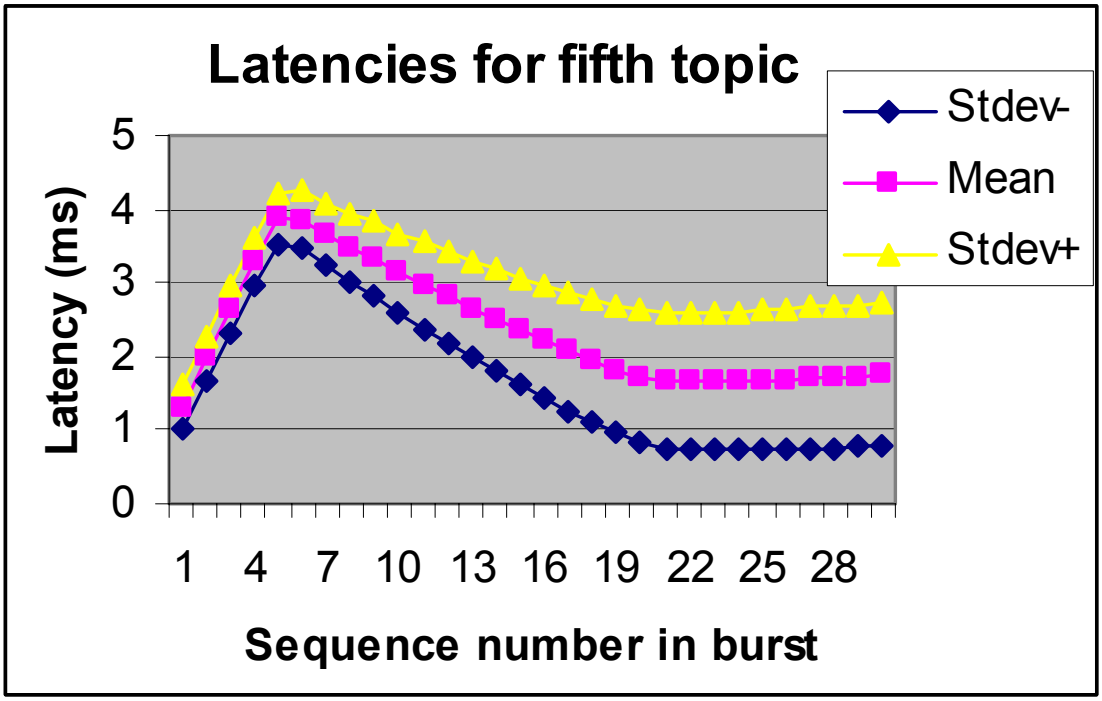

Figure 25

7.3.6 Results of test case 2.5

This test is omitted for reasons explained in section 7.3.4.2.

7.3.7 Results of test case 2.6

7.3.7.1 Parameters

We have one event generator on node 72 and a receiver on node 45.

Generator publishes 1000 issues for topics in elist1.txt with periods 100, 50 and 20ms.

Burst level 3 is used.

Subscription pattern is mixing/*/*/*/1-5, so only the first topic is subscribed to.

Table size is 62000, so the payload in one issue is slightly over 62kB.

Receiver's log files have names L2_6_X.txt.

Publisher's log file is PL2_6_X.txt

Where X is 20, 50 or 100, depending on the publisher period.

7.3.7.2 Statistics

Figure 26 displays the main statistics for each of the tests (where the sending periods were 20, 50 and 100ms).

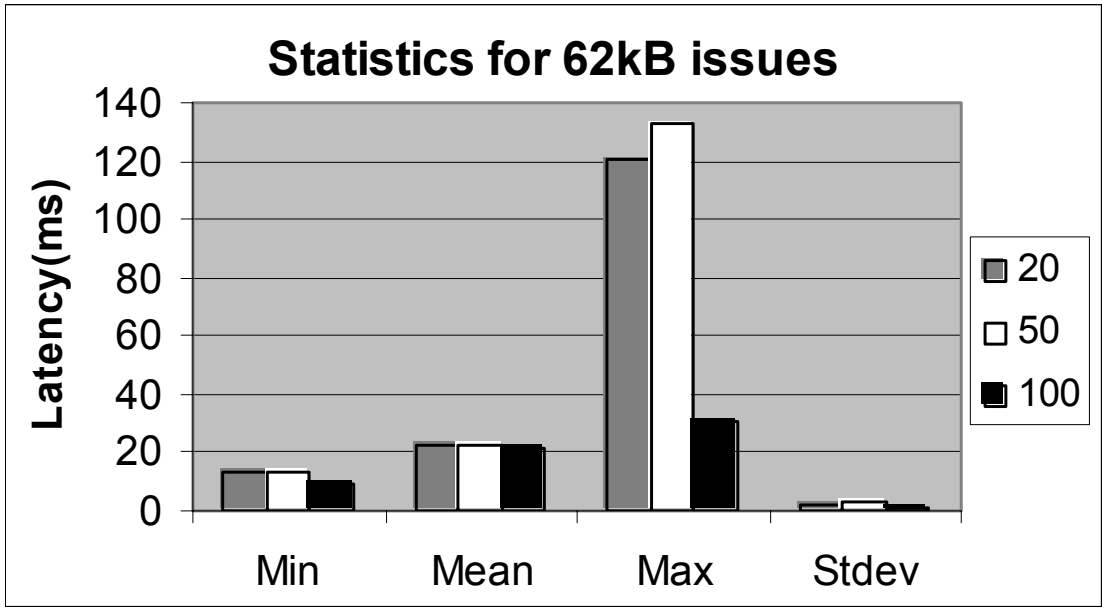

Figure 26

On the average, sending an issue took slightly over 22ms. We observed very little variation among the issues that were sent in one burst. This can be seen from Figure 27, where the issues are grouped according to their sequence number in the burst. (This is from the test that had a 100ms period; the others behaved similarly.) This is a clear difference to the previous results, where the latencies increased considerably within a burst. We suppose that sending the large issues takes so long that the receiver can handle them as they come (so the receiver is no longer a clear bottleneck).

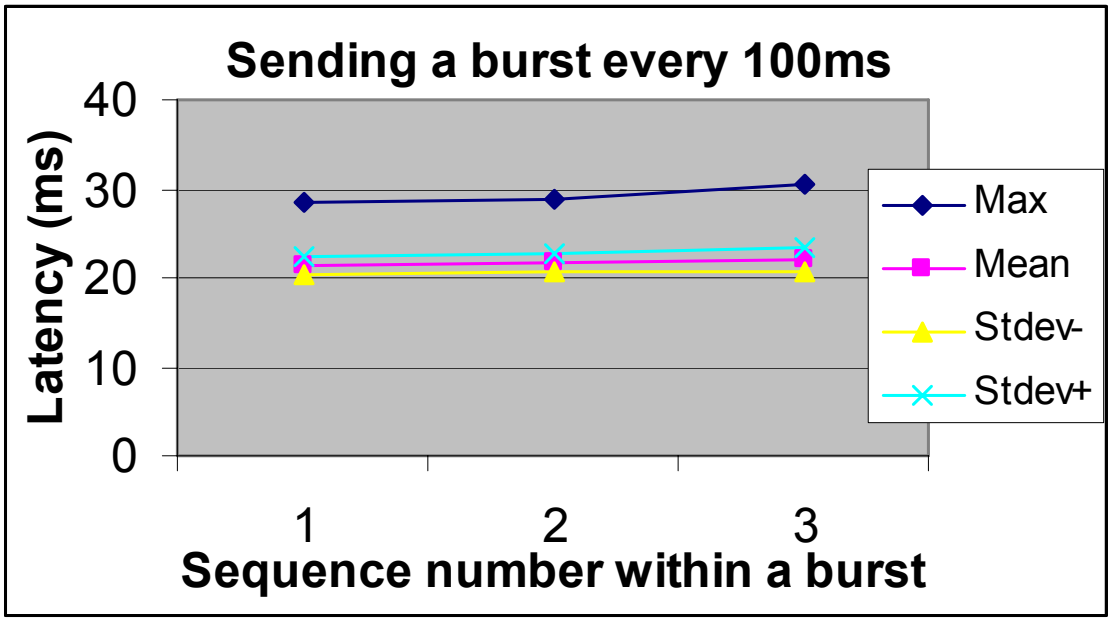

Figure 27

Since sending one issue takes close to 20ms, the whole burst should take nearly 60ms. When the sending period goes below this, the sender will just be sending new bursts one after the other without respite, since it cannot maintain the desired period. The total send times for 1000 issues at 20, 50 and 100ms were 50.4, 51.8 and 101 seconds, although the theoretically correct values would have been 20, 50 and 100 seconds, respectively. This indicates that the minimal sending period is around 50ms. A burst has size 3*62kB = 186kB, and 20 of these per second makes 3720kB per second. Since we are using 100Mbps Ethernet, it is clear that we did not run out of network bandwidth. The limiting factor was the interrupt handling and issue processing.

The total send times also explain why the minimum, maximum and standard deviation values were much lower with a period of 100ms: at that rate, the publisher was not running in a busy loop, so there was no fierce contention over the CPU.

7.3.7.3 Latency distribution

The latency distribution in Figure 28 has been obtained by sorting the latencies and taking the averages in groups of 300 (for the test with the 50ms period.) This supports the previous conclusions that the issues were sent and received at around the same rate, i.e. there was a steady flow of them from publication to subscription with no major bottlenecks.

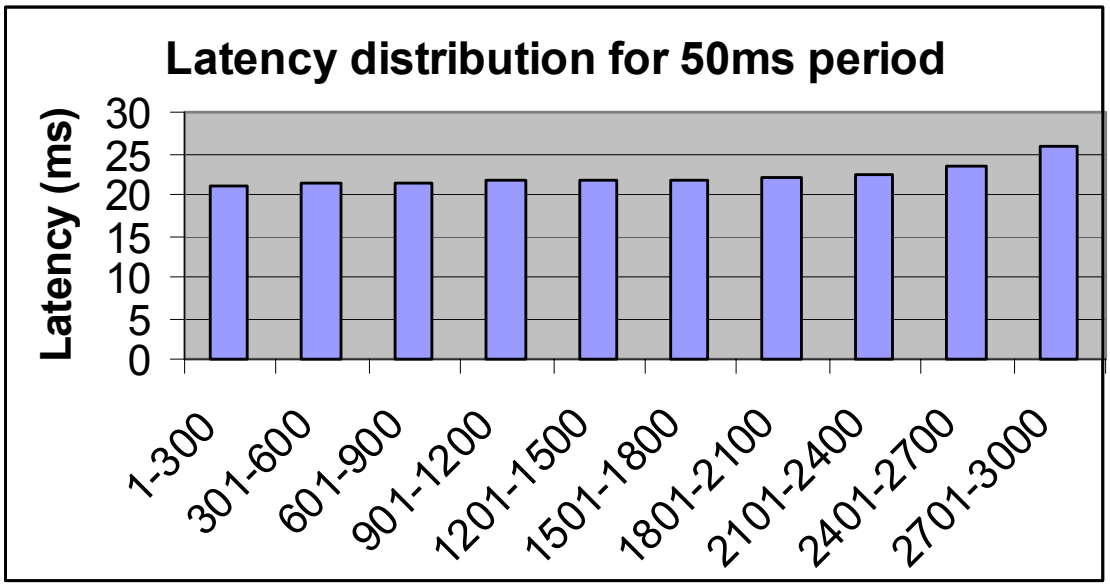

Figure 28

8 Scalability Test Results

8.1 Introduction

In this section, we describe the exact test arrangements and parameters used to run the scalability tests. We do not include all of the results, since the log files for this test suite alone have a total of over 100000 lines. Here we show summary data as well as any information that was unexpected or otherwise interesting.

We use the same components as in the functional tests. The difference is that the number of participating nodes and components as well as the volume of data is much greater. In the basic test scenario we generate traffic that is somewhat typical for a process automation system. In later test cases, we add a few components to observe how the behaviour is affected by components with special communication requirements. We will be observing behaviour that is in many ways similar to what has been reported in the functional test results. We will not repeat all of those conclusions here; rather, we will focus on any new insight that is gained from scaling up the tests.

8.2 Scalability Test Arrangements

8.2.1 Nodes and Network

All of our IP addresses start with 130.233.152, so we will refer to nodes by the last part of their IP address. The following nodes have been used:

45: (667MHz, 256MB RAM, Windows XP)

72: (400MHz, 128MB RAM, Mandrake 8.2 Linux)

199: (2GHz, 512MB RAM, Windows XP)

89: (533MHz, 128MB RAM, Mandrake 8.2 Linux)

127: (350MHz, 128MB RAM, Mandrake 8.2 Linux)

92: (533MHz, 64MB RAM, Mandrake 8.2 Linux)

72, 89, 92 and 127 are all connected to the same 100Mbps switch. This switch and 45 are connected to a switching router with 100Mbps Ethernet. 199 is connected to another switching router at 100Mbps Ethernet. The switching routers belong to the backbone, which operates at 1Gbps.

8.2.2 The Basic Configuration

8.2.2.1 Purpose

Here we describe the basic scalability test configuration. The test cases are variations of this; each variation tries to simulate some situation that might occur in an automation system.

In functional tests, we examined the behaviour of transferring data with a fixed period. Although we moved to short periods, the volume of traffic and number of participating nodes was small. Here we will have three nodes that publish 100 topics each with different periods. We want to see how much the performance deteriorates compared to the functional tests. Ideally, the middleware should prioritize the topics with shorter periods, but NDDS does not support this. RTPS is based on a best-effort principle, which just tries to minimize the latency caused by any middleware processing.

We also want to see how long it takes for the system to start up (i.e. when do subscriptions start getting issues.)

8.2.2.2 Acceptable performance degradation

When the load becomes heavy, the performance should degrade gracefully. It is acceptable if some measurements are missed and latencies increase. If such a thing happens during a peak load, the production process will still continue with some temporary drop in quality. (The damage caused by lost or late data is very process dependent; for the sake of safety, it is considered desirable that a process will remain stable even if it does not get new control signals for some time.) However, if the system behaves wildly or crashes, we will get serious problems with product quality and the likelihood for safety problems is greatly increased.

8.2.2.3 Setup and configuration

Node 92: Signal generator publishes 100 topics 'current1' to 'current100' with a period of 100ms. Signal receiver subscribes to all the temp topics with deadline 50.

Node 72: Signal generator publishes 100 topics 'pressure1' to 'pressure100' with a period of 50ms.

Node 199: Signal generator publishes 100 topics 'temp1' to 'temp100' with a period of 20ms. Signal receiver subscribes to all the pressure and temp topics with deadline 100.

Node 127: Signal receiver subscribes to all the temp, pressure and current topics with deadline 200.

Node 45: Signal receiver subscribes to all the temp topics with deadline 40.

Node 89: Signal receiver subscribes to all the current and pressure topics with deadline 200.

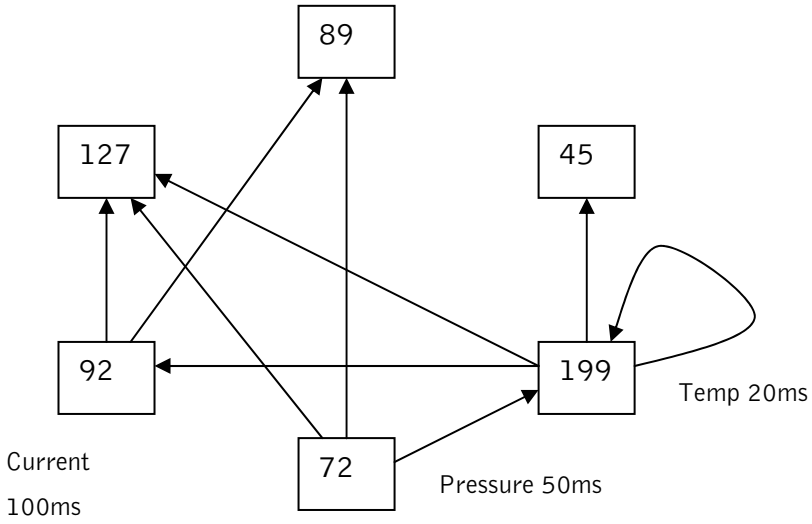

Figure 29 The basic scalability test configuration

Figure 29 shows the basic configuration. Next to each publishing node, we have the name of the topics that are published as well as the sending period. E.g. node 199 publishes topics temp1, temp2, …, temp100, so it will send a total of 100 issues (1 for each topic) every 20ms). An arrow from node A to node B indicates that B subscribes to all of the topics that B publishes. For example, 89 subscribes to 200 topics, (all the pressure and current topics).

Each signal generator will publish 1000 issues for each topic

Clock synchronization components were run at real-time priority on Windows and -15 on Linux (0 is default and -20 highest.) Other test components were run at high priority on Windows and -10 on Linux.

8.3 Test Cases

8.3.1 Multirate cyclic transfer of measurement data (TC-S.1)

8.3.1.1 Setup

Test-case id: S.1

Purpose: Here we test the performance and reliability of data transmission under a typical load (data is transmitted cyclically and at several different frequencies.) In this test case we do not generate any peaks or other problem situations.

Method: Exactly as described in 8.2.2.3.

8.3.1.2 Approach

In this test we had 300 topics, and 1000 issues were published for each topic. Each topic was subscribed to by at least one subscription. Therefore, we cannot present the data in much detail – detailed analysis has been carried out for the functional tests. Here we will display summary data and try to understand what kind of performance and reliability can be expected from a system which is under a realistic load.

8.3.1.3 Comparison of 2 signal receivers

In this section, we compare the statistics for all of the topics that were subscribed to by the receivers on nodes 127 and 199.

For each topic that was subscribed to, the signal receivers have recorded the minimum, maximum and mean latencies as well as the standard deviations. Remember that the following topics were published:

'temp1' to 'temp100' with period 20ms on node 199

'pressure1' to 'pressure100' with period 50ms on node 72

'current1' to current100' with period 100ms on node 92

Before looking at the results, it must be remembered that normal operation of NDDS is based on a first-come first-serve principle. [Wang 2002] The design goal has been to minimize the overhead caused by middleware processing, which means that any certain topics are not prioritized over others. Therefore, we will observe better latencies for those topics whose issues happened to be processed first. The order in which topics are processed is left unspecified, although we will make some practical observations about this. However, since we have grouped all the topics that an application publishes under a single publisher, we can never count on any particular topic being serviced before others. Therefore, in order to gain a comprehensive understanding of what kind of performance can be expected, we will need to look at all of the topics being published. This will give us an idea of the average, best and worst case performance.

Figure 30 illustrates the statistics for all of the topics that were subscribed to on node 199.

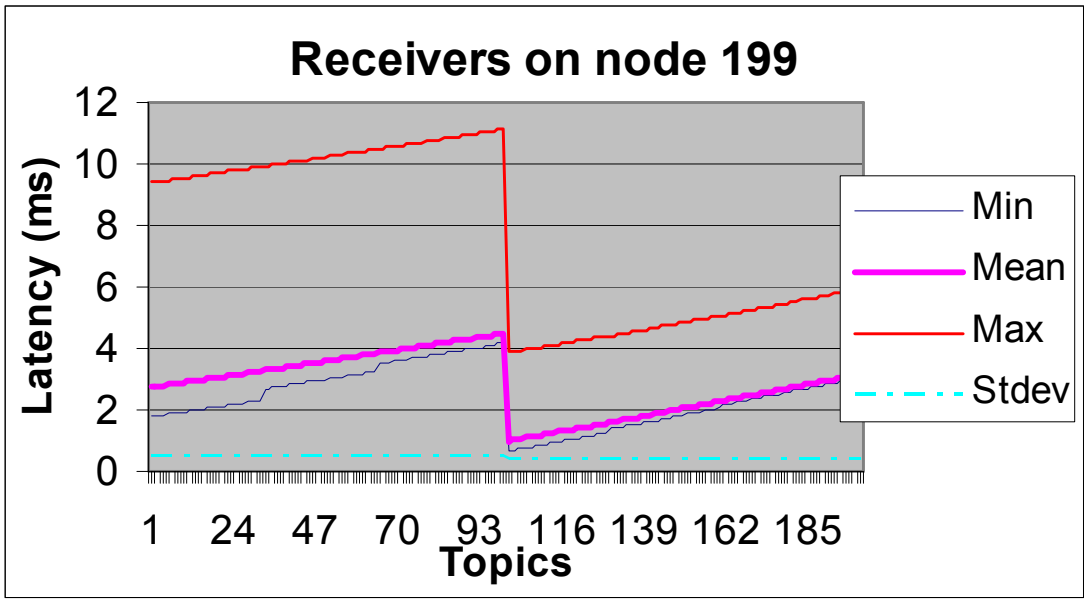

Figure 30

There were 200 subscriptions; one for every pressure and temp topic. For each topic, we have plotted the mean, minimum and maximum latencies as well as the standard deviations. (The plot should therefore be discrete, but with 200 topics the individual points could not be properly distinguished. We are using a continuous plot for the sake of clarity.) From the logs we can see that all of the pressure topics come first, followed by the temp topics (so that value 47 on the horizontal axis corresponds to topic 'pressure47' and 139 to the topic 'temp39'.)

Notice the very linear increase in latencies for the topics that were grouped under one publisher. We have observed and analysed this behaviour in test suite 2. Briefly, this is because the issues are sent in the order that the topics were added to the publisher, and the receiver's processing capacity is the bottleneck. Incoming issues will have to wait for the receiver thread to process the previous ones, and this wait time increases linearly as new issues arrive (the increase is linear when issues arrive at a constant rate and the processing time for one issue is constant.)

We can see that the temp topics (the last 100 on the chart) have been received more quickly than the pressure topics. This is because the publisher component for the temperature data was on the same node.

Now consider a Figure 31, which shows statistics for all of the 300 topics that were received on the node 127. First we have the 100 pressure topics on the left, then the 100 current topics and finally the 100 temp topics (e.g 253 corresponds to the topic 'temp53').

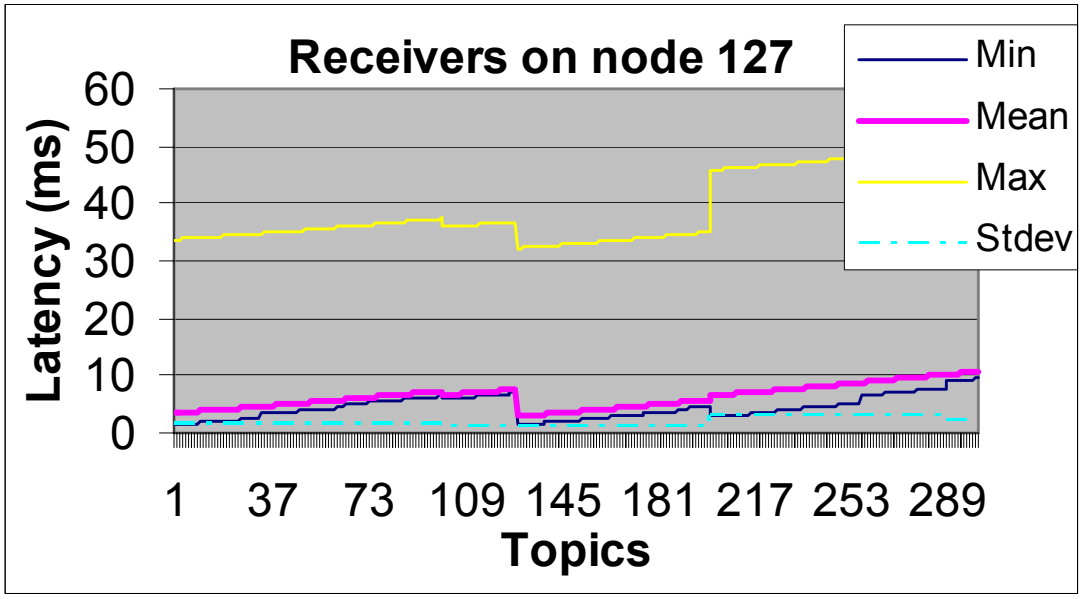

Figure 31

We can make the following observations:

- The curves are not always linear. Here we have an exception to the general behaviour, where topics are processed in the order that they were added to the publisher. However, from the previous chart we inferred that issues for the 'temp' and 'pressure' topics were sent in order, so they should also be received in order on every node. A closer examination of the receivers on 127 reveals that the nonlinear behaviour only applies to the 'current' topics, so there is no contradiction.

- The latencies on 127 (running Linux) are roughly twice as long as those on 199 (running Windows XP). However, 199 has 4 times more RAM and processor speed than 127. 127 has 300 subscriptions where 199 has only 200, but 199 also has a publisher component with the shortest period (20ms). Taking all of these factors into consideration, it is difficult to evaluate the performance of the operating systems.

- From the statistics on node 127 and 199 we see that there is a very strong relationship between the mean and maximum values (i.e. the shapes of the curves are very similar). This could be taken into consideration when setting limits on a system's average and worst case performance.

- The minimum latencies for 1000 issues should give a good idea of the best case performance. We see that the average and best case results are very close to each other. The worst case latencies were much higher, and we suspect bad scheduling and/or the execution of some kernel tasks to be the cause of this. Switching to a RTOS should clearly improve the worst case performance. The average performance is likely to improve when an OS with good interrupt handling capabilities is chosen. Here we are receiving thousands of issues every

second, and each incoming issue causes an interrupt. Windows and standard Linux have a bad reputation for handling interrupts in such quantities.

- Looking through the other logs, we see that the worst latencies were incurred by the subscriptions to the 'temp' topics on 127. The highest mean is nearly 11ms and worst standard deviation over 3ms. For most control algorithms, a latency of 11ms would be acceptable as long as new signals arrive at a fixed rate. The jitter of 3ms would probably not cause unstable behaviour from the part of a typical algorithm.

8.3.1.4 Comparison of the receivers of the pressure topics

The nodes 89, 127 and 199 had subscriptions to all of the pressure topics. In Figure 32, we have plotted the mean latencies for each topic. (e.g. at 56 we have the mean latency for the topic 'pressure56', which was nearly 4ms on node 199).

Since issues are processed as they arrive, we would expect that the curves for all of the receivers have the same shape. Indeed, they are all linear. The slope of the curve depends on how slowly issues are processed. Slow processing means that issues arriving later have to wait longer, so the increase in latency (the slope) is greater.

Node 199 has the fastest processor while the CPU on 127 is slowest. We observe that the curve for 127 is steepest and the curve for 199 is flattest, as was to be expected. Node 199 has a vastly superior processor, so we might have expected an even greater contrast. However, the publisher component on 199 must have taken its share of the processing resources.

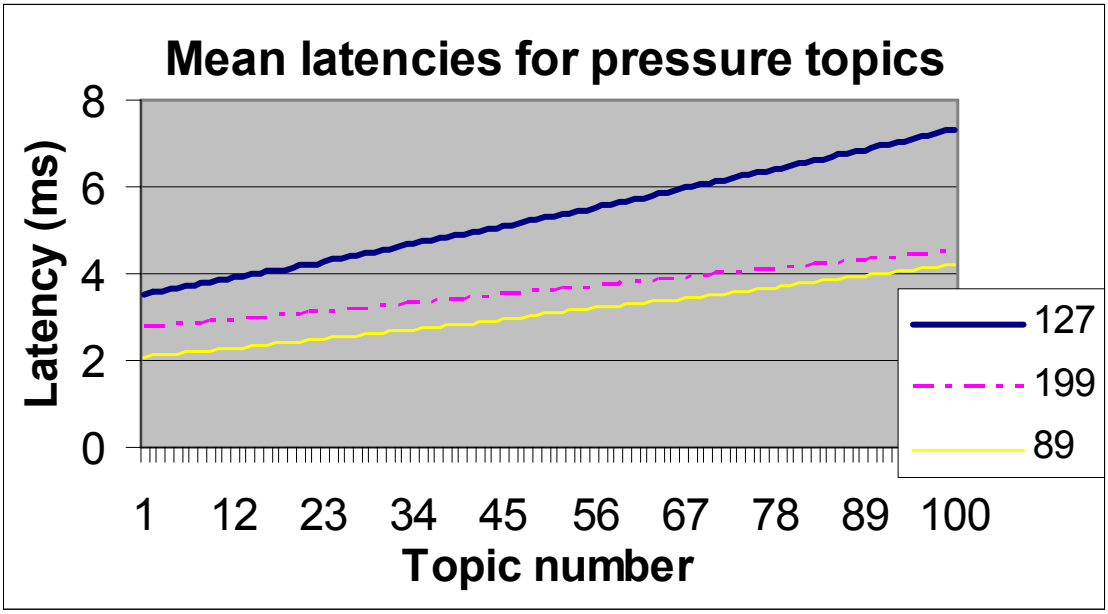

Figure 32

8.3.1.5 Comparison of Windows and Linux

The Windows node 45 and Linux node 92 both had signal receivers that only subscribed to the topics 'temp1' to 'temp100'. Figure 33 shows a comparison of the latencies:

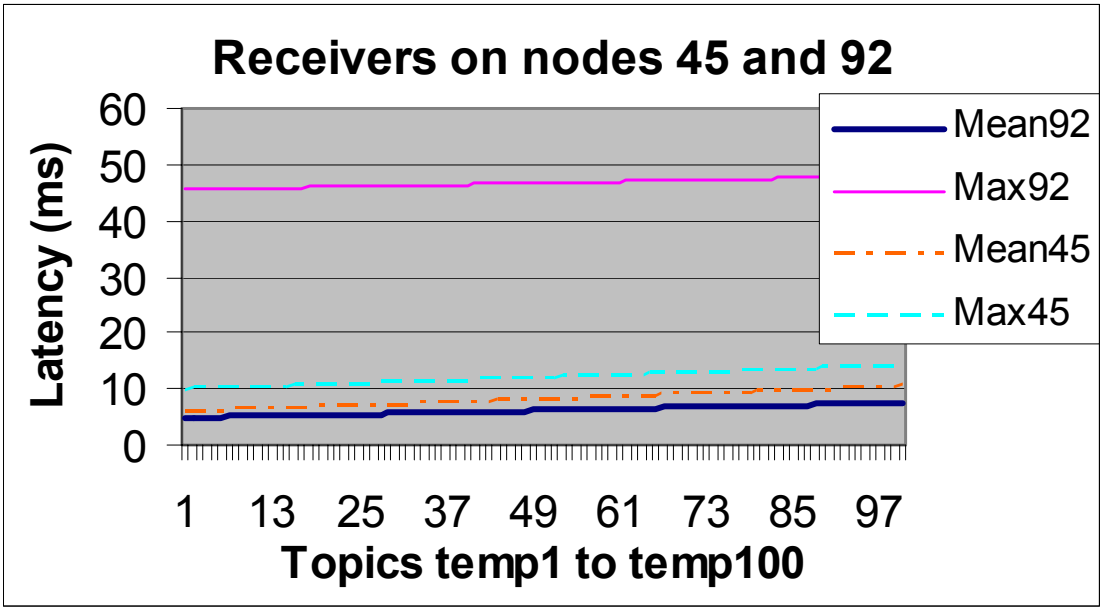

Figure 33

Although the Windows node had a 25% faster CPU and the Linux node had the extra task of publishing 100 issues for the current topics every 100ms, we see that the mean values are better on Linux. We conclude that Linux handles interrupts more promptly and with less operating system overhead. Issue processing is also more efficient, since the curve for the Windows machine has a steeper slope. However, in this and many other charts we can see that the Linux nodes often get a much higher maximum value. The main cause for high maximum values is the scheduler. Although these basically run the prioritized round robin algorithm, their behaviour is made unpredictable by modifications such as anti-starvation and priority boosting mechanisms.

8.3.1.6 Start-up times

The start-up times are measured by taking the difference of 2 timestamps: one stamp is taken before the subscriptions are created and the other when a subscription gets its first issue. Figure 34 shows the start-up times for all of the 100 pressure topics on the three nodes that subscribed to them.

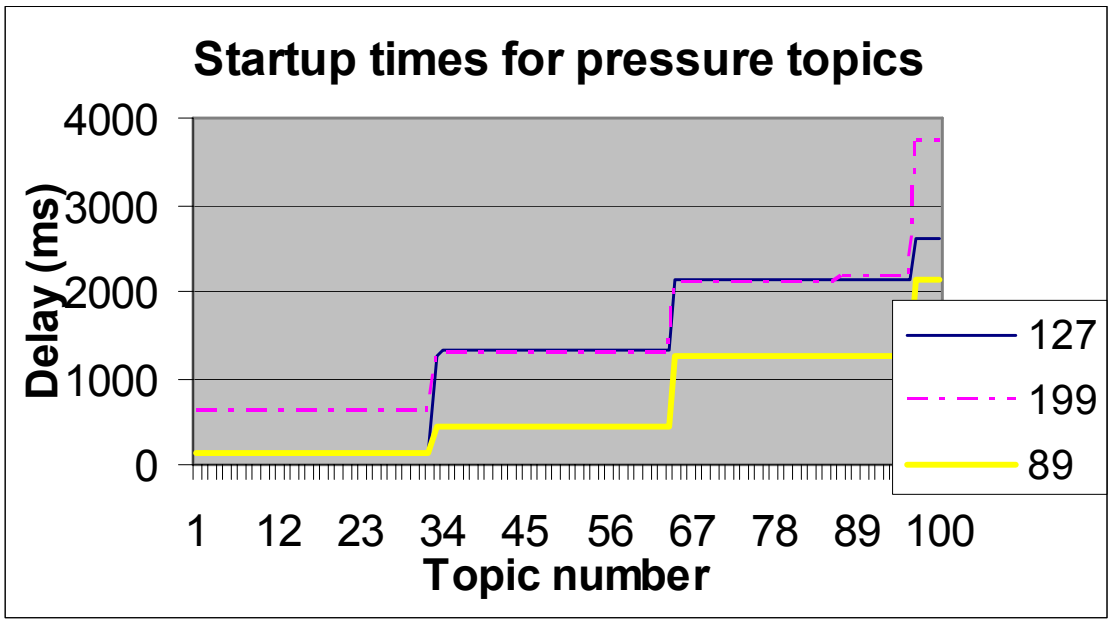

Figure 34

It looks like the NDDS managers on the different nodes communicate information about new topics to managers on other nodes in batches (in this case, one batch had information for around 30 topics and new batches came at intervals of roughly one second.) Although the curves are not linear, the worst case start-up times seem to be proportional to the number of topics.

From the other logs, we see that the average case behaviour is similar to what is indicated by these curves. However, the worst start-up time was 7.3s for topic 'current100' on node89.

8.3.2 Alarm bursts (TC-S.2)

8.3.2.1 Setup

Test-case id: S.2

Purpose: We generate bursts of alarms as in the functional tests in communication case 2. The purpose here is twofold:

- to observe the effect of the other real-time traffic on transmitting the alarms, and
- to observe the effect of a burst of alarms on the cyclic transfer of measurement data.

Method: The following components are added to the basic setup:

Burst level 3 is used.

Generator on 72 publishes 7000 bursts (each containing 3 issues) for topics in elist1.txt with period 20ms.

Receiver on 45 subscribes to topics with pattern */*/*/*/*.

Queue lengths are 5 and table size is 0. All publication's have sendMaxWait = 20s (i.e. how long to block the send call if the send queue is full. This should prevent sends from failing. The reasons for this approach are explained in section 7.3.4.2 of [Sierla 2003 B].)

Receiver's log is named LS2_Event.txt.

8.3.2.2 Effect of Load on Event Transmission

This test case is similar to 2.2. The differences are that the system is under a greater load from all of the cyclic data transfer. Also, the subscription pattern in this case matches all 10 published events, while the receiver on 45 for test case 2.2 subscribed to only half of these.

Since we sent bursts of 3, we again show statistics for the sequence number modulo burst size (i.e. the modulo remainder for the first issue of a burst is 1 and the modulo for the last issue in a burst is 3.)

Figure 35 shows the mean and standard deviations for the first topic received in the functional test case 2.2. The corresponding statistics for the first topic received in this scalability test (MeanS and StdevS) are also plotted. The means are nearly the same. This should be so, since the CPU load is far from 100% -- problems arise when issues are sent by the signal and event generators on node 72 at nearly the same time (Or when the signal and event receivers on node 45 both get issues – one of them has to wait.) The existence of other components that compete for the same resources explains why the standard deviations in the scalability test were so much worse than in the functional tests.

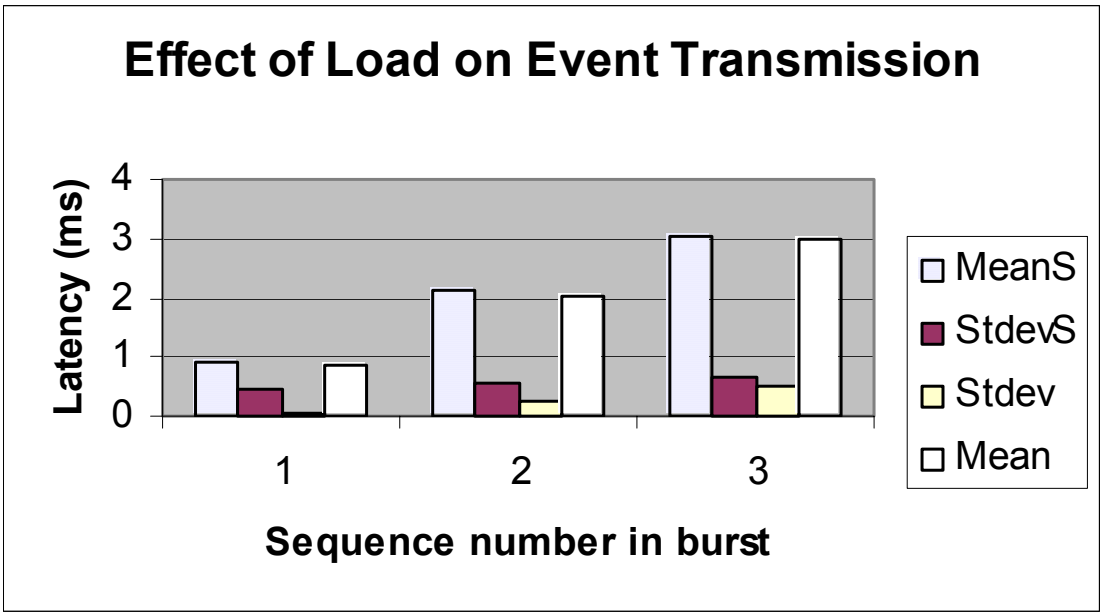

Figure 35

8.3.2.3 Effect of Event Bursts on Cyclic Data Transmission

In the previous section we looked at how the cyclic data transmission in the background affected the transmission of a burst of events. Here we observe the effect of event bursts on cyclic data

transmission. Figure 36 shows the mean and standard deviation for all topics in the signal receiver on node 45. (Remember that the event receiver was on that same node.) In test case S.1 we had the same signal receiver on node 45 with no events, so we have also plotted the values for that as well (MeanS.1 and StdevS.1).

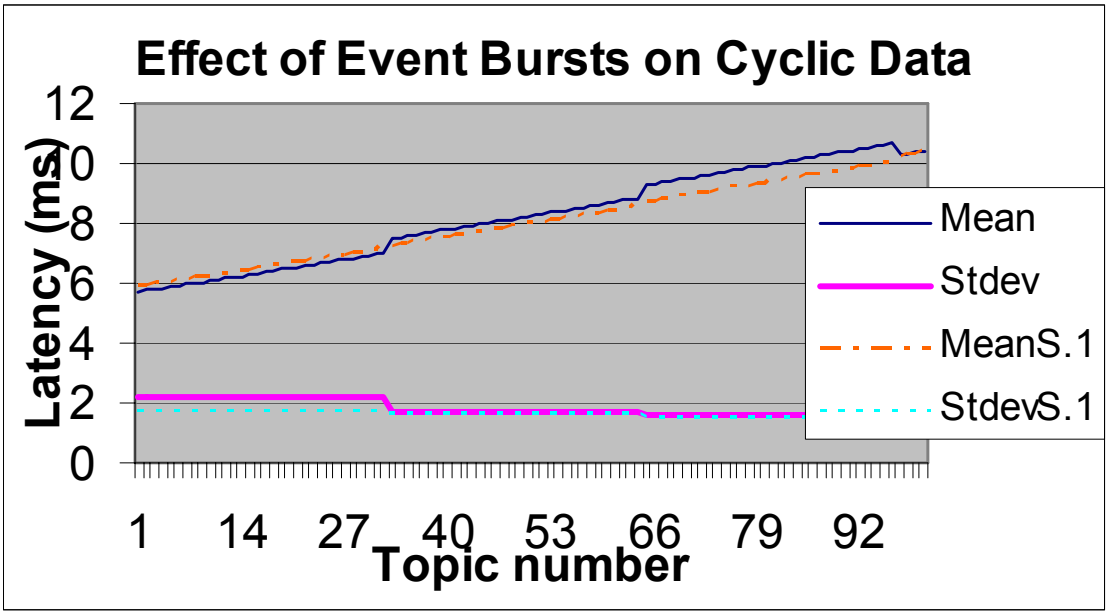

Figure 36

Again, we see little difference in the mean values, as this is presumably because the CPU load was not near 100%. However, the curves are less regular in this test case, which is probably because the event and signal components sometimes had to compete for the CPU. We can say that a few event bursts (3 events for 10 topics every 20ms) do not significantly impair the cyclic data distribution.

8.3.3 Fast rate control (TC-S.3)

8.3.3.1 Setup

Test-case id: S.3

Purpose: There might be some control loop or other process that needs to communicate faster than the 20ms cycle in S.1. In the functional tests we have observed that it certainly is possible to go below 20ms, but we had a light system load. Here we have the same arrangements as in S.1, but we also include a signal that is transmitted at a clearly faster rate.

Method: We add the following components to the basic setup:

A second signal generator on node 72 publishes topic 'level1' at a rate of 3ms. 25000 issues are sent.

A second signal receiver on 45 subscribes to this topic with a deadline 6ms. Its log file is LS3_fast.txt.

8.3.3.2 Results for the fast rate publication

Looking at these results, it must be remembered that the total send time for 'temp' topics was 20s, for 'pressure' topics 50s and for 'current' topics 100s. Since the node that published 'level1' had a resolution of 1ms, the sending period was close to 4ms and the total send time was close to 100s.

Sending 5000 issues every 4ms takes around 20s, and the latencies for later issues of 'level1' drop significantly after this, as can be seen from Figure 37. This is the time when the subscriptions to the 100 'temp' topics get their last issues. It is clear that the volume of incoming data affects the latency and jitter. This correlation is further illustrated by the beginning of the graph – notice that it takes around 8 seconds before we reach the worst case performance. Looking at the logs for the receivers on node 45, we see that around one third of the subscriptions got their first issue in less than 3 seconds after the component was started. For the rest, there was a delay of 6-8 seconds. In 6 seconds 1500 issues and in 8 seconds 2000 issues of 'level1' were sent. At this point there is a very obvious increase in the latencies for the 'temp' topics.

The quick flow of 'level1' issues does seem to slow down the start-up of some 'temp' subscriptions. In the basic case (S.1) the worst start-up time was 2.1 seconds, and in this case it was 8.1s.

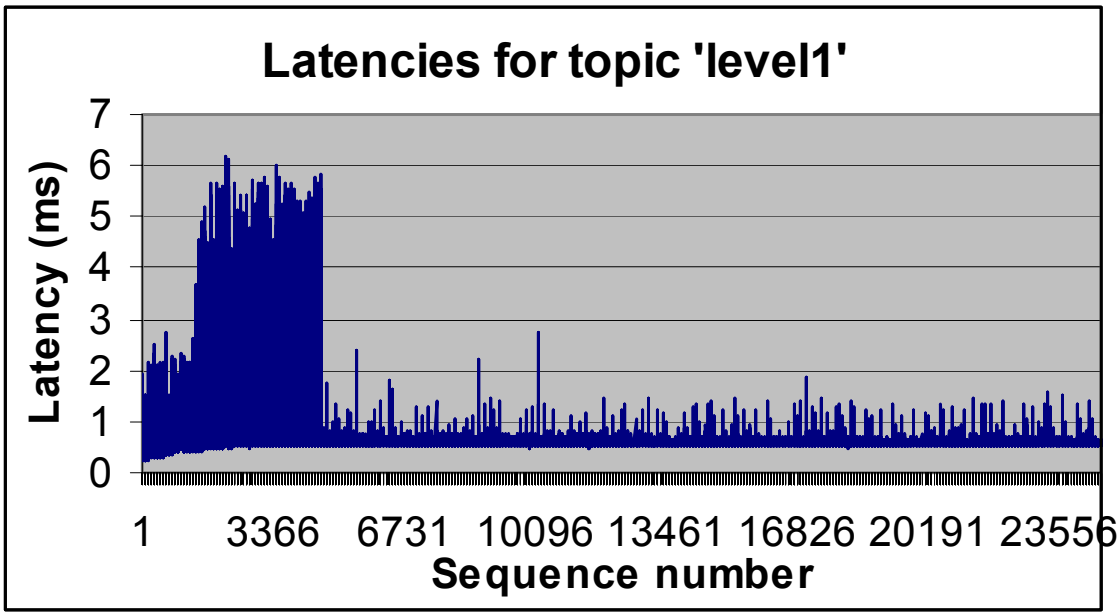

Figure 37

After the 'temp' publications shut down, we observe steady behaviour with 'level1'. After 50s, the 'pressure' publications (on the same node that published 'level1') shut down, but we see nothing remarkable at this point. This might be because the sending end is not the bottleneck, as we have seen before. However, the sender only takes the timestamp right before the packet is sent off, so we would not see the effect of congestion on the sending node.

8.3.3.3 The effect on the other subscriptions on node 45

In all scalability tests, we have subscriptions to topics 'temp1' to 'temp100' on node 45. We want to see if the fast rate transfer of issues for 'level1' will interfere with the 'temp' topics. Figure 38 shows the mean and standard deviation for each of the 100 topics in both the basic configuration (S.1) and in this case (curves labelled S.3). The different shape of the mean curve is only caused by issues for the 100 topics being sent in a different order. We conclude that there was no significant performance drop.

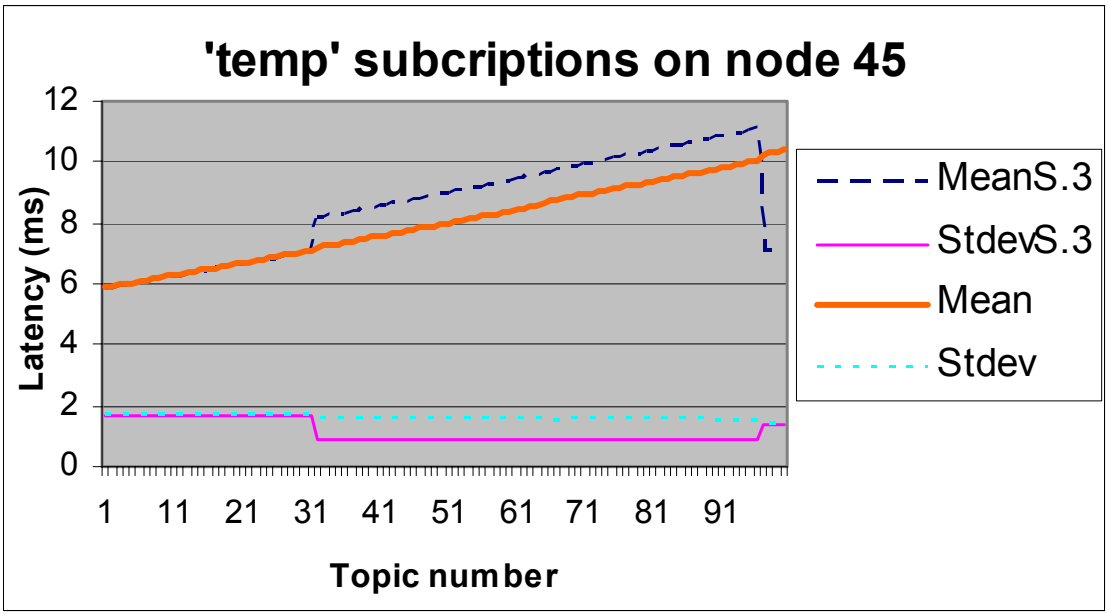

Figure 38

8.3.4 Dynamic start-up time (TC-S.4)

8.3.4.1 Setup

Test-case id: S.4

Purpose: Consider the situation where an operator in the control room opens a screen that displays various measurement data. We want to know how long it will take to establish the data flow from the sources to the control room application, i.e. how long will it take for us to get the first data for each signal.

Method: The signal receiver on 127 will have a delay of 10s. Otherwise, we have the same arrangements as in the basic situation.

8.3.4.2 Start-up times on node 127

Figure 39 shows the start-up times for all of the 300 topics that were subscribed to by the receiver on node 127. We again observe the same behaviour as in the basic case: subscriptions are started up in

groups of around 30. However, new groups are started more quickly in the case of S.4, where the subscriptions appeared at run time and the publications were already running. From the shape of the curve we are tempted to conclude that the start-up times for dynamically appearing subscriptions are also more predictable, but we would have to run more tests to be sure.

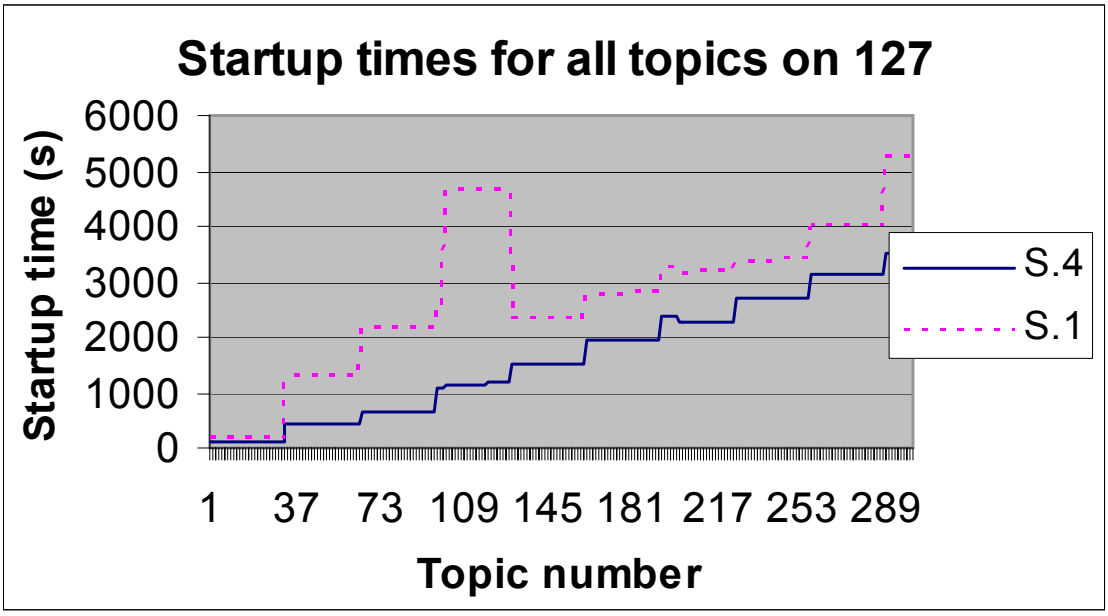

Figure 39

In this case it took 3 seconds to get data for all of the 300 subscriptions. This time can be decreased by grouping more data points under a single topic. Also, the CPU on 127 had only 350 MHz, so on a typical high-end control room workstation we could expect much better results. In this light it seems feasible to implement control room software that creates subscriptions to the necessary topics when a screen is changed. However, this is not the intended way of using NDDS [Wang 2002]. It is also clear that creating and destroying large numbers subscriptions and communicating the new status to the manager on the publishing node will consume resources.

8.3.5 Increasing the load with large data structures (TC-S.5)

8.3.5.1 Setup

Test-case id: S.5

Purpose: The event generator has a variable length array field which can be used to create packets of size 62kB. We will send these at a rate that is as great as the middleware can manage when there is no other load. The purpose of this test is to see that the performance of the middleware degrades gracefully.

Method: We will add the following components to the configuration in S.1:

An event generator is added onto Node 72; it sends a burst of 3 issues for each topic in elist1 every 50ms. A total of 1000 bursts are sent.

Queue lengths are 5 and table size is 62000. All publication's have sendMaxWait = 20s

Receiver on 45 subscribes to topics with pattern mixing/*/*/*/1-5.

The generator had a wait of over 10s, since there were not subscriptions to all topics in elist1. For convenience, we had a similar delay in starting up the pressure publisher component on node 72.

Receiver's log is named LS5_Event.txt.

8.3.5.2 The Latencies for the large data structures

The purpose of this test case is to observe the behaviour that results when the CPU is overloaded. From the results in test case 2.6, we concluded that parameterising the event generator in this way will consume all processing resources, i.e. new issues queue up and are sent as fast as possible. The queue never gets empty.

In this section, we look at how the reliable publish-subscribe mechanism handled the bursts of large issues. In test case 2.6, we only had this topic, while in S.5 we have all of the usual cyclic data transmission going on in the background.

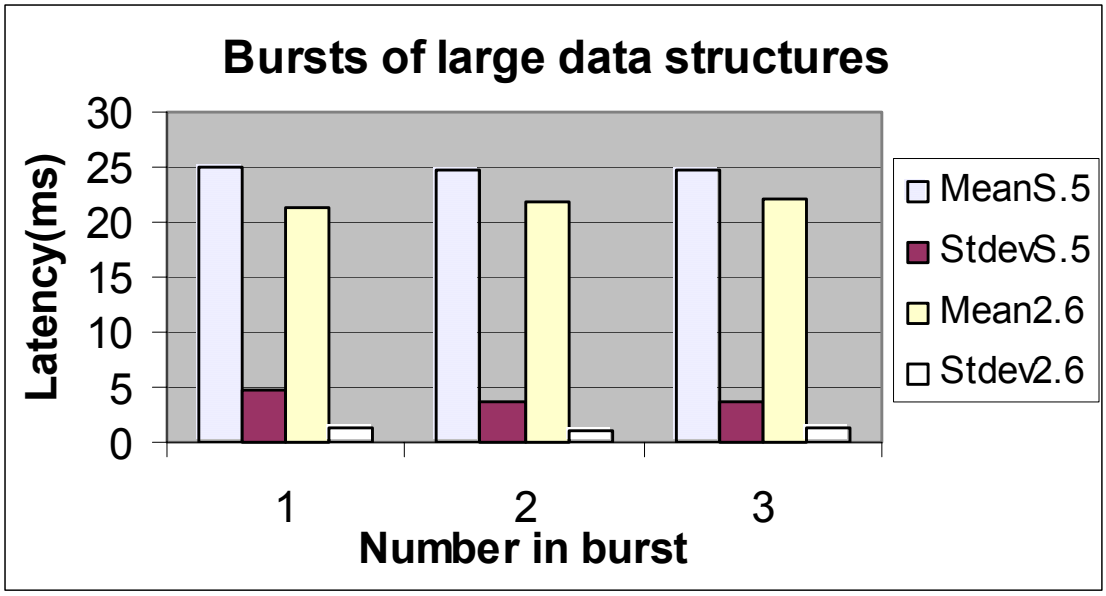

Figure 40

The reliable mechanism was able to transmit all of the issues despite the load. From Figure 40, we see that the mean latencies were slightly higher and jitter was clearly higher in the scalability test. Neither was it possible to maintain the 50ms sending period, since sending one burst took more than that. The performance degradation is therefore graceful, as it is caused by the load exceeding the processing capacity. There was no unexpected behaviour, and all issues were received in-order.

In this light, it is not necessary to invest in hardware and network resources in such a way that even the worst case situation will not strain the processing capabilities. When the rare worst-case peak load is experienced, there will just be a slight degradation in QoS, which in most processes will only result in a small and temporary drop in quality. Often, the cost of this is much less than having to invest in resources for handling the worst case.

8.3.5.3 The effect on the other topics

Now we look at how the transmission of large events affected the cyclic transmission of data. The node that published the events also published the 'pressure' topics. Figure 41 shows statistics for these topics when the event generator was running at the same time (S.5) and in the basic case with no events (S.1). We show the statistics for the signal receiver on node 89.

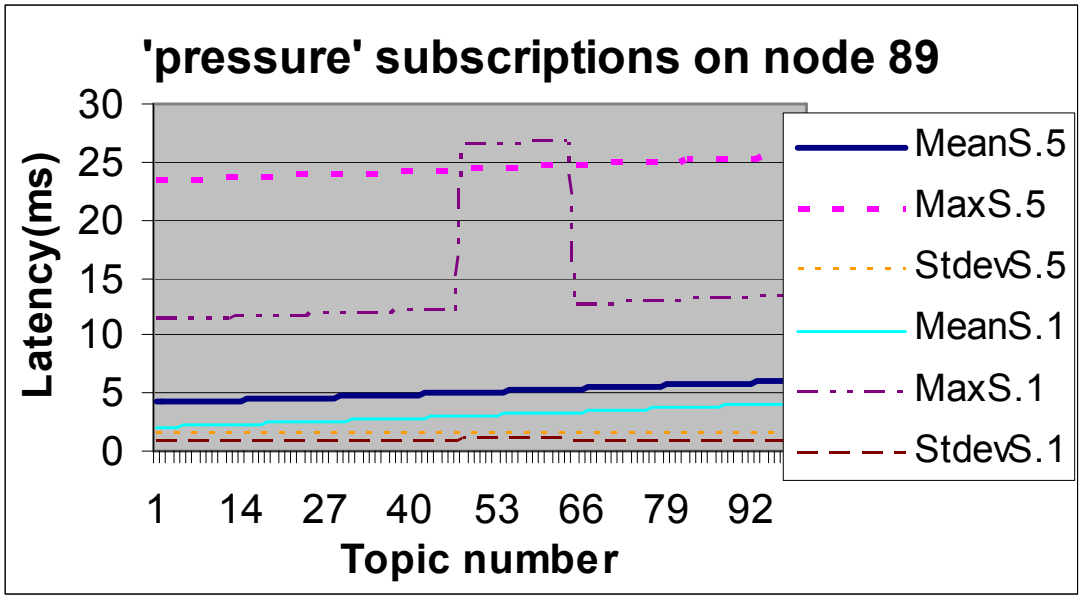

Figure 41

This graph must be viewed with caution. The sender took timestamps just before the data was actually sent, so the data might have had to wait a while before this because the event generator was running. Otherwise, there would presumably be a greater difference between the results of S.1 and S.5. Also, neither RTPS nor NDDS specify the order in which issues for the different topics are processed. This means that some subscriptions will get better service than others, and the behaviour might be different if the test is repeated. This could explain the peak in the MaxS.1 curve.

Having said this, the graph shows quite a clear pattern of moderate performance degradation under the peak load in S.5. However, even though issues for the pressure topics were sent in the best-effort mode, none of them were lost.

8.4 Conclusions

The suitability of a middleware product for an automation application cannot be expressed with any simple figure. Many different scenarios must be considered, since application components have diverse communication needs with varying performance and reliability requirements. A realistic system has to handle massive volumes of data, so the effect of scaling up the system on each of these scenarios needs to be known. For this reason, we have tested the middleware in a number of different circumstances that are relevant to process automation systems. After reading these results, we believe that the reader will have a fairly accurate idea of what kind of performance can be expected.

Finally, we again point out that there is much that RTPS leaves unspecified. Especially, the order in which publications and subscriptions are serviced usually cannot be reliably predicted at compile time. The programmer has limited control over the order in which things are done, as NDDS typically operates with the first-come first-serve principle. (It is possible to, for example, individually send issues for all publications synchronously in a certain order. However, using the middleware in unintended ways is unlikely to yield satisfactory results.) In the scalability tests, we noticed how the results for some topics were much better than others, although the application programmer had not prioritized any of them. This is why presenting a more limited amount of results would be misleading, since we would not know if we are looking at best, average or worst case behaviour.

9 Conclusions

9.1 General-Purpose Evaluation Criteria

In this section, we will review the main goals for middleware products and evaluate NDDS from this perspective. These goals are:

- encapsulation of OS and network levels
- control of real-time behaviour
- dynamic reconfiguration

In section 9.2 we will discuss some of the special requirements for process automation.

The middleware is above the operating system and network layers, so the programmer does not need to have any detailed knowledge of these levels. Therefore, the application is portable and can be easily reconfigured, since it depends only on the middleware's API. In our experience, NDDS lived up to these expectations. Scaling up the system by adding new nodes and components or moving components onto different nodes could be done without even recompiling any code. Porting components from Windows to Linux was also straightforward. First, we ran the nddsgen utility that generates C++ code from our data type definitions that we used to develop the Windows components. Then the source code we wrote for the Windows components could be recompiled on Linux without making any changes to the NDDS API calls.

Unfortunately, we are still inclined to believe that high quality real-time applications cannot be built without having some knowledge of the inner workings of the operating system and the network. Although it was possible to develop the application without fixing any components to physical resources, achieving the optimal real-time behavior is another thing. For example, with a group of slow Linux machines with very limited memory, we sometimes observed long delays (ca. 15s) before the subscriptions started receiving issues. The issues that were sent before this were lost. Presumably, NDDS was sending metatraffic in order to establish the communication links [Wang 2002]. With some systems, designers must have an accurate idea of how long it takes before these links are established, so it would then be necessary to understand what is happening and where the bottlenecks are. This insight cannot be gained only by debugging the application code, but an Ethernet packet sniffer would be useful. RTI also sells a product for monitoring the RTPS traffic on the network [Wang 2002].

RTPS does give the programmer control over a number of QoS properties, which can be used to influence the real-time behaviour. However, in our analysis of the test results, we had to mention frequently that the outcome will always ultimately depend on how the operating system schedules the various threads of the application and middleware. Since we did not use a RTOS, we noticed that it is not possible to guarantee a satisfactory upper bound for latencies, because even a high priority thread

might be kept waiting sometimes. When a high level of determinism is required, a suitable RTOS must be used [Kindel 2002], but the programmer should also understand the scheduling algorithm as well as the responsibilities and interactions of the threads in the application and middleware.

The preceding discussion is a good introduction for another powerful capability of middleware products: the dynamic reconfiguration of a system while it is running. This naturally requires that the system can be reconfigured statically, but it is equally important that the middleware will discover any configuration changes and update communication paths in a timely manner. In our tests, we never noticed that NDDS would have failed in this, and the response times were very satisfactory for most applications. However, in the next section we will describe a scenario encountered in process automation systems that places heavy requirements on the speed of dynamic reconfiguration.

9.2 The Special Requirements for Process Automation

9.2.1 The Logical Design of the Application

The process automation industry has some unique requirements for the logical design of the application that the middleware must support. Much of this has already been discussed in the sections about communication mechanisms. In general, we and our industrial contacts feel that RTPS is a very promising standard. The benefits have already been described at length throughout this thesis, so in the conclusion we will focus on potential difficulties.

1. Event-driven data distribution requires that the integrity of a group of signals must be guaranteed (i.e. they are from the same iteration of the algorithm that produced them). Yet it is not possible to group them into a single data structure, because this will lead to maintenance problems. It might not even be desirable to try to provide this functionality in middleware, since other design goals might be compromised. An application-level solution that utilizes appropriate subscription mechanisms and reliable delivery could be the most natural approach. In chapter 5, we proposed the container design pattern for this purpose.
2. The event subscription mechanisms could be more versatile, and we have discussed this in the section for the event notification and acknowledgement services (4.3.3).
3. Notifications must be acknowledged at the application level. There are many acknowledgement models that should be supported, such as one, some or all of the receivers of an alarm or notification must send an acknowledgement. No middleware standard that we have studied supports this, and it might well be the natural solution to build this functionality into system components that manage alarms.

9.2.2 QoS Control

There is a great volume of traffic in an automation system, but there are significant variations in the requirements for latency and determinism. Ultimately, achieving good performance is neither a matter of minimizing latencies or maximizing throughput but of making sure that the right data arrives at the

right destination at the right time. It should therefore be possible to prioritize the time-critical data. This does not only mean that the middleware should process the data in the right order, but the work must also be done by a thread with a suitable priority.

We have already said that the NDDS implementation tries to minimize any overhead caused by the middleware, so there are very limited ways for prioritizing any issues over others. The advantage of this is that publish-subscribe is a very efficient mechanism for data distribution tasks. The disadvantage is that time-critical data might be needlessly delayed by issues that could wait a whole second. For example, subscriptions in diagnostics and monitoring components or a history database could be kept waiting so that the signals for the real-time control of the process can be delivered before their deadline expires. Finally, with NDDS it is possible to adjust the relative priorities of the threads involved, but it is not possible to prioritize certain topics over any others in this way.

The designers of the CORBA Notification Service have taken a different approach, and so their standard has different advantages and disadvantages [OMG 2002]. There are sophisticated algorithms for processing event messages. A message can, for example, have a priority and a deadline. The application programmer could request that messages are sent with a first-come first-serve principle, or that the highest priority messages are sent first. If the processing capacity is exceeded, the deadlines can be used to determine which messages should be discarded. Real-time CORBA also gives a fine level of control for specifying the priorities of the threads that are used to carry out the services. These mechanisms provide very interesting possibilities for achieving the desired level of determinism for time-critical data. The major disadvantage of this approach is that the heavy mechanism is rather inefficient for distributing great quantities of data.

RTI is working together with the OMG to produce a DDS (Data Distribution Service) standard, and the specification should be published in the summer of 2003 [Wang 2002]. A publish-subscribe mechanism will be used to provide efficient data distribution services. Many of the Notification Service's strengths that we have described throughout this thesis will also be found in DDS.

9.2.3 Dynamic Configuration Changes

We mentioned in section 9.1 that NDDS's ability to handle configuration changes at run-time was very acceptable. We now describe one important scenario that places heavy requirements on the speed of dynamic reconfiguration.

In process automation, control room software typically needs nearly every topic at some point, so it will have hundreds or thousands of subscriptions. It is considered essential that data is not transmitted unnecessarily, to conserve bandwidth and processor resources on the subscribing node. There should be some better way of turning off the subscriptions when they are not needed (when the operator opens a different screen) than destroying it. Having control topics that are used to notify publishers when to send is troublesome when there are thousands of publications and subscriptions, so some support from the middleware would be useful.

It seems that creating subscriptions whenever the application needs them and destroying them some time after they have not been needed would be the most practical solution. It is desirable that the first issue would arrive within one second of creating the subscription. If the number of topics is limited by grouping related signals into data structures, this requirement might be met if powerful workstations are used in the control room.

10 References

1. [Dostoyevski 1988] Dostoyevski F.: Karamazovin veljekset, Karisto 1988
2. [IDA 2001] IDA: White Paper. 18.4.2001
3. [Metso] Metso Automation, http://www.metsoautomation.com 17.4.2003
4. [NTP] NTP, http://www.ntp.org 28.5.2003
5. [OHJAAVA] OHJAAVA, The Application of Java and Internet technologies in a Component Based Control System, http://www.vtt.fi/tuo/projektit/ohjaava/ 5.5.2003
6. [OHJAAVA-2] OHJAAVA-2, Modern Distribution Solutions in Open Control Systems, http://www.automationit.hut.fi/tutkimus/documents/Ohjaava/eohjaava-2.htm 5.5.2003
7. [OHJAAVA-212] Tommila T.: OHJAAVA-212, Automaation uudet hajautusratkaisut ja hajautuksen vaatimukset, OHJAAVA-project's technical document, unpublished, 31.3.2003
8. [OHJAAVA-216] Sierla S.: OHJAAVA-216, Evaluation of Communication Middleware – A General Test Specification, OHJAAVA-project's technical document, unpublished, 27.2.2003
9. [OHJAAVA-217] Sierla S.: OHJAAVA-217 Analysis of Middleware Solutions -- RTPS. OHJAAVA-project's technical document, unpublished, 10.2.2003
10. [OHJAAVA-220] Sierla S., Peltola J.: OHJAAVA-220 Analysis of Middleware Solutions – RT CORBA, OHJAAVA-project's technical document, unpublished, 12.2.2003
11. [OHJAAVA-227] Sierla S.: OHJAAVA-227, Evaluation of Communication Middleware – The NDDS Test Specification, OHJAAVA-project's technical document, unpublished, 20.2.2003
12. [OMG] Object Management Group, www.omg.org, 15.5.2003
13. [OMG 2002] OMG: The Notification Service http://www.omg.org/cgi-bin/doc?formal/2002-08-04, 21.10.2002
14. [Peltola 2002] Peltola J.: Distributed control systems — execution environment of a component based automation application, Report 6, Laboratory of Information and Computer Systems in Automation, Helsinki University of Technology, 2002
15. [Peltola 2003] Peltola J.: Automaatiojärjestelmien hajautusta palvelevat arkkitehtuurit, ÄLY-foorumi, Seinäjoki, 15.5.2003
16. [Raute] Raute Precision, http://www.rauteprecision.fi 17.4.2003
17. [RTI 2002] RTI: NDDS User's Manual Version 3.0. 330 pages, February 2002
18. [VTT] Valtion tieteellinen tutkimuskeskus, http://www.vtt.fi 17.4.2003
19. [ÄLY] Älykkäät automaatiojärjestelmät, Teknologiaohjelma, Tekes, 2001-2004, http://akseli.tekes.fi/Resource.phx/tivi/aly2000/en/index.htx 17.4.2003

Interviews and conversations:

1. [Kindel 2002] Robert Kindel, RTI, 9.2002 – 5.2003
2. [Wang 2002] Howard Wang, RTI, 9.2002 – 5.2003

HELSINKI UNIVERSITY OF TECHNOLOGY
INFORMATION AND COMPUTER SYSTEMS IN AUTOMATION

Report 1	Koskinen, K., Aarnio, P. (eds.), Internet-, Intranet- and Multimedia Applications in Automation. June 1998.
Report 2	Koskinen, K., Aarnio, P. (eds.), PC-based Automation Systems and Applications. June 1999.
Report 3	Mattila M., Prosessilaitteen etätukijärjestelmä – ohjelmistoarkkitehtuuri ja ohjelmistotekniset ratkaisut. March 2000.
Report 4	Strömman M., Ohjelmoitavan logiikan ohjelmointi ohjelmistotuotantoprosessina. March 2002.
Report 5	Aarnio P., Simulation of a hybrid locomotion robot vehicle. June 2002.
Report 6	Peltola J., Uudet automaatiojärjestelmät - komponenttipohjaisen automaatiosovelluksen suoritusympäristö. September 2002.
Report 7	Fortu T., Enterprise Resource Planning – Integration with Automation Systems. September 2002.
Report 8	Mattila M., Condition Monitoring of an X-ray Analyzer. February 2003.
Report 9	Sierla S., Middleware Solutions for Automation Applications – Case RTPS. June 2003.

ISBN 951-22-6604-0
ISSN 1456-0887
Picaset Oy, Helsinki 2003