Design and Implementation of a Distributed System for Synchronization and Supervision

L O V E   F L O R G Å R D

E x a m i n e r
D r .   C h r i s t i a n   S c h u l t e

Master of Science Thesis
Stockholm, Sweden 2006

ICT/ECS-2006-67

Design and Implementation of a Distributed System for Synchronization and Supervision

Love Florgård

love.florgard@rektron.se

Abstract

Development of distributed software does often suffer from the limited means of inter-process communication offered by the underlying platform. To solve problems involving communication, developers are forced to create complex and inflexible systems possibly with poor use of system resources. Systems distributed over local area networks do for example often use a shared central database as the only point of interaction. This thesis is focused on providing the means of flexible and reliable distributed computing to such a system. The insufficiencies with the current system are identified as well as the desired functionality. The conclusion is to design and implement a middleware system that provides direct inter-process communication with request/reply and publish/subscribe semantics. The system is decentralized, it does not rely on any database and its architecture is designed with respect to flexibility, transparency, accessibility and reliability. The implementation is based on the Ensemble distributed communication system and is written in C++.

Acknowledgements

Emil Åström and Daniel Persson contributed with valuable suggestions on several design decisions. All people at Rektron AB assisted in testing the system at different stages of development.

Table of Contents

1 Introduction

Development of distributed software does often suffer from the limited means of inter-process communication offered by the underlying platform. Developers are forced to create complex and inflexible applications, possibly with poor use of system resources, to solve problems involving communication. This can be the case especially if there are requirements on communication that the platform usually doesn't support, e.g. that communication should be asynchronous and real-time with high location transparency. This thesis is focused on providing the means of flexible and reliable distributed computing to an existing rigid client/server system. The desirable functionality is identified as well as the insufficiencies with the current system, with a central database and client applications running on different machines interconnected via a LAN. This document includes sections where the different techniques for reaching a solution are examined, the system architecture is designed, the work of the implementation phase is described and the final system is evaluated.

In this chapter the background to the assignment of the final degree project is surveyed, including an example of an application that is intended to use the system. The requirements of the system are then stated. Chapter 2 contains information on related systems and describes technology that the system is based on.

Chapter 3 considers the goals with the system and the design of the system architecture. The models of communication are described followed by a discussion on how they will be integrated in the specific system. The architecture design is specified on the basis of this discussion.

Chapter 4 handles the implementation of the system and related issues. The evaluation of the implementation is discussed in chapter 5 followed by conclusions of the final degree project in chapter 6.

1.1 The assignment

The final degree project that is the subject of this thesis was assigned by Rektron AB (http://www.rektron.se) for the company to acquire new possibilities of software development. Rektron develops software for statistical quality control (SQC) in the manufacturing industry. The applications give among other things functionality for measure data collection, real-time statistic calculations and report generation. A system in operation that uses Rektron's applications does often consist partly of computers where measure data is collected and partly of computers where the measure process is supervised or where reports are generated. The computers are usually connected to a local area network (LAN) and use a central database as shared storage.

Process instances of the applications run on different computers and do usually not transfer any information between each other more than indirectly via the database. Due to this it can be a rather complex task for a process to learn about relevant information on the state of another process. A general *distributed system* is required to make it possible for processes to request information on the state of other processes and to report changes in the own state. There is also a desire that operators not in front of any of the computers in the system should be able to receive messages about changes in the states of the processes.

The project resulted in a *middleware system* designed to support *request/reply* oriented and *publish/subscribe* oriented communication in a uniform manner. These two models of communication provide the tools for processes to share information on request and when internal events occur. The system is adapted to a LAN-based environment and primarily to be used in C++ applications. The design decisions are taken with respect to reliability, location transparency, low administrational requirements and flexibility in system structure and areas of usage.

1.2 A real scenario of usage

The intention is to use the system resulting from this final degree project in an application for supervision of the manufacturing process of plastic details. A number of *injection moulding machines* produce the details, which are then measured. Measure values are transferred to processes of statistical quality control software, *Rektron SQC*, provided by Rektron. There is one Rektron SQC process running for each injection moulding machine. The collection of measure values is called a *measurement process* and data from one measurement process is stored in an *order* in the database. A measure value is stored in the order as soon as it has been delivered to Rektron SQC.

The task of the supervision application is to show an overview of the machines and the status of the processes running on them. The status can be deduced from the interval between measure values delivered to Rektron SQC. When a process runs as expected the values are delivered before a certain interval limit so if no value has been delivered within this time the application should indicate a possible stop in the production.

The supervision application must know both in which orders measurement is active and when new measure values have been stored to be able to show the status for each process. The current system architecture offers only a rather ineffective solution of these two problems. To know which orders are active, run-time status information must be kept in the database. This information is not already in the database so it has to be added and maintained for this application solely. Measure values are stored in the database but the application will not be notified of them and needs therefore access the database frequently to notice new values. How often the database should be accessed is a balance between what time resolution is required during supervision and how often it is reasonable to do it due to the workload over-head.

A better solution would be to let Rektron SQC report directly to the supervision application when measurement is started or ended and when new values have been collected. This solution would completely remove the need both for alterations of the database and for superfluous database accesses. Further characteristics of a neat solution are to provide *location transparency*, so Rektron SQC does not need to know where the supervision application is located and vice versa, and flexibility with respect to the system structure, so processes can join and leave communication arbitrarily. These issues constitute part of the goals with the system designed and developed in this final degree project.

1.3 Requirements

At the project assignment a number of requirements were stated to define the characteristics of the final system. The primary requirements concern the fundamental patterns of interaction that should be provided and the additional requirements target the system properties in greater detail. The environment properties state a number of facts of the target system and constitute an additional base for design decisions.

1.3.1 Primary requirements

There are two primary requirements of the system, based on how applications must be able to interact. The first requirement conforms to a pattern of communication where it is the process that needs the information that begins interaction. The second requirement conforms to a pattern of communication where it is the process that has information to share that begins interaction.

P1. A process must be able to request information about the state of another process, possibly on another computer on the network.

P2. A process must be able to report changes in its state or in a shared state to another process, possibly on another computer on the network.

1.3.2 Additional requirements

In extent to the primary requirements a number of additional requirements is specified to control the properties of the architecture and semantics.

A1. The system should be generic and it should be possible to use it to solve arbitrary future problems related to the problems for which the system is designed.

A2. The system should be reliable and robust.

A3. The system should be accessible.

A4. The system interface should be transparent and easy to use.

A5. The system should be adapted to the development environment at Rektron AB.

A6. The system should be adapted to the environment of operation of the directly concerned customers.

A7. It is desired that the system is prepared for other possible environments of operation.

A8. The system should be easy to administrate.

A9. The system should be structured so it is fully adequate for real-time applications and scalable.

1.3.3 Environment properties

There are a number of known facts of the environment where the system will be in operation. These facts can be used to settle design decisions when there are several possible solutions to a problem.

E1. Computers that need to communicate are connected to the same LAN.

E2. Computers in the system are heterogeneous.

E3. Communication is asynchronous.

E4. Communication can be assumed to be sparse. A process can generate up to a few ten events per second, but usually less.

E5. The system can be assumed to be small. It will usually consist of up to a few ten nodes.

E6. The system might suffer from faults. Processes may crash. Network links may break.

E7. The primary software platform is Microsoft Windows NT/2000/XP.

1.4 Preliminary decisions

As a starting point for the final degree project, a few preliminary decisions were taken with respect to the requirements. The system will form a *middleware* layer, put between the application and transport layers to provide the required means of communication. This form makes the system easily deployed and applicable to general applications. Information will be transferred in messages to allow for different communication patterns in an environment of asynchronous communication.

Two different models of process interaction were put in focus for the state of the art study; *request/reply*-oriented communication and *event*-oriented communication, specifically with *publish/subscribe* semantics. The models are described in section 3.3. Group communication with message ordering protocols was taken into consideration to assure measure order consistency between processes in complex interaction patterns. This applies to situations were several processes can perform one-to-many send operations concurrently. Section 2.3 handles group communication including message ordering.

2 Background

2.1 Terminology

To understand the terminology used throughout this thesis the reader is assumed to have some knowledge in computer communication and software design. Basic knowledge in distributed computing will be helpful. The parts covering the implementation phase also touches terminology specific to object oriented programming and the C and C++ programming languages.

2.2 Related work

There are many middleware systems with different strengths and characteristics. The properties of the systems follow the numerous different goals that are targeted.

The *Common Object Request Broker Architecture* (CORBA) [OMG 2004a] is an RPC-based request/reply middleware solution. There is no hierarchy in means of client and server processes or machines; each process can be both server and client. Inter-process communication is taken care of by a small program module called *Object Request Broker* (ORB). An ORB provides stubs to objects in applications on other machines. Method calls are performed locally on the stub and the ORB passes the calls to the process holding the object, thus location transparency is achieved. Method definitions are predefined and cannot be changed without rebuilding the concerned applications. Data is transferred using the *Internet Inter-ORB Protocol* (IIOP) that specifies a typed binary data format.

omniORB [http://omniorb.sourceforge.net/] is an open source CORBA ORB for C++. It is for example used to connect C++ clients to the xmlBlaster *message oriented middleware* (MOM) system that provides point-to-point and publish/subscribe semantics [http://www.xmlblaster.org/].

The CORBA Event Service [OMG 2004b] offers distributed event handling in CORBA-based systems. It constitutes a platform for one-to-many communication using CORBA. The Borland VisiBroker Publish/Subscribe Adapter [Borland 2004 (chapter 4)] included in the VisiBroker ORB implements a publish/subscribe layer on top of the CORBA Event Service.

Ensemble [Hayden 1998] is a message-oriented distributed communication system that provides strong properties of reliability, fault tolerance and message ordering. It provides group communication in LAN-based environments. Ensemble is described in greater detail in section 2.4.

NaradaBrokering [Fox et al 2002] is a messaging infrastructure that provides publish/subscribe supporting several different schemes for subscription matching. There are topic-based matching based on strings, SQL-like queries, integers, tag-value pairs or regular expressions and content based matching of XML events using XPath queries. NaradaBrokering is implemented in Java and is JMS [Sun 2002] compliant. Communication in the system is performed via *brokers* that are connected to each other in clusters. Clusters are in turn connected in super-clusters and so on to provide scalability. Published events are routed on the different cluster levels to their destinations.

Web Services [W3C 2004a] provides request/reply communication in a client/server hierarchy. The server-side process is called *provider* and runs on a Web Services compliant web server. The client-side process is called *requester* and can run on any device that supports the standard Internet protocols [Gokhale et al 2002 (section 4)]. Data is transferred with the *SOAP* protocol using an XML-based data format.

The Apache Axis project [http://ws.apache.org/axis/] provides Web Service access in Java and C++ applications. It is used in the Globus Toolkit [http://www-unix.globus.org/toolkit/], an open source software toolkit used for building grids. The grid service of the Globus Toolkit provides a standard interface for applications and other services to inquire about their internal state.

WS-Notification [Graham et al 2004] is an event-driven interaction pattern specified for Web Services. Publish/subscribe is provided with topic-based event matching using XML topics.

2.3 Group communication

A system that involves more than two processes in communication does in many cases benefit from *group communication*. Group communication provides means of communication where a message is delivered to several processes in one single operation. With group communication it is also possible to acquire stronger properties of *reliability* and *message ordering* than in point-to-point communication [Défago et al 2003].

For processes to communicate there must be at least some basic operation of communication, a *communication primitive*. In point-to-point communication, the single communication primitive for sending a message is the *send* operation. This operation simply sends a message from one process to another. A corresponding *receive* operation is used by the destination process to receive the message. In group communication two additional communication primitives are added, *broadcast* and *multicast*. Broadcast is the operation for sending a message to all processes in a *communication group* consisting of an arbitrary number of processes. With multicast a message is sent to a number of specified processes. The broadcast operation requires the notion of a communication group but for the multicast operation that depends on the algorithm in use. From here on only the term *broadcast* commonly for both broadcast and multicast operations as long as the context doesn't require separation of the terms. It is also assumed that there is a communication group that defines which processes are involved in communication.

There are advantages to using a broadcast or multicast operation instead of several point-to-point send operations. *Reliable broadcast* assures that if a message is delivered to one of the destination processes, it is delivered to all. This simplifies the development of robust applications since the developer knows that when a process has received a message, all other concerned processes have also received it. In the point-to-point communication case there is a possibility that the sending process fails before the message has been delivered to all destination processes, leading to inconsistency in the different processes knowledge of messages. *Message ordering* guarantees that messages are delivered in order with respect to some policy. It can for example be guaranteed that all messages are delivered in the same order at all destinations, called *total order*.

2.3.1 Message ordering properties

Depending on the model of communication and how applications choose to interact, the communication patterns might grow complex with causal dependencies between messages and with multiple endpoints sending messages concurrently. To maintain a consistent view of the order of messages at all endpoints, synchronization of messages might be required. This is performed by the use of *message ordering* protocols [Chow and Johnson 1997 (section 4.1.5)] that assure that messages are delivered to the processes in the correct order. There are many algorithms on which the protocols are based, all with different strengths and weaknesses.

The difficulty with message ordering can be illustrated in three scenarios, each of which will show a possibility of message order inconsistency. The scenarios will prove that to achieve message consistency, three message ordering properties are needed, *FIFO (first in first out) order*, *causal order* and *total order*.

In the first scenario (figure 2.1) two succeeding messages are sent on the same communication *channel* (the path from the sender to one of the receivers), but due to the weak message ordering properties of the channel the messages are delivered in the reverse order. This could for example happen when using UDP/IP as the message transport protocol. FIFO order assures that messages in a channel are delivered in the same order they were sent.

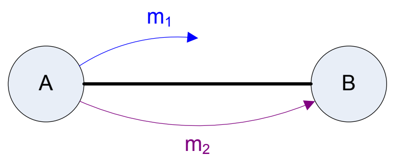


Figure 2.1 Possible message order inconsistency without FIFO order.

The second scenario (figure 2.2) shows three connected endpoints. Endpoint *A* broadcasts message m_1 and endpoint *B* responds by broadcasting message m_2. If the throughput is higher in the channels *A*-*B* and *B*-*C* than in the channel *A*-*C* it is possible that endpoint *C* delivers m_2 before m_1 even though m_1 causally precedes (happens before) m_2. Causal order assures that if an endpoint first delivers m_1 and then broadcasts m_2, then all endpoints deliver m_1 before m_2.

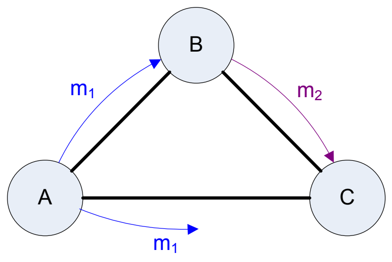


Figure 2.2 Possible message order inconsistency without causal order.

In the third scenario there are four endpoints (figure 2.3). Endpoint *A* broadcasts message m_1 and endpoint *B* broadcasts message m_2 at the same time. If the throughput in the channels *A*-*C* and *B*-*D* is higher than in the channels *A*-*D* and *B*-*C* it is possible that endpoint *C* and *D* delivers the messages in the opposite order, *C* delivers m_1 before m_2 and *D* delivers m_2 before m_1. Total order assures that messages are delivered in the same order by all endpoints.

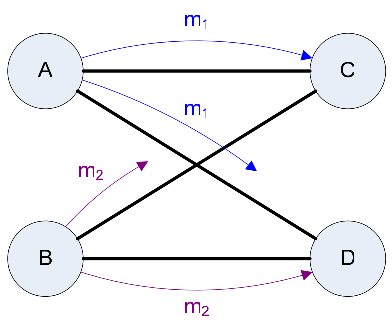


Figure 2.3 Possible message order inconsistency without total order.

The message ordering properties that must be fulfilled depends on the model of communication. If communication always is *one-to-one* (there is only one sender and one receiver for a message), the causal and total order properties will be fulfilled implicitly by fulfilling the FIFO order property. If communication on the other hand is *one-to-many* (there is only one sender but many receivers for a message), both the causal and total order properties must be fulfilled explicitly. The FIFO order property is implicitly fulfilled with the causal order property, but not necessarily with the total order property. However, the algorithms for total order do often fulfil the FIFO order property as well.

2.3.2 Total order broadcast algorithms

Total order broadcast is a complicated matter, especially when considering systems where faults might occur. This is most often the reality in distributed systems where processes can crash and network links can break. In the effort to provide robust total order broadcast communication a number of different algorithms have been developed. The algorithms use different techniques for message ordering and fault tolerance and are adapted to various fields of application. A detailed survey of total order broadcast algorithms can be found in [Défago et al 2003].

Three process types are used when describing how the algorithms work; *sender*, *destination* and *sequencer* processes. For each message there is a sender process that is the origin of the message and one or more destination processes where the message is delivered. A sequencer process is involved in making sure that messages are delivered in the correct order. Some but not all algorithms require a sequencer process. There are five major classes of algorithms, defined by the message ordering mechanism in use. The *fixed sequencer* (figure 2.4) and *moving sequencer* (figure 2.5) classes use, as the names suggest, a sequencer process for message ordering. In the *privilege-based* (figure 2.6) and *communication history* classes message ordering is provided by the sender processes and in the *destinations agreement* (figure 2.7) class the order is provided by the destination processes.

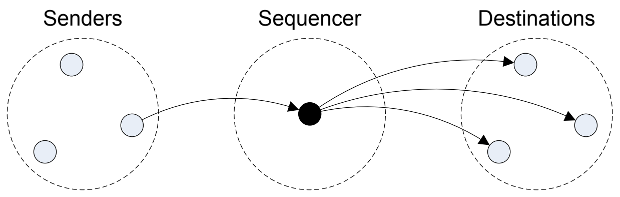


Figure 2.4 Fixed sequencer algorithm. Messages are sent point-to-point to the sequencer process that broadcasts the message.

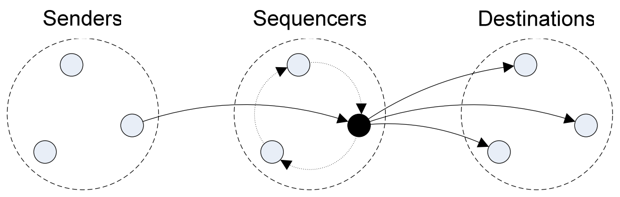


Figure 2.5 Moving sequencer algorithm. The principle is the same as for the fixed sequencer algorithm, but the role of sequencer is moved between processes, thus reducing the load on any single process.

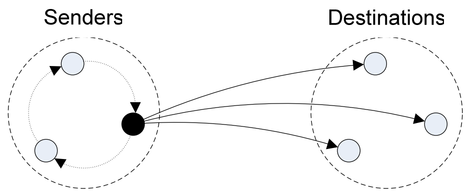


Figure 2.6 Privilege-based algorithm. A process can only broadcast when it is granted the privilege to do so. The privilege is propagated among the senders.

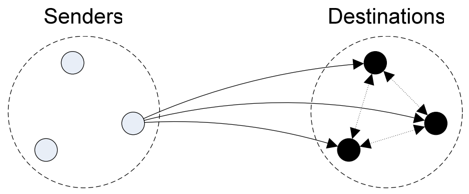


Figure 2.7 Destinations agreement algorithm. The delivery order results from an agreement between the destination processes.

2.4 Ensemble

Ensemble is an open system for distributed communication, developed at Cornell University and Hebrew University of Jerusalem. It has evolved from the *Horus* project [van Renesse et al 1996], which in turn was a redesign of the *Isis* group communication system [Birman and van Renesse 1994]. Ensemble provides one-to-one send and one-to-many broadcast communication primitives, as well as operations for joining and leaving communication groups. The system also offers properties of reliability, message ordering, fault tolerance, security and on-the-fly system reconfigurations. An extensive set of protocols [Hayden and Rodeh 2004 (part II)] that are assembled into protocol stacks provides the required properties in a modular way. A stack assembly that provides the required properties is decided upon in each communication group. All inter-process communication, both one-to-one and one-to-many, is performed in a communication group and there can be an arbitrary number of groups in an Ensemble system, each with its specific protocols. A protocol stack example is given in figure 2.8. Protocols are inserted into the stack as layers, which can be added or removed dynamically at run-time when the protocol requirements change [Hayden 1998 (section 2.1.8)]. For example, if total ordering is required and there is only one sender process in the group, FIFO ordering of messages is actually a sufficiently strong property and a corresponding protocol is inserted into the stack. If however other processes start to send, i.e. there is now more than one sender process, a total ordering algorithm is required. Due to the modularity of the protocol stack, the new protocol is simply plugged into the stack, silently and without the interacting application processes ever noticing.

Top	Top-most protocol layer
Heal	Partition healing
Switch	Protocol arbitration and switching
Migrate	Process migration
Leave	Reliable leave
Inter	Multi-partition view changes
Intra	Single partition view changes
Elect	Leader election
Merge	Reliable merge protocol
Slander	Failure suspicion sharing
Sync	View change synchronization
Suspect	Failure detector
Stable	Broadcast stability detection
Appl	Application representative
Frag	Fragmentation/reassembly
Pt2ptw	Point-to-point window flow control
Mflow	Multicast flow control
Pt2pt	Reliable, FIFO point-to-point
Mnak	Multicast NAK protocol
Bottom	Bottom-most protocol layer

Figure 2.8 A sample Ensemble protocol stack.

2.4.1 Groups and views

Ensemble is a group-based system, meaning that protocols providing fault-tolerant, reliable, ordered communication relies on the fact that the set of communicating processes is known and forms a group where all use the same protocol stack. Ensemble provides a set of *properties* from which an application can select a suitable subset. One property is for example Total that assures total ordering of messages. The actual protocol for total ordering that will be used (see the next section) is hidden within Ensemble, as are other protocols that the total ordering protocol relies on. Thus the developer doesn't need to worry which protocols to plug into the stack, only the properties of group communication must be selected. The protocol stack for a group is assembled as the first member joins a group and it may change as the group composition is reconfigured, e.g. if the new configuration requires stronger protocols to fulfil some selected property.

It is often considerably easier to assure that the selected properties of group communication are retained if the protocols can assume that all messages are delivered reliably to their destinations in the group. This is not a very complicated task as long as the group configuration doesn't change after it has been established, i.e. no members join or leave the group and no members fail due to process crashes or broken network links. The reality is however that processes do fail and it is often required that members must be able to join and leave a group at an arbitrary time for system flexibility. *Virtual synchrony* [Vitenberg et al 1999 (chapter 4)] is a technique that allows the protocols to assume that messages are delivered reliably to all destination processes between two well-defined points in time. This allows the protocol designs to be simpler leading to more robust protocols. The group configuration between two such points is called a *view* and virtual synchrony is therefore also referred to as *view synchrony*. The view defines the current set of processes and the protocol stack to use. A new view is installed when a group reconfiguration has taken place and the group is in a stable state. When something changes the group configuration, as when a member joins or leaves the group or crashes, a new view is installed as soon as the new configuration is stable. When a view has been installed each member receives a notification message including the view state with the set of members in the new view and other view related information.

Virtual synchrony is implemented as a stack of several independent protocol layers providing group and view management and reliable message delivery. Certain Ensemble protocols for total ordering rely on virtual synchrony.

2.4.2 Message ordering

Ensemble provides three communication primitives that group members, i.e. processes that have joined a communication group, can use to communicate. The send1() operation performs a reliable and FIFO ordered point-to-point send to a single member in the group. The send() operation performs a multicast to a list of members and the cast() operation performs a broadcast to all members of the group. Total and causal ordering of messages are provided by adding corresponding protocols to the stack.

Causal ordering is provided in a protocol layer that uses *timestamp vectors* (*causal vectors*) that are appended to messages. A timestamp vector contains *logical timestamps* of the messages that causally precede the message. There is one timestamp for each process in the group. A timestamp is simply an integer that is incremented by one for each new message a process sends. When a message reaches the layer, a check is performed to see if all messages with logical timestamps lower or equal to the timestamps in the causal vector have been delivered. If this is the case the message is delivered, i.e. passed on to the next layer above in the protocol stack. If there are causally preceding messages that have not yet been delivered, the message is delayed in the layer. Since the timestamp vector grows with the size of the group, possibly leading to a considerable over-head in message sizes, a second version of the protocol is provided. This variant compresses the timestamp vector by removing redundant information.

Adding one of the following protocol layers to the protocol stack provides total ordering of messages. SEQUENCER is a protocol based on a fixed sequencer. When a process want to broadcast a message it sends it point-to-point to the sequencer process that in turn broadcast the message. TOTEM is a token ring protocol, a variant of a protocol developed in the Totem project [Amir et al 1995]. A process may broadcast messages when it holds the token, then it passes the token to the next process in the ring. There are also two protocols for total ordering that are commonly not used. TOPS is also a privilege based protocol that orders messages lexicographically. Processes are ordered in a logical ring and a process may only broadcast a message when the preceding process in the ring has broadcast a message. This works fine as long as all processes in the ring broadcast messages regularly, but when a process doesn't broadcast a message when it is privileged to, the next process in order will have to wait until timeout. This leads to high latency when processes don't broadcast often. The last protocol, ZBCAST, is a probalistic protocol that unlike the other three admits a small probability of message loss. The arguments for using a probalistic protocol is that messages broadcast over LAN is usually spontaneously totally ordered and that a probalistic protocol improves system performance and scalability. ZBCAST consists of two sub-protocols. The first is an effective but unreliable broadcast protocol that makes a best-effort attempt to deliver the messages. The second is a 2-phase protocol that in the first phase detects message loss during the unreliable broadcast and in the second phase corrects the possible losses.

2.4.3 The Ensemble implementation

Server

The main Ensemble code is written in the Objective Caml programming language (O'Caml) [Leroy 1997], a dialect of the ML language. O'Caml was selected because it is a high level language with strong mathematical semantics, efficient memory handling and type checking. With the O'Caml language it is also possible to use theorem-proving tools like NuPRL [Constable et al 1986] to verify the correctness of protocols and to transform them into optimized forms.

To make it possible to use Ensemble from other languages than O'Caml, the core Ensemble code has been compiled into a separate Ensemble server executable. Applications written in other languages communicate with the server via TCP sockets[1], thus making it possible for the developer to use any language of choice, considered that TCP sockets are supported.

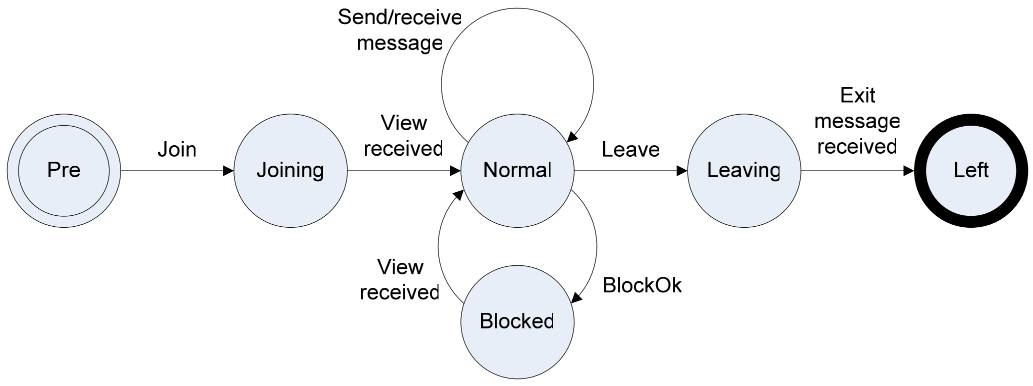
There are no constraints on the format of messages that can be transferred with Ensemble. Message contents are simply transferred as bulk data.

Client

An application that uses Ensemble acts as a client that communicates with the Ensemble server via a TCP socket. A code layer that maintains the socket connection and handles messages passed on the socket is required on the client side. The Ensemble development group provides client code in Java, C# and C that is ready to use.

The client is synchronized with the Ensemble server by obeying rules defined in six different states [Hayden and Rodeh 2004b], forming the client state-machine (figure 2.9). Initially, the client is in the Pre state. The only allowed operation here is Join. When the client requests to join a communication group, it moves to the Joining state and remains there until the first view is installed. When the view is installed it receives a view message and moves to the Normal state. In the normal state messages can be sent and received. The server sends a Block message to the client when it is about to install a new view. The client responds by sending a BlockOk message to the server, thus moving itself into the Blocked state. In the Blocked state it must not send any messages. When the new view is received the client moves back to the Normal state. If the client desires to end the communication session by leaving the communication group it performs the Leave operation, thus moving into the Leaving state. When an Exit message is received from the server, the client moves into the Left state. It has left the group and may no longer send messages to it.

[1] In earlier versions of the Ensemble system it was possible to link it directly into the application. This feature was removed in version 2 due to problems with ML memory management in poorly written C programs, leading to crashes that were difficult to debug [Hayden 1998 (section 4.2.4)].



Figur 2.9 The Ensemble client state-machine.

2.5 Messages

There are a number of ways to format the actual application messages. The issues to consider when choosing a message format are *content flexibility*, *data typing*, *platform independency* and *message size*. To provide content flexibility is to let applications decide what content should be in the messages and to allow that to change when needed. A flexible message format allows for the content as well as the structure of the content to change. This can for example be useful when a new application version is developed that can provide more information in its events. Then it should be possible to add the new content without breaking the communication with application instances of earlier versions. Data typing is the ability to define the type of the data content to enable type checking when applications are built and to make it unambiguous for the developer what data to provide in the message, thus making message formatting both easier and safer. Platform independency is achieved by choosing a format that can be read on various hardware and software platforms without data misinterpretation. The size of messages can be an issue when message handling must be highly effective, for example when messages are sent very frequently and the bandwidth is limited.

2.5.1 Binary versus text formats

A binary message format is usually more compact than a text-based format, thus reducing bandwidth usage. It may also speed up computational time if the message content has to be accessed several times, for example during routing to make routing decisions based on the content [W3C 2005 (section 3.13)]. A binary format does however require that the data representation is standardized, so all platforms interpret data in the same way. The classic example is the use of little versus big endian byte order in integers on platforms with different processor families. The format of more complex data structures must possibly also be defined, like tree structures containing arbitrary data objects.

If a binary message format is compact and effective, a text-based format is usually more flexible regarding the content. A structured text format can usually be changed without preventing applications written for the previous format to extract data from the new one. This is not always the case with binary formats that tend to be harder in their definition, thus possibly requiring software to be adjusted and recompiled when a format change is performed. Another property of text formats is that they can contain data of any type, since all data-types are converted to text anyway. This can be both an advantage and a disadvantage. The advantage is that the format itself doesn't delimit what kind of data that can be transferred, thus letting the system designer select the data types of choice. The disadvantage is that type checking is harder, there is specifically no static type checking at compile time.

2.5.2 Two format specifications

One standard for binary messages is the Abstract Syntax Notation One (ASN.1) [Chow and Johnson 1997 (section 4.2.1)], often used in RPC communication. ASN.1 provides the means of defining data structures in a standardized and type-safe way. There is a clear definition of what data types are supported, thus strong type checking can be performed and one source of errors is avoided.

The Extensible Markup Language (XML) [W3C 2004b] is a popular text format for general, structured data. The format is structured hierarchically with tags defining where data objects starts and ends. XML is widely used and there are many tools for managing XML documents. For general XML document handling in Java and C++ there is for example the Apache Xerces parser [Apache 2004]. There are also several languages defined for making the use of XML simpler and safer. XML Schemas [W3C 2002c] can be used to define the structure, contents and semantics of XML documents and further to validate that an XML document follows a certain schema. XPath [W3C 1999] is a language for finding information in an XML document. It uses path expressions to navigate through the document and can be used in filters that extract relevant information from it.

2.5.3 Content filtering

Effective event filtering and routing in content-based publish/subscribe systems require techniques of identifying the contents of a message and possibly extracting parts of it. Event routing will make decisions on the basis of the content and in advanced event filtering systems different parts of an event can be sent to different destinations. Except for a format that allows that such operations is performed in a reasonable simple manner, there must probably also be some sort of query language used to define the content of interest. XPath is such a language for XML and has been used for effective XML document filtering in systems as for example NaradaBrokering [Pallickara and Fox 2003] and YFilter [Diao et al 2002].

3 The DS3 system

This chapter handles the design of the system evolving from the final degree project, hereby called *Distributed System for Synchronization and Supervision* (DS3). The models of communication are described and discussed, followed by decisions on the system-specific variants. In section 3.5 the complete architecture is specified in greater detail.

3.1 Goals

The goals of the design and implementation phases of the project are as follows.

G1. To design a system that provides request/reply and publish/subscribe communication with respect to the requirements.

G2. To select and evaluate technology that provides the communication primitives required by the system.

G3. To select message formats to use for internal messages and application messages and to select appropriate tools for managing messages.

G4. To implement the system as a component that is readily integrated with the target applications.

G5. To evaluate the system.

G6. To integrate the system in an application for supervision, supposed that the system approaches a complete state within the time-span of the implementation.

3.2 Architecture

The system architecture must be designed on the basis of the models of communication that will be provided and with the goal to create a well-defined middleware solution. The models of communication imply that the system becomes rather complex, especially when the requirements of for example flexibility and transparency have to be fulfilled. The intention is to abstract the parts of the architecture that concern the models of communication from the overall architecture, including the message transport and application/middleware synchronization mechanisms.

Due to its strong properties of reliability and message ordering, Ensemble is chosen to be the foundation for message transport. The Ensemble-specific parts of the system are hidden under a layer that provides transparent and asynchronous message passing.

3.3 Model of communication

The model of communication is the part of the system architecture and semantics that defines how communication is performed as far as the application is concerned. It specifies for example whether communication is based on request/reply or event semantics, if it is one-to-one or one-to-many and if is direct or uses some central entity.

An application using the system will require that one or more of the primary requirements are fulfilled. The different nature of the primary requirements implies that two different models of communications must be supported by the system architecture. One model supports *request/reply*-oriented communication, hereby referred to as the *request/reply model*, and one model supports *event*-oriented communication, referred to as the *event model*.

3.3.1 The request/reply model

In the *request/reply* model, also referred to as the *client/server* model, processes consist of *clients* and *servers*. A process can be either one or both. Client processes can send requests for information and receive replies. Server processes can handle requests and send corresponding replies back to the clients.

Communication in the request/reply model can be *synchronous* or *asynchronous* [Coulouris et al 2001 (section 4.2.1)]. In synchronous communication a client process sends a request and then waits for the reply, thus blocking the execution until the reply has been delivered. The request and reply is synchronized. Asynchronous communication implies that the request isn't blocking, the execution continues without halt and the process is instead notified of the reply as it is delivered.

In the general request/reply system any number of server endpoints can be assumed to be active at a time. It is up to each endpoint what resources it provides so it is possible that several servers provide overlapping resources. If a client requests some information that more than one server provides there must be a policy that decides to which servers the request should be sent and how multiple replies are handled. The policies can be separated in three variants depending on the number of request and replies sent.

With one request and one reply, a request is only sent to one server regardless of the number of servers that actually provide the resource. The single reply is delivered to the client. This policy makes it clear how the request and reply are handled but there is a risk that information provided by other servers is missed. It is also possible that there are reply latencies if the selected server has low capacity or is heavily loaded. With many requests and one reply, the request is sent to all servers and the first reply that is received is delivered to the client. This will reduce response time and it will possibly equalize the load on the servers if all remaining requests are dropped as soon as one server has responded. To assure that a client receives all information that matches a request a third policy is required, with many requests and many replies. Then requests are sent to all servers and the replies are either collected and delivered all at once or delivered one by one. This lets the application decide what information is relevant and not, but it also requires more complex reply handling in the application.

Request/reply in DS3

The request/reply model is suitable for meeting primary requirement P1, where a process must be able to request information about the state of another process. In the first version of DS3 it is sufficient if request/reply communication is asynchronous, which will make the development considerably easier due to the asynchronous message-passing layer below.

The request/reply semantics is decided to follow the policy with one request and one reply. The decision was made because this policy is enough to provide the functionality needed and due to the simple handling of replies. A greater responsibility is put on the developer to make resource definitions unambiguous to prevent that processes miss relevant information.

The decisions on the request/reply model in DS3 are as follows.

- Replies are delivered asynchronously.
- A request is sent to only one server.

3.3.2 The event model

In the event model [Carzaniga et al 1998] (also called the *notification model*), processes consist of *producers* and *consumers*. When an event occurs in a producer process it notifies the concerned consumer processes. In this way the consumer processes receive information about events directly and without delay. An event might for example be a change in the state of the producer process. To accomplish the same functionality in the request/reply model the consumer processes would be required to repeatedly poll the producer process to notice when an event has occurred. The producer would also have to store the event information until it is certain that no other consumer will need it.

Communication in the event model is by nature asynchronous while events usually occur spontaneously.

3.3.3 The publish/subscribe model

Although the event model provides asynchronous one-to-many communication it has some limitations in flexibility and transparency. To be able to know where to send notification messages the producer must hold a predefined list of consumers or possibly use some broadcast mechanism to send notifications to all consumers. It must also be specified which consumers should be notified of which events. If couplings and event filtering is predefined, applications will become unnecessary rigid. If on the other hand notification messages are broadcasted and no event filtering is done there will probably be a considerable amount of over-head in the number of messages sent, leading to unnecessary high network load.

To acquire a dynamic and loosely coupled model of communication, there are three decoupling dimensions to be considered; *space*, *time* and *synchronization* decoupling [Eugster et al 2001]. Space decoupling means that the communicating processes don’t need to know the location of each other. A producer doesn’t need to know the consumers that will receive the notifications. Time decoupling means that the communicating processes don’t have to participate actively in the interaction at the same time. A consumer doesn’t have to run when an event occurs; it will be notified of the event when it joins the interaction. Synchronization decoupling means that the processes don't get blocked during interaction. A producer isn't blocked while consumers are notified and consumers are notified asynchronously. In this way interaction between processes doesn't disrupt the main flow of the processes.

With respect to the decoupling dimensions a new event-oriented model of communication has evolved, the *publish/subscribe* model [Eugster et al 2001]. In this model producers are *publishers* and consumers are *subscribers*. A subscriber *subscribes* to the events it has interest in. How the subscription is handled and where it is registered is completely transparent for the subscriber. The only thing it knows is how to perform the subscription operation. The actual subscription handling is done by the underlying and hidden publish/subscribe service. In the same way as with subscriptions, a subscriber can *unsubscribe* from an event when it is no longer interested in it. When an event occurs in a publisher process it *publishes* the event[2]. As for the subscriber and subscriptions, the publisher isn't aware of what happens with the event after it is published. It only knows the publishing operation. Except for publishing events, publishers can also *advertise* the events it will publish in the future to make it possible for the publish/subscribe service and the subscribers to make preparations for handling the specific event. When an event has been published, all subscribers that have subscribed to that particular event will be notified of it, either directly or whenever possible due to time decoupling. None of the operations in the publish/subscribe model require a reply and all operations can therefore be performed asynchronously, without blocking the process.

Even though the publish/subscribe model is designed with respect to the decoupling dimensions, it is up to the properties of the publish/subscribe service implementation what transparency is actually obtained. There are two architectures that are commonly used when implementing the service. In the *centralized* architecture there is a central entity, e.g. a central server that stores, filters and forwards events. Published events are transferred to the central entity and forwarded to the subscribers. The central entity can store an event until all subscribers have been properly notified of it to achieve time decoupling and high reliability. There are three major disadvantages with the centralized architecture: There must be a central entity, it constitutes a single point of failure and if the load on it is high throughput might decrease. In the *distributed* architecture the service is built solely on the publishers and subscribers alone. Events are sent directly from publisher to subscriber endpoints. To achieve time decoupling storage of events must be somehow be accomplish at the endpoints. Filtering of events can be done either on the subscriber or publisher side. On the subscriber side filtering is simple since the subscription information is already there but there will probably be a considerable over-head of sent events because all publishers must send events to all subscribers. With filtering on the publisher side, the subscription management is more complex but traffic will be minimized.

Publish/subscribe in DS3

An event-oriented model is suitable for meeting primary requirement P2. The publish/subscribe model is chosen due to its many benefits. It must be decided how the properties of time, space and synchronization decoupling should be fulfilled.

There is no requirement of persistent events in DS3. This eases the requirement for time decoupling in the publish/subscribe system to implement as compared with a full-blown publish/subscribe system. A subscriber process will as far as the system is concerned only be interested in events that are published when it is running. Events that were published before

[2] Note the difference between event as something that occurs at a point in time and event as an entity of information that can be published.

the subscriber process was started are of no interest. This makes the design of the publish/subscribe system considerably easier while there is no need for a central storage entity or smart event storing at the publisher and subscriber endpoints.

Space decoupling in the DS3 publish/subscribe system can be accomplished with broadcast operations. If event filtering is done on the subscriber side, publishers broadcast events to all subscribers. When a subscriber receives an event the subscriber-side publish/subscribe system filters the event according to the subscriptions made by this subscriber. If the subscriber has subscribed to the event the event is let through to the application, otherwise it is simply ignored. If on the other hand event filtering is done on the publisher side, subscribers broadcast subscriptions to all publishers and the publisher-side publish/subscribe system stores them for later use in event filtering. When the publisher publishes an event, the publish/subscribe system checks the subscriptions to see which subscribers are interested in the event. The event is then sent to those subscribers only, possibly in a single, effective multicast operation. In either case of event filtering, the underlying communication layer must provide a broadcast communication primitive for the distributed publish/subscribe system to be effective and clean in its implementation.

The event filtering should be based on content. A subscriber specifies what information it is interested in and events containing that information are sent to it. This subscription scheme eliminates the need for event metadata that describes what an event contains.

Publisher-side event filtering reduces the number of messages sent in publish/subscribe communication but it requires maintenance of the subscription state. Inconsistencies may appear, e.g. due to remaining subscriptions from subscriber processes that have failed. This can be handled by a sound *group membership service* that reports failed endpoints so the subscription management module can remove corresponding subscriptions from its state.

Synchronization decoupling requires that publishers don't have to wait until an event is sent to all destinations and that subscribers get notified of events asynchronously. It can be achieved by with threaded execution of publish and deliver operations in the publisher and subscriber processes. When an event is published, it is taken care of by a dedicated *sender thread* that performs the actual message send or cast operation. The main thread of the process puts the event message in a queue for outgoing messages, then immediately continues its execution of other tasks. In a subscriber process, there is in a similar way a dedicated *receiver thread* that receives events and either performs the necessary actions itself, lets a worker thread do it or notifies the main thread that the new event is available for processing.

The decisions on the publish/subscribe system can be summarized as follows.

- Time decoupling is not fulfilled.
- Event filtering is done on the publisher side to reduce the number of messages.
- Synchronization decoupling is achieved by using a threaded program model.
- Events should be filtered on content.

3.4 Message format

Since the goal with DS3 is to build a flexible system for a general set of applications, the message format shouldn't put restrains on what the applications can do. As applications evolve the information passed in messages could be a subject of change. Content flexibility is a highly valued property when taking these aspects into consideration.

Platform independency can be of importance even if the main system only runs on a single platform. If messages leave the system at some point a common and platform independent message format can restrain the need to reformat messages. To illustrate this, lets say that an application needs to pass certain messages via another protocol, e.g. SOAP to pass the message to a Web Service on the Internet. If the message is an XML document it can simply be inserted into the SOAP message envelope and sent as is. If the message on the other hand were of some other format, specifically a native binary format, it would most likely be required to reformat the complete message, object by object.

Communication in DS3 is assumed to be relatively sparse, thus relaxing the requirement for compact messages.

On the basis of these conclusions, the XML format is chosen as a suitable message format. It is a flexible, platform independent format and it has been shown that it makes effective content filtering possible. The non-typed data gives the developer a greater responsibility when writing and reading message content which could possibly increase the risk of data misinterpretation. This risk can however be reduced by using schemas for validation. There are also mature tools for handling XML documents, constraining the work to be done to integrating the tools into the system and defining the content structure of the different messages.

The message definitions and a few application specific examples are listed in appendix B.

3.5 Architecture specification

The DS3 architecture spans from the low-level Ensemble client to high-level communication models. Code with different purposes is separated as layers in an overall model of the system (see figure 3.1). The bottom-most layer is the Ensemble client code that performs the low-level communication with the Ensemble server and maintains the connection to it. The next layer is the threaded C++ layer for asynchronous sending and receiving of messages. This layer is responsible for hiding the blocking send and receive operations and for error handling in case of inconsistencies or client/server communication faults. The top-most layer, adjacent to the application, is the endpoint systems layer. Here the publish/subscribe and request/reply communication models are implemented. This layer provides the developer with an application program interface (API) to the system.

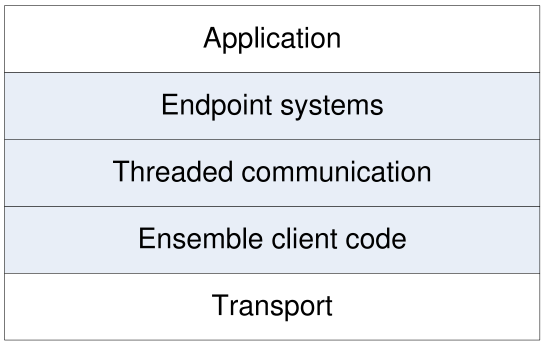


Figure 3.1 The layers in the DS3 architecture.

First, the API is specified on the basis of the application requirements. Then the properties of the threaded communication layer are specified with respect mainly to transparency and fault tolerance. Finally, the significant issues of the Ensemble client code are pointed out.

3.5.1 API

The API of DS3 should consist of a small set of interfaces corresponding to the different kinds of endpoints in the system. There is probably also reason to provide a set of helper classes, e.g. for message management.

There are four endpoint interfaces corresponding to the endpoint systems. The Publisher interface provides the method Notify() for publishing events, Advertise() for advertising a resource and Unadvertise() for announcing that no more events will be published for a resource.

There is a corresponding Subscriber interface with the method Subscribe() used to subscribe to resources and Unsubscribe() used to end subscriptions. It has also several handler methods can be over-ridden by an application class to take actions when messages are received. The handler methods are called asynchronously when messages are delivered from the system. OnNotify() receives published events. OnResourceFound() is called the first time a resource is advertised and OnAdvertise() is called on following advertisements. OnUnadvertise() is called when a publisher announces that it will not publish events for a resource any more and OnResourceDepleted() is called when there are no more publishers for a certain resource. OnResourceFound() and OnResourceDepleted() are provided for applications to be able to perform initialization tasks when there is a new resource and final tasks when no more events will be generated for a resource.

The Server interface provides the same methods for advertisement as the Publisher interface and the handler method OnRequest() that receives requests from clients. The reply should be given as an out or return parameter and there is therefore no need for a specific reply method.

The Client interface has a Request() method and a corresponding OnReply() handler method for the asynchronous replies.

All interfaces have in addition to their specific methods also Join() and Leave() methods used to join and leave communication groups.

3.5.2 Endpoint systems

With the API set, the functionality under the surface must be specified. For each interface a number of required abilities are described, followed by the modules that are responsible for providing them. The different types of modules that will be mentioned are *nodes* that can send and receive messages, *observables* that transform messages to well-defined method calls, *listeners* that those methods are called upon and *state objects* that keep state data. A module can be of more than one type; many consumers are for example also observables as they convert messages to method calls. Most of the nodes are either *producer nodes* that are used to create and send messages, *consumer nodes* that receive and handle messages or *update nodes* that can re-send previously sent messages.

Publisher

The publisher API specifies that a publisher must be able to send events and advertisements. Since the event filtering should be done on the publisher side, the endpoint must be able to receive and store subscriptions for event filtering. It should also handle notifications of group membership changes to remove subscribers that have left the group from the subscription state and to send advertisements to newly joined members. A number of different modules can be identified, each with a certain responsibility in building up the complete functionality. Each module can pass messages to other modules and the resulting *endpoint system* can be presented as a directed graph where each node corresponds to a module of functionality. The edges show how messages are passed and dashed lines show dependencies to state objects. The graph for the publisher endpoint system is shown in figure 3.2.

The Event Producer node provides the developer with the Notify() API method used to send application events. An event message is passed to the Event Filter node where it is tagged with the addresses to the destination subscribers. Filtering is based on the Subscription State object. The tagged event messages are then passed on to the Transport node that performs the actual send or broadcast operation. The Subscription Consumer node is responsible for updating the Subscription State when subscription and unsubscription messages arrive and when endpoints leave the communication group. The Advertisement Producer node provides the Advertise() and Unadvertise() API methods. Advertisement messages are passed both to the Transport node and to the Advertisement Consumer node. The Advertisement Consumer node updates the Advertisement State, used by the Advertisement Update node to re-send advertisements when new subscribers join the communication group. The Group Change Producer node generates events when members join or leave the group. These events are only distributed to the concerned nodes on the local system.

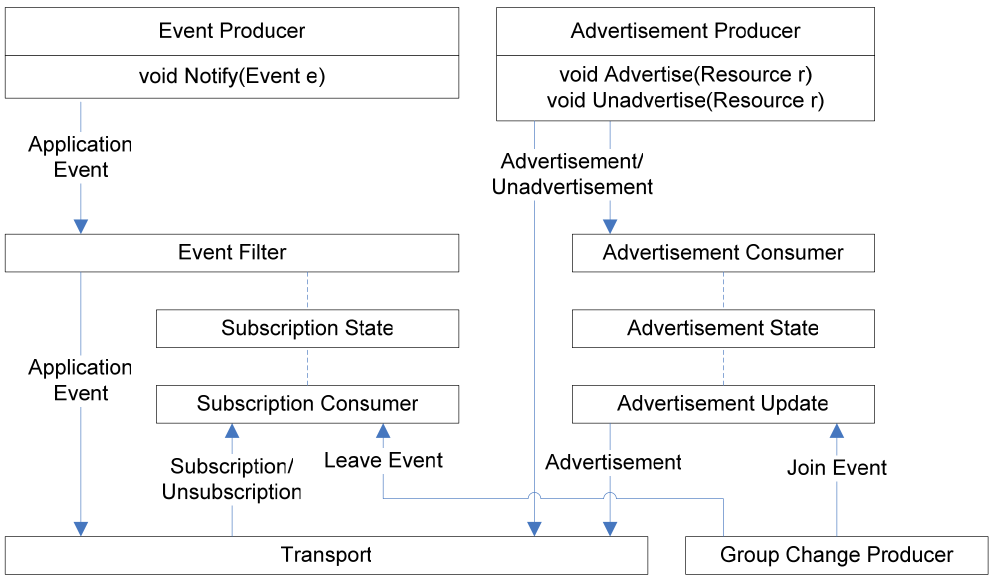


Figure 3.2 The publisher endpoint system.

Subscriber

The subscriber must be able to send subscriptions and receive application events. Subscriptions are stored at the subscriber side so they can be sent again, silently without notifying the application, when a matching resource is advertised. The subscriber must receive and store advertisements to be able to decide when there are no more publishers for a resource. This is due to the possibility that several publishers publish events for the same resource. If a publisher unadvertises the resource when there are still other publishers that have advertised the same resource, the resource should not be considered to be depleted. It is only when the last publisher unadvertises the resource it is depleted. The graph for the subscriber endpoint system is shown in figure 3.3.

The Subscription Producer node provides the Subscribe() and Unsubscribe() for sending subscription and unsubscription messages. The messages goes to the Transport node but are also caught by the local Subscription Consumer node. The Subscription Consumer node updates the Subscription State, used by the Subscription Update node to re-send subscriptions when a matching resource is advertised. The Event Consumer node receives and handles published application event messages. It calls the OnNotify() API method on all connected Event Listener objects. A listener class can be any application class that inherits a DS3 listener interface. The Advertisement Consumer node handles advertisements and calls the appropriate API methods on all connected Advertisement Listener objects. It keeps current advertisements in the Advertisement State to be able to decide whether a resource is found or depleted.

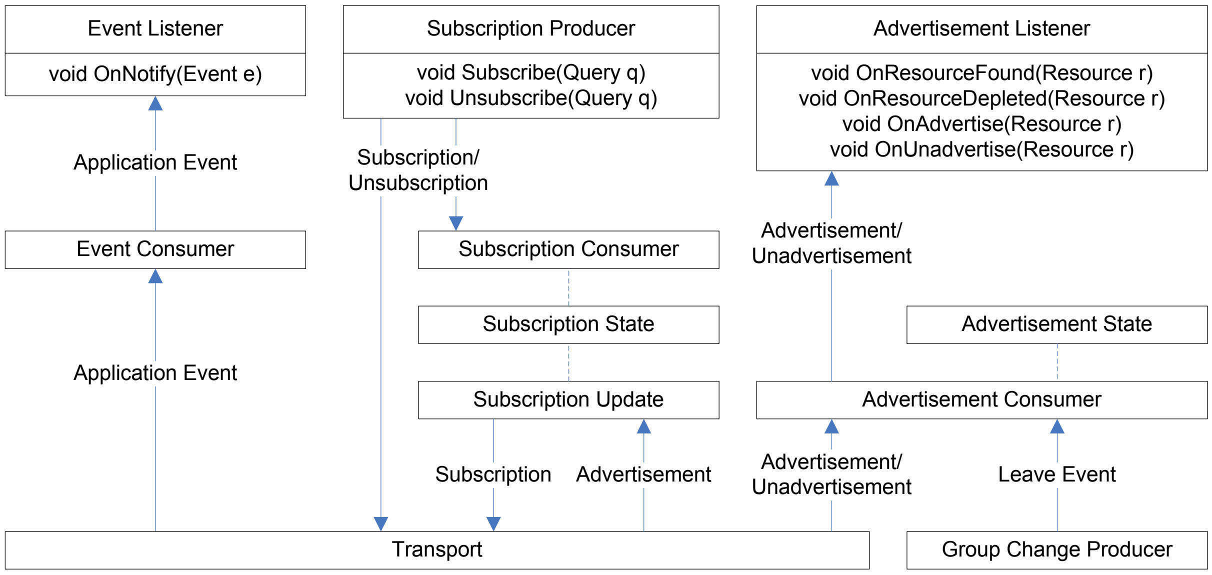


Figure 3.3 The subscriber endpoint system.

Server

The server receives request queries and sends one reply per query. It should advertise the resources it provides so client endpoints have the knowledge of the servers of interest. Figure 3.4 shows the graph.

The Request Consumer node receives requests and passes them on to the single Request Listener objects OnRequest() API method. It then sends a reply message with the reply from the application. Advertisements are managed in the same way as for the publisher endpoint system.

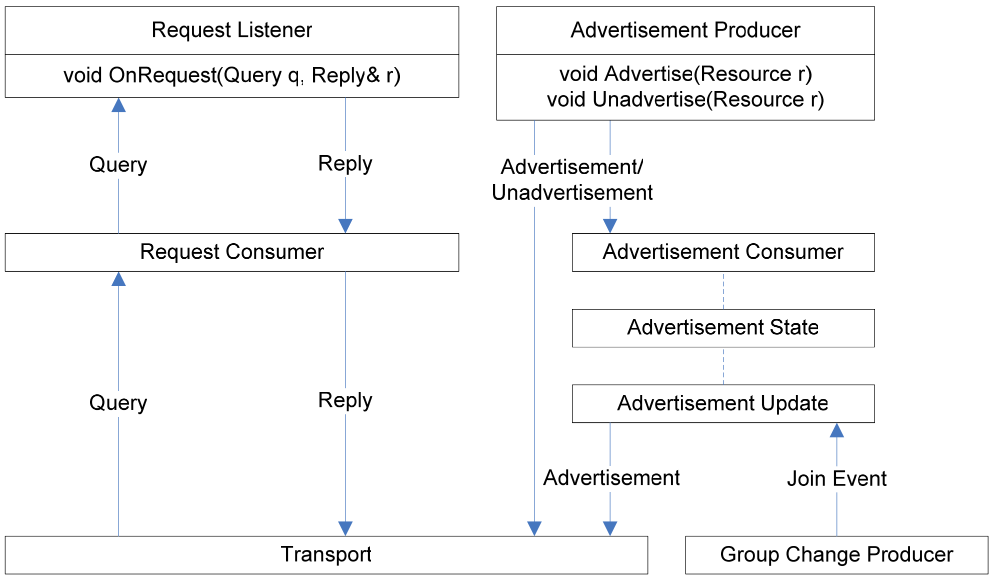


Figure 3.4 The server endpoint system.

Client

The client must be able to send requests to the appropriate servers and receive the replies (figure 3.5).

The Request Producer node provides the Request() API method that passes a request query to the Request Filter node. The filter node tags the query message with the address of one server that provides the requested resource. It uses the Advertisement State to retrieve the address. If no server provides the resource an empty reply is sent back to this client endpoint. The Reply Consumer node receives the replies and calls the OnReply() API method on the single Reply Listener object. Like in the subscriber endpoint system, an Advertisement Consumer node updates the advertisement state.

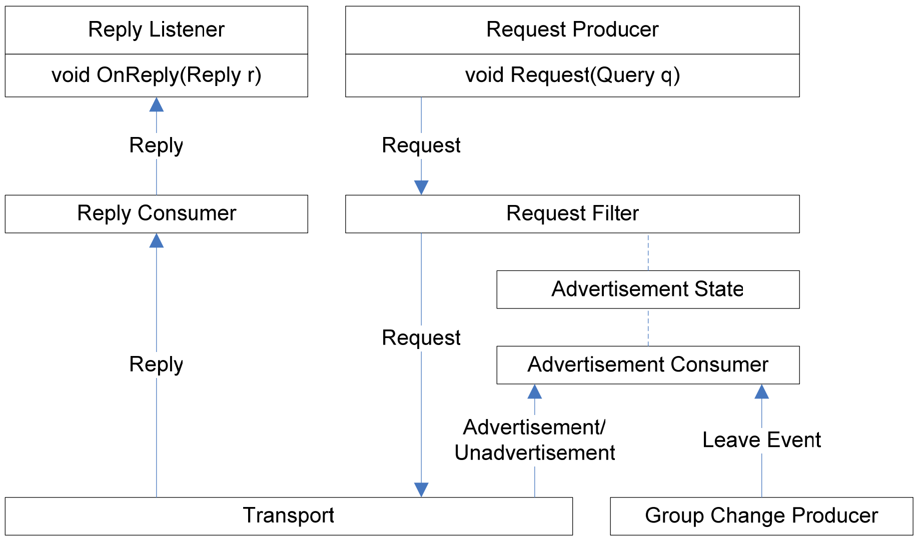


Figure 3.5 The client endpoint system.

3.5.3 Threads

Synchronization decoupling requires non-blocking message send operations and asynchronous message delivery. The communication layer is responsible for providing transparency with respect to the execution time of different operations and specific semantics of the layers below. An endpoint process will run at least four threads to make the execution of the tasks as smooth as possible. Except for the main thread of the application, the thread model in the DS3 system defines three additional threads with different purposes (figure 3.6).

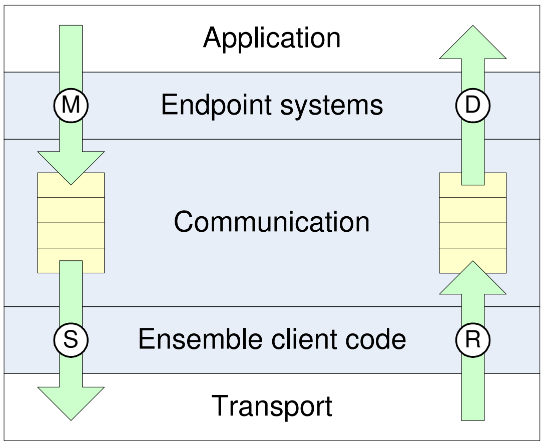


Figure 3.6 The thread model in DS3. The arrows correspond to the different threads and show in which direction messages propagate. The two blocks in the communication layer represent the message queues.

In most cases it is the main thread *M* of the application that commences communication interaction by a call to an API method. The endpoint system performs the actions specific to the communication model, including message formatting and filtering, then calls a send method in the communication layer. This method simply puts the message in a queue of outgoing messages and returns, letting the thread continue with other tasks immediately.

The send thread *S* runs a loop that retrieves a message from the queue and sends it to the Ensemble server via one of the functions provided by the Ensemble client code. The thread is responsible for holding messages when view changes occur. When a block before a view change is announced no messages are sent until the new view is installed.

The receive thread *R* runs a corresponding loop that polls the Ensemble server for new messages, receives them and puts them in a queue of incoming messages. The Ensemble client code provides the poll and receive functions. The thread is also responsible for announcing the special block and view messages for consistent handling of view changes.

The fourth thread is the *dispatch* thread *D*. This thread is responsible for processing the incoming messages in the endpoint systems and possibly delivering them to the application. It runs a loop that retrieves a message from the queue of incoming messages and passes it to the connected endpoint system nodes. The message can be consumed either by the endpoint system layer or by the application. For example, subscription messages are consumed by the subscriber endpoint system and the application never sees it, but event messages are delivered to the application in the OnNotify() API handler method. Some messages will cause a new message to be created and sent, either from the endpoint system layer or from the application. An advertisement message can for instance cause a new subscription message to be sent from the endpoint system layer and the application can respond to an event message by publishing a new event (figure 3.7).

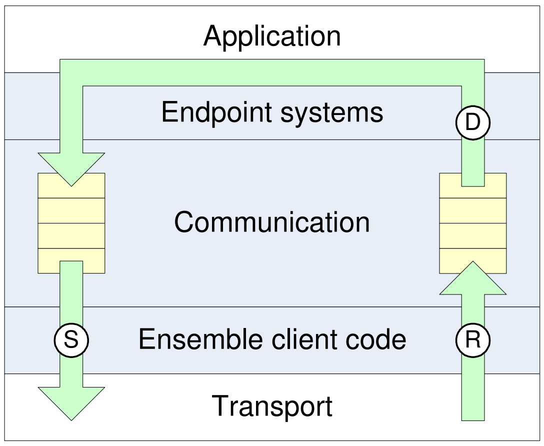


Figure 3.7 An incoming message can cause a new message to be sent either from an endpoint system or the application.

There are two main arguments for letting the dispatch thread deliver messages instead of letting the receive thread doing it. The first reason is that no messages will be retrieved from the Ensemble server while delivering a message. Thus important messages such as view change messages will be delayed, possibly causing several messages to be sent in an inconsistent view. The second reason is that even if the dispatch thread is blocked for some time, perhaps due to synchronization issues in the application, the receiver thread will continue to collect messages from the Ensemble server. This will relax the server with respect to internal message buffer load and algorithm states.

3.5.4 The Ensemble client

The Ensemble system is implemented as a server application running locally on the computers it is used on. Clients communicate with the server via TCP/IP. The Ensemble developer group provides client code in Java and C that handles the TCP/IP connection and passes messages to and from the server. Since DS3 is implemented in C++ the C client code is chosen to be the bottom-most code layer.

4 Implementation

The work of the implementation split into a number of tasks to be completed. The tasks were derived from the conclusions of the preparation phase. A total of about 4400 lines of code where written during the implementation (blank lines and comments excluded), roughly divided equally on the communication layer and the endpoint systems.

The class collaboration graphs in this chapter show the relations by inheritance and ownership/reference. Inheritance is shown with solid edges and ownership/reference with dashed edges. The graphs and member lists are generated from source code by the *Doxygen* documentation system [Doxygen 2005].

4.1 Development

The DS3 implementation is built into three different C/C++ libraries that are linked into the applications using the system. *libens* is the library built from the Ensemble client code for C, provided by the Ensemble development group at Cornell University. *DS3C* is the library with the communication layer that wraps the libens code and provides an interface for transparent communication. The *DS3* library implements the endpoint systems and relies on DS3C. The abstract interface between DS3C and DS3 actually makes it possible to completely replace the underlying group communication system. The DS3C library would need to be replaced, the new one possibly wrapping some other client module.

The development of DS3 is done in the Microsoft Visual C++ 6.0 environment. Three different projects were created for the different libraries.

4.2 Integration of the Ensemble client

The Ensemble C client is available both as a C library and as source code. An overview of the functions provided is given in appendix A.

The first attempt was to use the library directly but with no luck. The code crashed at execution and the reason was finally pinpointed down to be the use of incompatible function libraries in the Ensemble client code and the application. However unlikely, the Ensemble client code library seems to be built in the single thread model. The rest of DS3 obviously needed to be built in the multithreaded model. Creating a new library from the client source code solved the problem.

The client code uses the stdout stream to print information and error messages. In a windowed Microsoft Windows application all information written on the stdout stream is lost if the stream isn't redirected. Applications using DS3 can redirect the stdout stream to a file to catch messages.

Fatal errors in the client code are per default handled by printing an error message to stdout and exiting the application abruptly. This behaviour was changed so when a fatal error occurs an external handler function is called. This function was placed in the wrapping C++ code where a C++ exception could be raised for uniform error handling.

4.3 Implementation of the communication layer

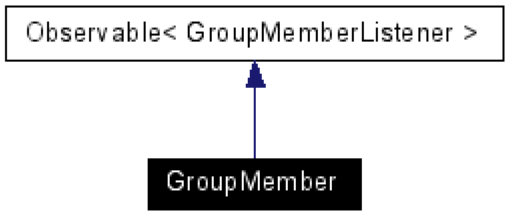
A great effort was put on building the DS3 Communication layer (DS3C) wrapping the Ensemble client code and providing an abstract and transparent interface for communication. The intention of the interface is to hide the behaviour specific to the Ensemble client. The interface to the endpoint system was made as clean as possible. It was defined in the abstract GroupMember and GroupMemberListener classes with just a few general methods for sending and receiving messages and joining and leaving groups.

4.3.1 The interface

An endpoint system uses a GroupMember derivative object to send messages and join/leave requests. The object is created by calling GroupMemberFactory::CreateGroupMember(), thus making the GroupMember implementation opaque to the endpoint system.

GroupMember

Used to communicate with the underlying communication group.



- virtual void **Join** (const **DS3_string** &sGroupName)=0
- virtual void **Leave** ()=0
- virtual void **Send** (const **DS3_string** &sMessage, const **DS3_set**< **DS3_string** > &destinations)=0
- virtual void **Send** (const **DS3_string** &sMessage)=0

GroupMemberFactory

Factory class that produces objects that implements the **GroupMember** interface.

- static GroupMember * **CreateGroupMember** ()

For the communication in the other direction, up from the underlying layer, the *Observer* pattern is used. The GroupMember is a GroupMemberListener *Observable*. The GroupMemberListener is registered to the GroupMember to receive messages and view changes. The GroupMemberListener class is inherited and implemented in the endpoint system.

GroupMemberListener

Enables a class to receive messages and view changes from the underlying communication group.

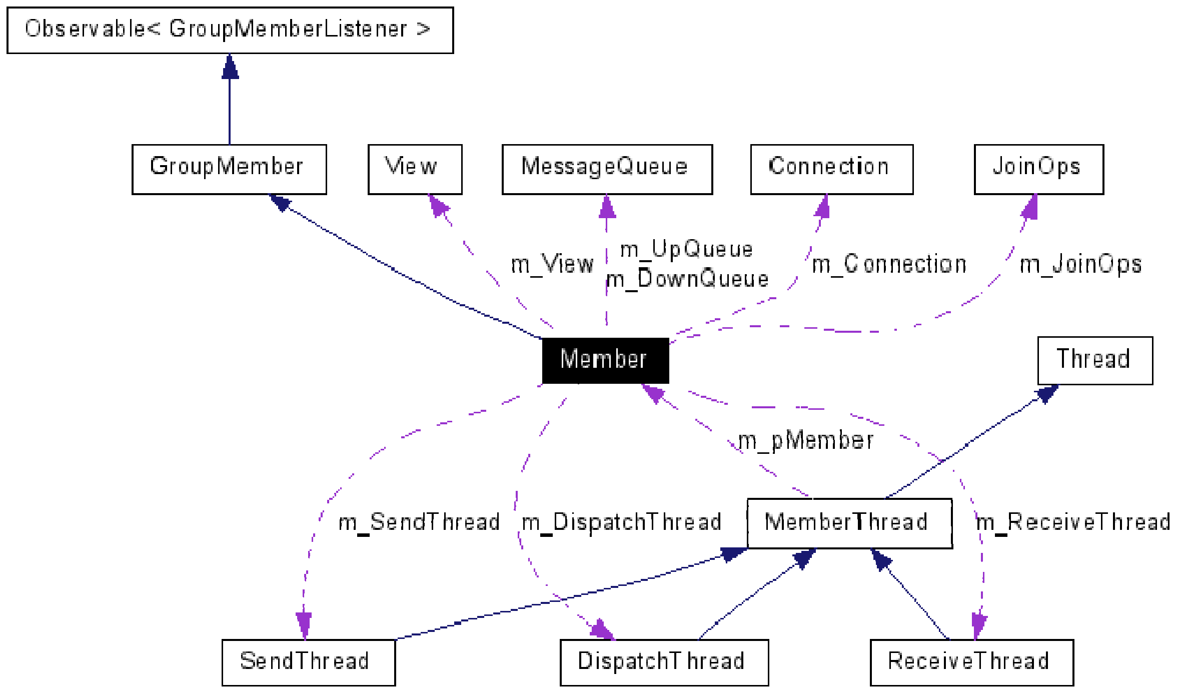
- virtual void **OnReceiveMessage** (const **DS3_string** &sMessage, const **DS3_string** &sSender)
- virtual void **OnReceiveView** (const **DS3_set**< **DS3_string** > &endpoints)

4.3.2 The member class

The GroupMember interface is implemented in the Member class to perform Ensemble-specific communication operations. The Member class provides asynchronous message passing between the endpoint system and the Ensemble system.

Member

The main class for threaded communication via Ensemble.



Member inherits **GroupMember** and implements its interface methods for joining and leaving groups and sending and receiving messages.

Member has internal methods for polling the Ensemble server, receiving messages, dispatching messages to the application and sending messages. These methods are called by different worker threads for asynchronous, non-blocking program execution. Incoming and outgoing messages are placed in queues and handled by the responsible threads.

The current Ensemble state data is kept in an **ens_member_t**, a **View** and a **Connection** object.

- enum **State** { PRE, JOINING, NORMAL, BLOCKED, LEAVING, LEFT }
- enum DS3C::Member::State **m_State**
- Connection **m_Connection**
  The Ensemble connection.
- View **m_View**
  The current view.
- ens_member_t **m_Member**
  The wrapped Ensemble member struct.
- JoinOps **m_JoinOps**
  The set of Ensemble properties to use in a communication group.
- MessageQueue **m_DownQueue**
  The queue for outgoing messages.
- MessageQueue **m_UpQueue**
  The queue for incoming messages.
- SendThread **m_SendThread**
  The thread that sends messages from the out-queue.
- ReceiveThread **m_ReceiveThread**
  The thread that receives messages from the server.
- DispatchThread **m_DispatchThread**
  The thread that dispatches received messages to the application.

- void **Connect** ()
  Resets the state to "PRE" and tries to connect to the Ensemble server.
- virtual void **Join** (const DS3_string &sGroupName)
  Join the communication group.
- virtual void **Leave** ()
  Leave the communication group.
- virtual void **Send** (const **DS3_string** &sMessage, const **DS3_set**< **DS3_string** > &destinations)
  Adds a message to the queue of outgoing messages. The message will be sent to the endpoints specified in "destinations".
- virtual void **Send** (const **DS3_string** &sMessage)
  Adds a message to the queue of outgoing messages. The message will be sent to all endpoints in the communication group.

Methods called in the send thread:

- void **Send** ()
  Sends all messages in the output queue.
- void **Send** (**QueueMessageOut** *pMsg)
  Sends a message to the Ensemble server.

Methods called in the receive thread:

- bool **Poll** ()
  Polls the Ensemble server to see whether there are new messages.
- bool **Receive** ()
  Receives messages, views and block and exit notifications from the Ensemble server.
- void **ReceiveMessage** (**ens_msg_t** &msg)
  Receives a message from the Ensemble server.
- void **ReceiveView** (**ens_msg_t** &msg)
  Receives a view from the Ensemble server.

Methods called in the dispatch thread:

- void **Dispatch** ()
  Dispatches all messages in the input queue to the local system.

The send procedure

To send a message, one of the GroupMember Send() overrides is called. The method places the message at the back of the queue of outgoing messages, then immediately returns thus making outgoing communication asynchronous. The thread function of the send thread m_SendThread waits for new messages in the queue. When a message appears it is removed from the front of the queue and it is sent to the Ensemble server. The send thread does only send messages when the Ensemble client is in a state where it is allowed to do so. If the send fails the message is put back at the front of the queue and the procedure is repeated when the error is corrected, e.g. when a new connection to the server is established after a server failure.

The receive procedure

The thread function of the receive thread m_ReceiveThread polls the Ensemble server regularly to notice new messages to collect. When there is a new message it is received and different actions are taken depending on the message type. The message can be a data (application) message, a Block message, a View message or an Exit message (see section 2.4.3). Messages containing application data are put in the queue of incoming messages for dispatch to the local system. When a Block message is received the system enters the Blocked state where no further data messages can be sent or received until the new view has been established. When a View message is received the new view is established and the system enters the Normal state. The new view is also dispatched to the local system to let the application take actions when endpoints join and leave the communication group. The Exit message is only received when the endpoint has requested to leave the group.

The dispatch procedure

The thread function of the dispatch thread m_DispatchThread waits for new messages in the queue of incoming messages. When a message appears, it is dispatched to the local system where it is handled by one of the OnReceiveMessage() and OnReceiveView() methods in the GroupMemberListener implementation. Since the dispatch is done in a dedicated thread messages arrive at the application in an asynchronous manner. It is up to the application to synchronize the handling of messages with respect to the main thread and other non-DS3 threads.

Thread synchronization

Critical sections are used to synchronize all access to the resources shared between the threads, e.g. the message queues m_DownQueue and m_UpQueue. The Ensemble client code functions are thread-safe by design.

The client state-machine

To be able to follow the rules defined by the Ensemble client state-machine the Member class keeps the current state in the m_State member variable. The state is checked in each method that interacts with the server and actions are allowed, cancelled or cause exceptions to be raised depending on the type of action and the current state.

Final notes concerning the member class

There is a considerable gap between the DS3C interface and the functionality provided by the Ensemble client code concerning synchrony, transparency and ease of use. This makes the Member class is the far most complex class in the whole DS3 system. Threads, message queues, send/receive operations, the Ensemble client state and exception handling must all work well together or the communication will definitely fail. Many hours were spent on putting all pieces together and to get them to work smoothly and effectively, as well as on tracking down several errors that could bring the system into an illegitimate state.

4.3.3 Server connection

The TCP/IP sockets connection to the server is created at DS3C initialization and kept throughout the session. If the connection is lost an exception is raised and attempts to reconnect are made.

The Ensemble server executable must be running at all times for DS3 to work. A primary attempt was made letting the server be started at operating system start-up and then be assumed to live as long as its services was needed. There have however been times when the server crashes, thus breaking all communication. To make the system more fault-tolerant, the server is checked to be alive at each connection attempt. At first, the actual check was performed from within DS3 itself, but due to some problems with reading certain Microsoft Windows registry information from within a *dynamic link library* (DLL) the code was moved to a stand-alone application called *DS3Externals*. The DS3Externals process is started from DS3 and makes sure that the Ensemble server is running, possibly by starting it if needed.

Another possibly more robust solution might be to build the Ensemble server executable as a Microsoft Windows Service thus letting the operating system be responsible for keeping the server alive. It has yet to be investigated whether it is possible to build the Ensemble server as a Window Service.

4.3.4 Group membership

The communication group is not joined automatically at initialization but requires that a join operation is performed from the application that uses DS3. The reason for this is that the join operation requires that both the send and receive threads are running to work. In some cases a thread isn't scheduled directly when it is started, causing an immediate join operation to time-out. This can be the case when trying to join directly after the threads are started from the initialization function of a DLL. The group can be joined as soon as the system is running properly and it's up to the application developer to choose when to do it.

4.3.5 Error handling

Error handling had to be done both for the Ensemble client code and for the communication layer code. All DS3C classes reports errors by raising C++ exceptions. Each thread has its own error handling semantics. Exceptions raised while sending messages are for example caught in the sender thread, possibly leading to an attempt to reconnect to the server. Errors from the client code are converted to C++ exceptions for uniform error handling. The function that handles fatal errors in libens was replaced with an external function in this layer. The function wraps the error information in an exception object and throws it. Error codes returned by the various libens functions are also thrown as exceptions.

4.4 Integration of the Xerces C++ XML library

All messages produced and consumed by the endpoint systems are XML documents. The Apache Xerces C++ XML library was integrated to facilitate the creation and handling of the documents. DS3 uses the Document Object Model (DOM) functionality provided by Xerces for effective internal handling of messages. In DOM the XML document is represented as a tree data structure that can be traversed and in which elements easily can be added or removed. This is helpful in the different endpoint systems where formatting and filtering takes place.

Before the message is passed to the GroupMember derivate it is serialized into a string for general handling.

Helper classes for creating and managing DOM documents were created to make message handling easier. This also includes XPath-to-XPath and XPath-to-DOM matching used in event filtering and query matching.

4.5 Implementation of the endpoint systems

The implementation of the publisher and subscriber endpoint systems was done more or less directly from the API and graphs showed in section 3.5.2. Each system was built from a number of node classes with clearly defined functionality. Several nodes could be used in more than one system reducing the total number of classes. All nodes have the class Node as base class. Node provides the interface for connecting nodes and passing messages between them.

Node

An endpoint system node. Contains a vector of connected nodes.

- DS3_vector< **Node** * > **m_Nodes**
  Vector of connected nodes.

- void **Connect** (**Node** *pNode)
  Connects a node to this node.
- virtual void **OnReceive** (**Event** &event)
  Handles event messages. The default implementation does nothing.
- void **Send** (**Event** &event)
  Sends the specified event message to all connected nodes.

The client and server endpoint systems were not implemented during the time-span of this final degree project.

4.5.1 Data representation

Messages passed between the nodes in the endpoint systems are objects of the Event class or any sub-class. The Event class holds a DOM document for the internal data representation.

Event

Extended DOM document class.

- DS3ErrorHandler **m_ErrorHandler**
  Handles errors from the parser.
- XercesDOMParser **m_Parser**
  A parser used for deserialization.
- DOMDocument * **m_pDocument**
  The managed document.
- DOMDocument * **m_pOwnedDocument**
  Set if this object owns the document.

- **Event** ()
  Constructs an event object without any document.
- **Event** (const **Event** &event)
  Constructs an event object with a document cloned from the given event object.
- **Event** (**DOMDocument** *pDocument)
  Constructs an event object that uses the given document.
- **Event** (const **TagName** &rootTagName)
  Constructs an event object with a single root element.
- virtual ~**Event** ()
  Reset the event object. The document is released if it's owned by the object.
- void **Reset** ()
  Reset the event object. The document is released if it's owned by the object.
- DOMDocument * **GetDocument** ()
- const DOMDocument * **GetDocument** () const

- void **Serialize** (**DS3_string** &sXmlText) const
  Serialize this objects DOM tree into an XML string.
- void **Deserialize** (const **DS3_string** &sXmlText)
  Build this events' DOM tree from the XML string.

- DOMElement * **GetRootElement** () const
- DOMElement * **InsertRootElement** (const **TagName** &tagName)
  Insert a new root element with the given tag name.
- DOMElement * **RemoveRootElement** (const **TagName** &tagName)
  Remove the root element in the DOM tree if the elements name matches the given name.
- void **AddElement** (const **TagName** &tagName, const **AttrName** &attrName, const Attribute &attribute)
  Creates and adds an element with the given attribute.
- void **AddElements** (const **TagName** &tagName, const **AttrName** &attrName, const **Attributes** &attributes)
  Creates and adds elements with the given attributes.
- void **AddReceivers** (const **Endpoints** &receivers)
  Adds receiver elements with the given endpoint identifiers set as attributes.
- void **AddSenders** (const **Endpoints** &senders)
  Adds sender elements with the given endpoint identifiers set as attributes.
- void **GetReceivers** (**Endpoints** &receivers) const
  Gets the endpoint identifiers from all receiver elements.
- void **GetSenders** (**Endpoints** &senders) const
  Gets the endpoint identifiers from all sender elements.
- void **GetAttribute** (const **TagName** &tagName, const **AttrName** &attrName, **Attributes** &attributes) const
  Gets the specified attribute for all elements with the specified tag name.

Some nodes keep their own internal state in some kind of data structure, e.g. strings, lists or trees. All data structures used are taken from the C++ Standard Template Library (STL). STL provides the means of effective and flexible handling of any data and also ensures portability of the code. Data between DS3 and the application are either passed as DOM documents or STL data structures. Subscription queries are for example XPath expressions passed as STL strings.

4.5.2 Communication layer interface

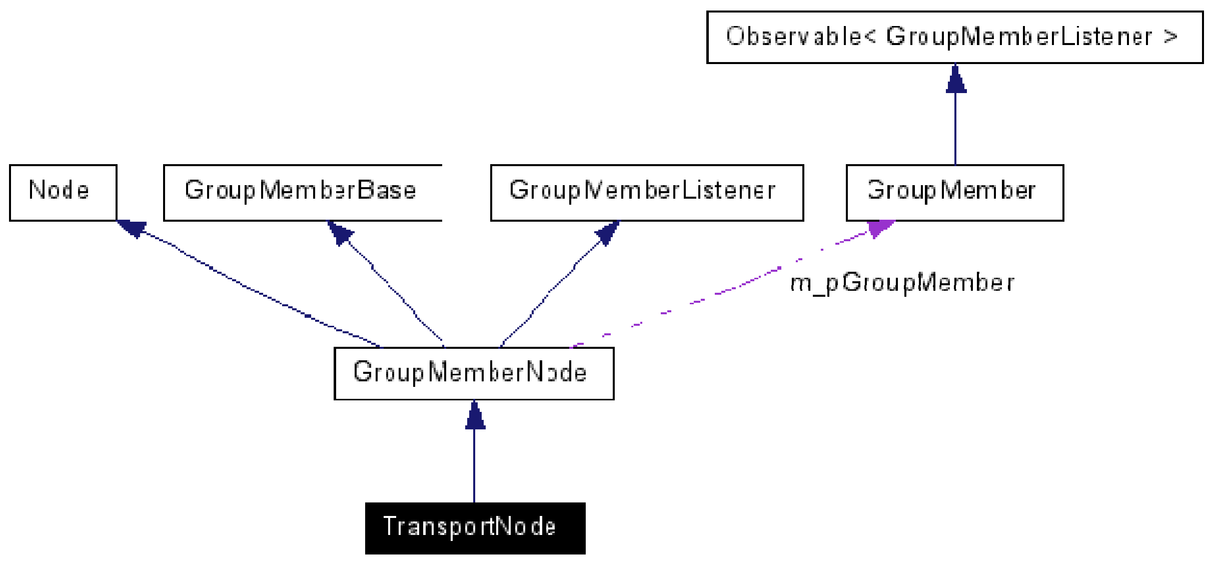
The endpoint system interaction with the underlying communication layer goes via TransportNode and GroupChangeProducerNode. In the current implementation all endpoint systems in an application share the instance of the GroupMember derivative, thus making the need for separate TransportNode and GroupChangeProducerNode instantiations superfluous. To avoid unintentional creation of multiple instances of the two classes they are made singleton.

The transport node

TransportNode transfers messages to and from the communication layer. Messages from the endpoint system are serialized from DOM documents to XML formatted strings and sent with the GroupMember::Send() interface method. Messages from the communication layer is handled by the OnReceiveMessage() handler method that parses the XML formatted string and creates a DOM document from it before passing the message to the endpoint system.

TransportNode

Handles the transport of messages over the communication layer.



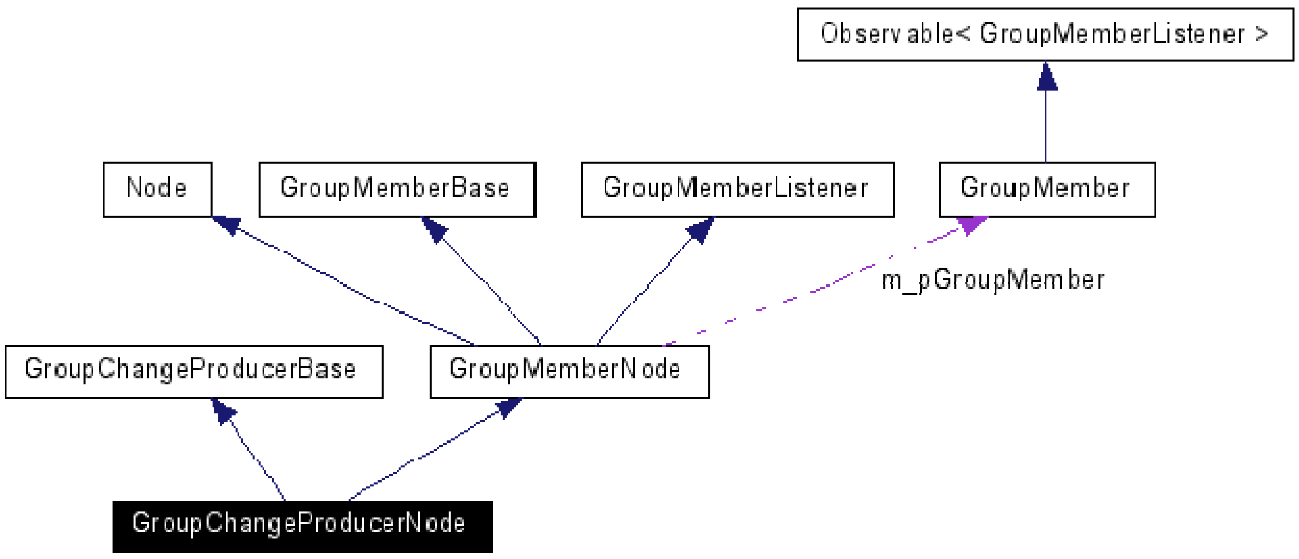
- virtual void **OnReceive** (**Event** &event)
  Serializes event and multicasts/broadcasts it via Ensemble.
- virtual void **OnReceiveMessage** (const **DS3_string** &sMessage, const **DS3_string** &sSender)
  Deserializes the message and passes it on to the connected nodes.

The group change producer node

GroupChangeProducerNode sends join and leave events on the local endpoint system when endpoints join or leave the communication group. It listens to GroupMember to receive a list of endpoints included in a newly installed view. It compares it with the list of endpoints in the previous view (the m_Endpoints member) to be able to notice which endpoints have joined and which have left.

GroupChangeProducerNode

Produces group change events.



- Endpoints **m_Endpoints**
- virtual void **OnReceiveView** (const **Endpoints** &endpoints)
  Sends group change notifications when a new view is installed.
- virtual void **Join** (const **Endpoints** &endpoints)
  Sends join events.
- virtual void **Leave** (const **Endpoints** &endpoints)
  Sends leave events.

The endpoint node

The EndpointNode class serves as a layer between the singleton interface node classes and the other endpoint system node classes. An endpoint system class inherits the EndpointNode class that will take care of the message passing to and from the communication layer. This makes an actual endpoint system differ slightly from the graphs shown in section 3.5.2. Figure 4.1 shows the structure of the interface between the endpoint systems and the communication layer in an application with two endpoint systems.

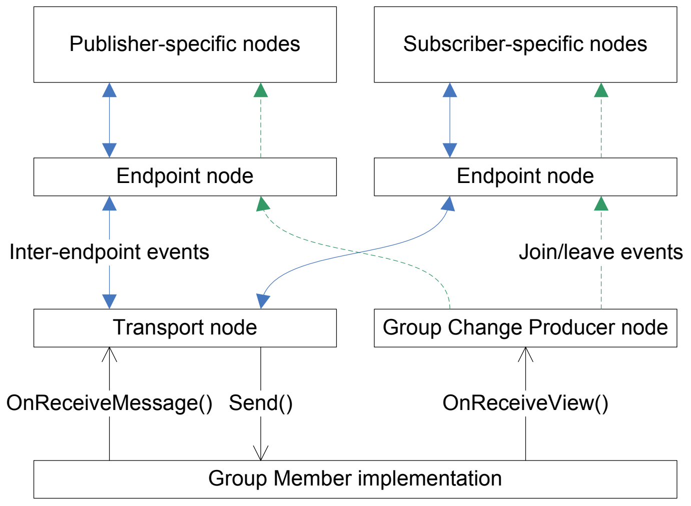
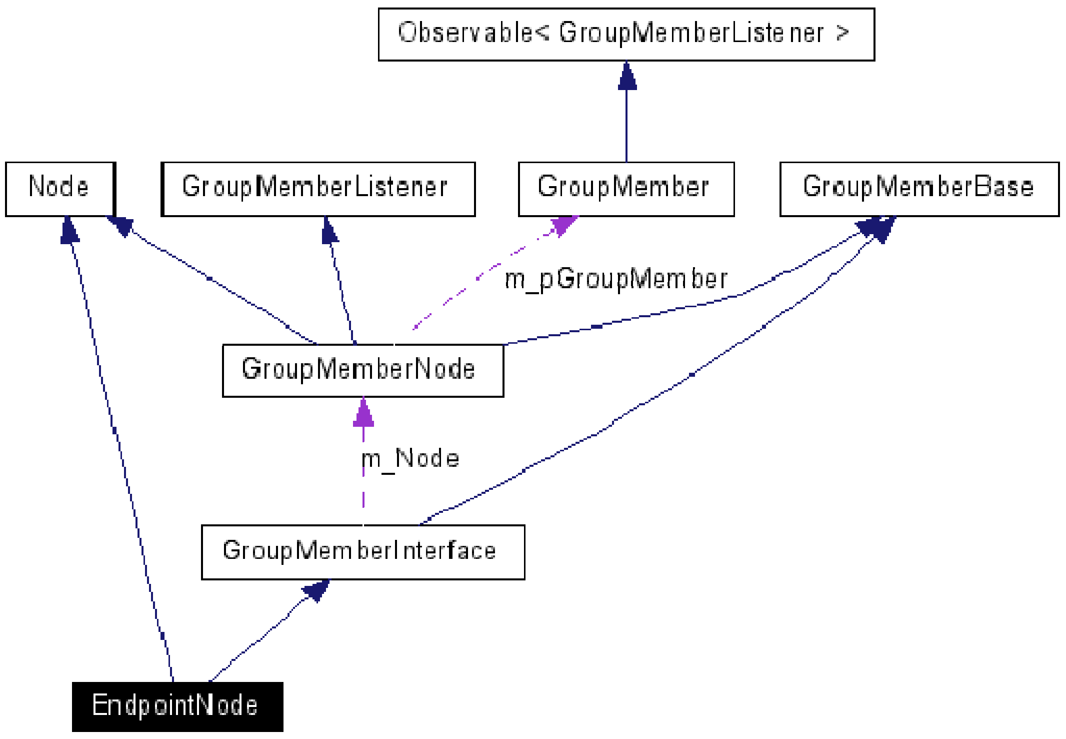


Figure 4.1 The endpoint nodes hide the singleton nature of the transport and group change producer nodes.

EndpointNode

A node that communicates with the DS3C layer.



- **EndpointNode** ()
  Connects the internal transport and group change producer nodes to this node.

- GroupChangeProducerNode * **GetGroupChangeProducerNode** ()
  Gets the singleton group change producer node.
- TransportNode * **GetTransportNode** ()
  Gets the singleton transport node.
- virtual void **OnReceive** (**Event** &event)
  Sends the event to all connected nodes.

4.6 The publisher endpoint system

To implement the publisher endpoint system several node classes had to be created. Events need to be produced and filtered, advertisements need to be both produced and consumed and subscriptions need to be consumed. Here follows a brief description of the node classes, how they process messages and what possible state data they keep. Refer to section 3.5.2 for graphs of how the nodes are connected to form the endpoint system.

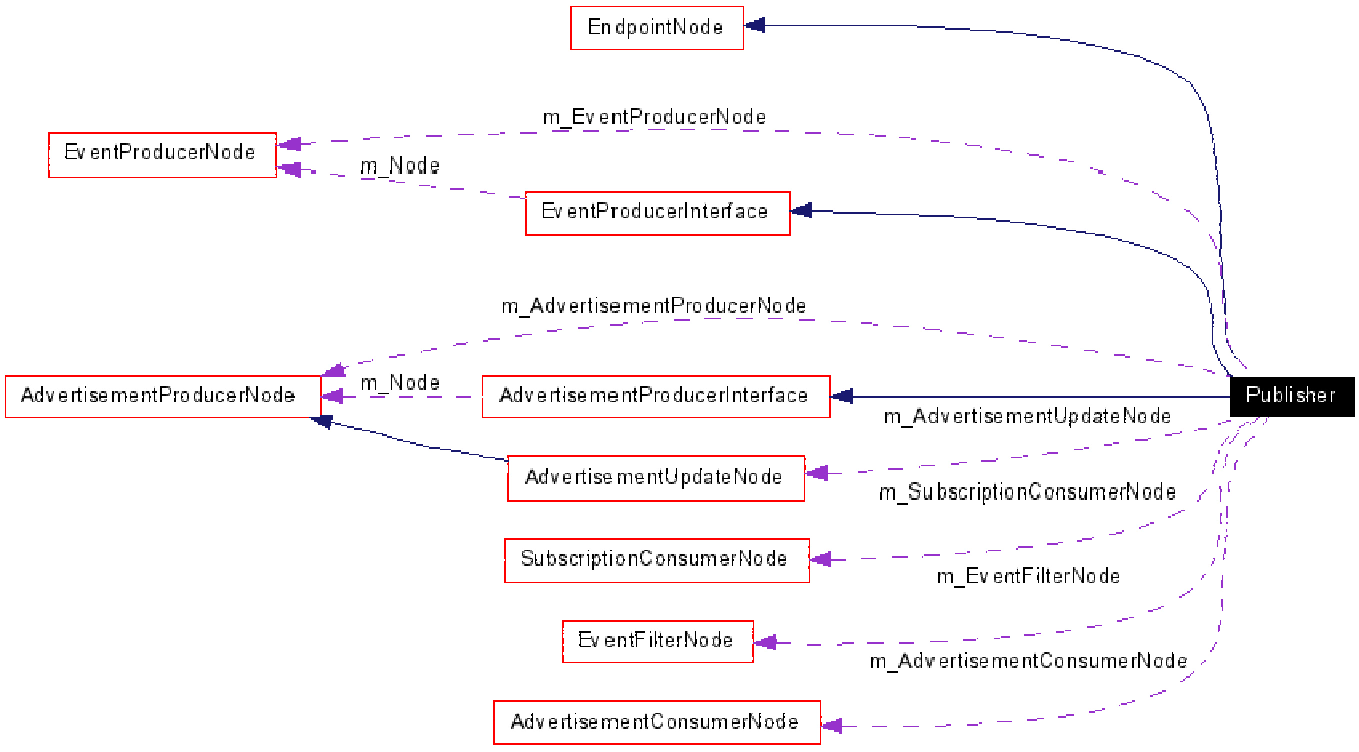
4.6.1 The publisher class

The Publisher class holds all nodes required by the publisher endpoint system. It connects the nodes for message passing and for state sharing. By inheriting the EndpointNode the Publisher class is itself a node class. It is responsible for sending events generated by the endpoint system to the communication layer and for dispatching incoming events to the endpoint system.

The Publisher class also inherits the two interface classes EventProducerInterface and AdvertisementProducerInterface. The interfaces supply the API methods provided by the corresponding node classes, namely EventProducerNode and AdvertisementProducerNode. The reason for using interface classes instead of inheriting the node classes directly is to make the polymorphic class heritance simpler, i.e. to avoid having several node base classes.

Publisher

The publisher endpoint. Contains the endpoint nodes required by the publisher endpoint system.


EndpointNode
m_EventProducerNode
EventProducerNode
m_Node
EventProducerInterface
m_AdvertisementProducerNode
AdvertisementProducerNode
m_Node
AdvertisementProducerInterface
Publisher
m_AdvertisementUpdateNode
AdvertisementUpdateNode
m_SubscriptionConsumerNode
SubscriptionConsumerNode
m_EventFilterNode
EventFilterNode
m_AdvertisementConsumerNode
AdvertisementConsumerNode

- AdvertisementConsumerNode **m_AdvertisementConsumerNode**
- AdvertisementProducerNode **m_AdvertisementProducerNode**
- AdvertisementUpdateNode **m_AdvertisementUpdateNode**
- EventFilterNode **m_EventFilterNode**
- EventProducerNode **m_EventProducerNode**
- SubscriptionConsumerNode **m_SubscriptionConsumerNode**

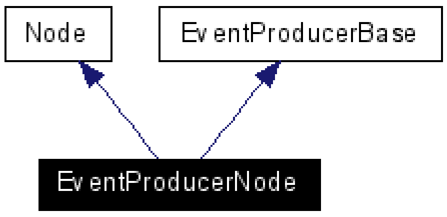
- **Publisher** ()
  Connects the endpoint nodes required by the publisher endpoint system.

4.6.2 The event producer node

EventProducerNode provides the Notify() API method. Notify() takes the application message as an Event object and wraps it in the event envelope before passing it on.

EventProducerNode

Provides methods for sending event messages.



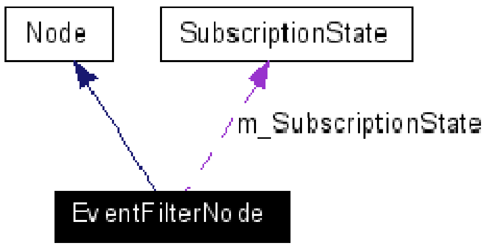
- void **Notify** (const **Event** &event)
  Send an event.
- void **Notify** (const **DS3_string** &sEvent)
  Send an event.

4.6.3 The event filter node

EventFilterNode filters the event messages according to the referenced SubscriptionState object. The content of the event message is matched to the subscription queries using the XPath-to-DOM matching helper classes. The comparison is done linearly on all queries and is therefore not very effective with many subscriptions. The event envelope is tagged with all endpoints corresponding to the matching subscriptions. If no matching subscriptions were found the event message is not passed on to the next node.

EventFilterNode

Node for filtering event messages depending on subscriptions.



- const SubscriptionState & **m_SubscriptionState**

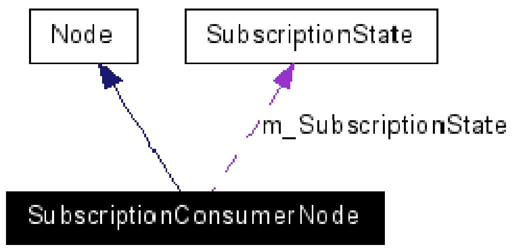
- void **GetReceivers** (**Endpoints** &receivers, const **XTree** &xtree)
  Get the endpoints that should receive this event.
- virtual void **OnReceive** (**Event** &event)
  Filters event messages depending on the subscription state.

4.6.4 The subscription consumer node and the subscription state

SubscriptionConsumerNode receives and stores subscription queries in the SubscriptionState. The state is basically a query-to-endpoints map where each subscription query is mapped to the endpoints that have subscribed with the query. When a subscription event with a new subscription query is received, the query is mapped to the subscribing endpoint. When other endpoints subscribe with the same query they are simply added to the list of endpoints for that query. When an unsubscription event is received the unsubscribing endpoint is removed from the list of endpoints for that query. The query is also removed if there are no more endpoints in the list. SubscriptionConsumerNode also listens for *leave* events from the GroupChangeProducerNode. When a subscriber endpoint leaves the communication group all its subscriptions are removed from the state.

SubscriptionConsumerNode

Updates the subscription state.



- SubscriptionState **m_SubscriptionState**

- virtual void **OnReceive** (**Event** &event)
  Handles subscription, unsubscription and leave events.
- void **OnSubscribe** (const **Endpoint** &endpoint, const **Queries** &queries)
  Updates the subscription state.
- void **OnUnsubscribe** (const **Endpoint** &endpoint, const **Queries** &queries)
  Updates the subscription state.
- void **OnLeave** (const **Endpoints** &endpoints)
  Removes all subscription queries for endpoints that have left the communication group.

SubscriptionState

Keeps the subscription state.

- typedef DS3_pair< Query, Endpoints > **Subscription**
- typedef DS3_map< Query, Endpoints > **Subscriptions**
- Subscriptions **m_Subscriptions**

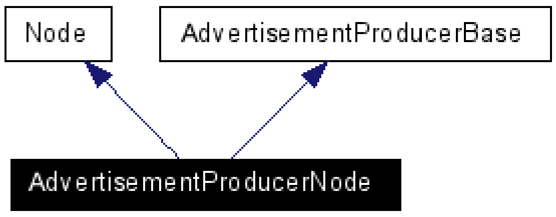
- const Subscriptions & **GetSubscriptions** () const
- void **GetEndpoints** (**Endpoints** &endpoints, const **Query** &query) const
  Gets the set of endpoints that have subscribed using the given query.
- void **GetQueries** (**Queries** &queries, const **Endpoint** &endpoint) const
  Gets the set of queries received from the given endpoint.
- void **OnSubscribe** (const **Endpoint** &endpoint, const **Query** &query)
  Inserts the specified endpoint into the list of endpoints that have subscribed using the given query.
- void **OnUnsubscribe** (const **Endpoint** &endpoint, const **Query** &query)
  Removes the specified endpoint from the list of endpoints that have subscribed using the given query.

4.6.5 The advertisement producer node

AdvertisementProducerNode provides the Advertise() and Unadvertise() API methods. Both methods take a resource identifier argument as an XPath expression string. The resource identifier is wrapped in an advertisement or unadvertisement envelope.

AdvertisementProducerNode

Provides methods for sending advertisement messages.



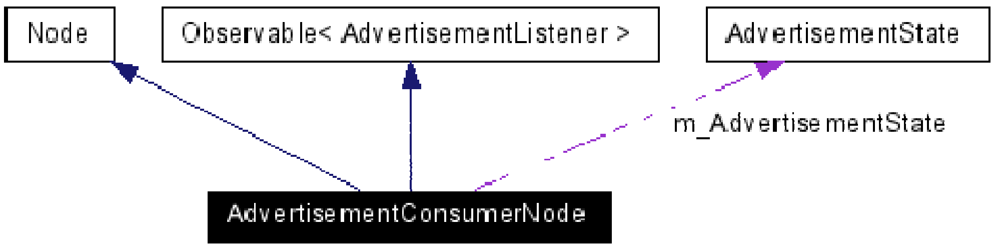
- virtual void **Advertise** (const **Resource** &resource)
  Sends an advertisement event.
- virtual void **Unadvertise** (const **Resource** &resource)
  Sends an unadvertisement event.
- void **Advertise** (const **Resources** &resources, const **Endpoints** &receivers)
  Sends advertisement events to the specified endpoints.

4.6.6 The advertisement consumer node and the advertisement state

AdvertisementConsumerNode is designed as SubscriptionConsumerNode except that it handles advertisement and unadvertisement messages. In the publisher endpoint system the only messages it receives come from AdvertisementProducerNode and the endpoint information in the state is therefore never used, nor does it receive leave events.

AdvertisementConsumerNode

Updates the state and dispatches events messages to the listeners.


Node
Observable< AdvertisementListener >
AdvertisementState
m_AdvertisementState
AdvertisementConsumerNode

- AdvertisementState **m_AdvertisementState**

- const AdvertisementState & **GetAdvertisementState** ()
- virtual void **OnReceive** (**Event** &event)
  Handles advertisement, unadvertisement and leave events.
- void **OnAdvertise** (const **Endpoint** &endpoint, const **Resources** &resources)
  Dispatches advertisement notifications among the listeners.
- void **OnUnadvertise** (const **Endpoint** &endpoint, const **Resources** &resources)
  Dispatches unadvertisement notifications among the listeners.
- void **OnLeave** (const **Endpoints** &endpoints)
  Unadvertises all resources for endpoints that has left the communication group.

AdvertisementState

Keeps the advertisement state.

- typedef DS3_pair< Resource, Endpoints > **Advertisement**
- typedef DS3_map< Resource, Endpoints > **Advertisements**
- Advertisements **m_Advertisements**

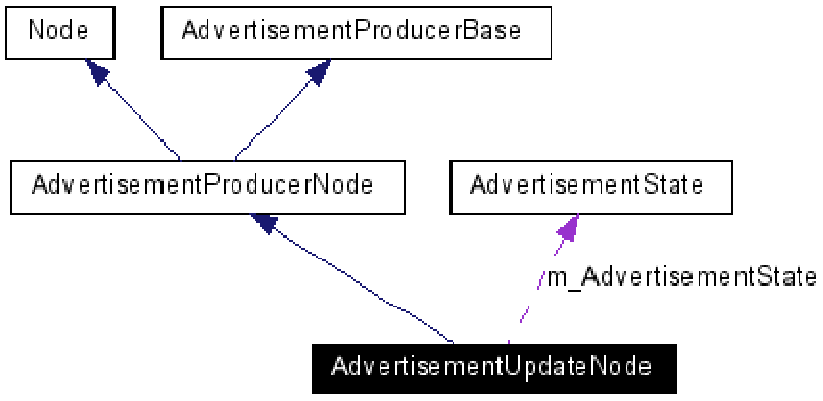
- void **GetEndpoints** (**Endpoints** &endpoints, const **Resource** &resource) const
  Gets the set of endpoints that have advertised the given resource.
- void **GetResources** (**Resources** &resources, const **Endpoint** &endpoint) const
  Gets the set of resources advertised by the given endpoint.
- bool **HasEndpoints** (const **Resource** &resource) const
  Tells whether any endpoints have advertised the resource.
- void **OnAdvertise** (const **Endpoint** &endpoint, const **Resource** &resource)
  Inserts the specified endpoint into the list of endpoints that have advertised the given resource.
- void **OnUnadvertise** (const **Endpoint** &endpoint, const **Resource** &resource)
  Removes the specified endpoint from the list of endpoints that have advertised the resource.

4.6.7 The advertisement update node

AdvertisementUpdateNode re-sends all advertisements in the shared advertisement state when new endpoints join the communication group.

AdvertisementUpdateNode

Re-sends advertisements to joining endpoints.



- const AdvertisementState & **m_AdvertisementState**

- **AdvertisementUpdateNode** (const **AdvertisementState** &state)
- virtual void **OnReceive** (**Event** &event)
  Handles the join event.
- void **OnJoin** (const **Endpoints** &endpoints)
  Advertises all resources again.

4.7 The subscriber endpoint system

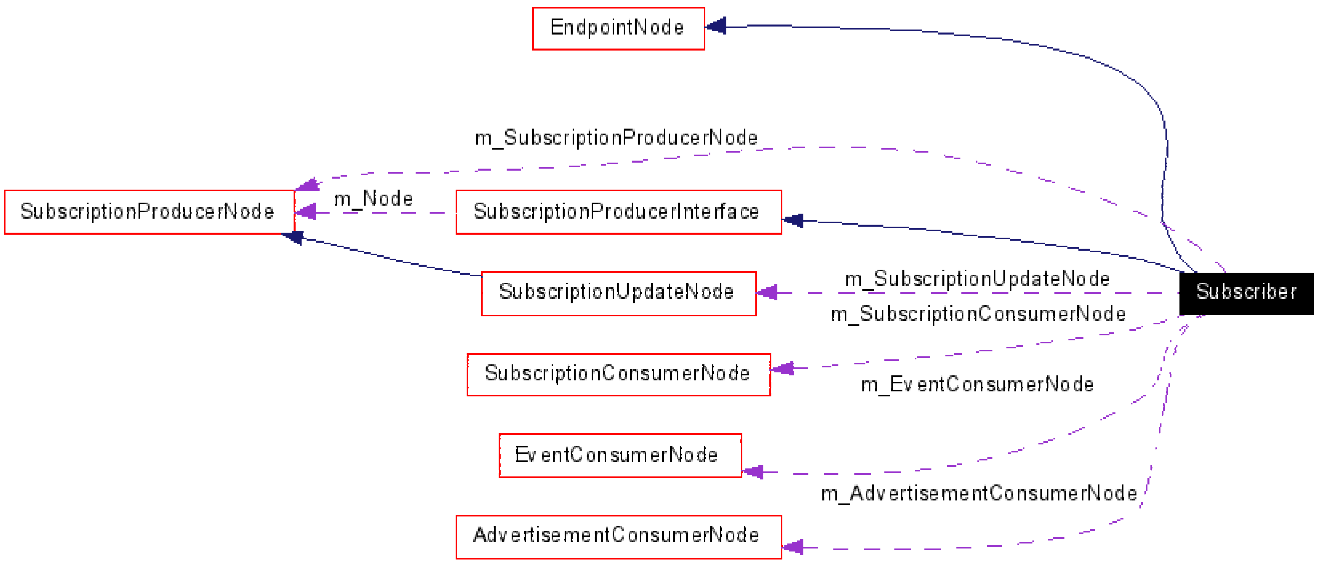
The subscriber endpoint system shares a few node classes with the publisher system but also adds several new classes. The consumer node classes are of certain interest since they have corresponding listener classes with handler methods that get called when a message is consumed. An application class that wants to take part of the information in certain messages inherits the corresponding listener class and over-rides the methods of interest. It registers itself at the proper consumer class and will then receive calls when new data is available. Here follows the classes that are new for the subscriber endpoint system.

4.7.1 The subscriber class

The Subscriber class holds all nodes required by the subscriber endpoint system. It corresponds to the Publisher class for the publisher endpoint system. Except for EndpointNode it inherits SubscriptionProducerInterface that supplies the API methods of SubscriptionProducerNode.

Subscriber

The subscriber endpoint. Contains the endpoint nodes required by the subscriber endpoint system.



- AdvertisementConsumerNode **m_AdvertisementConsumerNode**
- EventConsumerNode **m_EventConsumerNode**
- SubscriptionConsumerNode **m_SubscriptionConsumerNode**
- SubscriptionProducerNode **m_SubscriptionProducerNode**
- SubscriptionUpdateNode **m_SubscriptionUpdateNode**

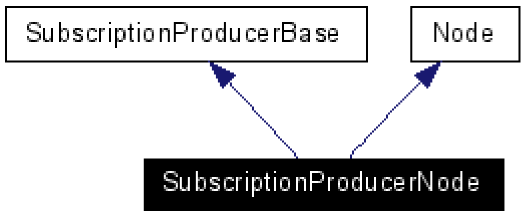
- **Subscriber** ()
  Connects the endpoint nodes required by the subscriber endpoint system.
- void **AddAdvertisementListener** (**AdvertisementListener** *pListener)
  Registers a listener with the advertisement consumer node.
- void **AddEventListener** (**EventListener** *pListener)
  Registers a listener with the event consumer node.

4.7.2 The subscription producer node

SubscriptionProducerNode provides the Subscribe() and Unsubscribe() API methods. Both methods take a subscription query argument as an XPath expression string. The subscription query is wrapped in a subscription or unsubscription envelope.

SubscriptionProducerNode

Sends subscription queries.



- virtual void **Subscribe** (const **Query** &query)
  Sends a subscription request to the system.
- virtual void **Unsubscribe** (const **Query** &query)
  Sends an unsubscription request to the system.
- void **Subscribe** (const **Queries** &queries, const **Endpoints** &receivers)
  Sends subcription events to the specified endpoints.

4.7.3 The subscription consumer node and the subscription state

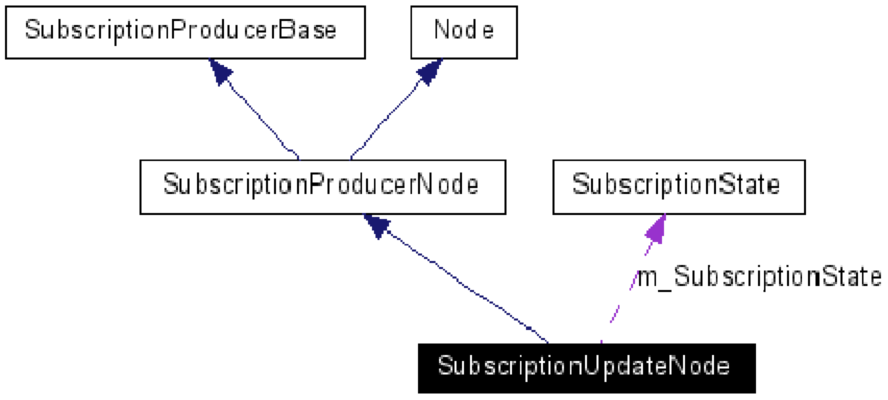
SubscriptionConsumerNode and SubscriptionState are the same as in the publisher endpoint system. Only subscriptions from the local endpoint are stored so SubscriptionConsumerNode does not have to receive leave events from GroupChangeProducerNode.

4.7.4 The subscription update node

SubscriptionUpdateNode re-sends subscriptions on advertisements. When the node receives an advertisement that matches a subscription in SubscriptionState, the subscription is sent again.

SubscriptionUpdateNode

Resends subscription queries when corresponding resources are advertised from newly connected publishers.



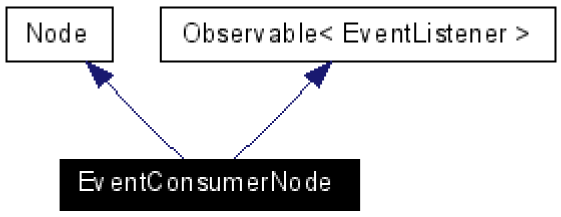
- const SubscriptionState & **m_SubscriptionState**
- **SubscriptionUpdateNode** (const **SubscriptionState** &state)
- virtual void **OnReceive** (**Event** &event)
  Handles the advertisement event.
- void **OnAdvertise** (const **Endpoints** &endpoints, const **Resources** &resources)
  Checks if the subscriber subscribes to the advertised resource and sends a subscription query in that case.

4.7.5 The event consumer node

EventConsumerNode receives event messages and extracts the application message from the envelope. It then calls the OnNotify() method on all registered EventListener objects.

EventConsumerNode

Dispatches events to the listeners.



- virtual void **OnReceive** (**Event** &event)
  Handles application event messages.
- void **OnNotify** (const **Event** &event)
  Dispatches application event notifications among the listeners.

4.7.6 The advertisement consumer node and the advertisement state

AdvertisementConsumerNode and AdvertisementState are the same classes in the subscriber system as in the publisher system. In the subscriber system the functionality of the classes are however used to a greater extent.

AdvertisementConsumerNode receives advertisements and unadvertisements and calls the proper handler method on the registered AdvertisementListener objects. It keeps current advertisements in AdvertisementState to be able to distinguish first time advertisements as well as unadvertisements that make a resource completely unadvertised. The state maps resource identifiers to lists of endpoints that have advertised them. When an advertisement is received and it is not already represented in the advertisement state, the OnResourceFound() method is called. On following advertisements for the same resource, the OnAdvertise() method is called. When an unadvertisement message is received, the sending endpoint is removed from the list of endpoints for the resource. OnUnadvertise() is called if there are still endpoints that have current advertisements for the resource. When an unadvertisement is received from the only endpoint that still has a current advertisement for that particular resource, the OnResourceDepleted() method is called. It can then be assumed that there are no more publishers for the resource. Leave events are received from GroupMembershipNode. If the leaving endpoint is represented in the advertisement state it is removed and the proper listener method is called.

5 Evaluation

The usability of DS3 has been evaluated during the different stages of the implementation by using it in various test applications. The system performance was tested thoroughly on the final implementation within the project.

5.1 Usability tests

5.1.1 Tests during the development

Early testing of the publisher and subscriber endpoint systems was done in a console application that prints log messages showing how messages propagate along the nodes and the resulting state changes. Several endpoint systems could be connected within the application and the actions and responses in endpoint interaction could therefore be analysed in a controlled manner. In this way the endpoint systems could be tested without requiring the communication layer to be implemented first. The tests resulted in several adjustments in the endpoint system designs.

The Ensemble system and client code was tested in a multi-user instant messaging application. The tests included joining and parting communication groups with resulting view changes, total and causal order message broadcasts and the reception of messages.

The instant messaging application was then extended with the threaded layer for asynchronous communication (asynchronous receive, dispatch and send operations) and the publish/subscribe endpoint systems. A positive aspect of the testing at this stage was that the publish/subscribe interaction and API worked very well, but also several stability problems appeared. The problems were partly the result of internal state inconsistencies. A lot of time was spent to pinpoint the source of the problems and the communication layer code was revised several times for reaching stability and Ensemble client state-machine compliance. This work was seriously time consuming but resulted in more correct and robust code with better error handling.

Another severe problem was that the whole Ensemble communication group could get corrupted for no obvious reason. Messages were delayed or delivered in incorrect order, communication groups got spontaneously fragmented and endpoints could sometimes not join the group at all. This problem remained an unsolved issue for a long time and was about to make the whole system useless due to unreliability. Finally it was discovered that it was IP multicast, the default Ensemble server-to-server communication network protocol, that didn't work well on the Microsoft Windows platform. When the protocol was changed to UDP everything worked as smoothly as expected. The change of network protocol does not affect the properties of the system since it is Ensemble itself that provides all required protocols for reliable, ordered message passing.

5.1.2 Target application

When the implementation of the DS3 publish/subscribe system was complete it was integrated in an application for supervision of measurement processes running on different machines. The integration went smoothly and proved the usefulness of the system in general and the publish/subscribe model with the flexibility of the XML messages in particular.

The application has successfully been tested in an environment that closely resembles the real environment of operation. It has also received positive reactions when demonstrated for people in the industry and will be the base for a more advanced supervision application.

5.2 Performance

To get a good comprehension of the capacity of the system in diverse conditions it was tested in a number of configurations, with variations in the number of endpoints, the number of computers, message frequency and message size.

The measurements were done with a distributed benchmark application that acts as both publisher and subscriber. During the benchmark interval the publisher publishes events at a given frequency and the subscriber receives all published events, from all benchmark application instances. The number of notify, send, receive and dispatch operations performed during the interval are measured, as well as number of other parameters. The benchmark results are then written to a file.

All tests were performed with benchmark application instances distributed evenly on up to 3 computers with equal performance[3]. Each instance is an Ensemble client and since it has both a publisher and a subscriber endpoint system the number of endpoints are twice the number of clients.

Tests were performed both on configurations with a single Ensemble server per computer and on configurations with one Ensemble server per application instance. Usually there is only one server per computer; the additional tests with several servers are performed to simulate an environment with more than 3 computers.

The results of the tests are shown in the respective tables. The parameters that define the system configuration are the number of Ensemble clients n_{client}, the total number of endpoint systems n_{endpoint} (both publishers and subscribers), the number of Ensemble servers n_{server} and the number of computers n_{computer} the applications are divided on.

The results of the benchmarks shown are the mean notification frequency $f_{\mathrm{n,tot}}$, the mean dispatch frequencies $f_{\mathrm{d,1}}$ and $f_{\mathrm{d,tot}}$ and the mean dispatch/notification quotient $f_{\mathrm{d,1}}/f_{\mathrm{n,tot}}$ with the standard deviation σ.

$f_{\mathrm{n,tot}}$ and $f_{\mathrm{d,tot}}$ are the mean values of the total number of events published and dispatched system-wide. $f_{\mathrm{d,1}}$ is the mean value of the number of events dispatched per endpoint. The quotient is based on $f_{\mathrm{n,tot}}$ and $f_{\mathrm{d,1}}$ rather than $f_{\mathrm{n,tot}}$ and $f_{\mathrm{d,tot}}$ to make the results comparable despite the complexity of the broadcasts (each published event is delivered to all subscribers). With this presentation the delivery frequency equals the notification frequency if the system has the capacity to deliver all events in real-time.

Note that $f_{\mathrm{d,1}}/f_{\mathrm{n,tot}}$ is the mean of all quotients rather than the quotient of the two mean values. $f_{\mathrm{d,1}}/f_{\mathrm{n,tot}}$ is 1 in a configuration where the system manages to deliver events at the same rate as they are sent. If $f_{\mathrm{d,1}}/f_{\mathrm{n,tot}}$ is below 1 the system can't deliver messages in real-time and will eventually break due to internal buffer overflows, if the notification rate doesn't decrease in time. It is possible for $f_{\mathrm{d,1}}/f_{\mathrm{n,tot}}$ to be above 1 if the delivery of messages are delayed and then performed in a short burst. That is considered as erroneous behaviour.

5.2.1 Normal load tests

In normal operation a system can be assumed to consist of up to a few ten endpoints, both publishers and subscribers. A publisher can be assumed to publish up to a few ten events per second but usually less. In the Ensemble documentation it is advised against using totally ordered broadcasts in systems of more than 12 clients due to performance issues. To get insight in whether the system performance is sufficient for normal operation and what impact the 12 client restriction has, the performance was measured on a few different system configurations.

In the normal load tests the attempt was for each publisher to send events continuously with a frequency of approximately 10 events per second, but in configurations where that proved to put too heavy load on the Ensemble servers the frequency was decreased to 1 event per second. The event message size is 1000 B[4]. Due to the number of tests that had to be performed and the time required to set the test configurations up, each test was limited to run for 100 seconds.

[3] Dell Dimension 8400, Intel Pentium 4 3.0 GHz, 1 GB RAM.

[4] This is the application message size. The size of messages sent to the Ensemble server can be considerably greater due to DS3 metadata overhead. The size of messages passed between Ensemble servers is unknown.

n_{client}	6	12	18	24	30
n_{endpoint}	12	24	36	48	60
n_{server}	3	3	3	3	3
n_{computer}	3	3	3	3	3
$f_{\text{n,tot}}$	60	120	176	24	31
$f_{\text{d,1}}$	60	120	176	24	31
$f_{\text{d,tot}}$	360	1440	3168	576	930
$f_{\text{d,1}}/f_{\text{n,tot}}$	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
σ	0.00	0.00	0.00	0.00	0.00

Table 5.1 Normal load tests with one Ensemble server per computer. Note that the 24 and 30 clients results are based on ~1 event per second and publisher.

n_{client}	6	12	18	24	30
n_{endpoint}	12	24	36	48	60
n_{server}	6	12	18	24	30
n_{computer}	3	3	3	3	3
$f_{\text{n,tot}}$	60	120	173	24	31
$f_{\text{d,1}}$	60	120	173	24	31
$f_{\text{d,tot}}$	360	1440	3114	576	930
$f_{\text{d,1}}/f_{\text{n,tot}}$	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
σ	0.00	0.00	0.00	0.00	0.00

Table 5.2 Normal load tests with one Ensemble server per application instance. Note that the 24 and 30 clients results are based on ~1 event per second and publisher.

The results show that the system handles ~10 events per second and publisher for configurations with up to 18 Ensemble clients. The same frequency at a higher number of clients makes the system get overloaded and break down. With a decreased notification frequency of ~1 event per second and publisher the system runs well for both 24 and 30 clients too. No difference is seen between the results for one server per computer and several servers per computer. Tests performed on configurations with a number of clients of between 18 and 24 would possibly indicate such a difference.

5.2.2 Extreme load tests

To see how the system reacts to in an over-loaded state several additional tests were performed. As in the normal load tests all benchmark application instances act as both publishers and subscribers. Events are published at a rate that is deliberately more than the system can handle and are consequently delivered at a lower rate. A test is performed by letting each publisher send event messages of 1000 B at a rate that makes the total (system-wide) number of events published per second to be approximately 1000 messages/s. Each test is only performed for about 1 second to get a value of the throughput without breaking the system due to buffer overflows.

n_{client}	1	2	3	6	12	18	24
n_{endpoint}	2	4	6	12	24	36	48
n_{server}	1	2	3	3	3	3	3
n_{computer}	1	2	3	3	3	3	3
$f_{\text{n,tot}}$	866	894	918	769	667	635	-
$f_{\text{d,1}}$	623	615	617	615	477	231	-
$f_{\text{d,tot}}$	623	1230	1851	3690	5724	4158	-
$f_{\text{d,1}}/f_{\text{n,tot}}$	**0.72**	**0.69**	**0.67**	**0.80**	**0.72**	**0.37**	**-**
σ	0.01	0.01	0.01	0.01	0.03	0.22	-

Table 5.3 Extreme load with one Ensemble server per computer.

n_{client}	1	2	3	6	12	18	24
n_{endpoint}	2	4	6	12	24	36	48
n_{server}	1	2	3	6	12	18	24
n_{computer}	1	2	3	3	3	3	3
$f_{\text{n,tot}}$	866	894	918	768	613	504	373
$f_{\text{d,1}}$	623	615	617	611	116	74	54
$f_{\text{d,tot}}$	623	1230	1851	3666	1392	1332	1296
$f_{\text{d,1}}/f_{\text{n,tot}}$	**0.72**	**0.69**	**0.67**	**0.80**	**0.19**	**0.15**	**0.14**
σ	0.01	0.01	0.01	0.01	0.03	0.02	0.02

Table 5.4 Extreme load tests with one Ensemble server per application instance.

Due to thread timing issues on the platform and to the limited processor resources available, the number of events actually published got considerably lower than the attempted 1000 events/s. This is particularly the case with many benchmark application instances running on each computer. Unfortunately this makes the quotients not entirely comparable, but they should still give a picture of the behaviour of the system at extreme loads.

For up to 6 instances the throughput hits the roof at about 615 delivered events/s. Note that the results are duplicated for 1 to 3 instances since the number of servers are equal anyway. At higher numbers of endpoints the performance decreases. The capacity seems to be higher with a single server per computer, but the system also gets unstable faster. For the 24 instance test no result is obtained at all due to system failure. In the tests with multiple servers $f_{\text{d,tot}}$ drops at 12 instances but are then only decreasing slowly and the system is still stable. The workload per server decreases with one server per Ensemble client, which is probably the reason for the higher stability. More servers are however involved when applying the communication protocols, for example message ordering, which can be the reason for the lower capacity.

The system tends to collapse rather easily if the communication is kept running for more than the one second test period. The failure usually appears as abnormal delays in communication of up to several minutes, followed by Ensemble servers exiting due to internal buffer overflows.

5.2.3 Summary

The following results were measured on 3 computers, with system configurations where all application instances published events simultaneously and received events from all other instances. The application message size was 1000 B.

In a system of up to 18 instances it is possible to publish at least 10 events per instance and second without overloading the system. In a system of up to 30 application instances it is possible to publish at least 1 event per instance and second.

The maximum number of delivered events per instance and second is slightly above 600.

If more events are published than the system has the capacity to handle, the system will eventually break down due to internal buffer overflows.

6 Conclusions

6.1 Review of the goals

This section discusses the conclusions made to reach the goals of section 3.1.

6.1.1 System design

The design phase was spontaneously split up in different parts. First the required parts of the system were identified with respect to the requirements. It was definitely helpful to have a clear set of requirements to base decisions on structure and semantics on. A sketch was made over the composition of the different software modules with their respective responsibilities as well as a preliminary API. The hard part at this stage was to decide how complex communication patterns should be allowed by the system. This is discussed further in the next section.

With an overall picture set the idea was to define the structure and semantics in greater detail for each part. At first this wasn’t supposed to be that difficult, but for many of the decisions several new questions appeared. This was for example the case when attempting to specify the semantics of the publish/subscribe system. It was at this early stage hard to see exactly what semantic properties would be best and in the extent it became very hard to make any decisions at all on the modules concerning publish/subscribe. To get along in the progress the implementation was carefully started parallel with the specification part.

6.1.2 Base communication technology

The decision of what technology to base the system on was a lot harder than expected. First it had to be decided what properties were required by the communication. It wasn’t obvious that causal and total ordering of messages would have to be fulfilled to support the applications that will possibly use the system. The conclusion was however that to spare the creative application developer from a lot of extra effort, both ordering properties should be supported. Otherwise it isn’t unlikely that an application that actually requires ordering is developed in the future, without taking into account what happens when messages are delivered in an inconsistent order. This could definitely lead to obscure errors that would be hard to trace down, and requirement A1 would be broken.

At first it was considered to implement the required ordering algorithms on top of some common technology of distributed computing such as CORBA. The decision was however to try to find a more complete technology to start with. Only the implementation of the algorithms would probably require a lot of work and it was uncertain how good the result would be, regarding for example complexity, reliability and ease of use. The decision did definitely pay off since the use of a group communication system did except ordering also provide membership handling, fault tolerance, bootstrapping (the possibility for a joining member to find the other members without knowing some specific address), etc.

It was not at all that easy find a technology for communication that fulfilled the desired requirements and that was suitable for the system environment. Most of the few projects found had either to weak properties, were commercial or limited to non-profit use, were inactive since long or were unnecessary complex, for example because they were targeted at WAN applications. Ensemble was selected because it was considered to be very suitable for DS3 and the completeness of the system did certainly reduce the effort of the implementation drastically.

The integration of Ensemble was however not painless. There were several issues concerning the threaded C++ client implementation, specifically to get the client state machine to behave correctly with the rather limited documentation as guide. This issue got excessively serious with the rough error handling in the C client code and limited possibility of insight in the Ensemble server state. A ready C++ client implementation, like the Java client implementation, would have been appreciated. Ensemble also suffered from severe stability problems until the network transport protocol was changed from the default IP multicast protocol to UDP. More information about this problem would have reduced the amount of work put into making the system stable.

It is hard to say whether the use of another system such as NaradaBrokering would have reduced the overall effort with retained properties and functionality.

6.1.3 Message format

It was a rather easy decision to select XML as the message format foundation. This decision proved to be justified due to the versatility and flexibility of the format and the ease of use in target applications. The tools available did also prove to be complete and useable.

6.1.4 Implementation

Even though the implementation of some parts did go through a few iterations the overall conclusion is that the implementation went well. The clear areas of responsibility, clear interfaces between the layers and the modularity of the endpoint systems made the implementation straightforward. No part was found to be ill defined and changes were mostly due to changes in decisions on semantics or increased robustness.

The publish/subscribe system and all underlying layers were fully implemented. The hardest part was definitely the threaded communication layer and its integration with the Ensemble client code. The client/server system is yet to be implemented.

6.1.5 Evaluation

The usability and flexibility of the publish/subscribe semantics of the system has been successfully evaluated in an application for supervision of measurement processes running on different computers. It has also been used to remotely control the parameters of a measurement process, simulating a client/server style application using publish/subscribe communication. The system has proven to be very suitable for these kinds of applications.

The performance of the system in different configurations was measured with a benchmark application on a small set of computers. The results, as presented in section 5.2, were measured on 3 computers, with system configurations where all application instances published events simultaneously and received events from all other instances. The application message size was 1000 B.

In a system of up to 18 instances it is possible to publish at least 10 events per instance and second without overloading the system. In a system of up to 30 application instances it is possible to publish at least 1 event per instance and second. If more events are published than the system has the capacity to handle, the system will eventually break down due to internal buffer overflows.

The maximum number of delivered events per instance and second is slightly above 600.

6.1.6 Target application

A preliminary version of the supervision application was developed as part of the system evaluation. The application provides all the desired basic functionality of supervising all running measurement processes. It will be developed into a final product with additional work required only on the end-user interface and possible extended functionality.

6.2 Work to be completed

There are still issues in the implementation concerning functionality and performance. For many applications the current implementation is complete enough, but to make the system as usable as intended work should be put into the solution of these issues.

6.2.1 Functionality

The client/server endpoint systems are yet to be implemented. Several node classes from the publish/subscribe implementation can be borrowed or imitated which would reduce the work considerably. At the time of writing an application with client/server behaviour has actually been tested where the server uses a subscriber endpoint to receive requests and a publisher endpoint to send replies. Correspondently, the client side uses a publisher endpoint for sending requests and a subscriber endpoint for receiving replies. Even though this solution works, a real client/server system would be much easier to integrate in applications.

6.2.2 Performance

Higher performance could probably be reached by relaxing the properties of communication in applications were such properties are unnecessary. This particularly applies to the costly total ordering of messages. Several communication groups with different properties could possibly co-exist.

The system could possibly also benefit in performance from optimization of the internal handling of messages. XML messages are for example converted between string and DOM tree representation in many cases where it is unnecessary. Overloaded methods that take respective data type and that only make conversions where necessary would reduce computational work. There are also some cases where unnecessary copies of DOM trees are made to preserve the structure of a message. If the message isn’t required to remain in its original state the structure change should be made on the message directly without performing the expensive copying.

7 References

[Amir et al 1995] Amir, Y., Moser, L. E., Melliar-Smith, P. M., Agarwal, D. A., Ciarfella, P. 1995. The Totem single-ring ordering and membership protocol. *ACM Trans. Comput. Syst. 13, 4 (Nov.),* 311-342.

[Apache 2004] The Apache Software Foundation. 2004. Xerces-C++ Documentation. http://xml.apache.org/xerces-c/

[Birman and van Renesse 1994] Birman, K., van Renesse, R. 1994. Reliable Distributed Computing with the Isis Toolkit. IEEE Computer Society Press, Los Alamitos, CA.

[Birman et al 2000] Birman, K., Constable, R., Hayden, M., Hickey, J., Kreitz, C., van Renesse, R., Rodeh, O., Vogels, W. 2000. The Horus and Ensemble projects: Accomplishments and limitations. In *DARPA Information Survivability Conference and Exposition (DISCEX 2000)*, 149-161. IEEE Computer Society Press.

[Borland 2004] Borland Software Corporation. 2004. VisiBroker VisiNotify Guide.

[Carzaniga et al 1998] Carzaniga, A., Di Nitto, E., Rosenblum, D., Wolf, A. 1998. Issues in Supporting Event-based Architectural Styles. In *Proceedings of the Third International Software Architecture Workshop (ISAW-3)*, 17-20.

[Chow and Johnson 1997] Chow, R., Johnson, T. 1997. Distributed Operating Systems & Algorithms. Addison-Wesley. ISBN 0-201-49838-3.

[Constable et al 1986] Constable R. et al. 1986. Implementing Mathematics in the NuPRL Proof Development Syste,. Prentice-Hall.

[Coulouris et al 2001] Coulouris G., Dollimore J., Kindberg T. 2001. Distributed Systems Concepts and Designs. Addison Wesley, third edition, 2001.

[Défago et al 2003] Défago, X., Schiper, A., Urbán, P. 2003. Total Order Broadcast and Multicast Algorithms: Taxonomy and Survey. Japan Advanced Institute of Science and Technology, Ishikawa, Japan, Research Report IS-RR2003-009, September 2003.

[Diao et al 2002] Diao Y., Fischer P., Franklin M. J. and To R. 2002. YFilter: Efficient and Scalable Filtering of XML Documents. In *Proc. of the 18th Intl. Conf. on Data Engineering*, pages 341--342, San Jose, California, February 2002.

[Doxygen 2005] Doxygen Release 1.4.5. http://www.doxygen.org/.

[Eugster et al 2001] Eugster, P., Felber, P., Guerraoui, R., Kermarrec, A.-M. 2001. The Many Faces of Publish/Subscribe. Technical Report DSC ID:2000104, EPFL, January 2001.

[Gokhale et al 2002] Gokhale A., Kumar B., Sahuguet, A. 2002. Reinventing the Wheel? CORBA vs. Web Services. In *Practice and Experience Track, Eleventh International Conference on World Wide Web (WWW2002).*

[Graham et al 2004] Graham, S. et al. 2004. Publish-Subscribe Notification for Web Services. http://www-106.ibm.com/developerworks/library/ws-pubsub/.

[Hayden 1998] Hayden, M. 1998. The Ensemble System. PhD thesis, Dept. of Computer Science, Cornell University.

[Hayden and Rodeh 2004] Hayden, M., Rodeh, O. 2004. Ensemble Reference Manual. Cornell University. http://dsl.cs.technion.ac.il/projects/Ensemble/doc.html

[Hayden and Rodeh 2004b] Hayden, M., Rodeh, O. 2004. Ensemble Tutorial. http://dsl.cs.technion.ac.il/projects/Ensemble/doc.html

[Leroy 1997] Leroy, X. 1997. The Objective Caml system release 1.05. INRIA, France, May 1997.

[OMG 2004a] Object Management Group. 2004. Common Object Request Broker Architecture: Core Specification. http://www.omg.org/technology/documents/formal/corba_iiop.htm.

[OMG 2004b] Object Management Group. 2004. Event Service Specification. http://www.omg.org/technology/documents/formal/event_service.htm.

[OMG 2004c] Object Management Group. 2004. Naming Service Specification. http://www.omg.org/technology/documents/formal/naming_service.htm.

[Pallickara and Fox 2003] Pallickara S. and Fox G. 2003. NaradaBrokering: A Middleware Framework and Architecture for Enabling Durable Peer-to-Peer Grids. In *Proceedings of ACM/IFIP/USENIX International Middleware Conference Middleware-2003.* pp 41-61. *Lecture Notes in Computer Science 2672 Springer 2003*, ISBN 3-540-40317-5.

[van Renesse et al 1996] van Renesse, R., Birman, K., Maffeis, S. 1996. Horus: A flexible group communication system. In *Communications of the ACM, 39(4):76-83, April 1996.*

[Rosenblum and Wolf 1997] Rosenblum, D., Wolf, A. 1997. A Design Framework for Internet-Scale Event Observation and Notification. In *6th European Software Engineering Conference/ACM SIGSOFT 5th Symposium on the Foundations of Software Engineering*, September 1997. Springer.

[Sun 2002] Sun Microsystems. 2002. Java Message Service Specification, version 1.1. http://java.sun.com/products/jms/docs.html.

[Vitenberg et al 1999] Vitenberg, R., Keidar, I., Chockler, G. V., Dolev, D. Group Communication Specifications: A Comprehensive Study. Tech. report CS99-31, Comp. Sci. Inst., The Hebrew University of Jerusalem and MIT Technical Report MIT-LCS-TR-790, September 1999.

[W3C 1999] W3C. 1999. XML Path Language (XPath) Version 1.0. W3C Recommendation 16 November 1999. http://www.w3.org/TR/xpath

[W3C 2002c] W3C. 2004. XML Schema Part 0: Primer Second Edition. W3C Recommendation 28 October 2004. http://www.w3.org/TR/xmlschema-0/.

[W3C 2004a] W3C. 2004. Web Services Architecture. http://www.w3.org/TR/ws-arch/.

[W3C 2004b] W3C. 2004. Extensible Markup Language (XML) 1.0 (Third Edition). W3C Recommendation 04 February 2004. http://www.w3.org/TR/2004/REC-xml-20040204/.

[W3C 2005] W3C. 2005. XML Binary Characterization Use Cases. W3C Working Group Note 31 March 2005. http://www.w3.org/TR/xbc-use-cases/.

Appendix A Message formats

A.1 Application message envelopes

For the DS3 endpoint systems to know which messages they should handle each application message is inserted into a specific *envelope*. The application message can be any well-formed XML document. The envelope shows what kind of message that it contains and can contain additional endpoint system specific data. In practice the envelope is an XML element inserted as the root node of the XML message document. All envelope elements have a *sender* attribute specifying the system-wide *endpoint identifier* of the sending endpoint. Each envelope element type can have additional attributes and elements. A producer node in the endpoint system usually creates the envelope. It is in some cases a small container for a large message but could also be the whole message.

The event producer node in the publisher endpoint system inserts application event messages into the *Event* envelope. The Event element has an attribute *approot* that holds the name of the application message root element. *Destination* elements specify which endpoints the message should be sent to. If there is no *Destinaion* element the message is sent to all endpoints in the communication group. The event consumer node in the subscriber endpoint system removes the Event envelope and delivers the application event message to the application.

```
<Event sender=”publisher ID” approot=”ApplicationMessageRoot”>
    <Destination endpoint=”subscriber 1 ID”/>
    ...
    <Destination endpoint=”subscriber n ID”/>
    <ApplicationMessageRoot>
        ...
    </ApplicationMessageRoot>
</Event>
```

The reply envelope created by the request consumer node in the server endpoint system is similar to the Event envelope. There is however only one destination.

```
<Reply sender=”server ID” approot=”ApplicationMessageRoot”>
    <Destination endpoint=”client ID”/>
    <ApplicationMessageRoot>
        ...
    </ApplicationMessageRoot>
</Reply>
```

Subscription and advertisement envelopes follow the same format. The query or resource identifier data they contain is XPath expression strings stored as attributes. Since subscription and advertisement messages are broadcasted no destinations need to be specified.

```
<Subscription sender=”subscriber ID” query=”query”/>
<Unsubscription sender=”subscriber ID” query=”query”/>
<Advertisement sender=”endpoint ID” resource=”resource identifier”/>
<Unadvertisement sender=”endpoint ID” resource=”resource identifier”/>
```

The request producer node in the client endpoint system sends query messages similar to the subscriptions but with a destination element.

```
<Request sender=”client ID” query=”query”>
    <Destination endpoint=”server ID”/>
</Request>
```

A.2 The structure of Rektron-specific application messages

All application messages that provide information on some resource should use the following hierarchy.

/Company/Product/Context/Content

Company is the name of the company that has developed the software. *Product* is the name of the product or product line that the resource belongs to. *Context* can be used to define that a message belongs to a certain domain of messages or a specific module in the application. *Content* is the actual data the message contains.

The company and product levels in the hierarchy should always be set but what values the other two layers can take is up to each application. For the Rektron SQC application suite the context level could for example take the value *Measurement* for resources regarding ongoing measurements. For events published in this context, the content could for example be *StartMeasure*, *EndMeasure* and *NewValue*. An event message indicating that Rektron SQC has collected a new value would then have the following hierarchy:

```
/Rektron/SQC/Measurement/NewValue
```

Here follows an example to illustrate how an event message would look like. Let’s consider a situation where a factory drills holes in metal sheets. The diameter of random holes is measured for statistical quality control (SQC). A measurement machine performs the measurement automatically and the measure value is collected by software for statistical quality control, running on a computer on the company’s LAN. The software in use is Rektron SQC that uses DS3 to publish measurement events. On another computer the events is caught by an application for supervision. Events are published in the /Rektron/SQC/Measurement context for a certain *article number* (in this case the identity of a certain sheet type) and *measure property* (the dimension that is measured). Note that both message content and format is simplified for brevity.

The XPath expression used to advertise the resource can look like this:

```
/Rektron/SQC/Measurement[@article="Aluminium, 5 mm" and @property="Diameter"]
```

The corresponding XML document base for event messages would then look like this:

```
<Rektron>
    <SQC>
        <Measurement article=”Aluminium, 5 mm” property=”Diameter”/>
    </SQC>
</Rektron>
```

When a new value is collected, it is inserted as a content tag in the document and the event is published.

```
<Rektron>
    <SQC>
        <Measurement article=”Aluminium, 5 mm” property=”Diameter”>
            <NewValue value=”9.012”/>
        </Measurement>
    </SQC>
</Rektron>
```

The publisher endpoint system inserts the event data into its specific *Event* envelope.

```
<Event>
    <Rektron>
        <SQC>
            <Measurement article=”Aluminium, 5 mm” property=”Diameter”>
                <NewValue value=”9.082”/>
            </Measurement>
        </SQC>
    </Rektron>
</Event>
```

The event filter node checks which subscribers have subscribed to the resource and add their system-wide endpoint identifiers to the envelope.

```
<Event>
    <Destination endpoint=”{Endpt:192.168.16.30:6793:976:2}”/>
    <Destination endpoint=”{Endpt:192.168.16.45:6793:630:6}”/>
    <Rektron>
        <SQC>
            <Measurement article=”Aluminium, 5 mm” property=”Diameter”>
                <NewValue value=”9.082”/>
            </Measurement>
        </SQC>
    </Rektron>
</Event>
```

When the message reaches the communication layer, the system-wide endpoint identifiers are translated to identifiers used by the subsystem and the destination elements are removed. If the XML document has been handled as a DOM tree it is serialized to a string. Then the message is sent to the subsystem for transport. At the subscriber side the *Event* elements are removed and the message is delivered to the application.