Selecting Middleware for the Intelligent Autonomous Systems Group

ing. Bas Terwijn
Juli 15, 2005

Supervisors:

Prof. Dr. D.J.N. van Eijck (CWI, Utrecht University)
Dr. ir. M. Maris (University of Amsterdam, TNO Fel)

Work conducted at:

IAS group, Faculty of Science, University of Amsterdam

Master thesis concluding one year master Software Engineering

Selecting Middleware for the Intelligent Autonomous Systems Group

Bas Terwijn

Author	: ing. Bas Terwijn
Email	: bterwijn@science.uva.nl
Date	: August 8, 2005
Supervisors	: Dr. ir. Marinus Maris
	Prof. Dr. Jan van Eijck
Conducted at	: Intelligent Autonomous Systems group
	Faculty of Science, University of Amsterdam
	Kruislaan 403
	1098 SJ Amsterdam, The Netherlands

Abstract

This work aims to select the most appropriate middleware product for the Autonomous Intelligent Systems group for the realization of distributed systems, specifically autonomous robot and distributed sensor systems. Each middleware product imposes structural and behavioral constraints on the systems it helps to realize. These constraints are a result of the specific architecture that a middleware product imposes on systems. In order to select the most appropriate middleware product, firstly interviews are conducted to come up with common requirements for the two types of systems. Secondly, different middleware induced architectural styles are evaluated based on their effects on the identified requirements which results in a selection of the request-reply and event based style for communication. Thirdly, different candidate products that support these styles and fulfill the requirements are selected for evaluation, namely TAO, ICE and SOAP compatible products. From these products ICE is chosen as the best suitable product mainly based on its low complexity, efficient support for the required operating systems and languages, and the tight integration it allows for the request-reply and event based style. Additionally, different enhancements for ICE are realized in order to better fulfill the requirements.

Acknowledgements

I would like to thank Dr. ir. Marinus Maris and Prof. Dr. Jan van Eijck for their comments, suggestions and patience during the supervision of my graduation project. For reviewing earlier versions of this thesis I would like to thank Dr. ir. Albert J.N. van Breemen, Sebastiaan Herman, Dr. Josep M. Porta, Daan van der Rol and Frans M. Terwijn. Furthermore, I would like to thank all the members of the IAS group that participated in the interviews described in this work.

Table of contents

1 Introduction 6

1.1 Overview 6

2 Middleware 7

2.1 Middleware Paradigms 7

2.2 Architectural styles 7

2.3 Architectural styles relevant to middleware 8

2.3.1 Pipes and Filters style 8

2.3.2 Message Passing style 9

2.3.3 Request-Reply style 9

2.3.4 Asynchronous Function Invocation style 10

2.3.5 Event Based, Publish/Subscribe style 10

2.3.6 Repository, Shared Data Space style 11

2.4 Communication properties 12

2.5 Styles mapped to communication properties 13

2.6 Different Approaches to Middleware Selection 13

2.6.1 Early key technology selection 13

2.6.2 Hierarchy of Middleware induced architectural styles 14

3 Intelligent Autonomous Systems group 15

3.1 Distributed systems of the IAS group 15

3.2 Distributed systems mapped to architectural styles 16

4 Approach to Middleware Selection 17

4.1 Strategy for middleware selection 17

5 Selection of Middleware 18

5.1 Interviews 18

5.1.1 Functional Requirements 18

5.1.2 Non-Functional Requirements 19

5.2 Evaluation of Architectural Styles 20

5.2.1 Effects on non-functional requirements 20

5.2.2 Overview of effects 23

5.2.3 Effects on Functional Requirements 23

5.2.4 Architectural styles in autonomous robot systems 23

5.2.5 Architectural styles in distributed sensor systems 24

5.3 Selection of architectural styles 24

5.4 Selection of Middleware product candidates 25

5.5 Selection of Middleware Product 26

5.5.1 Interoperability with other Middleware 27

5.5.2 Support for the Event Based style 28

5.6 Realize Middleware Enhancements 29

5.6.1 Development Time 29

5.6.2 Testability 31

5.6.3 Logger 31

5.6.4 Visualization 31
5.6.5 Automated tests 31
6 Results 32
6.1 Functional Requirements 32
6.2 Non-Functional Requirements 32
6.2.1 Development time 32
6.2.2 Interoperability 32
6.2.3 Modifiability 32
6.2.4 Performance 33
6.2.5 Testability 33
6.3 Conclusion 33
7 Evaluation 35
8 References 36
Appendix A Prioritized utility tree 40

1 Introduction

The subject of my graduation project is the selection of a suitable middleware product for the Intelligent Autonomous System (IAS) group which is part of the Faculty of Science of the University of Amsterdam. The IAS group has used various middleware products in various projects in the past. By selecting a standard middleware product the IAS group hopes to reduce the overhead of using different products for different projects. Additionally the IAS group is interested in using middleware as a means of integrating software written in different languages and running on different operating systems.

1.1 Overview

This document will start with a description of middleware in section 2. Here the differences between different middleware approaches are described. Additionally, two approaches for selecting a suitable middleware product are described. Section 3 focuses on the IAS group and the different distributed systems it has built and is building. Section 4 describes the approach to middleware selection used in this work. Section 0 describes the execution of this approach which results in the selection of a specific middleware product. Section 6 evaluates the selected product while section 7 will evaluate the selection approach used in this work.

2 Middleware

Middleware is software that is used in the context of distributed systems. A distributed system is composed of different applications that are running on possibly different heterogeneous computers and that form one system by communicating to each other. Middleware can be seen as the glue between separate applications as it is used to realize the communication between the separate applications. The term ‘Middleware’ comes from the fact that it abstracts from low-level platform-specific socket programming and thereby is positioning itself between the high level ‘Software’ layers, and the low level ‘Hardware’ layers as identified in the OSI reference model. As a definition for middleware we will use:

Middleware extends the platform with a framework comprising components, services, and tools for the development of distributed applications. It aims at the integration, effective development, and flexible extensibility of the business application [11].

The main goal of middleware is to enable the communication between the different distributed applications, but middleware is about more than just communication alone as follows from the definition.

2.1 Middleware Paradigms

Distributed systems are deployed in a wide variety of problem domains such as banking, telecommunication, aircraft control systems and online gaming. This diversity in distributed systems is reflected by the diversity of middleware that is used to build these systems. One of the main aspects of this diversity is the particular paradigm a specific product is based on. A paradigm can be defined as:

A set of assumptions, concepts, values, and practices that constitutes a way of viewing reality for the community that shares them, especially in an intellectual discipline [34].

A middleware product addresses the concerns for distributed applications from the perspective of the paradigm it adopts. Although a specific system can be implemented using most paradigms, the paradigm that best suits the problem at hand will generally result in the lowest complexity of that system in terms of its implementation and reasoning about the system.

2.2 Architectural styles

Closely related to the various paradigms are the architectural styles which embody structural and behavioral constraints [15] on the systems built with a specific middleware product. Each middleware product induces a certain architectural style on the system it helps to build. As a definition of a middleware induced architectural style we will use a slightly modified definition of [18]:

The behavioral and structural properties of, and constraints on components, connectors, and their interaction, that must be compliant with the corresponding middleware.

Architectural styles are not clear concepts but principled ways to structure a system, therefore only the main characteristics of the styles are considered in this work.

We are still far from having a well-accepted taxonomy of such architectural paradigms, let alone a fully-developed theory of software architecture. But we can now clearly identify a number of architectural patterns, or styles, that currently form the basic repertoire of a software architect [9].

Naming and classification of styles differ in literature, in this work I use the classifications and descriptions based on those in [9]. Styles can be used in combination where generally one style will be the dominant style and have the most influence on the non-functional characteristics of a system. In principle any architectural style can be used to implement the functionality of any system but each style results in a certain tradeoff of non-functional characteristics such as the performance, scalability and modifiability of the system. This influence on non-functional requirements is the reason that the architectural styles will play a central role in the selection of middleware in this work.

In [18] architecture description languages (ADL) are studied to be used for explicitly defining instances of middleware induced styles. In most cases the architectural style of a middleware product is not explicitly defined and hard to extract from the documentation. Therefore it will often be identified only after using the middleware to build an application possibly resulting in an architectural mismatch [10] with the architecture of the application.

2.3 Architectural styles relevant to middleware

In this work, I will only focus on the architectural styles that affect how components of a distributed system communicate. These styles are described below followed by a sequence diagram that shows typical interaction as a result from using the particular style. In a sequence diagram time is represented by the vertical lines below the names of components. A rectangle on these lines indicates activity of the component and an arrow between the lines indicates interaction initiated by the component at the base of the arrow to the component and the end of the arrow.

2.3.1 Pipes and Filters style

The pipes and filters style organizes a system as consisting of components with input and output ports. The ports are not typed so that any component can be connected to any other component and the data communicated between the ports is represented as a stream. A component has no knowledge to which components it is coupled and processes the data that is pushed to its input port and pushes the result to its output port. A component can also receive data that it does not operate on and which it would simply push unchanged to its output port. Data is often processed incrementally meaning that output can start before all the input data is received. A component can have more than one component connected to its input and output port. Pipes and filters systems are generally modeled as a series of components that realize one flow of data with no or few branches.

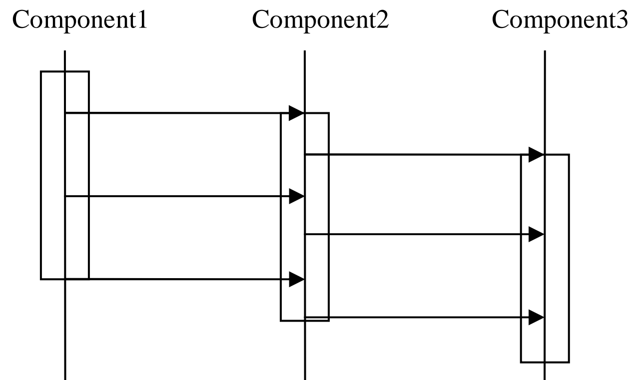

Figure 2-1, Sequence diagram that shows typical interaction for the pipes and filters style.

2.3.2 Message Passing style

The message passing style organizes the system as components that communicate by sending messages. The identity of the component sensing the message is added to all messages so that the receiving components can reply to a message when required. However, there is no explicit relation between the message requesting information and the reply to that message. The component initiating communication is often referred to as client and is requesting service, and the component that is contacted to provide this service is referred to as the server. When satisfying a request, a server often takes on the role of a client when it requests services of other components in order to satisfy the request. With the message passing style, a client component does not suspend execution while waiting for a reply message as messages are received asynchronously. A client component needs to know the identity of components with which it communicates resulting in a tight coupling.

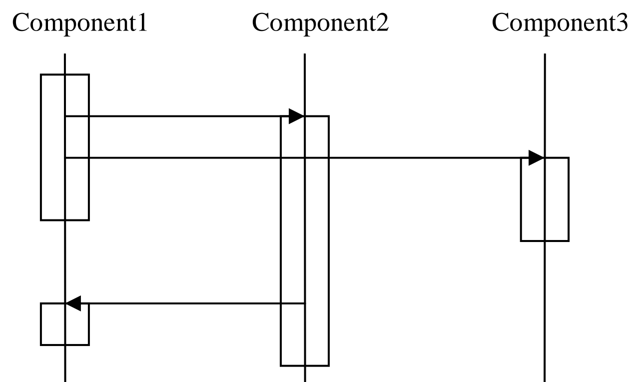

Figure 2-2, Sequence diagram that shows typical interaction for the message passing style.

2.3.3 Request-Reply style

The request-reply style organizes a system as components that have specific typed interfaces. Components interact by calling a specific function of the interface of another component that is to satisfy the request of the calling component and return a possible result. Therefore, this style requires components to know about the identity of the components it interacts with, resulting in a tight coupling. A client that has called an interface

function generally suspends execution until the server has finished the request. This style most resembles the structure of a single program with a procedural or object-oriented decomposition.

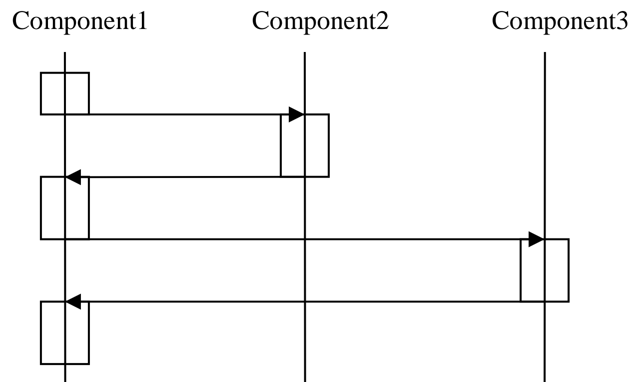

Figure 2-3, Sequence diagram that shows typical interaction for the message passing style.

2.3.4 Asynchronous Function Invocation style

The asynchronous invocation style [7] organizes the system, similar to the request-reply style, as components that have specific typed interfaces. The difference with the request-reply style is that the interface functions do not return any result, thus the calling component does not wait for the result but continues execution immediately after calling an interface function similar to the message passing style. However, it is possible for a component to receive a result by supplying a specific callback function as a parameter. The callback function can then by called by the component receiving the request to return the result. Similar to the message passing style, a calling component will not be waiting for the result but receives it asynchronously as a new function invocation.

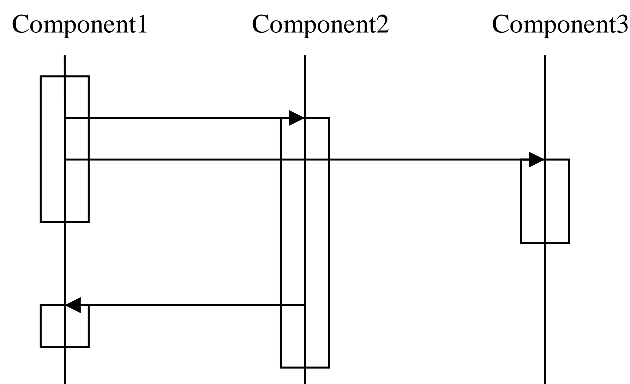

Figure 2-4, Sequence diagram that shows typical interaction for the asynchronous function invocation style.

2.3.5 Event Based, Publish/Subscribe style

The event based style organizes a system as components that publish data and subscribe to data with a particular topic. Data can be strongly typed or alternatively have a free format with a mandatory topic. A shared resource is responsible for managing the subscriptions and distributing the data to components that have indicated an interest

in the data of a particular topic. This resource can itself be distributed over several machines to avoid a potential bottleneck. This style results in a lose coupling as a component does not know how many, if any, components are subscribed to the data it publishes and what these components do with the data. Similarly a component does not know which components publish the data that it is subscribed to. Many publishers can publish data of the same topic which can be received by many subscribers.

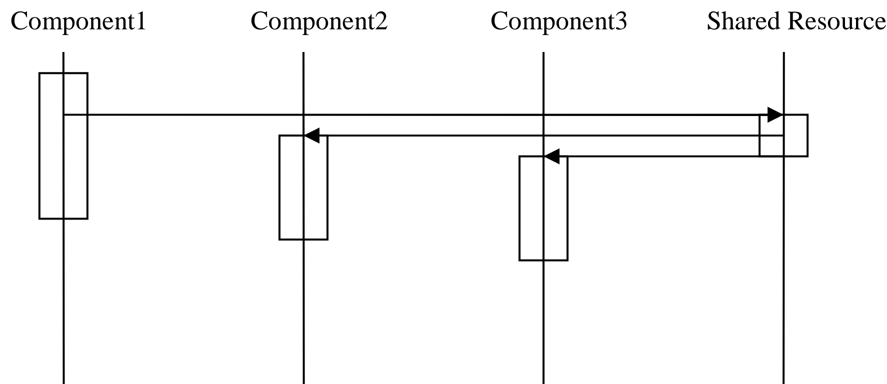

Figure 2-5, Sequence diagram that shows typical interaction for the event based style.

2.3.6 Repository, Shared Data Space style

The repository style organizes a system as components that implicitly communicate through reading and writing in a shared resource called the repository. The repository can be seen as realizing a shared data space between the components. The repository can be realized in many different forms, from a relational database to a free format data structure. The difference with the event based style is that the repository is not responsible for distributing the data it receives to the components interested in the data. Instead the components have themselves the initiative in accessing data in the repository.

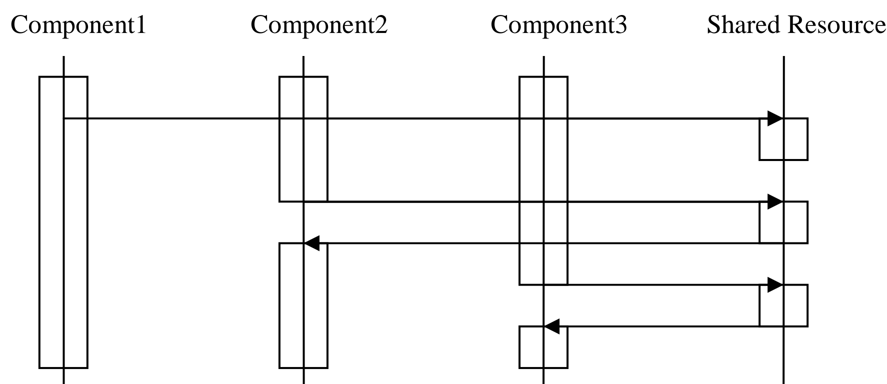

Figure 2-6, Sequence diagram that shows typical interaction for the repository style.

2.4 Communication properties

To avoid any confusion we will classify the architectural styles along the following identified communication properties. Note that not all combinations of properties yield a valid or practical style.

Two-way or One-way

Components can communicate by sending data to another component and then block execution while the other component receives and processes the information until it receives a possibly empty reply in return. This model on communication is referred to as two-way communication. Alternatively components can communicate by sending data to another component and immediately resuming execution without waiting for the data to be received and processed. This model is referred to as one-way communication.

Push or Pull

Information can be pushed on the initiative of the source of the information to the receiver. This is referred to as a push model. Alternatively, the receiver could take the initiative and request the information which is referred to as a pull model.

Single request or Subscription

When information is requested it can be either a singe request or it can be a subscription where the receiver indicates it wants to receive more instances of the information in the future. A subscription can supply information every time new information is available, or periodically, or when the information meets certain criteria. In the case of a subscription we speak of a push model, even though the information was initially requested by the subscriber, because after the initial request the publisher is the initiator of the communication. Such a communication model is known as a Publish/Subscribe model.

Interface or Message based Communication

Communication can be organized by having a component expose its interfaces to the other components. Components communicate by calling a specific interface function similar to the communication within a single application. In contrast, a component can also communicate by sending different messages to a single input port of other components which have to interpret the messages. A message, in contrast to a call to an interface function, can be an entity in itself and can therefore be more easily stored, prioritized and intercepted.

Predefined or Free Format

Information can have a predefined format (type and size) for example when calling an interface function with a fixed number of parameters with a fixed type. Alternatively, information can have a free format which leaves it open for interpretation by the receiver. Examples of this type include information represented as a sequence of bytes, as plain text or tagged text such as HTML and XML, or as SQL queries for database access. A message can either have a more free format or be more strongly typed.

Direct Communication or via a Shared Resource

With direct communication the sender sends information directly to a receiver. Alternatively information can be send to a shared resource from which all interested parties can receive it. This frees the sender from knowing the receiver.

2.5 Styles mapped to communication properties

The following tables map the architectural styles to the identified communication properties. As architectural styles are not clear concepts, this mapping reflects the way I view the different styles. Such a mapping can be a subject of long discussion and other mappings might also be valid. However, in this work this mapping will serve as the definition of the different styles.

	Two-way/ One-way	**Push/ Pull**	**Single request/ Subscription**
Pipes and filters	One-way	Push	N/A
Message passing	One-way	-	-
Request-Reply	Two-way	-	Single request
Asynchronous function invocation	One-way	-	-
Event based	One-way	Push	Subscription
Repository	One-way	Push	Single request

Table 2-1, The first part of the mapping from architectural styles to communication properties. The symbol '-' indicates that each alternative is possible and the symbol 'N/A' indicates that neither alternative applies.

	Interfaces/ Message based	**Predefined/ Free format**	**Direct Communication/ via Shared Resource**
Pipes and filters	Message based	Free format	Direct Communication
Message passing	Message based	-	Direct Communication
Request-Reply	Interfaces	Predefined	Direct Communication
Asynchronous function invocation	Interfaces	Predefined	Direct Communication
Event based	-	-	Via Shared Resource
Repository	-	-	Via Shared Resource

Table 2-2, The second part of the mapping from architectural styles to communication properties. The symbol '-' indicates that each alternative is possible.

2.6 Different Approaches to Middleware Selection

In literature, different approaches to middleware selection can be found. Here two approaches are described on which the approach in this work is based.

2.6.1 Early key technology selection

In [11] the selection process aims to provide a manager with an objective foundation for making a decision in the technical field of middleware which is unwieldy, complex and entangled. This is done in the following steps.

1) Create taxonomy

Where stakeholders are interviewed to identify the requirements of the applications to be constructed. These requirements are seen as non-functional requirements as discussed in [1] and are described in the vocabulary of the stakeholders and are organization specific.

2) Identification of useful technologies

Different middleware technologies are classified based on the available taxonomy in discussion with (technical) stakeholders. This results in a partially subjective classification which takes into account current views on middleware technologies within the company.

3) Selection of base technology

After prioritizing the requirements in the taxonomy and building change scenarios as described in [1], a middleware product is selected as the base technology.

4) Freedom to choose different middleware

To select middleware for a specific project the steps above are repeated. In a project developers are free to choose a different middleware product when such a choice better meets the requirements as long as a bridge is implemented for compatibility with the base technology.

2.6.2 Hierarchy of Middleware induced architectural styles

The selection process in [15] aims to select middleware based on the architecture of the particular application to be developed. It proposes the following steps:

1) Build a hierarchy of middleware styles

Describe the middleware induced architectural styles of middleware products using ADL and build a hierarchy with these styles. In this hierarchy the more general styles are at the top and branch down to more and more specific styles.

2) Map an architecture to the style hierarchy

Given the architecture of an application, the matching part of the hierarchy should be identified. The hierarchy now provides insight in the effects of additional architectural decisions on the possible choices of middleware products as further decision can reduce the matching part of the hierarchy.

3 Intelligent Autonomous Systems group

The IAS group, for which I aim to select appropriate middleware, studies methodologies to create intelligent autonomous systems, which perceive their environment through sensors and use that information to generate intelligent, goal-directed behavior. This work includes formalization, generalization and learning of goal-directed behavior in autonomous systems [22]. Distributed systems build by the IAS group are characterized by their real-time behavior and by dealing with uncertain information as a result of noisy sensors or partial observability of their environment. These systems contain experimental algorithms and their use is limited to members of the IAS group, partners in various projects and other researchers. The aim of the IAS group is not to develop software but to do research, which in most cases requires software to be build in the form of prototypes to serve as experimentation tools, verification of theory or proof of principle.

3.1 Distributed systems of the IAS group

The IAS group has developed different distributed system in the past and currently is developing a distributed system. The more recent examples of these systems are described shortly below.

Robocup Middlesize

Within the past years the IAS Group of the University of Amsterdam has, together with the Pattern Recognition Group of the Technical University of Delft, competed in the middle-size league of the international Robocup competition [23] where teams of autonomous robots compete in a soccer tournament. The communication between the robots and the different processes running on the robot is realized by three distinct middleware systems. Two of them, called MSG and Shared, are specifically developed for this project, and the other is a commercial middleware product called Splice developed by Thales [24].

Robocup Simulation League

Separate from the Robocup Middlesize competition the IAS Group is also active in the Robocup simulation league with the UvA Trilearn team [26]. In this soccer competition, the robots are simulated and the focus is on team strategy. Each simulated robot is controlled by a separate process which communicates with a server that manages a game. The communication is realized by low-level socket programming.

Predator-Prey

For educational purposes, the IAS group has developed a predator-prey system [25] which is used by students to do experiment with multi-agent systems. The goal is to develop cooperating predators that capture a prey as fast as possible in a discrete grid-like world. Similar to the Robocup simulation league, each predator is controlled by a separate process which communicates using low-level socket programming.

Ambience

The IAS Group participated in the Ambience project [27] together with Philips, Epictoid and the University of Leuven. In this project an indoor service robot was developed which is capable of simple conversations, showing emotions, estimating position and navigating rooms. The communication between processes running on both the Windows and Linux operating systems was implemented using middleware developed by Philips called DML.

Cogniron

Within the Cogniron project [28] the IAS group joined the University of Bielefeld in developing a service robot that is aware of its environment by combining different models of the environment. The communication between the different processes running on several computers is implemented using the XCF middleware developed by the University of Bielefeld [29].

Combined systems

The Combined systems project [30] aims to develop a collaborative intelligent system that fuses the information of heterogeneous distributed sensors and humans and makes decision based on this information. One of the subprojects where the IAS group is involved in is the Distributed Perception Networks (DPN) project [31] where the middleware product Cougaar [32] is used.

Distributed Multi-camera Surveillance

In the Distributed Multi-camera Surveillance project [33], the IAS groups aims to develop techniques for tracking multiple objects with multiple cameras. The focus is on problems where the cameras are sparsely distributed over a wide area and thus have no overlapping field of view.

3.2 Distributed systems mapped to architectural styles

Table 2-1 maps the middleware products used in the various projects to architectural styles. Note that where products use a combination of styles, the first style listed is the most dominant style. The Distributed Multi-camera Surveillance is not listed as at this time the prototype is a monolithic system build without any middleware product.

System	Middleware	Architectural styles
Robocup middle-size	MSG	Message passing
	Shared	Repository
	Splice	Event based
Robocup simulation	Socket programming	Message passing
Predator-Prey	Socket programming	Message passing
Ambience	DML	Request-reply, Event based
Cogniron	XCF	Request-reply, Event based
DPN	Cougaar	Request-reply, Event based

Table 3-1, Mapping from middleware product to architectural styles.

4 Approach to Middleware Selection

This section describes the approach that is followed in the selection of a suitable middleware product for the IAS group.

4.1 Strategy for middleware selection

As the main selection strategy for this project I followed the approach of 'Early key technology selection' [11] described in section 2.6.1 as this seems to provide the clearest way to come up with useful requirements for a specific organization. For the requirements I make a distinction between the functional requirements that need to be fulfilled with the help of the middleware product, and the non-functional requirements, for which the middleware product should realize an optimal tradeoff.

However, the approach is not clear in how to select a middleware product given a set of requirements. Therefore I extend this approach to include a selection based on architectural styles. However, I will not go as far as to describe the styles using an ADL and build a hierarchy of styles as proposed by the 'Hierarchy of middleware induced architectural styles' [15] described in section 2.6.2 as it will not be strictly necessary to build a hierarchy. Instead I use a similar approach as in chapter 3 of [3] where high level architectural styles are mapped to non-functional requirements for a specific type of application.

Given the requirements and the architectural styles that should be supported, various candidate middleware products can be identified. From these candidate products I select a single product that best meets the requirements. The last step in the selection approach is to identify and realize enhancements to the middleware product that makes for a better fulfillment of the requirements.

By dividing the selection process into first selecting architectural styles and then selecting specific products, I hope to significantly reduce the complexity of the selection process. The selection approach is visualized in Figure 4-1.

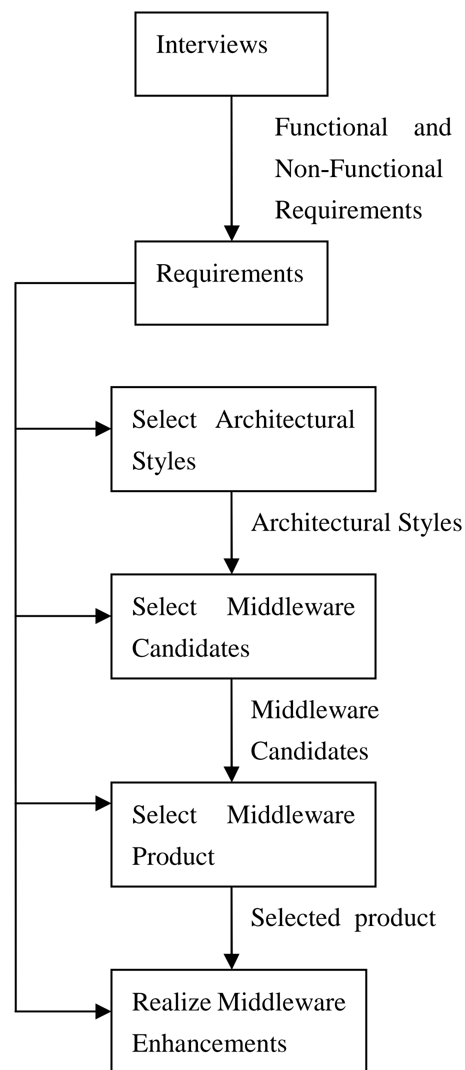

Figure 4-1, Visualization of the middleware selection process.

5 Selection of Middleware

This section describes the execution of the chosen approach to middleware selection. First it describes the outcome of the interviews. Then the different architectural styles are evaluated, after which two styles are selected for use by the IAS group. Given the selected styles and other requirements, different middleware products are evaluated after which a specific product is selected. The section concludes by describing enhancements that make the product better fulfill the requirements.

5.1 Interviews

In order to come up with requirements, interviews are held with seven people from the IAS group with experience in distributed systems. These interviews are aimed to produce both functional and non-functional requirements. The functional requirements are extracted in an open interview by having the person being interviewed describe the different systems that he or she has worked on, describe the reasons for building the systems and point to any available descriptions. The non-functional requirements are extracted in an open interview where past experience with distributed systems is discussed followed by a structured interview where the person being interviewed is asked to come up with concrete scenarios for a set of non-functional requirements that he or she indicated to be important. These non-functional requirements are represented in a utility tree which elaborates high level non-functional requirements to concrete scenarios as described in [1]. After all interviews were completed the participants were asked to rate the various scenarios of the utility tree in order to prioritize them. The resulting prioritized utility tree can be found in Appendix A. Both the functional and non-functional requirements could not be described in detail but remain general as a result of eliciting requirements for a number of future projects in contrast to a single concrete project.

5.1.1 Functional Requirements

Systems developed by the IAS group generally are real-time and operate on current data using relatively complex algorithms that deal with uncertainty. The systems only have limited user interaction and have a small user group which is generally restricted to persons in the IAS group. The lifetime of the system is limited although parts of the systems are reused in other systems. The functionality of the systems described in the interviews can be mapped to two different types of systems; autonomous robot systems and distributed sensor systems.

Autonomous robot systems are characterized as real-time systems where a robot continuously perceives its unstructured and dynamic environment through sensors, builds a belief about its environment, plans actions to reach its goal and execute these actions using its actuators. The autonomous robot systems developed by the IAS group, focus primarily on indoor exploration, localization and navigation and multi-agent coordination.

As stated in [8], three types of architectures are common in this type of system; reactive or behavioral architectures [5], centralized architectures [20], and hierarchical architectures [4]. Often these architectures are used in combination as in [12] [13] [8].

Distributed sensor systems on the other hand are characterized by fusing data of many distributed sensors [16]. The computations in these systems are decentralized for performance and robustness reasons. The network of sensors often automatically configures itself based on a request for specific information and should be robust in the event of failures of sensors [14]. Distributed sensor systems developed by the IAS group focus primarily on person-tracking and situation awareness in large environments.

Both types of system generally have a relatively tight coupling of at least part of the system where

components know the identity of other components. This allows a component that processes sensor data, to have control over when sensor components produce sensor data. This allows for making sensor observations at specific moments and allows for simultaneous observations of several sensors.

5.1.2 Non-Functional Requirements

During the interviews regarding the non-functional requirements it became clear that the IAS group only builds prototype systems so that non-functional requirements such as security and usability are of little importance and were soon discarded. Non-functional requirements that were considered important are listed below and are typical for prototype systems.

Development time
The amount of time and effort required to build a system. The ease at which components can be reused in other systems. The extend in which support on the middleware product is available in the form of a helpdesk, support forum or active user community.

Interoperability
The extend in which the system can cooperate with components written in different languages, namely C, C++ and Java, and running on different operating systems namely Linux, Unix and Windows. The extend in which the system can cooperate with other systems that use different middleware products and with components running on a host in a different network with only a local IP address or behind a firewall.

Modifiability
The extend in which changes to components do not require other components to be changed. The extend in which a system can be changed at runtime and continues to function when components are temporarily removed or replaced.

Performance
The extend in which a system responds in a timely fashion.

Testability
The extend in which error messages are communicated to the component causing the error. The ease of constructing test software. The extent in which component interactions of the system as a whole can be visualized and logged. The extent in which errors are reproducible by partially fixing the order of interactions between components.

Besides the requirements for the projects, the interviews also resulted in requirements for the middleware product itself. The middleware product should be generally applicable for building distributed systems in contrast to a framework for specific types of systems. Additionally the middleware should be open source so that it can be adapted to work around problems that might arise.

5.2 Evaluation of Architectural Styles

In order to evaluate the architectural styles, I investigated their effect on the requirements. First the effects on the non-functional requirements studied. Then I survey different architectures for autonomous robot systems and distributed sensor systems to see which styles are common for systems with such functional requirements.

The architectural styles do not affect the non-functional requirement of interoperability as an architectural style does not affect the choice of programming language or platform. Nor does it affect the ability to communicate with components in another network. However, architectural styles do affect the interoperability of a middleware product with other middleware products but this relation depends on the details of the implementation which will be considered when selecting a specific middleware product.

5.2.1 Effects on non-functional requirements

First I studied the effects that can be related to the identified communication properties. These effects are listed in Table 5-1 which shows the differences between using different alternatives for each communication property. The effects are ordered by the associated non-functional requirement and are preceded by an abbreviation for the alternative causing the effect followed by a '+ 'or '-', respectively indicating a positive or negative effect.

Styles/ non-functional requirements	**Effects**
Two-way vs. One-way	
Development time	[Two, -] With Two-way communication a component waits before executing the next statement until the result of a request is received. Therefore a single request that results in a cascade of more requests to other components will make those components run sequentially, not in parallel. Extra code that implements multiple threads, is required to make components run in parallel. [One, -] With One-way communication the source code is harder to understand because a sequence of operations that acts on a single data element is distributed throughout the source code. Two-way instead localizes a sequence of operations in a single function as is the case with common procedural and object oriented programs [10].
Modifiability	[One, -] Because One-way communication makes the source code harder to understand it also makes the source code harder to modify.
Performance	[Two, -] Because Two-way communication results in components running sequentially, it has a negative effect on the performance [17].
Testability	[Two, +] Because a component waits to receive the result of a request, it is also able to receive possible error messages. This allows error messages to be returned to the part of the code that issued the request which makes it easier to localize errors in the code. [Two, +] A single invocation of a function using Two-way communication, results in fix sequence of potential subsequent invocations of functions. This fixed sequence avoids race conditions and helps error reproduction. However, multiple invocations only have a partially fixed order.

Push vs. Pull	
Development time	[Push, -] Using a push model a component has to wait until data is sent to it. Extra code is required to allow a component to request the data at any specific time.
Modifiability	
Performance	[Push, +] Because data is pushed without a request, no processor or network resources are needed to issue the request.
Testability	No effects identified
Single request vs. Subscription	
Development time	[Single, +] Similar to a pull model, a single request allows a component to have control over when it receives data. In contrast, subscriptions data normally is received periodically and extra code is required to request data at any specific time.
Modifiability	[Single, +] When at runtime a component is replaced by another component using the same name, a single request can automatically connect to the new component. In contrast, a subscription established with the old component, will not be automatically transferred to the replacement component.
Performance	[Single, -] Data needs first to be requested each time it is needed. This requires processor and network resources. [Single, -] A component using single requests to receive data, is not automatically notified when new data is available, instead polling is required to check for new data. [Subsc, +] When using a subscription, the load of a component that publishes data is independent of the number of components that are subscribed to the data. In contrast, when using single requests, each request of each component needs to be handled by the component that produces the data.
Testability	No effects identified
Interface vs. Message based	
Development time	[Message, -] A component that receives messages needs code to first identify the different messages and secondly to redirect the messages to the function that processes them. When using interfaces this is not required as the parameters of a function invocation are automatically sent to the function that processes them.
Modifiability	No effects identified
Performance	No effects identified
Testability	[Interface, +] Interfaces allow for compile-time type checking of all the arguments.
Predefined vs. Free format	
Development time	[Free,-] The Free format requires extra code for putting together and parsing of information.

Modifiability	[Free, +] With a robust parser small changes to the format of the data does not require any additional changes in the code. In contrast, changes to the format of predefined data will require changes to the code that interprets this data.
Performance	[Free, -] Putting together and parsing of information, required by free format data, results in decreased performance.
Testability	No effects identified
Direct communication vs. shared resource	
Development time	[Direct, -] As a component requires knowledge of other components to communicate directly, the system is tightly coupled which make reuse more difficult.
Modifiability	[Direct, -] The tight coupling often requires other components to change as a result of changes to a single component. [Direct, -] Due to the direct communication, removing a component at runtime affects other components which might block until the component reappears or times out and stops. [Direct, -] Components that communicate directly require to be active at the same time, in contrast the shared resource allows for time decoupling [7].
Performance	[Direct, +] Direct communication avoids the overhead of communication via a shared resource.
Testability	[Shared, +] All communication via the shared resource can be automatically logged and visualized allowing for better understanding of the system as a whole. [Shared, +] As all communication goes via the shared resource, it can define an order in the interaction which is useful for error reproduction. It is harder to define and reproduce an order in the interactions of several component that communicate directly in parallel.

Table 5-1, Effects of the architectural styles related to the communication properties on the non-functional requirements.

Besides the effects of the styles on the non-functional requirements that can be related to communication properties, there are additional affects caused by each style as a whole. These effects are discussed below.

The pipes and filters style models the system as a flow of data where the data is represented as a stream. Modeling the autonomous robot and distributed sensor systems in such a way requires extra code for creating and parsing the streams. This has a negative effect on the development time. However, modeling these systems in such a way results in high modifiability as components of these systems can arbitrarily be connected to any other components at runtime.

The message passing and asynchronous function invocation styles are identified as providing a subscription mechanism. But although the styles can support subscriptions, the subscriptions need to be implemented for each component that provides it. This implementation comes down to code for keeping track of which components are subscribes to what data and sending data to the subscribed components when required. Although part of the implementation can be shared over all components, this still results in a negative effect on the development time.

The event based style can provide a mechanism to give different priorities to different kind of data elements

which has a positive effect on the performance. Additionally, the event based style often allows filter criteria to be attached to a subscription. This filter mechanism only allows data that passes the filter criteria to be sent to the subscribed component. This reduces the network load and thus can increase the performance of the system.

Using the repository style has a negative impact in the performance as the shared resource can not be as efficiently distributed as with the event based style. The reason for this is that it is not known which component requires what data. Therefore, the shared resource can become a bottleneck which has a negative effect on the performance.

5.2.2 Overview of effects

Table 5-2 gives an overview of the relative effects of the different architectural styles on the non-functional requirements. Please note that this table gives an oversimplified representation that is not valid for systems in general but only applies to systems with similar characteristics to autonomous robot and distributed sensor systems. The table is the result of adding and canceling out the effects identified in the previous paragraphs. This simple approach is taken as the relations between the different effects are to complex for alternative approaches. Assigning weights to the effects is unpractical as a specific effect can be reduced at the cost of increasing other effects by an unknown factor. Therefore all the effects are considered as roughly having the same weight.

	Development time	**Modifiability**	**Performance**	**Testability**
Pipes and filters	--	+/-	+	--
Message passing	--	-	+	--
Request-reply	+	+/-	+/-	+
Asynchronous function invocation	+	--	++	-
Event based	-	-	++	+/-
Repository	+	+	-	+/-

Table 5-2, Simplistic overview of all the effects of the architectural styles on the non-functional requirements

5.2.3 Effects on Functional Requirements

Although each style can in principal be used to realize systems with any functional requirements, I investigate what styles are more commonly used in architectures for both the autonomous robot and distributed sensor systems.

5.2.4 Architectural styles in autonomous robot systems

One such architecture for autonomous robot system, described in [12], uses the asynchronous message passing style (here called ‘bidirectional pipes and filters style) in combination with the repository style (here called ‘blackboard’ style). Here the choice for the asynchronous message passing style is explained by its concurrent execution property which it shares with the asynchronous function invocation style. This paper states that it uses a repository style, however, as the initiative for communication lies completely at the shared resource, the architecture should actually be seen as using an event based style but this is not completely clear from the paper.

The architecture described in [8] uses a request-reply style for the activation and deactivation of a component (or service) which during execution writes data to a ‘poster’. A poster is defined as structured shared memory readable by any component but only writable by its owner and seems similar to a repository style. The

posters allow for efficient communication between components running at different frequencies or different temporal constraints [8]. When instead an event based style was used, a component with a low frequency could be overloaded by the poster of a component running at a high frequency. However, as there is no single poster but instead each component has it own poster, the poster of a specific component must be addressed. Therefore the architecture can be regarded as having a request-reply style.

The architecture described in [13] uses a combination of the request-reply and event based style. In general activation of components is done by a central supervisor component using the request-reply style and any feedback is returned using an event based style. This allows other components to efficiently tap into the feedback without increasing the computational load of the component sending the feedback.

5.2.5 Architectural styles in distributed sensor systems

As for the autonomous robot systems, I also studied different architectures for the development of distributed sensor systems where the focus is on the adopted architectural styles.

For the realization of the DPN [16] and ASN [14] distributed sensor systems the request-reply style is used exclusively. However, the event based style can be used to potentially reduce the complexity and development time of these systems in relation to using the request-reply style. This reduction is caused by having reasoning components simply subscribe to data that they are interested in without needing to first find sensor components that can provide the data. However, there are several reasons for not using the event based style exclusively.

One is that in the ASN project reasoning components do not only process sensor data but also actively control the sensors. In order to control a specific sensor component, the reasoning component requires some kind of reference to that sensor component which fits better with the request-reply style than the event based style where components are unaware of the existence of other components.

A second reason is that in the DPN project a sensor component can activate otherwise possibly dormant reasoning components when it detects interesting sensor measurements. Such activation is not possible when reasoning components would exclusively initiate communication with sensor components by subscribing to sensor data. To solve this problem a complex event based architecture is required where both reasoning components and sensor components can initiate communication. Such communication is more easily implemented by the request-reply style.

A third reason is that sensor components that use the event based style, are required to produce sensor measurements periodically even though the measurements might only be sporadically needed. When there is a cost associated with the use of the sensor, this approach can be inefficient. When instead the request-reply style would be used, the sensor would only produce sensor measurements when required by a reasoning component.

A fourth reason is that distributed senor systems can potentially have many components which might result in a bottleneck at the shared resource. This would require the shared resource to be efficiently distributed over many nodes. Not many middleware products allow for such an efficient distribution. One exception is Splice by Thales that efficiently distributes the shared resource over all nodes in the system.

5.3 Selection of architectural styles

Based on the evaluation of the different styles I select the combination of the request-reply style with the event based style as best solution to fulfill the requirements of the IAS group. The request-reply style allows for a low development time, good testability properties and a high level of control over other components. The low performance, due to the limited concurrency properties of this style, can be resolved by using multiple threads. On the other side the event based style provides high-performance one-to-may push communication and a low

coupling between components.

For example, when accessing a sensor component, the request-reply style can be used to request a senor measurement at a specific moment in time. However, when there are many components and those components are only interested in a measurement when there is a significant change in relation to the previous measurement, then the use of the event based style is much more efficient.

When the event based style uses interfaces instead of messages, it can be tightly integrated with the request-reply style which is interface based. This tight integration of the two styles would provide designers the flexibility to choose between these two styles for individual lines of communication and will reduce the development time.

The pipes and filters style is discarded even though it has relatively good modifiability properties. However, it seems to be more suitable for systems that can easily be modeled as a single flow of data and where different sequences of components can be required. Examples are systems for telecommunication that have a variety of optional components for data compression, decryption and noise reduction. The message passing style is discarded based on the long development time it induces and on the fact that its concurrency properties are matched by that of the event based style. The asynchronous function invocation combines properties of both the request-reply style and the interface based event based style. Therefore it can be discarded as a similar result can be achieved by using a combination of the two selected styles. The repository style seems to be more suitable for systems that store large amounts of data for a long time and have relatively low performance requirements such as administrative systems.

5.4 Selection of Middleware product candidates

Now that I have chosen the request-reply style in combination with the event based style, I can select candidate middleware products that support these two styles. I only select candidates that fulfill all the requirements, especially the interoperability requirements and requirements for the middleware product itself as described in paragraph 5.1.2. These include requirements for platform and language independence, requirements on the general purpose character of the middleware and the requirement that the middleware is released as open source. Based on these requirements the following three candidates are identified.

The ACE ORB (TAO)

TAO [35] [36] is an implementation of the OMG CORBA specification [37] which started in 1991. The specification was made specifically to develop middleware with high interoperability. TOA is developed by research groups at Washington University, University of California - Irvine and Vanderbilt University. It uses an object-oriented model, is released as open source and is free to use.

Internet Communication Engine (ICE)

ICE [38] can be seen as an adapted implementation of the CORBA standard and is therefore not CORBA compatible. ICE is developed by the commercial company ZeroC but is released as open source under the GNU General Public License for non-commercial projects. It uses an object-oriented model.

SOAP compatible product

Soap [39] is the XML based communication technology specified by the World Wide Web Consortium and is used for the realization of Web Services. The long term goal of Web Services is to enable autonomous service discovery for application [2]. This would enable applications to automatically find and interoperate with other formerly unknown applications in an ad hoc manner in order to fulfill a specific service request. This would not

require human intervention. Many different SOAP compatible products are available for various platforms and languages [40] including the Microsoft '.Net' Framework.

5.5 Selection of Middleware Product

Making a selection over the identified products is difficult as a result of the complexity of the products. To make a selection I again made a mapping to the non-functional requirements that are identified. This mapping in Table 5-3 is mostly based on practical experience with these products and information found in [41], [42] and [43].

As no significant effect on the modifiability could be identified, the modifiability requirement is not included in the table. The modifiability property of a system is mainly determined by the architectural styles used in its implementation.

Product	**Effects**
TAO	
Development time	- The CORBA specification is relatively complex resulting in a relatively long period to get familiar with CORBA products such as TAO. + In the source code remote function calls to other components are similar to function calls within a component. - TAO does not allow for a tight integration between the request-reply style and the event based style. The reason for this is that the event based style uses message based communication instead of communicating via interfaces as with the request reply style [19].
Interoperability	- For Java programs another product called Zen is required which is compatible with TAO. + A CORBA product does in principle interoperate with other CORBA compatible middleware products. In practice different implementations of the CORBA specification require additional work to make them interact due to ambiguities in the CORBA specification.
Performance	+ The performance is high as TAO is specifically designed for real-time systems. It allows priorities to be set to the different types of data, it support filter criteria for subscriptions, and can therefore give guarantees on the time of data delivery [19]. + The shared resource can be distributed to avoid a potential bottleneck.
Testability	+ Faults can be detected by compile time type checking.
ICE	
Development time	+ Remote function calls to other components in the source code are similar to function calls within a component. + ZeroC provides a responsive online helpdesk that is not restricted to paying customers. + Has a relatively simple and clearly documented language binding to C++ and Java.
Interoperability	+ ICE is readily available for Windows, Unix and Linux for the required languages C++ and Java and other languages.

	- ICE is not compatible with other middleware products. To interoperate with another middleware product, a 'bridge' to that specific product must be developed. + Provides a facility to access components behind a firewall or with a local IP address residing in a different network.
Performance	+ The performance is high without any guarantees on the time of data delivery. ICE outperforms TOA in different simple tests [47]. + The shared resource can be distributed to avoid a potential bottleneck.
Testability	+ Faults can be detected by compile time type checking.
SOAP product	
Development time	- Different SOAP products are required for different languages and different platforms. - As the event based style is not an intricate part of the SOAP specification, different SOAP products have different implementations of this style, if any. - A function call requires an XML message to be constructed that holds among others the function name, parameters and the name of the component to be called. As a result a lot is time is required to create source code that builds these messages.
Interoperability	+ The XML representation eases interoperability with other middleware products as XML is increasingly recognized as a standard in all sorts of software systems.
Performance	- The conversion to and from XML results in a performance penalty.
Testability	- SOAP products do not allow for compile time type checking. Type checking is only done after calling a function at runtime using XML schemas.

Table 5-3, Effects of the different middleware products on the non-functional requirements.

Based on this mapping to non-functional requirements, ICE is chosen as the best middleware product to be used by the IAS group. The choice is mainly based on the tight integration it allows for the request-reply and event based style. Additionally, the low complexity of the product and language bindings and the availability of the single product on the required platforms supporting the required languages are important benefits.

The main reasons why TAO is discarded are its high complexity and the lack of support for a tight integration of the request-reply and event based style. The main reasons that SOAP compatible products are discarded are the lack of support for the event based style and the elaborate work required for building and parsing of XML messages.

5.5.1 Interoperability with other Middleware

As the IAS group frequently needs to cooperate with other organizations in projects, the ability to interoperate with other middleware products is an important factor. Both SOAP compatible products and TAO have better interoperability properties then ICE because they are implementations of the two most widely accepted middleware specifications, respectively CORBA and SOAP. However, in practice considerable work is required to realize interoperability between different products of the same specification. This is partially caused by

ambiguities in the specifications and additional features of the products that are not covered by the specification. Therefore the interoperability with other middleware products is not given much weight in this evaluation. Additionally, this is supported by the fact that, until now, the IAS group has not cooperated with other organizations that used a middleware product conforming to either the CORBA or SOAP specification. Instead the products that were used could not automatically interoperate with other middleware at all.

The problem of interoperating with another middleware product therefore needs to be solved on an ad hoc basis, for example by writing an intermediate conversion component that serves as a bridge between two middleware products. In the worst case, this approach requires source code to be developed for the conversion of data for each line of communication that crosses the boundary between two middleware products. In this situation, products that adhere to a widely accepted specification have no advantage over products that do not.

5.5.2 Support for the Event Based style

Because the event based style that is provided by ICE is interface based instead of message based, the two styles can be tightly integrated as will be shown in section 5.6.1. In contrast, the TAO event based style is message based where a component receives all messages in a single fixed function which prevents a tight integration. As the SOAP specification does not include communication using the event based style, SOAP compatible products that do support this style, will have different implementations that are hard to integrate.

The event based style that ICE provides does not allow for time decoupling, meaning that two components that communicate have to be running at the same time. This is because the shared resource only distributes the data to components that are active at the time the data is published. It does not store any data so that a component that subscribes to data will never receive data that is published before the time it subscribed. Although this feature would increase the modifiability of systems, it is not a critical factor and this feature is not supported by TAO either. Because ICE is distributed as open source, it is possible to realize this feature in the future when required. Until now this feature is not used by any of the distributed system in the IAS group.

The shared resource that ICE provides to realize the event based style can be distributed to prevent a bottleneck. However, this distribution is limited to one branching level as described in chapter 41 of the ICE manual [46]. When this one level is insufficient it is possible to use several independent shared resources to divide the load but this would add complexity to the system as components would have to publish and subscribe to a specific resource.

ICE does not provide a means to set priorities to different types of data communicated using the event based style such as TAO does. This might cause important data not to be sent in time because of high numbers of relatively unimportant data that keep the shared resource busy. When this problem arises, again several shared resources could be used but again this would increase the complexity. An alternative solution would be to reduce the amount of unimportant data sent to the shared resource.

ICE does not provide any mechanism to supply a filter for a subscription as is possible with TAO. A filter can indicate that a component is only interested in data that passes the filter criteria. Alternatively, filtering can take place at the component receiving the data but then network resources are wasted. Another alternative is to split the single subscription into several subscriptions that separate the data along some criteria. However, this results in more coarse grain filtering than using filter criteria and is less flexible.

Although some advanced features related to the event based style are not supported by ICE, the event based style is sufficiently supported as those advanced features are not of critical importance for the systems build by the IAS group. I refer to [48] for an example of a product that does support all these advanced features for the event based style.

5.6 Realize Middleware Enhancements

In order to better fulfill the identified requirements, different enhancements on the selected middleware product ICE are realized.

5.6.1 Development Time

To decrease the development time and increase the changes of acceptance within the IAS group, the middleware is enhanced by incorporating it in a simple platform-independent build environment which uses the GNU autotools [44]. This allows developers to start working quickly on a new project without having to first select and set up a build environment. At this time the build environment is only working on the Linux and Unix platform but should be relatively easily made available on the Windows platform using software like Cygwin [45].

Additionally I developed a class called 'IAS_Application' that further reduces development time by hiding away much of the complexity of using ICE at the cost of loosing some flexibility and tightly integrates the two selected styles. Using the build environment and this class allows for rapid development of distributed systems. This is illustrated below by showing the four steps required to build a simple distributed sensor application showing both the request-reply and event based style of communication. This example also serves to show the tight integration between the request-reply and the event based style.

1) Define the interface of the service in Slice.

Slice is the language-independent interface-definition language of ICE and is described in chapter four of the ICE manual [46].

```
interface Sensor
{
        int     readSensor(int i);
        void adjustSensor(int i);
};
```

2) Implement the interface as a class.

Based on the interface, ICE can produce a working but empty implementation in any of the supported languages in the form of a class. For this example I chose a C++ class. The developer should supply the implementation of the empty methods of this class. The implementations in this example are highlighted and simply print the name of the method and the argument. Additionally method 'readSensor' returns an integer that represents a sensor measurement.

```
::Ice::Int SensorI::readSensor(::Ice::Int i, const Ice::Current& current)
{       cout<<"readSensor("<<i<<")"<<endl; return i*10; }

void SensorI::adjustSensor(::Ice::Int i, const Ice::Current& current)
{       cout<<"adjustSensor("<<i<<")"<<endl; }
```

3) Make a server component that presents the class to the outside world.

Here the new SensorServer class inherits from the IAS_Application class and overloads the IAS_run method that is automatically called after initialization. In this method the server creates an instance of the Sensor class and registers this object with the name 'SensorName1' for communication using the request-reply style. Additionally the same object is subscribed to a topic with the name 'SensorName2' in order to use the event based style of communication.

```
class SensorServer : public IAS_Application
{
        public:
        virtual int IAS_run(int argc, char* argv[])
        {
                Ice::ObjectPrx proxy1=registerObject(new SensorI(),"SensorName1");
                Ice::ObjectPrx proxy2=subscribeToTopic(proxy1,"SensorName2");
                waitForShutdown();
                return EXIT_SUCCESS;
        }
};
```

4) Make a client component that gets a reference to the class and interacts with it.

Similar to the server the new SensorClient class inherits from the IAS_Application class and overloads the IAS_run method. There it creates two references to the Sensor object called 'sensorRequestReply' and 'sensorEventBased' for communication using respectively the request-reply and event based style. Then it calls both functions using reference sensorRequestReply resulting in the last started server component printing the names of the methods and arguments. Then it uses reference sensorEventBased to publish to the topic 'SensorName2' which is received by all running server components that all print the method name and argument.

```
class SensorClient : public IAS_Application
{
        public:
        virtual int IAS_run(int argc, char* argv[])
        {
                SensorPrx sensorRequestReply =
                        SensorPrx::checkedCast( getRegisteredObject("SensorName1"));
                SensorPrx sensorEventBased =
                        SensorPrx::uncheckedCast(publishToTopic("SensorName2"));
                int i= sensorRequestReply->readSensor(111);
                senorRequestReply->adjustSensor(222);
                sensorEventBased ->adjustSensor(333);
                return EXIT_SUCCESS;
        }
};
```

The example above clearly shows the tight integration between the request-reply and event based style as the adjustSensor method can be called on the same Sensor object using both the request-reply and event based style. Please note that the event based style can not be used to call method ‘readSensor’ as this method is defined with a return value. Only a style with Two-way communication properties, such as the request-reply style, can be used to receive return values. Calling this method using the ‘sensorEventBased’ reference, that uses the event based style, will result in an error at run time.

5.6.2 Testability

There is no official test strategy in the IAS group for testing the prototype systems. Software testing is done ad hoc during development and use of a system. Thorough test strategies for complex experimental systems that deal with dynamic uncertain environments are considered to be too costly in relation to the risks it could eliminate.

To support this ad hoc testing I have developed a central logging facility, I propose to develop a standard way of visualizing the internal state of a component and I recommend using the build environment to run automated test to check for faults.

5.6.3 Logger

The logger facility lets all components of the distributed system send log messages to a single logger facility. A log level is associated with each message which represents the level of abstraction of the message. The logger facility writes all messages to disk for later analysis and prints all messages for which the log level is enabled so that the developer can focus on specific levels. The logger supports testing software by capturing the state of the distributed system as a whole.

5.6.4 Visualization

As components developed by the IAS group can be complex, a means of visualizing the internal state of components is helpful for understanding the system. Therefore I propose to develop a component specifically for visualization purposes. Such a component should provide an interface for drawing graphic primitives such as lines, circles and rectangles. Optionally, this interface could be extended for a specific project to draw higher level concepts such as graphs. The implementation of such a visualizing component is future work due to the limited time available. The component can best be implemented in Java as it provides a build in graphical library available on all required platforms.

5.6.5 Automated tests

The GNU build environment enables running all the declared test software that is developed in a project. This allows developers to automatically test the software after modifications to the source code. Test software contains user defined checks on the responses of the system and takes the form of a client component that can interact with one or more server components.

6 Results

This section evaluates the selected middleware product ICE, based on the extent in which it fulfills the identified requirements. First I evaluate in what extent the selected architectural styles can fulfill the functional requirements. Secondly, I evaluate the level of fulfillment for the non-functional requirements and the requirements for the middleware product itself.

6.1 Functional Requirements

As ICE is general purpose middleware, it can be used to realize distributed systems with any functional requirements. Additionally, section 5.2.3 shows that the selected request-reply and event based style are often used in architectures of autonomous robot and distributed sensor systems.

6.2 Non-Functional Requirements

To evaluate the level in which the non-functional requirements are fulfilled, I evaluate the level of fulfillment of the non-functional requirements as described in paragraph 5.1.2. The requirements for the middleware product to be open source and generally applicable are met by the selection of ICE.

6.2.1 Development time

By integrating ICE in a build environment and creating the IAS_Application class to hide unnecessary complexity of using the request-reply and event based style, the development time is reduced considerably. Only a short manual is required to enable developers to build distributed systems. In case of unanticipated problems with ICE, a support form is available that is used by an active user community and by the developers of ICE. Integrating existing software into a distributed system is possible by wrapping it in a desired interface. The possibility of architectural mismatches is reduced by providing both the request-reply and event based style.

6.2.2 Interoperability

By using ICE, developers are able to build distributed application consisting of components written in the desired languages C++, Java and others. As C code can be used in a C++ program it also provides support for the C language. At this time the build environment is available on Linux and Unix platforms but it will be made available on Windows shortly. ICE offers a facility to deal with firewalls and to access components that reside in a different network and only have a local IP address as described in chapter 39 of the ICE manual [46].

Components that use ICE as middleware can only interoperate with other components that use the ICE middleware. Interoperability with other middleware products needs to be realized by developing a bridge to other specific middleware products.

6.2.3 Modifiability

By providing both the request-reply and the event based style, developers are able to make a tradeoff between a tightly coupled system with control over other components and a loosely coupled system that can more easily be modified.

When using the event based style, removing a component that publishes data has no direct effect on the remaining components other then stopping the receiving of data. Removing a component that is subscribed to specific data has no influence on other components at all. With the event based style, components do not automatically reconnect to any replacement components.

When using the request-reply style, removing a server component will block all client components which will try to reconnect to any replacement server component with the same name. When after 30 seconds (the default set in the build environment) no new connection can be established, the client components time out and stop with an error message. Removing a client component does not affect the remaining components other then to reduce the load of the server component that is was calling.

6.2.4 Performance

Although ICE does not provide any filtering mechanism or a means to prioritize different types of data for the event based style, it still provides a good performance due to the efficient communication between components. This is possible because ICE, as a single product, defines its own language and platform independent representation of data which is optimized for speed. This representation is hidden from developers. Other solutions where several products are used to support different languages and platforms, often suffer from a more complex representation that reduces performance.

6.2.5 Testability

The testability of components is positively affected by the request-reply style that allows error messages to be returned to the component causing the error. Additionally the build environment allows for test software to be easily added to the automatic test suite and the logging facility provides an awareness of the state of the system as a whole.

However, several shortcomings can be identified based on the testability requirements. One of these is the inability to automatically visualize all the active components and interaction between them. As a result a developer does not have a good intuitive overview on what the system as a whole is doing. A second shortcoming is that with the current state of ICE and the enhancements, there is no way to automatically fix the order of interaction between components running in parallel although the interactions using the request-reply style are partially fixed. This makes error reproduction more difficult when a specific order of interactions is causing the error.

6.3 Conclusion

By choosing ICE, the functional requirements and most of the non-functional requirements are sufficiently satisfied so that the chosen solution is suitable for use in the IAS group. The biggest problem identified is that of the limited testability properties resulting from not being able to visualize all components and interaction and fix the order of interactions. Due to the limited time available for this work, no good solution could be provided for this shortcoming.

A middleware product that uses a central architecture, such as the shared resource in the event based style, would be able to automatically visualize all components and interactions. However, the request-reply style does not communicate via a central point but interacts directly with other components. Modifications to the ICE middleware product are required to automatically notify a visualization component of this type of direct interaction which would result in a performance penalty.

At this time I am not aware of any middleware product that allows to automatically fix the order of interactions between different concurrent components. This is a difficult problem as the order in which data is send is not necessarily the order in which data is received. I do not see how, in the short term, ICE can be adjusted to fix the order of all interactions for at least the request-reply and event based style.

7 Evaluation

After completion of this work I am satisfied by the chosen middleware selection strategy as described in section 4. Especially the central role of the identified requirements has allowed me to evaluate different middleware approaches without focusing on aspects that have little effect on the requirements of the IAS group. A middleware selection process, like any other design decision, must focus on both fulfilling the functional requirements and finding the optimal tradeoff between the non-functional requirements.

The middleware induced architectural styles are identified as the aspect of middleware the most influence on the requirements and therefore plays a central role in this work. By defining these architectural styles using the identified communication properties, the otherwise abstract architectural styles could be evaluated based on their effects on the requirements.

It could be argued that this selection approach results in a merely subjective and possibly biased selection of a middleware product. However, this is a result of the nature of the selection problem and not of the approach that was taken. The subjective nature of the middleware selection problem is caused by the large variety and complexity of middleware products, by the complex relation of the middleware induced styles on the requirements, and by the fact that only general requirements can be identified for future projects. As a result no objective weights can be assigned to the different requirements or to the effects different middleware products have on these requirements. In this work different weights are implicitly assigned by the rational that is given for the different decisions that are made.

Other middleware selection approaches have a stronger focus on the different tools that the middleware product supplies. These include tools for starting and stopping components, tools for runtime configuration of the connections between components, and tools for automatic distribution of new versions of components. The reason that there was little focus on such tools in this work was that these tools only had a small effect on the identified requirements. With different requirements these tools might have played an important role in the selection process.

Yet other middleware selection approaches focus on the interoperability with other systems that are already in use such as databases systems or other special purpose middleware products. The reason that there was no focus on these interoperability issues is that the IAS group does not use such system at this time and thus there were no requirements to do so. The only systems the middleware is required to interact with are the unknown middleware products used by future partners in future projects. As described in paragraph 5.5.1, interoperability between middleware remains a problem due to the lack of standardization. Therefore, interoperability between different middleware products is generally realized on an ad hoc basis by developing special-purpose conversion components that realize a bridge between the products.

Yet other middleware selection approaches focus on the software-development methodology and the support that different middleware products offer for a particular methodology. This work does not focus on this aspect as the IAS group has no standard methodology for software development, presumably because of the overhead of using methodologies for developing experimental prototype systems with a relatively short lifetime. Therefore, this aspect did not appear in the requirements and is thus ignored.

As concluded in the section 6, the selected middleware product ICE and the enhancements, sufficiently satisfy the identified requirements. Future projects, in which distributed systems are developed, will validate if the right requirements are identified or that important aspects were not discussed in the interviews that resulted in the current set of requirements. Judging from the previous projects undertaking by the IAS group, I belief that no important requirements are missing that can not be fulfilled by the selected product or other means.

8 References

Books

[1] Len Bass, Paul Clements, Rick Kazman. **Software Architecture in Practice, Second Edition**. April 2003, 560 pages, ISBN 0-321-15495-9.

[2] Chris Britton, Peter Bye. **IT Architectures and Middleware: Strategies for Building Large, Integrated Systems**. 2nd edition, May 2004, 337 pages, ISBN 0-321-24694-2.

[3] Roy Thomas Fielding. **Architectural Styles and the Design of Network-based Software Architectures**. Doctoral dissertation, University of California, Irvine, 2000, ISBN 0-599-87118-0.

Papers

[4] J.S. Albus, H.G. McCain, and R. Lumia. **NASA/NBS standard reference model for telerobot control system architecture (NASREM)**. Technical Report 1235, NBS, 1987.

[5] R.A. Brooks. **A robust layered control system for a mobile robot**. IEEE Journal of Robotics and Automation, April 1986.

[6] A. W. Brown. **Mastering the Middleware Muddle**. IEEE Software, vol. 16, no. 4, pp. 18--21, July 1999.

[7] Patrick Th. Eugster and Pascal A. Felber and Rachid Guerraoui and Anne-Marie Kermarrec. **The many faces of publish/subscribe**. ACM Computing Surveys, pages 114--131, 2003.

[8] S. Fleury, M.Herrb, and R. Chatila. **Genom: A Tool for the Specification and the Implementation of Operating Modules in a Distributed Robot Architecture**. In Proc. of the IEEE/RSJ International Conference on Intelligent Robots & Systems (IROS), pages 842--848, Grenoble, France, September 1997.

[9] D. Garlan and M. Shaw, **An introduction to software architecture**, in: V. Ambriola and G. Tortora, editors, Advances in Software Engineering and Knowledge Engineering, World Scientific Publishing Company, pp. 1--39, 1993

[10] David Garlan, Robert Allen, and John Ockerbloom. **Architectural Mismatch or, Why it's hard to build systems out of existing parts**. Proceedings of the 17th International Conference on Software Engineering (ICSE-17), April 1995.

[11] Michael Goedicke, Uwe Zdun. **A Key Technology Evaluation Case Study: Applying a New Middleware Architecture on the Enterprise Scale**. In Proc. of 2nd Int. Workshop on Engineering Distributed Objects (EDO 2000), Davis, USA, Nov 2-3, 2000.

[12] Barbara Hayes-Roth and Karl Pfleger and Philippe Lalanda and Philippe Morignot and Marko Balabanovic. **A Domain-Specific Software Architecture for Adaptive Intelligent Systems**. IEEE Trans. Softw. Eng., 1995.

[13] M. Kleinehagenbrock, J. Fritsch, and G. Sagerer. **Supporting advanced interaction capabilities on a mobile robot with a flexible control system**. In Proc. IEEE/RSJ Int. Conf. on Intelligent Robots and Systems, volume 3, pages 3649--3655, Sendai, Japan, September/October 2004.

[14] Alexei Makarenko, Alex Brooks, Stefan Williams, Hugh Durrant-Whyte. **A Decentralized Architecture for Active Sensor Networks**. IEEE International Conference on Robotics and Automation (ICRA'04) April 26 - May 1, New Orleans, LA, USA, 2004.

[15] Elisabetta Di Nitto, David, S. Rosenblum. **On the role of style in Selecting Middleware and Underwear**. In Proc. of Engineering Distributed Object 99, ICSE 99 workshop, pages 78--83, Los Angeles, May 17-18, 1999.

[16] G. Pavlin, M. Maris, and J. Nunnink. **An Agent-Based Approach to Distributed Data and Information Fusion**. In IEEE/WIC/ACM Joint Conference on Intelligent Agent Technology, Beijing, 2004.

[17] N. Pryce and S. Crane. **Component Interaction in Concurrent and Distributed Systems**. In Proc. Fourth International Conference on Configurable Distributed Systems, Anapolis, MD, USA. IEEE Computer Society. May 4-6 1998.

[18] E. Di Nitto, D.S. Rosenblum. **Exploiting ADLs to Specify Architectural Styles Induced by Middleware Infrastructures**. In Proc. 21st Int. Conference on Software Engineering, Los Angeles, CA, May 1999.

[19] Douglas C. Schmidt and Carlos O'Ryan. **Patterns and Performance of Distributed Real-time and Embedded Publisher/Subscriber Architectures**. Journal of Systems and Software, Special Issue on Software Architecture -- Engineering Quality Attributes, 2002.

[20] R.G. Simmons. **Structured control for autonomous robots**. IEEE Transactions on Robotics and Automation, pages 34--43, 1994.

[21] Jess Thompson. **Avoiding a Middleware Muddle**. IEEE Software, vol. 14, no. 6, pages 92--98, 1997.

Webpages

[22] http://www.science.uva.nl/research/ias/

[23] http://www.science.uva.nl/research/ias/research/multi_agents/robocupmid

[24] http://www.thales-naval.com/newsroom/naval-press-clips/29-oct-2004.htm

[25] http://staff.science.uva.nl/~jellekok/software/

[26] http://staff.science.uva.nl/~jellekok/robocup/2003/

[27] http://www-iri.upc.es/people/porta/ambience/

[28] http://www.cogniron.org/

[29] http://aipc1.techfak.uni-bielefeld.de/projects/xcf/

[30] http://combined.decis.nl/

[31] http://combined.decis.nl/tiki-index.php?page=Distributed%20perception%20networks

[32] http://www.cougaar.org/

[33] http://staff.science.uva.nl/~wzajdel/project/

[34] http://dictionary.reference.com/search?q=paradigm

[35] http://www.theaceorb.com/

[36] http://www.cs.wustl.edu/~schmidt/ACE.html

[37] http://www.omg.org/gettingstarted/corbafaq.htm

[38] http://www.zeroc.com/

[39] http://www.w3.org/TR/soap/

[40] http://www.soapware.org/directory/4/implementations

[41] http://www.xs4all.nl/~irmen/comp/CORBA_vs_SOAP.html

[42] http://groups-beta.google.com/group/comp.object.corba/browse_frm/thread/55cd6ee5a7010ee8

[43] http://www.zeroc.com/iceVsCorba.html

[44] http://sources.redhat.com/autobook/

[45] http://www.cygwin.com/

[46] http://www.zeroc.com/download.html#doc

[47] http://www.zeroc.com/performance/

[48] http://www.xmlblaster.org/

Appendix A Prioritized utility tree

Utility Tree			W. Zajdel	N. Vlassis	J. Kok
Quality Attributes	**Attribute Refinements**	**Scenarios**	**grades**	**grades**	**grades**
Interoperability	Language	A developer can choose to use C, C++ or Java to develop a component of the distributed system.	10	5	10
		A developer can choose C# to develop a component.	5	1	2
		A developer can choose Matlab to develop a component.	5	2	7
	Platform	A developer can choose to develop a component on the Windows, Linux or Unix operating system.	10	2	10
		A developer can choose to develop a component on a PDA or Palmtop.	5	5	0
	Middleware	It should be easy to develop a bridge for communication with other middleware products (e.g. when cooperating with other organizations).	1	8	5
	Network	It should be possible to easily connect components that reside in different local networks (where their IP addresses have only local meaning).	3	8	3
Modifiability	Component changes	Changes to a component should have a minimal effect on other components with which it communicates.	4	5	7
	Runtime changes	The system continues normal operations after a temporarily unavailable component reappears (e.g. in the event of temporal network failure).	0	9	5
		Data that is send to a temporarily unavailable component is stored and resend to the component after it reappears. This way the system behaves as if the component has always been available.	0	5	3
	Modularity	A component should have minimal dependencies with the rest of the system enabling efficient reuse of components.	5	7	7
	Open source Middleware	The middleware itself should be open source and simple to understand so that it can be extended and modified when required.	5	10	7
		The middleware should have an active user community which can give advice in the event of problems.	5	4	7
Performance	Network latencies	The overhead of the middleware should result in no more then 100% increase in network latency.	5	5	10
	Priorities settings	A high priority can be set to the communication of a specific type of information which requires fast network throughput.	7	5	3

M. Maris Grades	M. Spaan grades	G. Pavlin grades	O. Booij grades	B.Terwijn Grades	Mean grades	STD grades
9	9	10	4	9	8.25	2.37546988
4	1	5	1	4	2.875	1.80772153
8	9	10	5	7	6.625	2.55999442
8	9	10	5	9	7.875	2.90012315
3	3	5	2	2	3.125	1.80772153
6	7	7	5	6	5.625	2.13390989
8	8	10	3	7	6.25	2.81577191
9	6	10	4	8	6.625	2.26384628
6	7	10	4	6	5.875	3.09088522
6	7	7	3	4	4.375	2.38671921
9	6	10	9	8	7.625	1.68501802
9	9	10	8	8	8.25	1.66904592
8	8	7	10	7	7	1.8516402
6	6	5	2	5	5.5	2.20389266
8	6	7	2	5	5.375	2.06587927

Testability	Exception handling	Errors and exceptions within a component should be communicated to the component causing the exception.	9	8	10
	Test generation	Development time of test software (that tests interacting with one or more components) should be minimized. For example by partial code generation that takes the interfaces of components as input.	8	8	8
	Visualization	Visualization of the distributed components and their interaction should provide a high level overview of an otherwise inherently illusive distributed system.	0	10	8
		The visualization should be able to pause the distributed system, run it in slow motion, inspect interaction data and should allow runtime interaction with the system.	0	7	5
	Debugging	Debugging facilities should be available to analyze the distributed system at runtime possibly in combination with the high level visualization.	7	8	7
	Logging	Logging facilities should be available to analyze the past activities of the distributed system.	7	8	10
		Log messages of all the distributed components should end up in a single file with guarantees on the relation between the order of the messages in the log file and the order in which the messages were produced.	7	8	5
	Error reproduction	It should be possible to (partially) fix the order of interactions between the components when debugging.	0	8	5
		It should be possible to record the states and interaction of the components so that it can be repeated on demand for error reproduction when debugging.	0	6	5
Development time	Familiarity time	The time needed for a developer to familiarize with the development of quality components is 1 week at maximum.	0	8	5
		Simple example application should help shorten the time to learn how to use the middleware.	7	9	3
	Development	As much of the complexity of the middleware as possible should be hidden from the developer to enable better overview and faster development. For example by providing a default component where all other components are derived from.	7	7	3
		Complex features should be hidden until a developer actually decides to use them.	7	8	3
		An easy to use methodology should be provided that fits the middleware to helps developers to, decompose the problem, make an architecture and design the system.	5	9	3
	Legacy interoperability	It should be possible to wrap existing software into a distributed component within on average 8 hours.	0	4	8

8	4	5	9	9	7.75	2.12132034
6	6	7	9	7	7.375	1.06066017
6	8	5	9	7	6.625	3.11390889
8	7	7	7	7	6	2.56347978
8	8	10	9	8	8.125	0.99103121
9	9	10	9	9	8.875	0.99103121
9	5	7	6	7	6.75	1.38873015
7	5	7	7	8	5.875	2.64237447
9	6	7	8	5	5.75	2.71240536
9	2	10	2	10	5.75	4.02669663
9	6	10	2	7	6.625	2.87538817
7	5	10	3	9	6.375	2.55999442
7	8	10	1	9	6.625	3.06768875
8	8	5	1	6	5.625	2.72226271
7	6	7	1	5	4.75	2.91547595