Technische Universität Braunschweig
Institute of Operating Systems and
Computer Networks

Diploma Thesis

Development of a new Middleware for Real Time Image Processing in Remote Sensing

by

Oliver Meynberg

Supervisors:

Prof. Dr.-Ing. L. Wolf
Dr.-Ing. U. Thomas
Dipl. Wirt.-Inf. M. Doering

Braunschweig, February 5, 2009

Erklärung

Ich versichere, die vorliegende Arbeit selbstständig und nur unter Benutzung der angegebenen Hilfsmittel angefertigt zu haben.

Assertion

I hereby declare that I wrote this thesis myself without sources other than those indicated herein. All parts taken from published scripts are indicated as such.

Braunschweig, February 5, 2009 _____

Signature

Kurzfassung

Zur Zeit wird am Deutschen Zentrum für Luft- und Raumfahrt (DLR) an einem Projekt zur Verkehrsüberwachung und zur Katastrophenhilfe gearbeitet, welches in nahezu Echtzeit vorverarbeitete Bilder der Erdoberfläche von einem Flugzeug an eine Bodenstation übermitteln soll. An Bord des Flugzeugs befindet sich ein Computernetzwerk, welches die in kurzen Abständen aufgenommenen, hochaufgelösten Bilder verarbeiten soll. Diese rechenintensive Verarbeitung findet nach modernsten Methoden zur Bildverarbeitung statt und ist auf mehrere Module, die auf verschiedenen Rechnern laufen, verteilt. Der Verarbeitungsablauf dieser hochkorrelierten Module erfordert einen ständigen Austausch von kleinen und großen Datenmengen verschiedensten Typs. In diesem Zusammenhang ist der Einsatz einer Middleware erforderlich, die eine einfache, effiziente Kommunikation zwischen den Modulen gewährleistet und gleichzeitig deren räumliche Verteilung vor ihnen verbirgt.

In dieser Arbeit wird nun zuerst untersucht in wieweit bereits existierende Middlewares Verfahren zum Austausch von großen und kleinen Datenmengen unterstützen. Anschließend wird die in dieser Arbeit entwickelte neue Middleware DANAOS vorgestellt, die im Vergleich zu anderen Middlewares, sowohl die effiziente Kommunikation zwischen den Modulen mittels Message Passing gewährleistet, als auch den schnellen Austausch großer Bilddatenmengen mittels eines Distributed Shared Memory unterstützt. Dazu wird dem Modul-Programmierer eine Programmierschnittstelle angeboten, die außerdem weitere Dienste zur Gruppenkommunikation (Publish/Subscribe-Konzept) und einen Namensdienst bereit hält.

Abstract

At the German Aerospace Center (DLR) a new project aiming at the efficient control of traffic and at providing help in cases of natural disaster is being developed which is supposed to transmit pre-processed images of high resolution from a plane to a station on the ground. Aboard the aircraft a computer network is installed which is to process these images shot at short intervals. This intensive computational work is done with the latest methods of image processing and is allocated to several modules running on different computers. The process of this computationally intensive work on highly correlated modules requires a permanent exchange of small and big amounts of data of totally different data types. Here a middleware is needed, which guarantees easy and efficient communication between the modules and simultaneously hides their spatial distribution from them.

The first part of this thesis analyses to what extent the process of exchange of small and big amounts of data is supported by existing middlewares. Then the new middleware DANAOS, which is developed in this thesis, is presented, a middleware which compared with other middlewares not only guarantees an efficient communication between the modules using message passing but also supports the exchange of big amounts of image data using a distributed shared memory. For this purpose the module programmer is offered an application programming interface, which - in addition to this - provides further services for group communication and name service.

[Hier wird später die Aufgabenstellung eingefügt.]

Contents

Chapter 1

Introduction

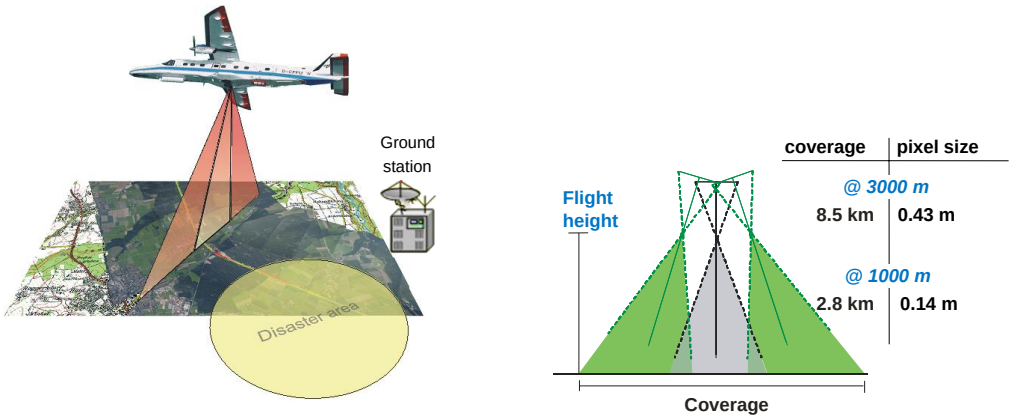
During mass events or natural disasters security authorities and emergency services are dependent on valuable and precise information about the operational area. It can be gained from various sources including maps and *digital surface models* (DSM), which provide digital information about the earth surface including buildings, vegetation and roads. If these DSMs are both accurate and up-to-date they can be an essential source for command centers of mass events or disasters. Preprocessed images of the current traffic situation during mass events like Oktoberfest or of affected areas during river floodings are examples of such DSMs.

Their availability in near real time is subject of several research projects at the German Aerospace Center (DLR). One of it - Project ARGOS[1] - is an airborne monitoring and processing system (Fig 1.1(a)). Images which are taken from the affected area are processed in near real time on-board and are sent to a ground control station via S-Band microlink.

The 3K-Camera system (Kurz et al. (2007)) consists of three off-the-shelf cameras (Canon EOS 1Ds Mark II), each capturing pictures with a size of 16 MPixel and at a rate of 3Hz. Moreover the camera system is connected to a GPS/IMU unit to allocate GPS coordinates and motion data to each image. As shown in Fig 1.1 the covered area and consequently the pixel size increase with a higher flight altitude.

Huge amounts of data need to be processed and require a high performance on-board imaging and processing unit. But before its hardware design is described in more detail, a short look at some of the (image processing) steps must be taken to get a basic idea of the computational effort which has to be made to realize near real time image processing during a flight campaign. To georeference all captured images, the position and orientation of the airplane must be permanently measured by a GPS/IMU navigation system. Also the interior orientation parameters of the camera system must be known. Then pre-orientation of the images can take place. An important stage of the processing flow

[1] ***a****irborne wide a****r****ea hi****g****h altitude m****o****nitoring* ***s****ystem* (ARGOS)



(a) An airborne monitoring and processing system.

(b) The looking directions of the three airborne cameras.

Figure 1.1: Project ARGOS. Figures from Kurz et al. (2008)

is the orthorectification procedure. Figure 1.1 illustrates the direction in which each of the cameras looks during a flight campaign: While one camera looks in nadir direction (i.e. vertically to the ground), two cameras look in oblique direction and produce high-resolution images. Before the next processing step, these images need to be geometrically corrected, or othorectified. That means after orthorectification the distances on the image are an accurate representation of the real earth's surface. The earth, however, is not flat at all and a *Digital Elavation Model* (DEM) of the photographed region must be taken into account in order to achieve an orthophoto. To increase the overall performance, the GPU of the PC's graphics card can take over this time consuming task (Thomas et al. (2008a)). Another aspect in the processing flow is the surface point matching where every point of the captured image is matched with a point in an already aquired orthophoto. This part of the DSM generation takes about 92% of the total processing time (Kurz et al. (2008)).

The major advantage of Project ARGOS is the provision of the above described digital surface models of the *current* situation, which broadens the spectrum of different applications e.g. the operation during natural disasters to support rescue and emergency services. To test the on-board systems in situations like these a flight campaign was accomplished in Voralberg, Austria, where a DSM of a test area with a torn down slope was created to compare the accuracy of this DSM caluclated by ARGOS with reference DSMs from the Austrian cartographic office of the same area. The resulting vertical variation was 40 cm and the resulting horizontal variation was 14 cm, which is absolutely sufficient for this test area (Kurz et al. (2008)).

Other possible applications are the determination of building heights in urban areas after earthquakes or automatic traffic monitoring. Traffic monitoring is surely not only useful during disasters but also for traffic planning i.e. the daily commuter can benefit from ARGOS as well. Existent measurement systems like terrestrial cameras have the disad-

vantage of their low spatial resolution and do not give a good overview of the actual traffic situation in contrast to the highly resolved images from ARGOS. Moreover its captured images are processed in a way that roads and vehicles are detected automatically and additionally these detected vehicles are tracked during their drive (Thomas et al. (2008b)). This information gives the traffic planner useful information other sources cannot provide. The traffic monitoring module is a part of the image processing system on-board and works basically as follows: Firstly prior information of a road database is used and compared with the captured image. So the road detection algorithm is applied only on areas marked with the information from the road database. Then the roads are extracted by putting a buffer zone around the roads and with the aid of edge detection algorithms the white road marking lines can be detected (Rosenbaum et al. (2008)). As a further step the vehicle detection algorithm is applied on these already detected roads. Special attention must be given to white cars as they can be mistaken for the white mid line marking on the road. Finally tracking of the already detected cars becomes possible by creating a search window in the following images of the sequence based on the cars' position in the first image (Thomas et al. (2008b)). All the implemented programs, of which a few were mentioned above, are called *modules* throughout this thesis.

On the one hand ARGOS has obviously a very good spatial resolution due to its installation on an airplane but on the other hand the temporal resolution firstly depends on the flight frequency in general and secondly on the image acquisition rate in particular (Thomas et al. (2008b)). The latter point puts high demands on the camera system and the image processing system. Live traffic monitoring requires an image acquisition rate of 3Hz, so cameras with frame sensors (as built in in Canon's EOS 1Ds Mark II) are necessary.

Three cameras taking high-resolution pictures three times a second produce a lot of data. If we assume an image size of $4992 \cdot 3328$ pixels and a color depth of 24 bits the overall output data rate of the camera system is

$$Number\ of\ Cameras \cdot Image\ Resolution \cdot Color\ Depth \cdot Acquisition\ Rate$$

$$= 3.5\ GBit/s \approx 428\ MByte/s.$$

With JPEG compression within the cameras this data rate can be reduced to 90 MByte/s assuming a data size of about 10 MByte per image.

This high input data rate on the one hand and the processing intensive modules on the other hand put high demands on the on-board image processing hardware, which consequently leads to a multi-host solution with five PCs in total (Fig 1.2). Due to lack of space more than five PCs would not fit into the aircraft. Each camera produces an output data rate of 30 MByte/s and is connected via Firewire IEEE 1394a to a dedicated host computer with DSM-processes running on it. These processes deal with image acquisition and time stamping with GPS/IMU data, orthorectification of images and street segmentation as described above. Therefore the PCs are equipped with up-to-date hardware, esspecially a high-end graphics card for the GPU-based orthorectification was chosen. The fourth and the fifth PC are dedicated to traffic mapping and microlink communication respectively.

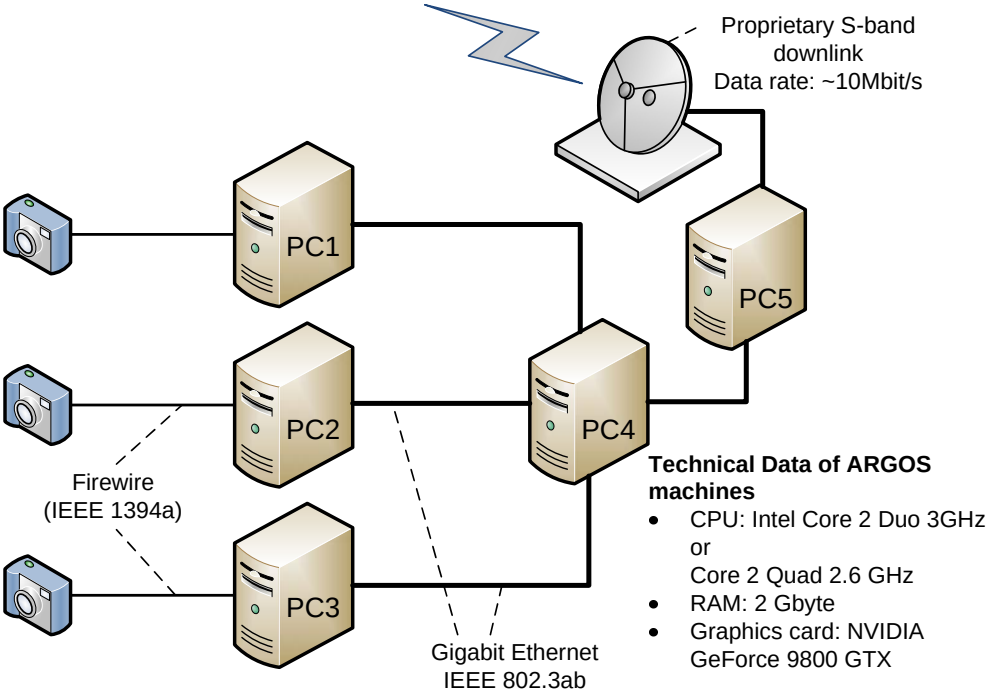


Figure 1.2: Topology of the on-board network.

S-Band microwave communication is used to send the processed data from the airplane to the ground. Currently this is done with a proprietary antenna system consisting of a 2.0m wide dish on the ground for receiving and a 28 cm wide dish in the plane for sending. This simplex downlink achieves data rates of up to 10 MBit/s within a distance of 100 km (line-of-sight).

1.1 Motivation

If ARGOS is equipped with a middleware this offers several advantages. Interprocess communication between modules becomes possible regardless of how these modules are distributed across the network. This fact makes the user's work easier while programming his image processing module. He need not concentrate on how he transfers data from module A to B any more because he can simply call the appropriate function of the middleware.

As described above several computers are needed to process the large amounts of data in an acceptable processing time. To achieve this the different image processing modules running on each machine are highly correlated and need to interchange information and image data with each other frequently. A common scenario is that one module has finished

its work on an image and one or more other modules need the resulting output, i.e. its availability must be announced. This announcement must be distributed to all machines where interested modules are running on it. One module may only need the information that it can start its own task now, another module may need the whole image to start its processing on it. So the amount of data which must be moved either within a host or also between different hosts can vary greatly depending on the transferred information.

In particular several message types for regularly distributed information are needed. In some cases module A sends something to module B and has to wait (i.e. it blocks) for the result from B. In other cases A just wants to inform B about information X but it doesn't care about B's answer and continues its work immediately.

Also the number of recipients for a specific processing result can vary. So the approriate sending module needs a method to announce or publish the data to every interested module without spamming other modules which are not interested in its announcements. Here a way must be developed to subscribe and unsubscribe to certain services. Of course broadcasting a message to all modules must be possible as well.

The programmer of a module might also be interested in a comfortable way to address other modules like typing a name in his browser's address field rather than typing the exact IP address. For this reason a name service could support the programmer and the user of ARGOS when every running module in the whole system has a unique identifying name.

1.2 Purpose

For the foregoing reasons we suggest to establish the on-board network as a distributed system to ensure an effective inter-module-communication without dragging down the overall performance. This thesis describes the development of the middleware ***D**istributed Middlew**a**re for a **N**ear Re**a**l Time **Mo**nitoring **S**ystem* (DANAOS)[2]. DANAOS is placed as an additional software layer between the operating system and the image processing modules to handle incoming and outgoing data transfers between the modules, i.e. modules are always *indirectly* connected via an instance of the DANAOS middleware (Fig 1.3).

Here the question arises if it makes sense at all to install an additional software layer and to deal with the subsequent overhead. Are the benefits greater than the drawbacks? Of course many middleware supported distributed systems already exist, so before we concentrate on implementation issues we will look in this thesis at some of the main concepts of distributed systems. It will be discussed what a distributed system is, how a middleware can transform a network of loosely coupled computers into one transparent system and what we can gain from already existing middleware solutions for the development of

[2]Danaos was a Greek mythological character and king of Argos in the Peloponnese.

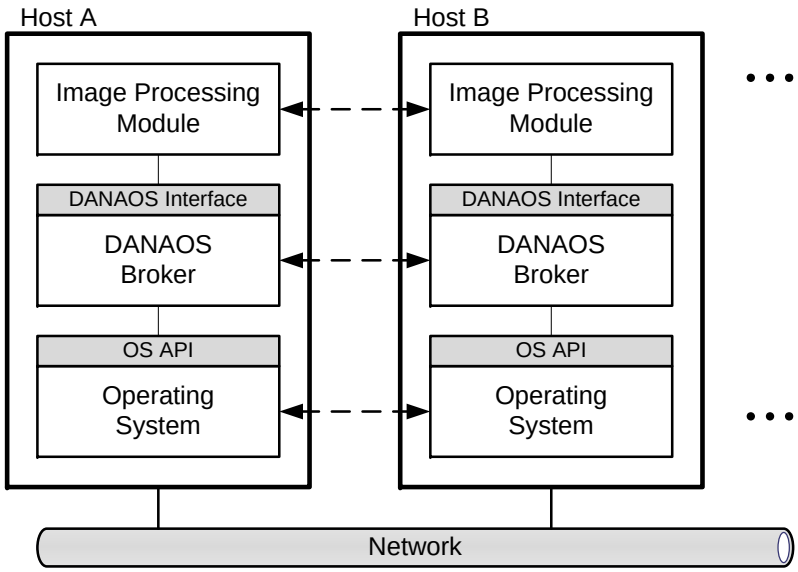


Figure 1.3: Software stack of the ARGOS machines with the additional DANAOS middleware layer

DANAOS.

The program consists of two parts and uses two main concepts for transferring data. One concept is *message passing* and is used for the exchange of administrative messages, which usually occur frequently but its size is usually small. *Distributed shared memory* (DSM) is the other concept and is required by the modules to transfer their large images across the network.

These two transfer strategies are supported by DANAOS consisting of two parts: The first part is called *DANAOS Interface* (DI). As you can see in Fig 1.3 communication with other modules either on the same host or on a different host at first requires a communication with the DANAOS instance running on the same host. Therefore we design an application programming interface with several functions for administrative communication and data transfer. Exactly one DI is dedicated to one image processing module. It provides functions for asynchronous and synchronous communication between modules as well as functions for publishing service messages and subscribing to these services. The module programmer decides by choosing one function of DI how a certain amount of data is transferred. So the degree of transparency is an issue and must be taken into account by the programmer of the module who is in turn the user of DI. If he uses the publish/subscribe concept for example he has chosen an efficient way for addressing more than one reciptient. DANAOS uses it in different scenarios as for example the announcement of a newly available data block at location X in the shared memory which several modules might need.

The second part of DANAOS is the middleware itself or one instance running on each participating host. It is called *Broker*. The Broker coordinates the delivery of messages and the administration of the distributed shared memory. We can now imagine if a large number of modules engage services of DANAOS simultaneously, an efficient processing concept is obviously needed to minimize the processing time of each query.

During and after the implementation of the designs and concepts the performance of DANAOS must be tested. Several metrics illustrate the behaviour of the system in different scenarious where special interest is given to the overall processing rate of the system. Therefore we measure, in addition to the input acquisition rate, the output processing rate. It will be checked if this rate of 3 Hz, which is necessary for traffic monitoring, is kept throughout the processing chain or if there is some kind of shortage in the system.

1.3 Scheme of this Thesis

After this short introduction of Project ARGOS, chapter 2 gives an overview of distributed systems and middlewares. Several concepts and existing middleware solutions are compared, thereby focusing on real time capabilities and the support of distributed shared memory. Chapter 3 covers the two main transfer strategies, namely message passing and distributed shared memory. Chapter 4 deals with several performance meassurement strategies used on DANAOS and chapter 5 firstly summarizes the results of this work and secondly gives an outlook on design issues which could be implemented in the future to improve both performance and usability of DANAOS.

Chapter 2

Distributed Systems - A Middleware Approach

This chapter gives an overview of distributed systems and the usage of middlewares. The characteristics of a distributed system are described as well as the goals which must be achieved to call a system a *distributed* system. Its realization can be done in several ways, one of it is implementing a middleware. The usage of a middleware has several advantages, e.g. it is independent of the used hardware because it can be implemented and run in the user space of the machines' operating system. Several existing middleware solutions are discussed, their advantages and disadvantages will be compared and we finally propose the implementation of DANAOS at the end of the chapter.

2.1 Definition and Characteristics

The extremely wide range of existing services and applications, that are based on a distributed system, makes it hard to give only one single, universally valid definition of a distributed system. The internet is one example of a distributed system, it has roughly 600 million permanently online hosts, another is the computer network ARGOS, which has 5 hosts. Different hardware configurations, coming along with different operating systems, do not simplify a defintion. All standard works, dealing each with the basic concepts of distributed systems, give a different definition. I have chosen the one of Tanenbaum and van Steen (2002) because it concentrates on the main aspect:

> *"A distributed system is a collection of independent computers that appears to its users as a single coherent system."*

If it actually appears as one coherent system in all possible scenarios, depends on the degree of transparency of the system. Other authors like (Coulouris et al. (2005)) and

(Schill and Springer (2007)) call a computer network a distributed system if all hosts communicate *only* via passing messages.

So, in any case, a distributed system is always a computer network, but not the other way around. There are three main characteristics, described in the following, which you must take into account while designing a distributed system: *concurrency*, *lack of a global clock signal* and *independent failures* (Coulouris et al. (2005)).

In a computer network many users work simultaneously with programs on different hosts; concurrent program execution is the norm. If we imagine ourselves as some co-workers in a company's intranet where we want to print documents, backup today's work or stream audio/video content, it is not practical and almost impossible that each of us has his own printer, backup server, file server and multimedia server. To reduce costs and maintenance effort we share these devices, or more abstractly, we share *resources*. Resource sharing is indeed very abstract because the term must comprehend several kinds of resources, and it supports the conclusion that many cases must be taken into account during implementation. It is one of the main problems when you deal with the design of distributed systems. In contrast to a computer network the handling of resources is mostly concealed from its users. On the one hand the usage of network resources becomes more comfortable, but on the other hand more effort must be put in the design of a distributed system in the first place.

Synchronization of processes is another important issue. Normally if two processes A and B have finished writing to two files, each makes a system call and the kernel can respond with the exact system time. Now the system knows when the two files were last modified and a process C, which is only interested in the newer one, knows which one to take. If we now imagine that processes A and B haven't run on the same host but on two different hosts, it is not clear which one of the two files is newer and process C could mistakenly read from the older file. This is caused by the fact that on user level as well as on kernel level, there is no way of absolute synchronization in a network because according to Coulouris et al. (2005) every inter-host communication relies solely on passing messages over a network and is therefore limited to its accuracy. On one host one global clock signal exists, but such a signal is impossible to implement in a network due to its latency and jitters. There are, however, different solutions for these timing problems.

Distributed systems conceal the network from the user. As we mentioned above this can significantly improve the usability - but only as long as no failure occurs. Because the location and cause of a failure is also concealed automatically. For example your mounted home directory becomes suddenly unavailable during your day-to-day work. You may possibly know that it has been mounted over the network but you do not know if it is unavailable because of a network fault or a fault of that computer where your home directory is located. So we basically deal with two different types of failures. Network failures cause the isolation of certain hosts, but the hosts might be still up and running. A host's program crash can affect only parts of its, the own system or can cause other processes, maybe on different hosts, to terminate.

2.2 General Design Goals

The foregoing characteristics of a distributed system must be considered during its design. For that reason several goals should be defined in advance to avoid commonly known pitfalls and to improve the usability and performance of a loosely coupled computer network. Tanenbaum and van Steen (2002) defined the following design goals:

- Connecting Users and Resources
- Transparency
- Openess
- Scalability

2.2.1 Connecting Users and Resources

Of course the main goal is connecting users and resources these users can commonly work with. Without this goal the attempt of designing a distributed system would make no sense at all and that is why it was mentioned as a characteristic at the beginning of the chapter. But in addition this primary goal comprehends two other interesting aspects which some authors (e.g. Coulouris et al. (2005)) consider as additional, separate goals: *handling heterogeneity* and *security*.

The internet itself illustrates best how different hardware, different communication protocols, different operating systems and different applications written in different programming languages work with each other seemingly seamlessly. Each of these components has a share in achieving that. Communication protocols for example hide the differences in networks, whereas an operating system could deal with applications.

Security issues must be taken into account as soon as the distributed system expands to another possibly insecure domain. If it transfers confidential data like bank account details or personal medical data, the used communication channels must be secure or the data itself must be encrypted. Moreover not only the message must be protected but also the peer's identity must be validated. Otherwise we, as the receiver of the secure data, don't know who has actually sent it.

2.2.2 Transparency

Ideally the normal user should not know that his application is running on a distributed system. So if the system makes the user think that he is working on only one system, you can call the system *transparent*. Transparency is a comprehensive term, a system can be transparent in several ways and to a certain degree. The Reference Model of

Open Distributed processing (RM-ODP), see ISO (1998), defines eight different kinds of distribution transparency:

Access transparency hides different data representations and formats (e.g. little endian and big endian), which can occur when different operating systems and programming languages are used. It must be considered when using heterogeneous computer architectures.

Location transparency is given if the user can access a resource without knowing the logical or physical location. A good example is the Network File System (NFS) where a client computer can access files over a network as easily as if they were on his local computer.

Migration transparency is a subset of location transparency. A system is additionally migration transparent if the user does not know if a resource has recently moved to a new location. NFS is also migration transparent.

Relocation transparency is again a subset of migration transparency. The user can continue working seamlessly *while* the resource moves to another location. An example scenario is making a call with a GSM mobile phone during driving a car from one network cell to its adjacent cell. The network hands the call over without interrupting it.

Replication transparency exists when if the user does not know how many copies of a resource exist. Distributed shared memory algorithms usually provide it.

Concurrency transparency allows several users to access a resource concurrently and without interfering each other. To achieve that locking mechanisms can be provided which guarantee that after each access the resource is left in a consistent state. This is again an important issue in the design of distributed shared memory.

Failure transparency masks computer crashes or network failures. The user may only be partly affected and can complete his task while the distributed system recovers from that failure.

Persistence transparency hides allocation and deallocation of resources from the user. Moreover it provides that these resources can be shared. A typical example is a machine which on a read request copies a data block from disk to RAM and makes it available to the users. After reading the data block can be written back to disk.

2.2.3 Openness

The openness of a distributed system depends on its extensibility. Primarily that means how easy new resources can be integrated and become available for the users. Extensibility always requires a certain set of open standards and protocols, so that newly written

programs of other developers can faultlessly interact with the system. This approach is commonly known in computer science, examples are the *Request for Commons* (RFC), where all major network protocols are standardized, or the *C++ Standard Library* which provides useful generic containers and functions for every day tasks in programming. There are of course more means of publishing standards, the widely known CORBA middleware is published through a series of technical documents including a complete interface specification. This is also known as the *interface definition language* (IDL) of CORBA (Cha. 2.5.1). An IDL specifies the services which are made available more precisely, it defines all functions and exceptions the programmer needs to either extend or rebuild the distributed system. Ideally two completely different implementations of the distributed system's interface can interact without errors and behave in exactly the same way.

To summarize, an open distributed system has public interfaces, which standardize communication between resources, and these resources' hardware and software could also be provided by different vendors.

2.2.4 Scalability

Scalability of a distributed system becomes an important factor as soon as it grows. According to Neuman (1994) it can grow in three different dimensions, namely *numerically*, *geographically* and *administratively*.

Numerical scale deals with the number of nodes, users and services in the system.

Geographical scale deals with the distance between the participants.

Administrative scale deals with the number of authorities which are responsible for parts of the network.

So, in general, a system is said to be scalable if it remains stable and effective when it grows in one or more of these dimensions. At this point things can get complicated, therefore firstly a look must be taken at how a network is affected if it scales and secondly what techniques exist to keep the system up and running.

2.2.4.1 The Effects of Scale

Scale affects basically three different aspects of a distributed system: *system load*, *administration* and *reliability*.

If the number of nodes in the system increases, the number of not working hosts will probably increase as well. The same applies to geographical expansion. The longer the distances between the network nodes are, the more likely a connection will abort. So briefly speaking, the *reliability* of the distributed system can be affected. To avoid this

a system can be subdivided into autonomous regions or one can create several copies of resources in the system.

The average *system load* of a distributed system must be taken into account, as soon as the system grows beyond a certain point. If a database of all registered users exists on each single node, it might be possible to keep it up to date. But that means that as soon as the account of one user changes, all databases on all nodes need to be updated, which produces a lot of traffic. If the system scales up a centralized approach in storing the user data has advantages as well as disadvantages. Only one database has to be updated and additionally the system is less vulnerable to attacks if sensitive user data is only stored on one secured server. But if the system grows further, neither all nodes nor one node can hold all user information. The central server would become a bottleneck.

Moreover the system could expand across administrative boundaries and some authoritives might want to have control over their own domain. This *administrative* effect of scale could also be handled by distribution and replication.

2.2.4.2 Scaling Techniques

If the system scales up, the distributed system should operate in an effective and stable way or it should at least pretend this state to its users as long as possible. Therefore several scaling techniques exist, we present three of them, namely *distribution*, *replication* and *caching*.

Distribution: Distribution is a technique to spread the information maintained by the system across several servers. This has some advantages. As the number of server requests climbs up, each server - instead of only one - can handle a certain part of the requests. The most famous example is probably the Domain Name System of the Internet. There are way too many domains to store and query them on only one server, moreover the global dimensions would let make a user in New Zealand wait for several seconds, let alone the (political) quarrel about the administration of this one and only server. So we need a large number of DNS Servers, but where should they be placed and how does a user find the right DNS Server? Distributed systems exhibit locality, users of the TU Braunschweig are more likely to access websites of the TU-BS.DE domain than users from another instution in a remote country. So the university's DNS server should be located somewhere near the campus. The right DNS server can be found through a hierachical name space which is also a nucleus part of DNS. If another user from New Zealand tries to connect to a host WWW in the CS.TU-BS.DE domain for instance, his DNS server does not know anything about a strange DE domain and would ask one of the 13 global root servers[1] where this can be found. The server responsible for the DE domain in turn knows who is responsible for TU-BS, and so on. So the major advantages are that firstly the locality of the servers conceal the internet's global scale to the user and secondly due to

[1] In fact there are 167 today, see http://root-servers.org

the hierarchical naming root servers are only bothered by top-level-domain DNS servers - at least theoretically. Concerning this, Wessels and Fomenkov (2003) published a very interesting paper about the discrepancy between the theory of DNS and how it is put into practice.

Replication: Replication is not less important than distribution. Distribution means that several resources share information but one entry never exists twice at the same time. Applying replication means that several copies of a resource exist, where each additional copy takes away workload from the existing ones. Maintaining multiple copies lead again to the question where these copies are placed best in the system. Furthermore it is crucial to keep all copies consistent, which can be a challenging task, see Cha. 3.3.7 for an example.

Caching: Caching, like replication, results in making copies of resources and it is a special form of it. Again the placement and the consistency of a cache are important factors to consider. The difference to replication is, however, that a cache is only a temporary replica of the original. If we use pure replication, there is no original, or better there are only originals but no copies. Using caching the data is copied, but if the original is updated the cached data must not necessarily be updated as well. So the maintenance of a cache resides with the client of a resource and not with the server.

2.3 Design Principles

Until now we have a general idea of what a distributed system is and what goals we should keep in mind while designing it. In this section we take a closer look on the design principles. Many software concepts for distributed systems exist like *multicomputer operating systems* but we will concentrate on distributed systems, which basically consist of an arbitrary number of loosely coupled computers with their own set of CPUs, main memory and operating systems. The middleware running on each host gives the network the additional features of a distributed system. In this context we focus on questions like: How do different processes communicate with each other? How are services actually distributed and which part of the system deals with their reception? How can we keep several replicas consistent? These are some of the questions we are going to answer in this section.

2.3.1 Communication Mechanisms

A distributed system is a network of loosely coupled computers and should appear to the user as a coherent system, i.e. in terms of communication two conditions must be fulfilled: Firstly the participating machines must be able to communicate and exchange data and secondly the network must be able to conceal that fact to the user. So before

implementing the actual distributed system we should have at least a brief understanding of the underlying communication layers. In ITU (1994) seven layers of abstraction are defined. Each layer provides a service via a *service access point* (SAP) to its adjacent layer. Due to the wide range of different physical media and communication protocols these layers need to guarantee compatibility between different systems. In literature a five-layer model is often used. It consists of the physical layer, the data link layer, the network layer, the transport layer and the application layer. This model is perfectly sound as long as we only consider a computer network. A distributed system needs an additional layer between the transport layer and the application layer to provide its services to the user. Consequently we will use a modified model with six layers in this thesis, which is illustrated in Fig. 2.1 below. These layers are described in the following:

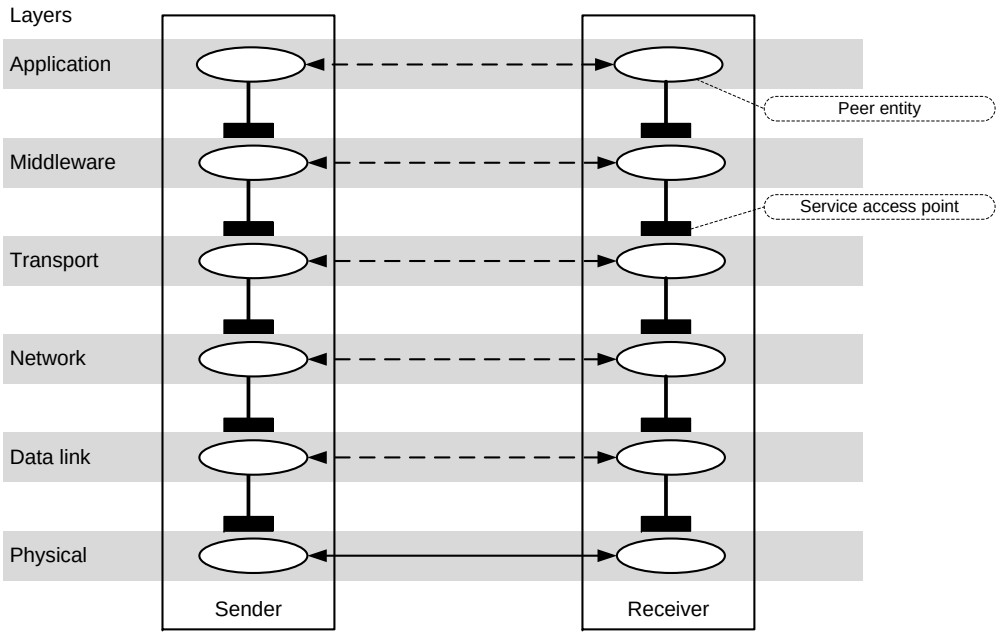


Figure 2.1: Modified protocol stack of network layers

Physical layer The basic layer of all networks is the physical layer. It translates incoming electronic or optical signals into a raw bitstream for the data link layer. It includes mechanical and electrical specifications, as well as frequency allocation, signal strength, modulation and bandwith. It is probably the most complex layer, therefore it is hard to name one source which covers the whole subject.

Data link layer The data link layer manages the data transfer between *adjacent* stations. It transforms the raw bit stream offered by the physical layer into a stream of frames for the network layer. Besides framing, methods for error detection, error correction and flow control are provided. Media access control is another important task

of this layer. It handles multiple concurrent accesses to the medium and thereby occuring collisions.

Network layer The network layer ensures data delivery from source to destination, no matter how many intermediate systems the data may pass. To accomplish this task it puts data into packets and provides a route to the destination to each of it. The most wide-spread network protocol is IP. Additional tasks of this layer are congestion control and quality of service.

Transport layer The hosts a packet may pass during its journey are identified by their network address. To map the incoming packet to its destination application within the host the tranport layer is necessary. It provides a *transport service access point* (TSAP) to the application. If TCP or UDP is used, it is called a socket and consists of the host's IP address and of the 16-bit number local to that host, called a *port*.

Middleware layer These sockets mentioned above act as the TSAP to the middleware layer. It processes the received data and passes it to the applications. This additional layer provides a wide range of services which can basically be divided into two categories. For one thing there are internal services and for another thing there are services for high-level communication. The internal services implement for example synchronization like mutual exclusion algorithms or server functionality to handle multiple threads. High-level communication represents the service access point to the specific applications using the distributed system. An application may call a function which actually runs on a different machine and returns to the application after completion without the user's notice.

Application layer This layer is the top of the protocol stack. In general both applications and application-specific protocols as the *hypertext transfer protocol* (HTTP) are located here. In the case of project ARGOS the image processing modules are applications which communicate via the DANAOS Interface with the underlying middleware layer.

2.3.1.1 Forms of Communication

Several forms of communication exist. If an application wants to exchange data with another application possibly running on a different host, the middleware guarantees the data to be sent to its destination, but it can not guarantee its successful delivery in all cases. This depends on the implemented form of communication. Tanenbaum and van Steen (2002) distinguish between six different forms shown in Fig. 2.2 below.

Communication can be persistent or transient. *Persistent communication* between process A and B means that the receiver B does not actually need to run when A sends a message. It is up to the middleware to deliver the message as soon as process B starts. In contrast to that, *transient communication* does not offer this service. The message is simply forwarded to the destination, but if B is not running, the message will be discarded.

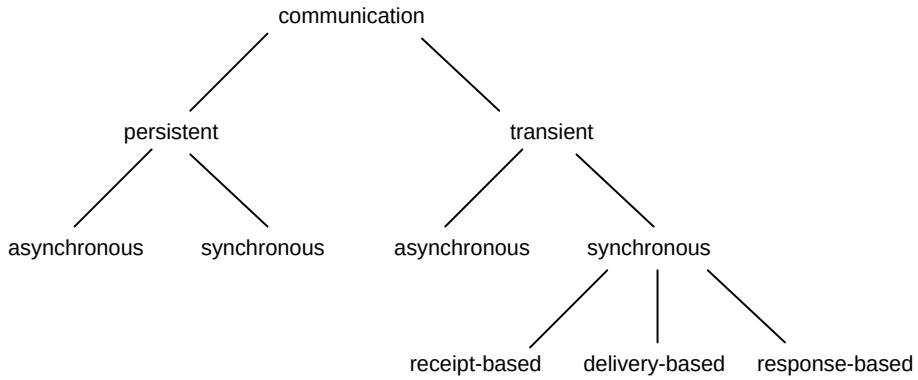


Figure 2.2: Forms of communication

Moreover communication can be asynchronous or synchronous. Using *asynchronous communication* process A can continue immediately with a different task after sending a message to B. But this also means that A will never get to know if B really received A's message, which is no problem at all, if A does not care anyway. On the contrary, if using *synchronous communication* A is blocked until the system sends back a reply. This reply either can be sent by the receiver itself or by some intermediate system.

So these two characteristics - persistence and synchronicity - can be combined as shown in Fig. 2.2. The IMAP protocol for email services can be regarded as a persistent asynchronous communication protocol because the receiver does not need to be online while the composer is sending the message and in addition the composer does not get an acknowledgement by default.

There are three extra forms of transient synchronous communication. These forms differentiate in the point of time when a receiver sends back its acknowledgement. *Receipt-based* transient communication means the acknowledgement is sent as soon as the message arrives at the receiving process B. B may currently have some other work to do. If process B sents back the acknowledgement as soon as it processes the sender's request, it is called *delivery-based*. And finally *response-based* means, that B waits with sending the acknowledgement until it has finished processing the request.

2.3.1.2 Hiding Communication

Until now nothing has been said about hiding communication from the user. The user should be unaware of the fact where his invoked procedures are actually processed when he uses a distributed system. Therefore an additional level of abstraction implemented in the middleware is needed. We will discuss briefly three mechanisms, each of which can provide this extra degree of transparency: *remote procedure calls* (RPC), *remote method invocation* (RMI) and *message oriented middleware* (MOM).

The idea of using RPC stems from programming on one host. If you implement something on a single machine it goes without saying that you use a procedure to compute something and wait for it to return. A RPC seizes this idea and extends it to multiple hosts. So your program calls a procedure, the procedure itself manages communication with another host where the actual computing takes place and waits for the reply. The reply will be forwarded to the caller of the procedure as a normal return value. This form of *interprocess communication* (IPC) is shown in Fig. 2.3. Two machines, one client and one server, exchange data via an RPC. You can break it down into ten subsequent steps. The client thinks it calls a "normal" local function, but instead it calls a so called *client stub*(1) which in turn creates a message and calls some network routines (2) to send it to its destination (3). The server stub on the destination machine gets the message from its network routines (e.g. berkeley sockets)(4) and calls the server procedure for processing the requested data (5). The return value of this procedure is passed to the client procedure via the same way as the request came to the server (6) - (10). The process of transforming the memory representation of an object to a data format suitable for transmission is called *marshalling*. The reverse process is called *unmarshalling*. (Tay and Ananda (1990)) as well as (Birrell and Nelson (1984)) give a good survey of RPCs.

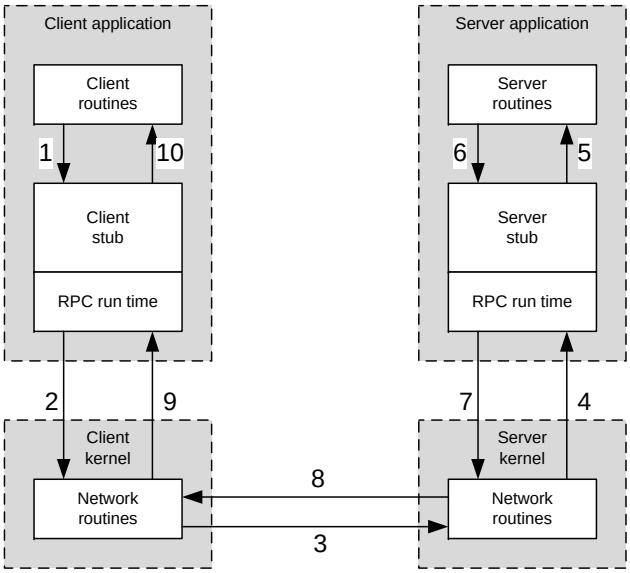


Figure 2.3: Client and server communicating via remote procedure calls.

The idea behind RPCs is both simple and powerful as it gives the client procedure the possibility calling a function located on a remote host as simple as calling a local function. But life is of course not that easy and there are some serious drawbacks which limit the scope of possible applications. The first one is that function parameters in the C

programming language can be *call-by-value* or *call-by-reference*. In the first case the parameters are simply copied on the stack and it also works out perfectly well with RPCs. But in the latter case a reference would be sent to the destination machine and of course the reference is not valid for the destination's address space. To fix that, there is a special form of RPC called remote method invocation. RMI can handle references whereby the scope of possible applications is broadened to the entire object oriented programming languages. The basic idea is that local methods invoke a local dummy object, called a *proxy*, which in turn communicates with the real *remote object*. Several existing middlewares like *Java RMI*[2] or Jini (Cha. 2.5.5) use this form of communication.

However, RPC and RMI require transient communication. As we described earlier that means that the receiving site must be online in order to communicate with it. Not all applications could fulfil this precondition, e.g. email services cannot assume that their users always have turned on their machines. Message-oriented systems can close this gap. First of all message-oriented communication can be both persistent and transient.

Transient message-oriented communication can be in fact the base of RPCs. But it has the additional advantage that messages can be stored more easily in the intermediate communication system e.g. via message queues. How long messages stay in these queues depends on the length of the queue and on its processing time but theoretically there is no upper bound and they will stay there until the recipient goes online and queries its messages. These message-queing system are generally referred to as message-oriented middlewares.

2.3.2 Client-Server Model

The client-server model describes a widely used way to distribute computing power in a network. Users are interacting with clients, which are in turn connected with each other via one or more centralized servers. The crucial aspect is balancing the functionality of network applications. In former times the clients were designed as terminals. Then more functionality migrated to the client's site, particularly the user interface code. Since a decade ago many companies have bucked this trend and move towards a "thin client/fat server" solution. In this latter scenario the client machine supports only a graphical user interface while most computing takes place on the server.

2.4 Middleware in Distributed Systems

Having discussed both the design goals and the design principles we need to have a closer look at the implementation of a distributed system. As we stated earlier we want to focus on the usage of middlewares as an additional layer in the software and the protocol stack.

[2] http://java.sun.com/docs/books/tutorial/rmi/index.html

We have already discussed the middleware layer and have positioned it between the transport layer and the application layer. Of course other options are in existence to enable support of distributed applications. From support at the hardware level all the way to extensions of applications a variety of different architectures provides distributed services. For example a *distributed operating system* distinguishes itself by managing the hardware of tightly coupled computer systems. This kind of system runs either on one multiprocessor host or on homogeneous multicomputers. Another approach would be the use of a *network operating system* where on each host an independent operating system runs and supports communication over the network.

Neither of these two solutions is satisfactory. Distributed operating systems do not consist of *independent* loosely coupled computers. On the contrary network operating systems do not conceal the underlying network so the system does not appear to the user as one coherent system. So the definition we stated at the beginning of the chapter asks for another solution - the middleware. The middleware runs on top of a network operating system of independent machines as an additional software layer and hides by means of this layer the physical distribution of resources.

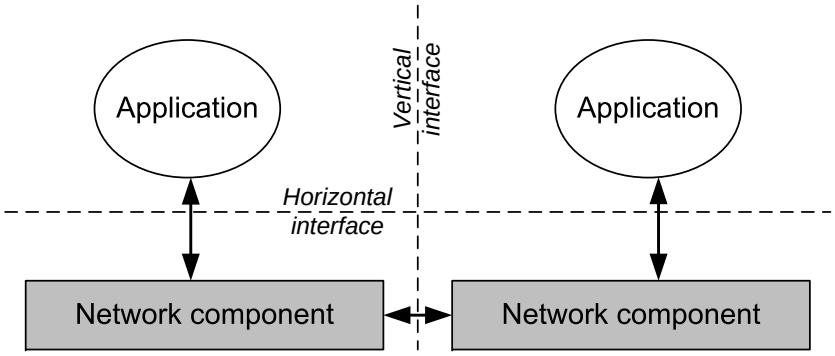


Figure 2.4: Horizontal and vertical interfaces to provide portability and interoperability.

This middleware model uses two types of interfaces. Firstly the middleware must communicate with the underlying operating system and accesses the provided *application programming interface* (API) like berkeley sockets for communication or functions for mapping shared memory. This type can be refered to as the horizontal interface shown in Fig. 2.4. The second type of interfaces is the vertical interface type and is placed between the hosts and defines the message structures being sent. These messages are also defined as *protocol data units* (PDU) and are defined in each of the first four layers of the OSI model.

A proper definition of the horizontal interface allows an easy migration to other hosts and results in the *portability* of applications and middlewares. The definition of a vertical interface results in interoperability between heterogeneous hosts.

Of course you lose these advantages as soon as an application skips the middleware layer and communicates directly with the underlying services. In that case severe data inconsistencies can occur because the application does not know how the middleware handles internal services and the other way arround. So the adherence to the layers and interfaces is crucial.

2.5 Examining Existing Middlewares

In this section we examine several existing middlewares for the support of all necessary features required by project ARGOS. These features are described in the following:

Sharing data by replication The middleware must support a fast way of exchanging large amounts of data between processes. The distributed shared memory model introduced by Li and Hudak (1989) shall provide this feature. Because replication of data objects is an integral part of distributed shared memory, existing solutions are primarily checked for replication mechanisms. Several ARGOS modules need to exchange captured or orthorectified images. The main advantage of a distributed shared memory is that the I/O data consumed/produced by the modules resides in main memory the whole time. It should be one of two supported ways of interprocess communication.

Synchronous/asynchronous communication The other interprocess communication mechanism must support synchronous and asynchronous communication between modules. The modules need to exchange information at high rate, so the designated IPC mechanism must perform accurately for small, frequently sent messages.

Group communication More than one module might be interested in a published service. To reduce the amount of sent messages in the system group communication must be supported to reach several receivers with one sent message.

Name service To address modules in a more convenient way a name service should be implemented. Each module should have a unique name it can be identified with.

Timing There are certain real time criteria to be met. Images should be made available to other modules at a rate of 3Hz. Because of the size of the images of about 20 MByte the transfer rate of large amounts of data plays a decisive role here.

The most important criteria are firstly easy communication between the modules and secondly the exchange of large amounts of data. Consequently we must give special attention to these two points while examining existing middlewares.

2.5.1 CORBA

The *Common Object Request Broker Architecture* (CORBA) is a widely known middleware developed by the *Object Management Group* (OMB)[3]. CORBA is a freely available specification which is implemented by various vendors both as proprietary and as open solutions. So CORBA is not a middleware itself, but it defines a specification to allow different implementations of the same architectural component irrespective of the used programming language. In Cha. 2.2.3 we asked for openness and CORBA is the answer. Due to the CORBA interface description language the developer can specify data structures without sticking to one programming language syntax. So it is another layer of abstraction in the software development process (Fig. 2.5). That means the specification

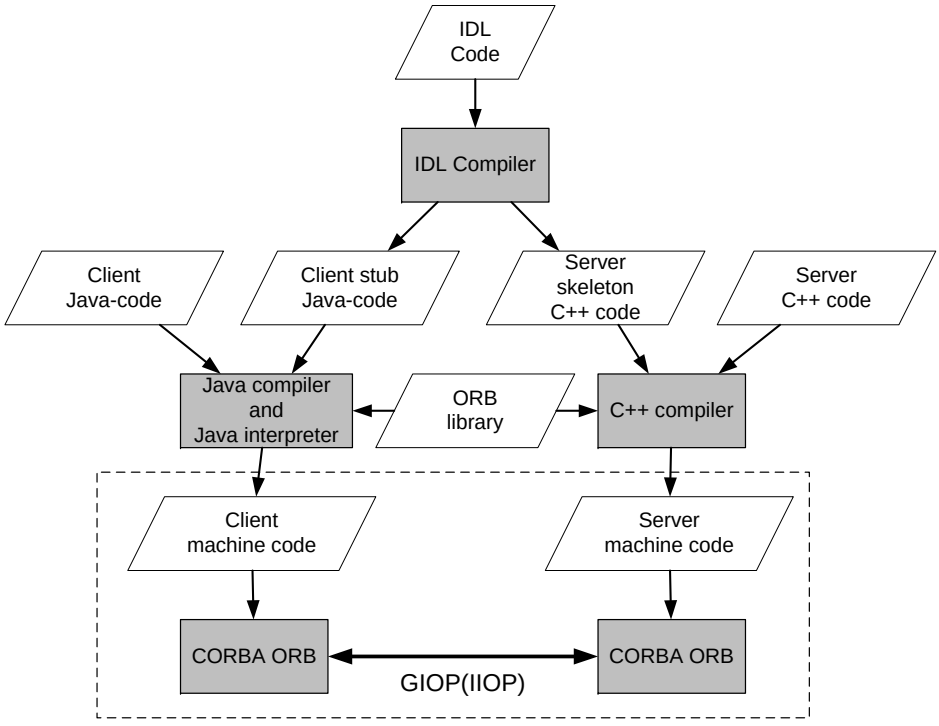


Figure 2.5: Exemplary overview of applications compiled for the use with CORBA. The participating entities can be written in any of the CORBA-supported programming languages. In this example the client is written in Java and the server in C++.

is written in an IDL, which an IDL compiler translates into source code of the chosen programming language, e.g. C++ or Java. These pieces of code comply with the client stub and the server stub or skeleton respectively. We have already mentioned this construct in Cha. 2.3.1.2. The stubs and the code with the actual program are compiled together with an ORB library to form the machine code which runs on the ORB. This *Object Re-*

[3]http://www.omg.org/

quest Broker (ORB) is the core of every CORBA running host. It enables communication between processes while concealing all issues dealing with distribution and heterogeneity.

2.5.1.1 Communication

In CORBA all communication between hosts takes place by invoking an object on a remote host. Fig. 2.6 shows an overview of this process. The client application calls a stub which is responsible for initiating the communication towards the server skeleton. It wraps client object functionality and translates the calls from the caller object so that they can be marshalled and transfered over the underlying network. One of the tasks of the ORB is sending and receiving messages conform to a standardized protocol, named the *General Inter-ORB Protocol* (GIOP). The *Internet Inter-ORB Protocol* (IIOP) is the most commonly used version of it and runs atop the TCP transport protocol. This protocol enables communication between clients and object servers from different vendors. On the server site the skeleton translates the incoming data from the client stub to the correct up-calls to server objects. It is responsible for unmarshalling the parameters and passing them on to the server object. Moreover the return values from the called server objects needs to be marshalled again and sent back to the client stub. There are in fact two kinds of invocations in CORBA: static and dynamic. Static invocation is used when the remote interface for the remote object is known at compile time. However, if it is not known at compile time, a dynamic invocation must be used. In this case the client calls the *Dynamic Interface Invocation* (DII) instead of the client stub. On the server site the *Dynamic Skeleton Interface* (DSI) is used instead of the static skeleton.

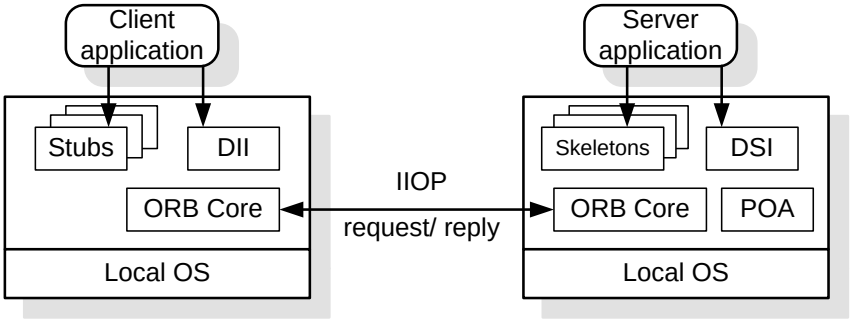


Figure 2.6: Remote object invocation: a client application invokes a server-object without being aware of its location.

As we stated earlier remote object invocation is a transient form of communication and this model is an inherent part of CORBA. To support persistent communication as well, two additional models of communication have been introduced to the CORBA standard which nvertheless are based on remote object invocation. One is the *callback model* and the other is the *polling model*. Both support persistent communication by using a form of

asynchronous commmunication. See Tanenbaum and van Steen (2002) for details.

2.5.1.2 Sharing Data through Replication and Caching

Unfortunately CORBA does not offer generic support for caching and replication. There is an extension called CASCADE, see Chockler et al. (2000), which offers transparent caching. However, it is designed for wide area networks with a dynamically changing number of participants. These features might be very useful in a different scenario but are not appropriate for the usage in project ARGOS.

Another approach is made by Fleisch and Hyde (1998). They want to combine the advantages of distributed object systems like CORBA with the advantages of distributed shared memory. The idea is to place the distributed objects into the virtual address space created by the DSM. Introducing a *virtual distributed object* (VDO), as it is called by the authors, is promising, but it requires at first the implementation of both a DSM system and a distributed objects supporting middleware. Interaction between these initially independent systems might be a challenging task.

2.5.1.3 Name Service and Group Communication

Due to the extensive usage of CORBA in a wide variety of different applications our previously defined criteria name service and group communication are well supported by CORBA. Object references in CORBA can be identified by using an (id,string)-pair, where both the id and the string are character-based names.

Group communication can be implemented using the CORBA event service. Comsumers and suppliers are connected via a logical event channel. Consumers are either notified of an incoming event or actively ask the suppliers for new events.

2.5.1.4 Timing

As mentioned above the image processing modules need to load/store large data blocks at a specific rate. However, CORBA is not able to fulfil predefined timing constraints. The original CORBA standard does not support real-time applications. So it is not possible to make a point about how long a remote object invocation actually last. Moreover the client does not have the possibility to express timing constraints on its request to the server. To improve CORBA in this respect the CORBA standard has been extended with support for real-time applications by the *Real-Time Special Interest Group* (RT SIG), which was founded within the OMG in 1995. Today several implementations of Real-time CORBA exist, e.g. TAO which is introduced in Sec. 2.5.2.

2.5.2 TAO

The ACE ORB (TAO) is a high-performance middleware specification. It is compliant to the real-time CORBA standard and targets for applications with deterministic and statistical QoS requirements. It extends the conventional CORBA standard by the following improvements:

Real-time Interface Definition Language In contrast to the conventional CORBA IDL, TAO's IDL enables applications to meet timing requirements which are enforced from end-system to end-system.

High-perfomance Object Adapter The Object Adapter is the interface between the actual server application and the ORB. It also demultiplexes incoming client requests. The time to dispatch server operations of CORBA's Object Adapter increases with a growing number of client requests. In contrast to that this time remains constant with TAO's Object Adapter. It dispatches servant operations in constant O(1) time, regardless of the number of active connections.

Real-time scheduling TAO's ORB core supports both static and dynamic real-time scheduling strategies. The deadline miss ratio can be minimized through static preemptive scheduling strategies.

Priority-driven I/O-Subsystem TAO's I/O subsystem assigns priorities to the OS's real-time threads. So TAO has a strong influence on the schedulability of its applications in the operating system.

Because TAO is based on the CORBA standard the previously specified criteria about communication, synchronicity and name service are met. The compliants to CORBA's real-time extension provides functions for timing constraints which could retrieve information about how long it takes to access image data. However, the most prominent criterion, the support of a distributed shared memory, is not supported. So we may be able to determine memory access times with TAO but it would certainly be not so fast as it would be with a distributed shared memory system.

2.5.3 Ice

The *Internet Communications Engine* (Ice) is an object-oriented middleware implementation developed by ZeroC. ZeroC is a company from the United States employing several developers who formerly worked at the development of CORBA. Consequently Ice inherits several features and components from CORBA but more notably it also avoids several pitfalls and dead-ends which have been noticed during CORBA's development process. Ice supports language mappings for C++ and Java among others and is available under the terms of GNU *General Public License* (GPL) as long as it is not used in closed-source software. In the following some of Ice's key features are presented and compared to

CORBA in order to show its advantages. The *Specification Language for Ice* (Slice) is a file format which can be seen as Ice's IDL. According to Ices' developers (ZeroC (2008a)) it has fewer constructs than CORBA's IDL but offers a greater flexibility.

Because CORBA wants to support as many different systems as possible, many CORBA implementations suffer from this fact in terms of performance. A good example is CORBA's IIOP in which the object reference encoding prevents an efficient marshaling. ZeroC (2008a) states that the Ice protocol is designed in a simpler and more efficient way. To evaluate performance of Ice, its devlopers compared it to TAO which is known as one of the most performant CORBA-based middlewares (ZeroC (2008b)). The results show that Ice can compete easily with TAO in latency, throughput and event distribution tests, as well in networks with slow connections. But as we said earlier these tests have been run by developers of ZeroC and we did not recheck them.

According to Laukien (2005) there is and will be no support of shared memory in current or future releases of Ice. The reason for that is that the performance gain would not compensate the development effort. In particular the support of shared memory would require a fundamental change for the whole system, e.g. additional protocols and skeletons would have to be defined. These extensions don't make sense if real-time criteria like latency play the decisive role.

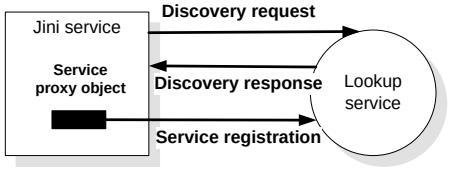
But for project ARGOS this is not the case. Latency does not play a decisive role here, but copying large data chunks does. And there is certainly a performance gain by using shared memory because it is often not necessary that data must leave the fast main memory.

2.5.4 DCOM

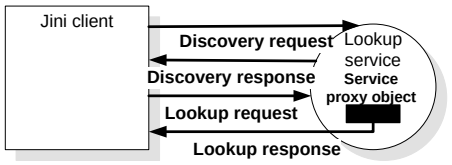
Microsoft's *Distributed Component Object Model* (DCOM) is the most widely used middleware because of the native support of Windows-PCs. DCOM is a closed standard and is developed by a relatively small number of programmers, whereas CORBA is "designed-by-committee". It can be compared to CORBA in terms of complexity and functionality and is therefore for the same reasons, namely replication, timing and complexity, not appropriate for our application.

2.5.5 Jini

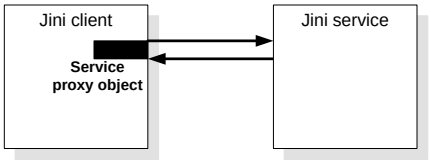
After discussing three CORBA-based middlewares (CORBA itself, TAO and Ice) and Microsoft's DCOM middleware, let us have a brief look at a different kind of architecture for distributed systems - named Jini. Jini has been originally developed by SUN Microsystems and is solely based on Java. The approach Jini takes to design distributed systems is different from other middlewares. The developers intend to make the network itself the central part of a distributed system and not the single nodes it consists of. To make this clearer Jini's architecture is described here in more detail.



(a) Registration of a Jini service.



(b) After receiving a proxy the client can make service requests.



(c) Direct communication between Jini service and Jini client.

Figure 2.7: Scenario of a client-server communication with Jini. Figures from Waldo (1999).

Jini runs ontop of the Java Remote Method Invocation using, as the name says, RMI for communication (Cha. 2.3.1.2). Jini extends the Java platform by two components: the discovery protocol, which lets an entity peer find a lookup service, and the lookup service itself, where services can register and clients can look up services.

Fig. 2.7 shows the initialization of a client-server communication. Three parts participate in this process: the Jini client, the Jini service and the lookup service. In Fig. 2.7(a) the Jini service starts a discovery request to find the lookup service. The lookup service replies with a discovery response, which includes a *service proxy object*. This object is filled with some information about the service, e.g. its ID. Then it is used to register the Jini service in the lookup service. If the Jini client wants to use a service it sends a discovery request to the lookup service to ask if the desired service is available. The lookup service can then respond with a service proxy object which allows the client to communicate with the service *directly* (Fig. 2.7(b)). So basically the only reason to bother a lookup service is to get information about the Jini service's interface, i.e. the service proxy object. Due to the fact that the client now knows the interface of the service it is able to communicate directly with it (Fig. 2.7(c)). This interface-based communication enables both client and service to communicate by way of whatever protocol they need.

Moreover the service can actually move code into the client's address space because the client communicates to the service only via its interface. The internal communication between service proxy and service can change, as well as their implementation, as long as the interface to the client remains the same. This approach makes it possible to move object-oriented programming techniques out of the process' address space onto the network. This feature distinguishes Jini from other systems like CORBA or DCOM.

The portability of code during runtime in a network with different operating systems becomes possible because of the usage of the Java programming language. Java allows the byte code the Java source code is compiled into to be moved from one machine to the other. Therefore it is possible to dynamically load and unload object code during runtime.

With the architecture provided by Jini asynchronous and synchronous communication, name service and group communication become possible. In addition *JavaSpaces*[4] allows the implementation of a distributed shared memory. It uses *ObjectSpaces* to write and take objects out of an ObjectSpace. While an entity is taking an object out of an ObjectSpace it owns it until it is pushed back into the ObjectSpace. Due to this process mutual exclusion is naturally inherited.

Waldo (1999) introduced Jini as a catalyzer for a new architecture of computing systems. The old *disk*-centric architecture with its close bound between main memory and virtual memory is supposed to be out-of-date and a new *network*-centric architecture should be introduced not at least with the help of Jini. One decade later virtual memories are still dominant and Jini is used in some applications but it has not stirred up a revolution in computer architecture.

Moreover Java hides the memory management from the programmer. Deallocation of objects and memory is done automatically by the garbage collection process running in the background. On the one hand programming becomes easier because the developer does not need to worry about memory management but on the other hand the automated garbage collection might not be as efficient as a manual garbage collection, which is necessary when programming in C++.

Due to the fact that during runtime image data and the number of concurrently running modules already use a lot of memory, we have come to the conclusion that direct influence on memory allocation is more important than comfortable programming and therefore we have chosen C++ instead of Java.

2.5.6 Conclusion of Examination

Fig. 2.8 summarizes the results we get from the examination. Despite the large variety of implementations of CORBA there does not seem to be an appropriate variant for project ARGOS. One reason is certainly its complexity and lack of real-time capabilities but the decisive point is that there is no built-in support for replication and distributed shared memory concepts.

TAO and Ice are heavily influenced by CORBA but do a far better job in real-time-enabled systems especially because of their light-weight implementations. However, the support of hard-edged real-time is not necessary for DANAOS to that extent and TAO and Ice do not support shared memory concepts. For our purpose DCOM can be put in the same category as CORBA.

[4] http://www.jini.org/wiki/JavaSpaces_Specification

	CORBA	TAO	Ice	DCOM	Jini
Sharing data / replication	*No support of replication or shared memory.*				*JavaSpaces*
Timing	*poor*	*good*		*poor*	*poor*
Communication sync./async.	*supported*				
Group communication	*weakly supported*		*well supported*	*weakly supported*	*supported*
Name service	*supported*				

Figure 2.8: List of exisiting middleware solutions. Each middleware has been tested for the necessary features Project ARGOS requires.

Jini chooses a different interesting approach to the implementation of distributed systems by using the platform-independence of Java and by supporting interface-based communication. JavaSpaces is an integral part of Jini and implements a distributed shared memory concept using ObjectSpaces. Nevertheless trading off rapid programming (Java) against less memory consumption (C++) kept me off from using Jini.

Because of the foregoing reasons we decided to implement a new middleware from scratch. There was no appropriate solution which both supports a light-weight, high-performance message passing system and a distributed shared memory system for the exchange of large amounts of data.

Chapter 3

DANAOS - A New Middleware

In this chapter DANAOS is introduced, a new middleware specialized for the use in Project ARGOS. At first an overview of DANAOS' components is given by explaining features and tasks of the *DANAOS Interface* and the *Broker*. The succeeding section describes the services being supported by DANAOS in order to fulfil the criteria stated in Chapter 2. The internal mechanisms and data structures, which are necessary to offer those services, are explained in the last section of this chapter.

All components are written in C++ and run on Windows XP machines. This combination has been chosen because C++ is a comprehensive, mature and widely-used object-oriented programming language. Windows XP supports it very well because it is firstly written in C++ itself and secondly its constituent Visual Studio is a good *Integrated Development Environment* (IDE). Of course, a POSIX-compliant operating system could have been used instead, but the camera drivers are only available for Windows systems, so the whole system is kept homogeneous by using solely Windows-operated machines.

3.1 Components of DANAOS

As described in Cha. 1 the machines of the on-board network of Project ARGOS host a set of programs for several image processing tasks (Fig. 1.2). In order to focus on the relevant internal parts of each host an excerpt of this network is displayed in Fig. 3.1. The communication messages between the programs - or modules - are always forwarded by the components of DANAOS, so there is never a direct communication between the modules. This concept ensures the compliance to the layered model we described in Cha. 1.2. In Fig. 3.1 Module a.x on Host x can communicate with Module i.x on the same host by sending a message with its DANAOS Interface to the Broker, then the Broker processes this message and forwards it to Module i.x. Is the message destined for Module a.z on Host z instead, the message is forwarded from Broker x to Broker z, which in turn

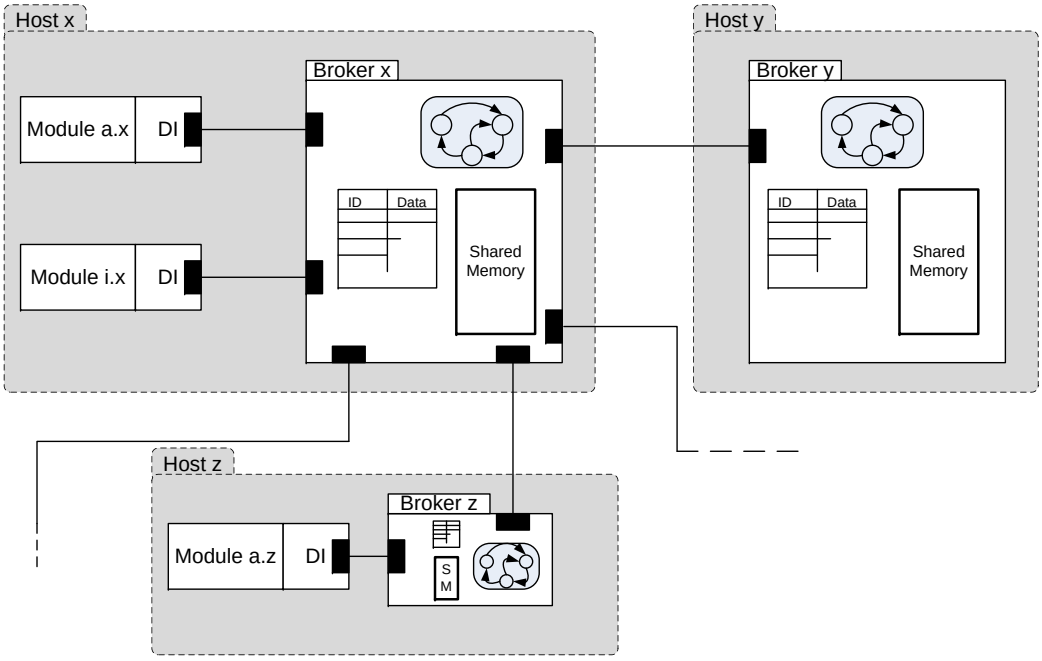


Figure 3.1: Block diagram of DANAOS. Image processing modules use their DANAOS Interface to communicate with each other via one or more Brokers.

forwards it to Module a.z.

The DANAOS Interface consists of some header files and a library. It basically offers the user functions to address modules, send messages, communicate in groups and share data. Exactly one interface is dedicated to one module.

The broker simultaneously processes requests of all modules and all brokers directly connected to it. Therefore a number of worker-threads perform several tasks like marshalling of messages, maintaining global tables (e.g. the name-service table) and controlling access to the distributed shared memory. Exactly one broker exists per host. Each of them includes several state machines running all the time in parallel and serving all connected modules. The number of these state machines depends on the concurrency value (Cha. 3.3.2).

3.2 Services Offered by DANAOS

In this section the services offered by DANAOS are introduced. DANAOS is designed to meet the needs of a middleware for Project ARGOS, so each or a combination of DANAOS's services fullfils one of the required criteria. In the following some function signatures of class `DanaosInterface` are mentioned to clarify the usage of DANAOS.

Listing 3.1: Excerpt of the header file of the DANAOS Interface showing the most important functions, which can be used to invoke DANAOS's services.

```
1  class DanaosInterface
2  {
3     ...
4     Danaos::CSocketHandler *cs_handler;
5     Danaos::MessageQueue *di_msg_queue;
6     char label[LABEL_LENGTH];
7     char domain_name[LABEL_LENGTH];
8     SOCKADDR_IN my_id;
9     ...
10
11 public:
12    int Initialize(void);
13    int RegModule(char *label);
14    int DeregModule(void);
15    ...
16
17    //functions to send and receive messages
18    bool AsynchronousSend(Message *msg_send);
19    bool AsynchronousSend(char *raw_bytes);
20    int SynchronousSend(Message *msg_send, Message **msg_recv);
21    void CheckMessageQueue(void);
22    char GetNextMessage(void);
23    ...
24
25    void Subscribe(char *service_name);
26    void Unsubscribe(char*);
27    void Publish(char*,char*);
28    void Broadcast(Message *msg_bcast);
29    ...
30    //functions to access distributed shared memory
31    int DSMWriteRequest(...);
32    int DSMReadRequest(...);
33    int DSMWrite(...);
34    int DSMRead(...);
35
36 };
```

3.2.1 Name Service

In a TCP/IP network applications are usually defined by the combination of the host IP address the application is running on and the local port number. If the application acts as a client, its port number will be randomly chosen by default, so a possible combination might be `10.0.0.5:1210`. As long as the connection is kept up, the application can be uniquely identified throughout the network.

Nevertheless the user of DANAOS longs for a more convenient way to identify his modules in the network without memorizing complicated (IP,Port) combinations, which might change as well if the connection needs to be re-established at some point. Therefore the user can give a name to each of his modules before he lets it communicate with other modules in the network. From now on we define the name of a module as a *label* in order not to mistake it for a *service name* used for the publish/subscribe service. The user can assign an arbitrarily chosen character string of at least three characters, which does not already exist on the *same* host. For communication DANAOS appends the host's *Fully Qualified Domain Name* (FQDN) to this *local label*. So if the user chooses the local label IGI for instance for a module running on host KIRK.AF.OP.DLR.DE, which is unique throughout the whole network, the label IGI.KIRK.AF.OP.DLR.DE will be assigned. This label will be called a *global module label* from now on.

Before a module can communicate with other modules and can use all network services it must be registered with DANAOS. Therefore the DI offers the following function to the user:

```
int RegModule(char *label);
```

Calling this function invokes several steps in the middleware as shown in Fig. 3.2. The function `RegModule()` is called with the parameter IGI as the label of the module. The DI, which is a static library and runs with the module itself as one process in the system, creates a message and sends it to the broker running as another process on the *same* host. There is always exactly one broker per host and a new module always makes a connection attempt to the broker running on the same host. This broker checks in one of its tables, if the name IGI[1] is already used. If not, the new global module label IGI.KIRK.AF.OP.DLR.DE is stored in the table and an acknowledgement message is sent back to the DI. Moreover the broker sends an update to all other brokers in the network to announce the successful registration of a new module. The DI parses the message and `RegModule()` returns with an error code. From the perspective of an ARGOS module the call of `RegModule()` is like an RPC. But in addition it is handled in a very efficient way in the broker as described in Cha. 3.3.2.

If a module does no longer need to communicate to other modules it can call the function `DeregModule()` to make the broker remove all entries about this module from the tables of all brokers in the system. This procedure corresponds to the procedure of registering a module.

3.2.2 Synchronous and Asynchronous Communication Service

DANAOS provides an easy-to-use set of functions for sending and receiving messages. Two different functions for an asynchronous-send command are available:

[1] IGI stands for the name of the company "Ingenieur-Gesellschaft fuer Interfaces mbH" - a manufacturer for GPS/IMU systems.

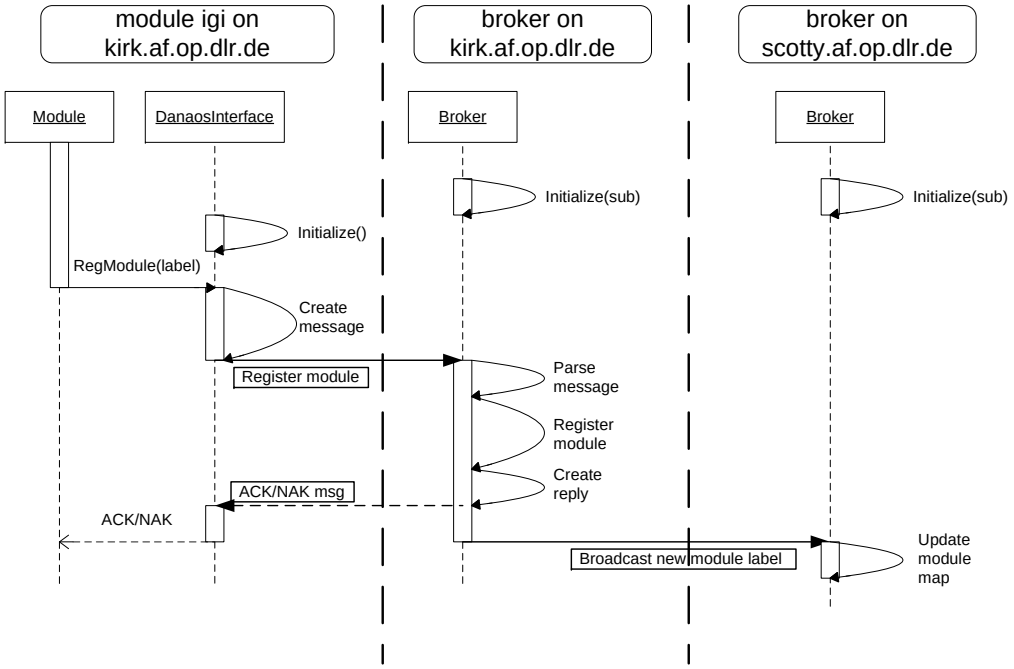


Figure 3.2: Registering a new module: This figure shows three processes, which are separated by a vertical dashed line. Process B on host X registers with DANAOS. With the help of its DI it sends a message (“Register module”) to the broker on the same host. The broker registers the module, acknowledges it (“ACK/NAK message”) and broadcasts a message to all other brokers (“Broadcast new module label”).

```
bool AsynchronousSend(char *raw_bytes);
bool AsynchronousSend(Danaos::Message *msg_send);
```

The first one takes a buffer of raw bytes. These bytes must contain a message already encoded, which includes destination address and payload, i.e. DANAOS assumes that the user knows what he is doing when using this function. It is dedicated for sending messages without creating a message object. Of course a more convenient way is the second function. It takes a pointer to a *Message Object*, which represents a message with all the header elements and transmitted data types in the payload. Class `Message` is decribed in Cha. 3.3.6.1.

Messages from other modules or brokers might be sent from a remote host but in the end the last node on the route through the network is the broker the module has connected to during its registration process. So all communication of a module is controlled by a directly connected broker. Consequently all incoming messages arrive at one socket at the DI of the module and therefore only one message queue for incoming messages is necessary.

```
void CheckMessageQueue(void);
```

```
char GetNextMessage(void);
```

If `CheckMessageQueue()` is called this function will block until a new message has arrived in the message queue of this DI. Moreover the new message is stored in the array `DanaosInterface::recv_buffer[]` for further processing. This is done by calling the function `GetNextMessage()`, which the user can call directly as well.

Synchronous communication is provided by the function

```
int SynchronousSend(Message *msg_send, Message **msg_recv);
```

The first parameter is a pointer to an object of class `Message` which contains the message to be sent. The function blocks until an answer to the sent message was received. Then the second parameter points to an object of class `Message` which contains the reply. So the user of this function only needs to provide two messages: one contains the data to be sent and the other is an "empty" message, which is filled with the received data by the function `SynchronousSend()`. The advantage of this function signature is that the user does not have to deal with the marshalling or unmarshalling of data and can access parts of the message by calling member functions of class `Message` and `MObject` respectively.

3.2.3 Publish/Subscribe Service

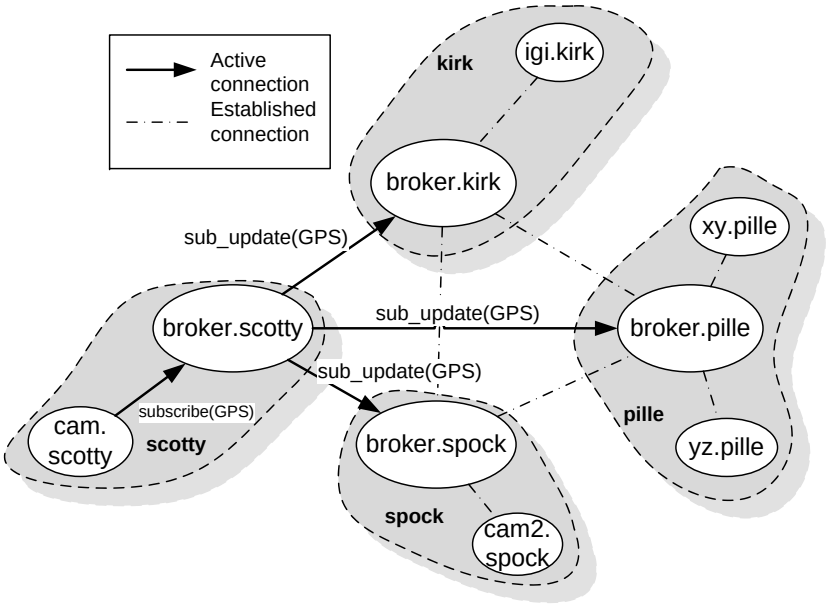
This service enables the user to reach many recipients by sending only one message. It is a form of group communication where modules can *subscribe* to certain services. As soon as a message is *published* as a service "GPS" for instance, only those modules will receive this message, which have previously subscribed to that service.

The subscription process is illustrated in Fig. 3.3(a). Module CAM.SCOTTY on host SCOTTY, supported by its DI, sends a message to BROKER.SCOTTY on the same host by calling the function:

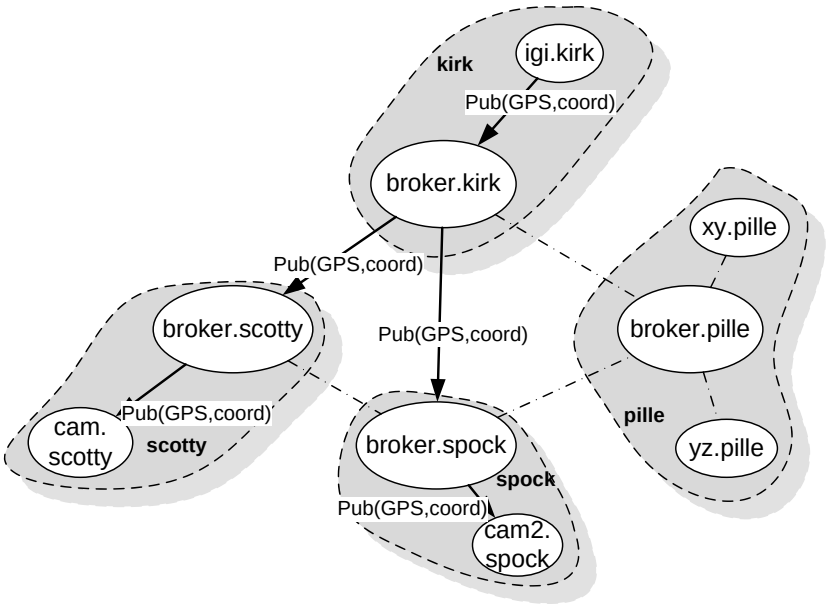
```
void Subscribe(char *service_name);
```

`service_name` points to "GPS" in this case. Under this name GPS coordinates might be published regularly. BROKER.SCOTTY stores the information that CAM.SCOTTY is now subscribed to all messages which will be published under the name "GPS" in the following. Then it broadcasts this information to all brokers in the network.

In Fig. 3.3(b) module IGI.KIRK wants to publish new acquired GPS coordinates. It sends a message of type `PUBLISH` containing the service name and these coordinates to BROKER.KIRK. Then this broker looks up "GPS" in its subscription table where every subscribed module is listed. It turns out that in addition to CAM.SCOTTY also CAM2.SPOCK is subscribed to this service. Now the only thing BROKER.KIRK does is sending two independent messages to each broker of these hosts. BROKER.SPOCK and BROKER.SCOTTY respectively forward the incoming message to the destination modules directly connected to them. Note that IGI.KIRK only sends *one* message.



(a) Subscribing to a service.



(b) Publishing data as a service.

Figure 3.3: Group communication in DANAOS. Every subscribed host is informed when a service is published. The publishing module needs to send only one message.

This distribution strategy offers several advantages. Each broker knows all subscriptions of a module, i.e. a message of type publish is analyzed by the first broker it is sent to

and then it is forwarded to only those hosts which have a subscribed module directly connected. All other hosts are not affected by this publishment. Furthermore the brokers of intermediate hosts lying on the route are not affected as well because the routing is done by the underlying operating system. On the other hand each broker must be updated as soon as a new module connects or disconnects. But due to the small number of hosts in the network and the long time a connection remains established the number of sent update messages is acceptable and should not have a serious influence on the network performance.

3.2.4 Service to Share Data

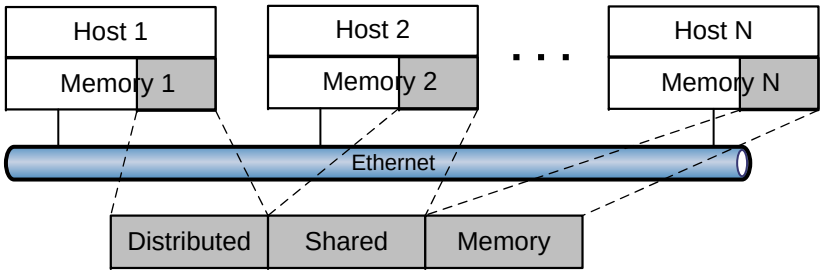


Figure 3.4: The distributed shared memory is a virtual address space, physically distributed on several hosts but logically appearing as one address space.

Because of having to exchange large amounts of data frequently between processes an efficient mechanism to share data is provided. This *Distributed Shared Memory* consists of parts of the physical memory from each participating host and is logically "fused" as one large virtual address space (Fig. 3.4). DANAOS provides the necessary functionality to conceal the physical distribution of the DSM from the user and let it work almost like a conventional memory. Of course sharing data between two processes running on the same host is faster than sharing data between two processes running on different hosts, as in the latter case the data need to be transferred after all via a network with a limited data rate. Moreover this IPC mechanism requires not quite a simple access control to avoid memory coherency problems. Memory coherency and issues on *how* the DSM works are described in Cha. 3.3.7. For now we concentrate on the *usage* of the DSM functions provided by the DI.

```
int DSMWriteRequest(
                char *dst_host_name,
                SIZE_T allocation_size,
                char **dst_addr,
                MEMORY_ID *new_memory_id
                );
```

Before you can actually access the DSM to perform some read or write operations, access must be granted by DANAOS. While the interface functions for requesting read or write

access are similar, the internal processing triggered by a write request is more complicated than that triggered by a read request, so two different functions are provided by DANAOS. `DSMWriteRequest()` requests a memory block of size `allocation_size` on host `dst_host_name`. When the function returns `dst_addr` points to the first byte of the newly allocated memory, if the physical memory is located on the same host. The user benefits from it as now he has a pointer to a memory block he can use in his program as he would use any other "local" pointer. He has exclusive write access to this memory block of size `allocation_size`. Moreover a `MEMORY_ID` is provided and is used to uniquely identify a specific memory block. It is a structure with two `DWORD` member variables: `first_page` and `last_page`. The whole distributed shared memory is subdivided in pages and the variables indicate that these two pages and all pages between these two pages belongs to this memory id. The internal funtionality is described in Cha. 3.3.7.

The actual write operation is performed with the following function:

```
int DSMWrite(
          char *dst_host,
          char *dst_addr,
          SIZE_T buffer_size,
          char *src_addr,
          MEMORY_ID mem_id
          );
```

The pointer to the memory block returned by `DSMWriteRequest()` can now be used by this function, if that is located on the same host. Alternatively the memory ID can be used to address a memory block on another host. `buffer_size` bytes of data are copied from `src_addr` to `dst_addr` where `src_addr` is a pointer pointing to a memory block in the module's address space and `dst_addr` is a pointer pointing to a memory block in the Distributed Shared Memory.

3.3 DANAOS Inside Out

The previous section has given an overview about *what* kinds of services are offered. Let us now have a look at *how* the internal mechanisms providing those services actually work. In this section internal parts like algorithms and data structures of DANAOS are described. The class diagram of DANAOS (Fig. 3.5) gives an overview of the main components of the system. Basically it can be divided into three packages of classes. `DI` is the first package, which implements classes like `DanoasInterface` itself and supporting classes like `CSocketHandler`, which handles incoming and outgoing messages. `Broker`, the second package, includes the class `SSocketHandler`, which handles all connections to all modules and brokers. It is by far the biggest class and continuously writes data entries to and reads data entries from three supporting maps, which are represented by the classes `ModuleMap`, `Router` and `SubDbase`. Class `DistributedSM` creates

and maintains the distributed virtual address space, whereas class `SharedMemory` and its derived classes provide the physical memory the distributed virtual memory consists of.`DanaosLib` is the third package and comprehends all data structures used both by classes of the `DI` package and by classes of the `Broker` package.

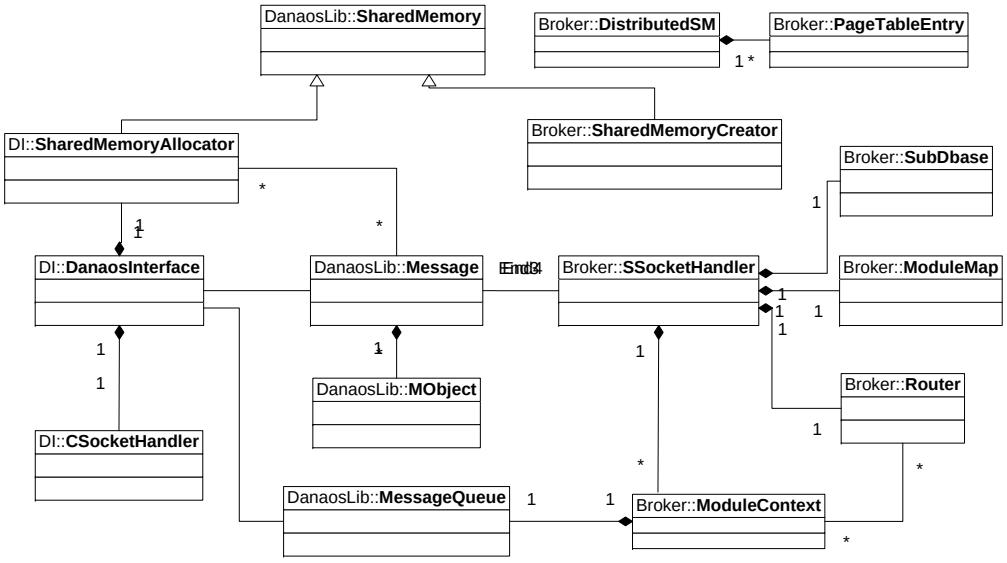


Figure 3.5: Class diagram of DANAOS. Member functions and attributes are not displayed herein.

3.3.1 Interprocess Communication

How does interprocess communication work in DANAOS? Nine mechanisms are provided by Windows for facilitating interprocess communication from which we have to choose. A comprehensive overview of these mechanisms is given on the MSDN sites[2]. We have chosen File Mapping and Windows Sockets for the following reasons:

Clipboard and its extension Data Copy do not have network support. We have already discussed the usage of DCOM in Cha. 2.5.4 and have concluded that it is not appropriate for our task. *Dynamic Data Exchange* (DDE) is considered to be not as efficient as newer technologies and therefore can be excluded as well. Mailslots and anonymous pipes only provide one-way communication. Named Pipes provide all the necessary functionality but one thread per pipe-client is necessary, which is unacceptable for our application because of reasons described in Cha. 3.3.1.1.

Remote Procedure Calls: RPCs provide a programming model with a rich API to control client/server communication. The programmer does not need to deal with the

[2]http://msdn.microsoft.com/en-us/library/aa365574(VS.85).aspx

details of network data representation, traffic, data integrity, etc. and it is portable to other operating systems.

On the contrary it sits on top of the transport layer, so we have no influence on the message structures. Moreover it is a way of synchronized communication as described in Cha. 2.3.1.2. Of course concepts like asynchronous RPC exist, but even then the client has to wait for the server's acknowledgement before it can continue to work. Microsoft does not provide any details if one-way RPCs are supported. These RPCs abandon the otherwise mandatory acknowledgements. Given that Microsoft's RPC does not support one-way RPC, each instance must acknowledge its receipt additionaly to the connection-oriented support of the transport layer (e.g. TCP). These unncessarily sent acknowledgements on top of the transport layer double the number of sent messages compared to the usage of Windows Sockets. So by using RPC our software-stack would grow by an additional layer. Microsoft's RPC would sit on top of the Transport layer, on the RPC layer DANAOS would sit and on DANAOS in turn the module applications would be located. Furthermore the programmer needs to learn Microsoft's Interface Definition Language before he can start using RPCs efficiently.

Due to the fact that we want to have a custom-tailored implementation after a short period of time with no additional software-layer, I decided to not use RPC but pure Windows Sockets instead. They are described in Cha. 3.3.1.1.

File Mapping is an efficient way of quickly addressing large amounts of data and is described in Cha. 3.3.7.1.

3.3.1.1 Windows Sockets

The Windows Sockets API or shortly Winsock[3] specifies how network software can access network services, especially TCP/IP and UDP/IP communication services. Functions and attributes are based on the Berkeley sockets API model.

Several reasons for using Winsock as a message delivery mechanism exist. The client/server model is suitable for the DANAOS architecture because the DI requests information from the broker or attempts to use offered services. Without the broker the DI cannot operate. So the DI acts like a client and the broker like a server. Winsock supports this model very well. Typical server functions like LISTEN() and ACCEPT() deal with incoming connection requests and are as well supported as typical client functions like CONNECT(), which starts a connection attempt to a known server address. So Winsock provides the functionality to reach every process in the network, it must only have a server listening for incoming connections. It doesn't matter if this server is located on the same host or not. Due to this fact you need to develop only one mechanism for both remote and local process communication.

[3] http://msdn.microsoft.com/en-us/library/ms740673.aspx

3.3.1.2 Identification of Modules

When you use Winsock another problem is solved automatically. There must be a way to identify a module througout the distributed system. Of course the global module label is an identifier but it is only convenient for the user of DANAOS but not for the internal program structure. The combination of IP address and port is ideal for internal identification processes because it is used for the same purpose by the operating system.

3.3.2 Perfomance Issues and IO Completion Ports

When we recall the number of communication peers connected to one broker (Fig. 3.1) we face an interesting design issue, which needs some additional explanation. Each communication peer (either a broker or a module) is connected to one broker by a socket. The number of established connections depends on the number of hosts in the network and on the number of running modules on its own host. In a typical ARGOS scenario each broker has to deal with 15 to 20 open sockets simultaneously. A straight-forward or intuitive way of serving all sockets would be to create one thread per socket and to apply some kind of scheduling strategy to serve all sockets in a more or less efficient way.

But this strategy, also called *thread-thrashing*, would have a serious drawback. If every socket had a dedicated thread the operating system would have to switch contexts every time it wants to serve another socket. A context switch, however, is a computationally intensive task, because the thread state and its data must be unloaded and stored in the memory and the next active thread needs to load its own state and data again. So the best strategy to increase performance is to avoid as many context switches as possible to reduce the computational overhead for the CPU.

Every major operating system has a solution for this problem. A Linux-like OS uses *Asynchronous I/O* and Windows has been using *I/O Completion Port* (IOCP) since Windows NT 3.5. It provides a set of APIs and an efficient threading model to handle multiple asynchronous I/O requests. Fig. 3.6 illustrates the way an I/O Completion Port works. After an IOCP has been created in the process every newly created socket is associated with this IOCP. When a packet from a communication peer arrives on one of these sockets it will be placed as a completion packet into a FIFO queue. Now a prior specified number of worker-threads (here: four), which have already been waiting for incoming packets, take these completion packets and process them. Each packet is processed by one worker-thread. The number of concurrent worker-threads is specified by a previously defined parameter and will almost never be exceeded. When a running worker-thread has finished the processing of one packet, it makes checks to see if there is another packet waiting in the FIFO queue - if there is then it simply grabs it and starts processing it. If not, it becomes inactive and waits in the LIFO queue for the next packet.

The main advantage of this concept is that, provided that there are constantly incoming packets waiting to be processed, a running worker-thread is not dedicated to a specific

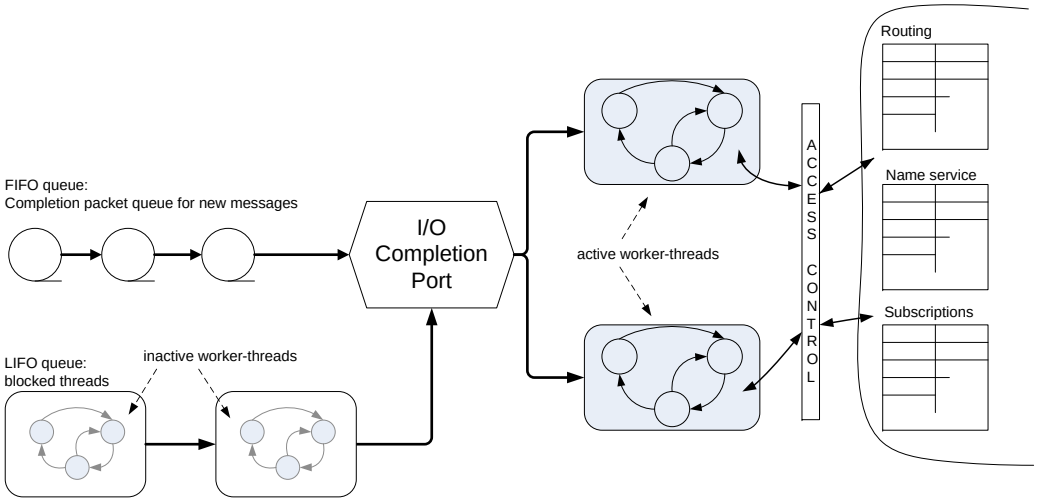


Figure 3.6: An I/O Completion Port at work. Two active worker-threads process incoming completion packets. If the concurrency value of this IOCP was increased to four the two worker-threads from the LIFO queue could join the other worker-threads and process completion packets.

socket. So when taking another package out of the queue there is no context switch, and the CPUs are utilized to near their full capacity.

The reason to let inactive threads wait in a LIFO queue is the following. When new completion packets arrive the *most recent* thread from the queue is woken up because threads that block for long periods of time can have their stacks swapped out to disk. Waking up the most recent one minimizes the work to process the in-memory footprints.

What functions are provided to use IOCP? During the initialization phase of DANAOS the function `CreateIoCompletionPort()`[4] of the Windows API is initially called with the parameter `NumberOfConcurrentThreads`, which is the concurrency value. It specifies the maximum number of concurrent worker-threads which will serve the sockets and should be chosen carefully. Microsoft's guidelines are to set this concurrency value roughly equal to the number of processors in a system.

The worker-threads are notified of new packets by the `GetQueuedCompletionStatus()`[5] function. It blocks one thread until a new packet is ready to be processed by this thread.

Another powerful function is `PostQueuedCompletionStatus()`[6]. `GetQueuedCompletionStatus()` removes a packet from the FIFO queue and `PostQueuedCompletionStatus()` adds a packet. The broker can use this function for communication between threads, like forwarding messages from one internal

[4] http://msdn.microsoft.com/en-us/library/aa363862(VS.85).aspx
[5] http://msdn.microsoft.com/en-us/library/aa364986(VS.85).aspx
[6] http://msdn.microsoft.com/en-us/library/aa365458(VS.85).aspx

socket to the next.

An in-depth coverage of IO Completion Ports can be found at (MSDN (2008)). (Russinovich (2006)) gives some backrground information as well as a description of the internals of IOCP.

3.3.3 Broker's State Machine

Each broker creates exactly one I/O Completion Port. After being accepted by the broker's listening socket each local module as well as each remote broker is registered with this IOCP. Lst. 3.2 shows a code snippet with some important states. `GetQueuedCompletionStatus()` in line 8 blocks until a new completion packet from either one of the sockets or from another thread arrives. When this function returns because of incoming data at one of the sockets this data is received by `WSARecv()` in line 23. The function gets the right socket from an object of the class `ModuleContext`. This class stores data structures regarding each socket. Important members of this class are for example the socket variable itself or a pointer to a message queue. After the message is stored in a buffer it can be parsed and further processed in one of the thread states.

`ModuleWorkerThread()` is called in a for-loop for a certain number of times during the initialization phase. Each call of this function runs in a separate thread. The number of threads is identical to the concurrency value mentioned in Cha. 3.3.2.

Listing 3.2: Code snippet of the broker's state machine. The original state machine has about 25 thread states and roughly 700 lines of code making it the most complex function of DANAOS.

```
1  void SSocketHandler::ModuleWorkerThread(...)
2  {
3  // ...a lot of local variables
4
5     while(WaitForSingleObject(g_hShutdownEvent, 0)!=WAIT_OBJECT_0)
6     {
7        //Blocks until new completion packet arrives.
8        GetQueuedCompletionStatus(completion_port,....);
9
10 // .....
11
12
13       while(thread_state != STATE_DONE)
14       {
15          //state-machine
16          switch(thread_state)
17          {
18          case STATE_INIT:
19
20 // .....
21             //current_context stores information
```

```
22              //for every socket.
23              result = WSARecv(current_context->GetSocket(),...);
24
25
26          case STATE_PARSE:
27  // .....
28          case STATE_FORWARD:
29  // .....
30          case STATE_BROADCAST:
31  // .....
32          case STATE_WRITE_FAULT_REQUEST:
33  // .....
34          case STATE_DONE:
35  // .....
36          default:
37  // .....
38          }
39          thread_state = next_thread_state;
40       }
41    }
42  }
```

3.3.4 Internal Message Handling

Fig. 3.7 illustrates how messages are handled by DANAOS during its way from the source to the destination module. Assume that module IGI.KIRK sends a set of GPS coordinates to module CAM.SCOTTY. At first the GPS coordinates, which are stored as double values in the modules' address space, are stored in a message object, serialized and copied into a send buffer. The class `CSocketHandler` of the DI sends this buffer to BROKER.KIRK via the Winsock-API, where it produces a completion packet at the broker's IOCP.

After being dequeued from the FIFO-queue by one worker-thread, the destination address is retrieved from the message. This address, which is an (IP,Port)-combination, is looked up in a table, mapping each address to a local socket. This table is represented by the class `Router`. Once the socket has been found by the broker, the message can be put into the socket's message queue. This message queue arranges messages according to its priority. The priority is stored in the message itself and is determined by the source of the message. When the message is put into the queue the function `PostQueuedCompletionStatus()` is called to put a completion packet into the FIFO-Queue. This completion packet is nothing more than a reminder for a worker-thread to actually send the message. So when the completion packet is dequeued by a worker-thread it gets the current state of this socket and sends the message to the next module.

This module is either the destination module on the same host or that broker on a dif-

ferent host where the destination module is connected to. Because we want to send the GPS coordinates to CAM.SCOTTY the message is at first received by BROKER.SCOTTY. The internal forwarding process of the message corresponds to the process just being described for BROKER.KIRK. The next step is to send the message to the destination module CAM.SCOTTY, which is directly connected to BROKER.SCOTTY and sites on the same host. The DI's socket of this module has also implemented a prioritized message queue where the incoming message is stored. Finally the module can dequeue the message by calling the DI function `CheckMessageQueue()`.

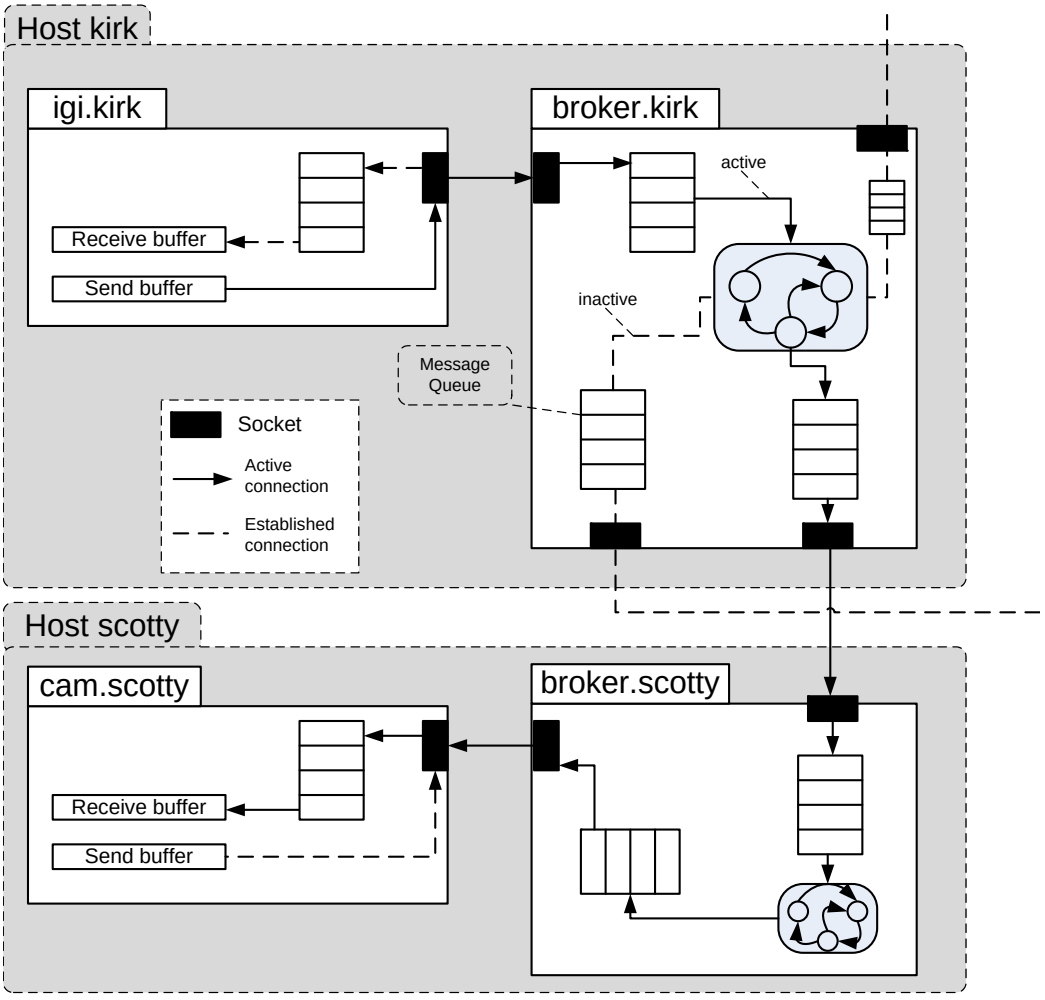


Figure 3.7: Illustration of the stages a message takes on its way from source to destination. The route which the message takes is indicated with a continous line whereas currently established but inactive routes are indicated as dashed lines.

3.3.5 Routing

The DANAOS routing service uses the information provided by the underlying operating system. Brokers and modules are identified by an (IP,Port)-combination. This combination is assigned during the TCP/IP connection establishment. Messages routed through the network are forwarded like normal IP packets and are routed directly from application to application. The only difference to other TCP/IP communication is that both source and destination applications are always DANAOS brokers and not the users' applications themselves. Intermediate hosts, where the message might be routed through, simply forward the message without any knowledge about the contents of the message according to the IP routing policy (Fig. 3.8). Of course, a router between two communicating hosts is not necessary as the messages produced and consumed by DANAOS are encapsulated as normal payload data. The resulting traffic can handled by layer 2. However, the hosts which are used in Project ARGOS are connected by a software router.

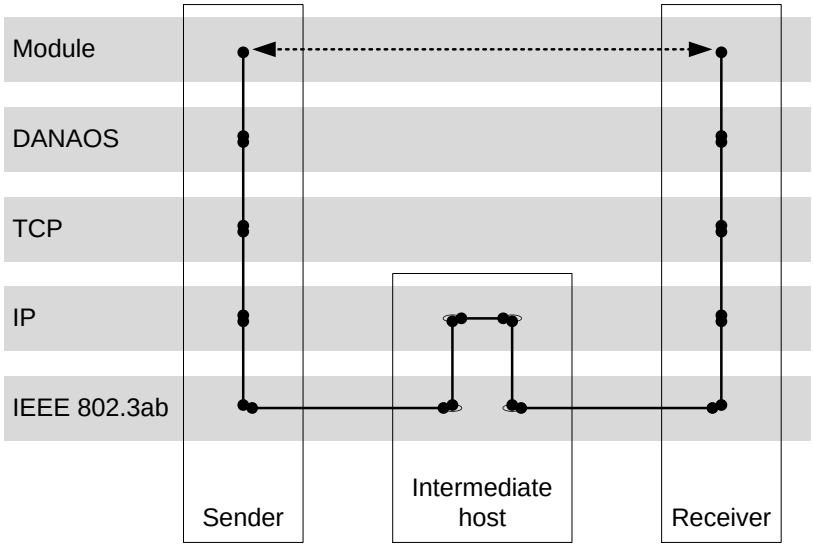


Figure 3.8: DANAOS uses IP routing mechanisms to deliver messages. Consequently brokers running on intermediate hosts are unaware of these messages.

From the perspective of the transport layer DANAOS messages are sent from the broker's source port to the broker's destination port on the destination host. The destination broker parses the message and gets the necessary information to which destination module the message has to be forwarded.

The message communication services we have discussed until now are always based on connection-oriented TCP communication. So before a message can be sent a connection has to be established. Subscription and name updates are delivered in a different way. Each broker creates during its initialization phase in addition to the TCP sockets also two UDP sockets. One socket is used to broadcast the supscription and name updates to every

broker using the networks broadcast IP address. The other socket is bound to a pre-defined port and is used to receive these updates. Unsecure UDP communication is used in this case because a TCP broadcast would produce a lot of overhead messages. Each received message results in sending back an ACK/NAK-message when using TCP. So if the number of hosts increases the number of acknowledgements will increase proportionally to it and will cause scalability problems.

3.3.6 The DANAOS Message

Category 1	Category 2	Category 3
ADMIN	SUBSCRIBE	SUB_UPDATE
SEND	PUBLISH	NAME_UPDATE
	BROADCAST	DSM_WRITE_FAULT
		DSM_READ_FAULT

Figure 3.9: The DANAOS message structure supports three different categories depending on the message type.

A DANAOS message consists of a message header and a payload, which is in turn subdivided into message objects. The header structure and the payload varies slightly depending on the type of the message. These different structures can be sorted into three categories: In the first category are messages of type `SEND` and `ADMIN` used for administrative purposes and the transfer of (a)synchronous messages respectively. Messages of type `SUBSCRIBE`, `PUBLISH` and `BROADCAST` can be found in the second category. The third category comprises all messages for internal message transfers, i.e communication between broker instances (Fig. 3.9).

(Lst. 3.3) illustrates a message structure of category 1. After the message header ending with the last byte of the field `SOURCE_LABEL` one or more message objects are transmitted. The message object type further specifies the type of the transfered information. If the message type indicates that the message is of type `ADMIN` then the message object type may indicate whether it is a message to register a module or a message to acknowledge a successful registration. Because each message object can vary in its length it is stated after the message object type and is part of each message object header. The number of message objects in a message is only capped by the overall length of a message.

Listing 3.3: Message structure of category 1. One row corresponds to 32 bits.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| MESSAGE TYPE  |         MESSAGE LENGTH        |I|   PRIORITY  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  DST_LENGTH   |                DESTINATION LABEL              |
+-+-+-+-+-+-+-+-+                                               +
|                                 ...                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  SRC_LENGTH   |                   SOURCE LABEL                |
+-+-+-+-+-+-+-+-+                                               .
|                                 ...                           |
+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+
|    MO_TYPE    |      MESSAGE OBJECT LENGTH    |               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+               .
|                      MESSAGE OBJECT PAYLOAD                   |
.                                 ...                           .
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
.                                 ...                           .
```

In the following each header field is described briefly.

Message type: Specifies the type of the message. It is used by the receiving module to identify the type of the message and to parse it accordingly. **1 byte**

Message length: Specifies the total length in bytes of the message including header, payload and terminating NULL-character. **2 bytes**

I flag: Immediate-Send flag. It is for internal processing and indicates whether a message has been aready parsed or not. **1 bit**

Priority: Specifies the priority of the message (`0x00`=low, `0x3E`=high). **7 bit**

DST_LENGTH: Specifies the length of the global module label of the destination in bytes. **1 byte**

Destination Label: Specifies the global module label of the destination as a character string. **<DST_LENGTH> bytes**

SRC_LENGTH: Specifies the length of the global module label of the source in bytes. **1 byte**

Source Label: Specifies the global module label of the source as a character string. **<SRC_LENGTH> bytes**

MO_TYPE: Specifies the type of this message object. **1 byte**

Message Object Length: Specifies the length of this message object in bytes. **2 bytes**

Message Object Payload: This is the payload of this message object. It stores the actual information of the message. Multiple message objects can be transfered one after the other in one message. **<MESSAGE OBJECT LENGTH> bytes**

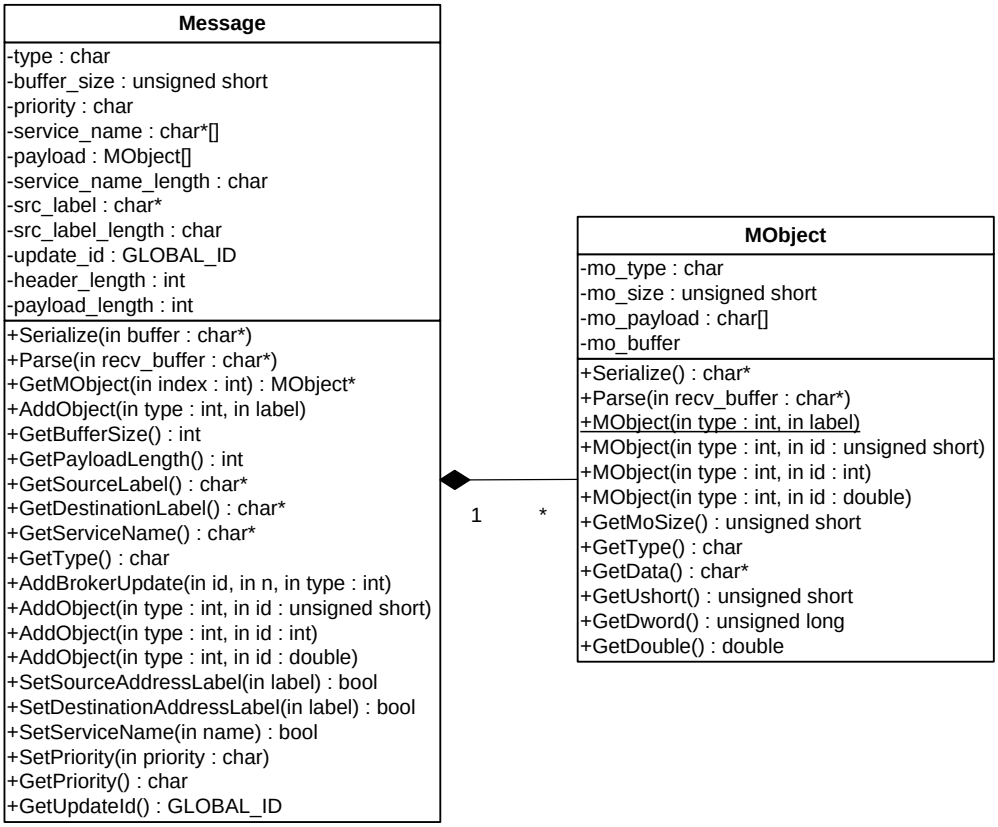


Figure 3.10: Class `Message` and class `MObject` provide a set of functions to assemble a message object.

3.3.6.1 The Message Object

Class `Message` and class `MObject` provide an object-oriented approach to handle DANAOS messages (Fig. 3.10). After creating an "empty" message object the user can choose from several functions to equip the message object with the information to be sent to the destination. For instance, message objects are added with `Message::AddObject()` the message priority is set with `Message::SetPriority()`, and so on. When the message object contains the necessary information, the module calls `Message::Serialize()` to transform the message in an array of characters.

The receiving site reads the character-array into a buffer and hands it over to the function `Message::Parse()`, which fills a prior allocated message object with the information from the buffer. Now the module can use functions like `Message::GetMObject()` to retrieve the received information.

Fig. 3.11 describes a message transmission in detail using the example of Fig. 3.2 where message creation and parsing were handled as "black boxes".

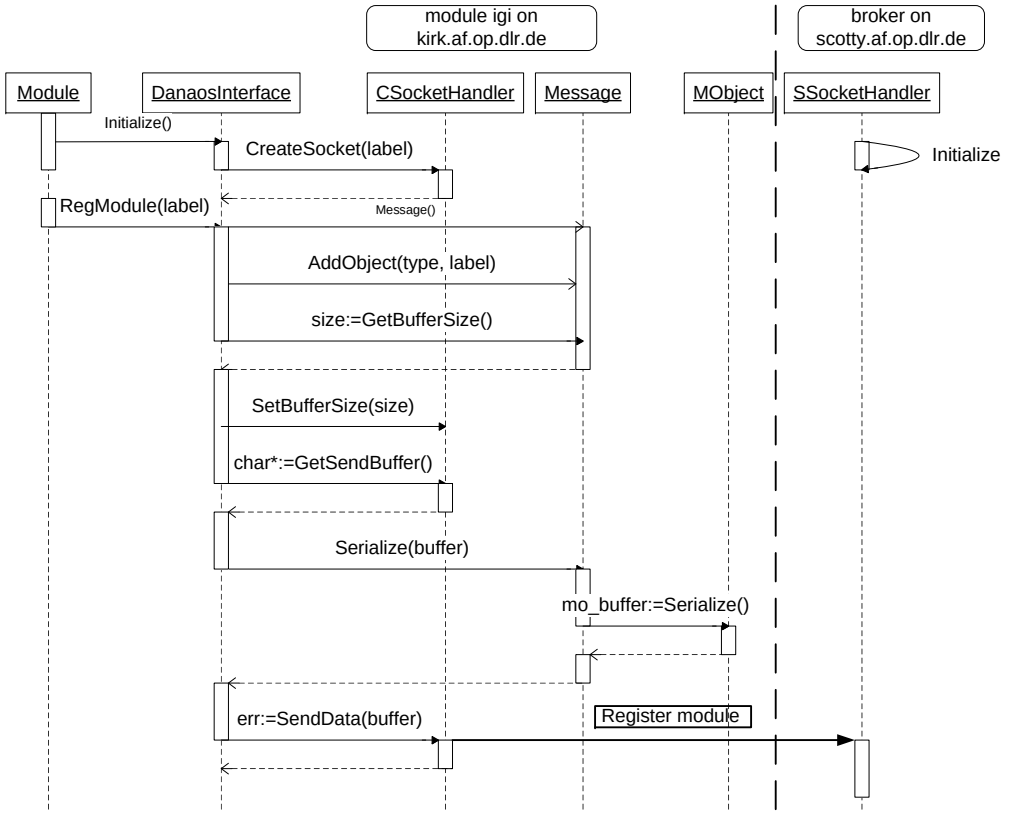


Figure 3.11: In Fig. 3.2 one processing step of an object of class `DanaosInterface` has been named “Create Message”. This step is illustrated in detail in this figure. Class `Message` and class `MObject` create a data structure to easily add and modify information whereas class CSocketHandler provides the buffer and the socket to send the information via the Winsock-API.

3.3.7 Distributed Shared Memory

The distributed shared memory is a virtual address space which is physically distributed on several hosts. The DSM being used in DANAOS consists of DSM pages of a constant size and must be allocated to the physical address space. In the following this relationship between local and distributed address space is described by firstly explaining the local shared memory concept and by secondly explaining the distributed shared memory concept.

3.3.7.1 Windows File Mapping

The Windows-API provides the FileMapping concept to create a memory, which can be shared with other processes on the same host. At first the function `CreateFileMapping()` called with a unique character-based string creates a file

mapping object. This object is used to call the function `MapViewOfFile()` which returns a pointer to the first byte of the newly mapped address space. This pointer can be used to copy data into this shared memory (e.g. by using `memcpy()`). All other processes on the same host firstly open the shared memory by calling `OpenFileMapping()` and secondly by calling also `MapViewOfFile()`. The second process must ensure that `OpenFileMapping()` is called with the same character-based string as only already created file mapping objects can be opened with this function and this character-based string serves as the object's system-wide ID.

To subdivide this shared address space into local pages `MapViewOfFile()` is called with the parameter `dwFileOffset`. Also the size of this mapped view can be specified. With these parameters it is possible to make a part of the shared memory available to the address space of the calling process. The file offset must be a multiple of the system's allocation granularity which is usually 65536 bytes.

The creation and allocaion of these file mapping objects is managed by the DANAOS broker class `SharedMemoryCreator` and by the DI class `SharedMemoryAllocator`.

3.3.7.2 Implementation of the Distributed Shared Memory

Because of the allocation granularity it makes sense to set the page size of the distributed shared memory to a multiple of the allocation granularity. It is defined with the parameter `DSM_PAGE_SIZE` in DANAOS. The first and the last DSM page on a host are defined with the parameters `FIRST_PAGE` and `LAST_PAGE`.

The crucial part in implementing a distributed shared memory is the implementation of the access control to the single DSM pages to maintain memory coherency. This area was subject to intesive research in the last two decades. A memory is said to be coherent if a read operation to a page returns the same data the last write operation has written to that page. Basically all algorithms in literature deal with the solution of this problem. For DANAOS the algorithm proposed by Li and Hudak (1989) is appropriate. It supports multiple read access to a page and single write access. In this paper the algorithm has been proven to keep the DSM coherent. The same algorithm has been evaluated and compared to other algorithms (e.g the *Hot Potato* algorithm) by Kessler and Livny (1989) and has been finally chosen for DANAOS.

Chapter 4

Evaluation

In this chapter tests of the DANAOS Middleware are minuted to give an overview of its behaviour in critical situations. Special emphasis is placed on timing measurements because the image processing modules must perform their own tasks in a certain period of time and therefore the processing rate of the DANAOS middleware must be known in advance.

4.1 General Test Set-Up

For the following test scenarios two machines have been chosen out of the ARGOS-network as shown in Fig. 1.2. They are directly connected to each other via Gigabit Ethernet (IEEE 802.3ab). The relevant technical data is also specified in that figure. Slight differences in configuration come from the different tasks of the machines in the ARGOS network. PILLE is directly connected to one camera whereas KIRK has several tasks like traffic monitoring.

In addition to the powerful CPUs and graphics cards being used, which are necessary for the computationally intensive image processing modules, the support of jumbo frames by the ethernet adapters is worth mentioning. A jumbo frame is an Ethernet frame with a *Maximum Transmission Unit* (MTU) of more than 1,500 bytes. This upper limit varies by vendor and the hosts KIRK and PILLE are equipped with three and two Intel PRO/1000 GT Desktop Adapters respectively, which all support jumbo frames of 4088 and 9014 bytes in size. This higher MTU results in greater efficiency because each IP packet can hold more payload data and it does not need to be fragmented and reassembled so often. In the specialized ARGOS environment, where large amounts of data need to be transferred frequently, the support of jumbo frames seems to be promising.

All ARGOS machines run on Windows XP SP2 with Visual Studio 2008. To measure the time on Windows operated machines as accurately as possible the so-called “High-

Resolution Timer" of the Windows-API is used by calling the following two functions in our test programs:

```
BOOL QueryPerformanceFrequency(LARGE_INTEGER *lpFrequency);
BOOL QueryPerformanceCounter(LARGE_INTEGER *lpPerformanceCount);
```

Initially the function `QueryPerformanceFrequency()` must be called to determine if the system supports a high resolution counter. If it does, the pointer `lpFrequency` points to a variable that receives the counter frequency in counts per second, otherwise the bool-return value is zero. Most often the counter-frequency is equal to the CPU's clock rate. Every time the function `QueryPerformanceCounter()` is called the variable `lpPerformanceCount` receives a pointer to a 64-bit signed integer value, which represents the absolute number of counts since the system's start-up. Whenever the function is called its value is stored and then it is possible to calculate the difference between two subsequent calls. With the help of the first function it can be converted into seconds or milliseconds, for instance. A short check on the continuity of this counter has not revealed any leaps in continuity and is therefore regarded as capable of measuring time.

4.1.1 Configuration of DANAOS

Some parameters must be given before starting the broker. For now these parameters must be set as `#define`-directives in the header files before compilation. Reading these prameters from an XML configuration file would be a more convenient way. In this case appropriate classes for parsing XML files must be provided.

- Parameter MAX_INIT_CONNECTIONS defines the number of previously started brokers in the network, which are waiting for incoming connection attempts. This can also be done automatically by sending a ping to the network's broadcast address. Assuming that all replying hosts also have a DANAOS broker running this ICMP request will reveal the IP address of each active broker in the network. This solution is sufficient for networks with a small number of active hosts (e.g. the ARGOS network) but in larger networks an ICMP request might not be forwarded by the majority of routers due to security reasons (smurf attack).
- LISTENING_PORT defines the port where the broker listens for incoming connection attempts both from local image processing modules and from brokers running on other hosts.
- The parameter BROADCAST_ADDR and BROADCAST_PORT define the broadcast address where broadcast messages must be sent to and the broadcast port every broker must listen to in order to receive broadcast messages.
- The parameter BUFFER_SIZE sets the size of the receiving buffer in bytes. It must be set high enough if jumbo-frame support is activated at the network interface

drivers.

- PAGE_OFFSET indicates the first byte of that part of the distributed shared memory which is physically located on this host.
- FIRST_PAGE indicates the first page of that part of the distributed shared memory which is physically located on this host.
- LAST_PAGE indicates the last page of that part of the distributed shared memory which is physically located on this host.

By running the command

```
Broker.exe
```

the broker is started. After initializing the local shared memory it tries to connect to already running brokers. Then it starts its own socket to listen to incoming connection attempts from modules and brokers.

The DANAOS Interface and its messages can be used by performing the following steps:

- Link to the static library `DanaosInterface.lib`.
- Include `DanaosInterface.h` in your source code.
- Call the constructor of class `DanaosInterface`.

These three steps are implemented in the test programs supplied on CD.

4.2 Measuring the Average Message Round Trip Time

During this test the *Message Round Trip Time* (MRTT) is determined. Two different aspects are to be evaluated in this test. The first one is the comparison between the MRTT and the pure RTT where no message processing takes place (Cha. 4.2.1). The second aspect is the influence of high message load on the MRTT. Therefore several modules start sending messages via DANAOS while the MRTT of one module is being measured.

4.2.1 Comparison of MRTT and RTT

The goal of the first aspect of this MRTT test is to measure the processing time a module needs to create a message, sends it to another module, receives the reply and processes it. Of course, the receiving module needs to parse and process this message as well.

Moreover the brokers of the sending module and of the receiving module also need to analyze and forward this message. This sequence of processing steps is shown in Fig. 4.1. In this context an interesting aspect is to what extent the computational overhead caused by message serialization and parsing influences the MRTT. It is compared with a measurement of the round trip time of ICMP packets.

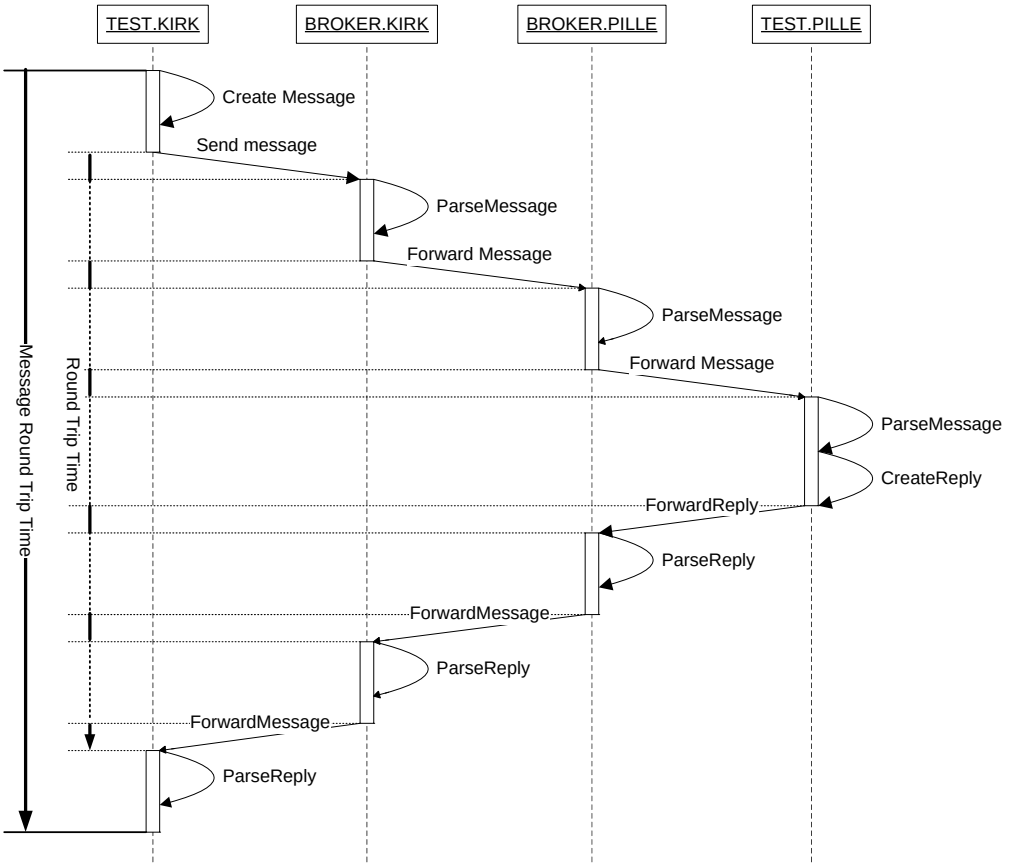


Figure 4.1: Measurement of the message round trip time. To calculate the message processing overhead the RTT must be subtracted from the MRTT.

4.2.1.1 Execution of the Test

Therefore a test program called `DummyModule.exe` can be started from the command prompt:

```
DummyModule.exe <start_delay> <my_label> <dst_label> <op_mode>
```

From the viewpoint of DANAOS it behaves like a real image processing module. `start_delay` delays the registration of this module with the broker by `start_delay`

seconds. my_label is the label of this module. dst_label is the label of the destination module where the test messages are sent to. op_mode indicates the test mode. To print a menu, choose mode 7.

At first it is important to know how long it takes DANAOS to deliver one message from module TEST.KIRK to module TEST.PILLE. To achieve this the MRTT can be measured by starting test mode 2 because in this case only one timer triggered and stopped by one module is necessary and synchronization problems caused by different timers on different hosts can be avoided.

The code snippet in Lst. 4.2 shows the relevant lines of code of this test scenario. To get the current time the function Timer::GetTime() is called. It calls the high-resolution timer described above. The test program measures the time between line 6 and line 17. Attention should be paid to the if-statements in lines 5 and 15 which cause the timer not to measure the duration of each loop iteration but of every N_LOOPS_PER_SAMPLE iteration. If this value is 50,for instance, the sum of 50 MRTTs will actually be measured. Moreover after every loop iteration the test program waits for WAIT_INTERVAL milliseconds.

Listing 4.1: Code snippet of the test program to measure the message round trip time.

```
1   utils::Timer *timer = new utils::Timer();
2   //...
3   while(!_kbhit() && loop_iter < N_LOOPS)
4   {
5      if(!(loop_iter % N_LOOPS_PER_SAMPLE))
6         timer->Reset();
7
8            //Create message objects msg_send and msg_recv
9
10     //Send and receive message
11     my_di->SynchronousSend(msg_send,&msg_recv);
12
13     //Parse message object msg_recv
14
15     if(!((loop_iter+1) % N_LOOPS_PER_SAMPLE))
16     {
17        mrtt = timer->GetTime();
18        //write into log file
19        send_test_log << mrtt << "␣";
20     }
21           //delete message objects
22     loop_iter++;
23     Sleep(WAIT_INTERVAL);
24  }
```

4.2.1.2 Evaluation of the Test

The program writes its output into the log file `MRTT_test_1.log`, which can also be found on CD. The most relevant lines are the following:

Listing 4.2: Snippet of log file `MRTT_test_1.log`.

```
#Test module label: test.kirk
#MRTT = message round trip time
#WAIT_INTERVAL 50 ms
#total number of loops 2000
#NUMBER_OF_LOOPS_PER_SAMPLE 200
#ETHERNET_FRAMESIZE: 1500 bytes
#TEST_MODE: 2
#Copied 999 bytes to test message array.

#START: 30918.972212195

30931.404347511 199 12.430646315
30943.904510302 399 12.438605886
30956.404684696 599 12.438666919
...
...
#END: 33043.966974848
```

If the test program did not wait for the period of time indicated by `WAIT_INTERVAl`, DANAOS would crash after approximately 800 sent messages of 1000 bytes in size. The lower limit of 50 milliseconds for this value has been empirically measured, i.e. if a module waits for 50 milliseconds before it sends the next message DANAOS will not crash. A possible explanation for this behaviour could be that one of the message queues in the broker or in the module is overloaded and causes succeeding messages to be discarded. This assumption can be substantiated by the fact that the minimum WAIT_INTERVAL is not correlated to the number of concurrently running modules on the system. Each module is dedicated to one socket and each socket has one message queue both in the DANAOS Interface and in the broker. So a critical "filling level" of one message queue does not affect other message queues.

Three variables are logged in three columns in Lst.4.2: system uptime, number of loop iterations and message round trip time. This value in the third column indicates the time a message needs for 200 round trips. From this value the average message round trip time can be calculated. In this way a much more accurate value will be achieved than by measuring only one round trip. Of course the `WAIT_INTERVAL` must be subtracted as well. The average MRTT is defined by the following equation:

$$Average\ MRTT = \frac{Measured\ MRTT}{NUMBER\ OF\ LOOPS\ PER\ SAMPLE} - WAIT\ INTERVAL$$

For the above values it is 12 milliseconds in average. The elapse time for an ICMP echo request on both test machines is less than 1 millisecond for packet sizes smaller than 3000

bytes. Consequently more than 90 percent of the average MRTT is caused by the message serialization and parsing steps in both the participating brokers and the modules (Fig. 4.2).

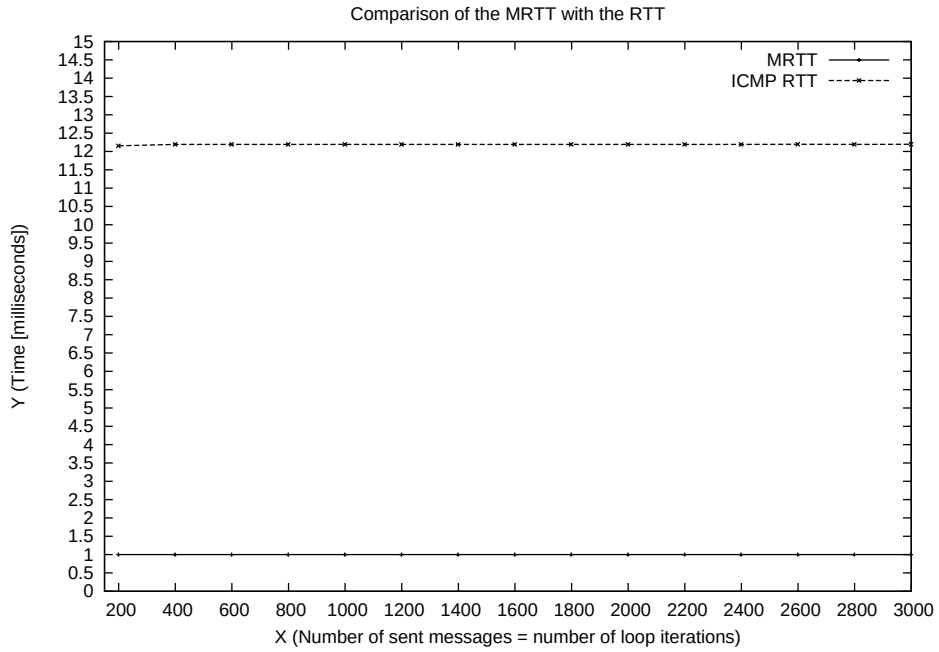


Figure 4.2: Comparison of the average MRTT with the RTT gained from the ICMP echo request.

4.2.2 Influence of Background Traffic on the MRTT

The second part of this MRTT test is to show the influence of other running modules, which consume processing time and produce network traffic.

4.2.2.1 Execution of the Test

The MRTT is measured in the same way as in the first MRTT test. In addition several instances of the test program described above run in the background and send data continuously via DANAOS. Thus the performance of DANAOS during high traffic loads can be simulated. Always two test instances send data to each other, i.e. one half of the test modules runs on host KIRK and the other half runs on host PILLE. Moreover the program Wireshark captures the data on one host, which is being sent from the relevant ports

during the test. Due to the distribution of the modules the whole incoming and outgoing traffic of DANAOS is captured by Wireshark.

From the beginning of the test every 5 seconds another module starts sending data via DANAOS. The question is if the MRTT is significantly affected by the increasing number of running modules.

4.2.2.2 Evaluation of the Test

It turns out that the MRTT is not affected. The MRTT is not correlated to the number of running modules basically because of two reasons.

Firstly Wireshark's capture file allows us to calculate the data rate of the incoming and outgoing traffic. Seven running modules per host, which are continuously sending data, produce an overall data rate of 241 Kbyte/s. This data rate is far below the maximum data rate of Gigabit Ethernet, which is about 100 Mbyte/s. Consequently the maximum data rate is only reached when about 400 modules are sending data. Before this value would be reached the participating hosts would have run out of memory. Of course, the data rate would rise significantly as soon as the wait interval of 50 ms is decreased. In this case, however, the dependence of the MRTT on the number of active modules cannot be measured, because messages are discarded in the broker. During a real flight campaign messages will not be sent at such high rates and therefore DANAOS' performance is sufficient in this case.

The second reason is the internal message handling of DANAOS. As already described in Cha. 4.2.1.2 each active socket has its own message queue. Packets which are not routed through a specific socket are not placed in their message queue either. The implementation of the class `MessageQueue` as a composition of class `ModuleContext` makes this handling possible. It supports the conclusion that the I/O Completion Port of the broker in conjunction with class `ModuleContext` provides scalability in terms of connecting clients.

4.3 Increasing Complexity and Size of DANAOS Messages

The foregoing tests have been executed with a constant total message size and the messages have been constructed with one instance of class `MObject`. Now the question arises what happens if the number of MObjects per message is increased or if the total length of a message is increased significantly. Increasing the message size is an important feature as soon as large data blocks are copied via DSM functions from host to host. Therefore two test modes have been written.

4.3.1 Execution of the Test

The first mode uses a modified `Message::AddObject()` function with an additional parameter to set explicitly the size of the payload of the thereby constructed MObject. Then it is possible to increase the number of MObjects while the total length of the message remains constant. Each MObject size must dynamically decrease with an increasing number of MObjects. In this way it is possible to meassure the time which is needed to create and parse a message without increasing the data rate which is finally sent over the network.

In the second mode the number of MObjects per message is left constant and the payload of each object is increased every test iteration. This test can be executed with the help of the modified `Message::AddObject()` function described in the previous paragraph.

4.3.2 Evaluation of the Test

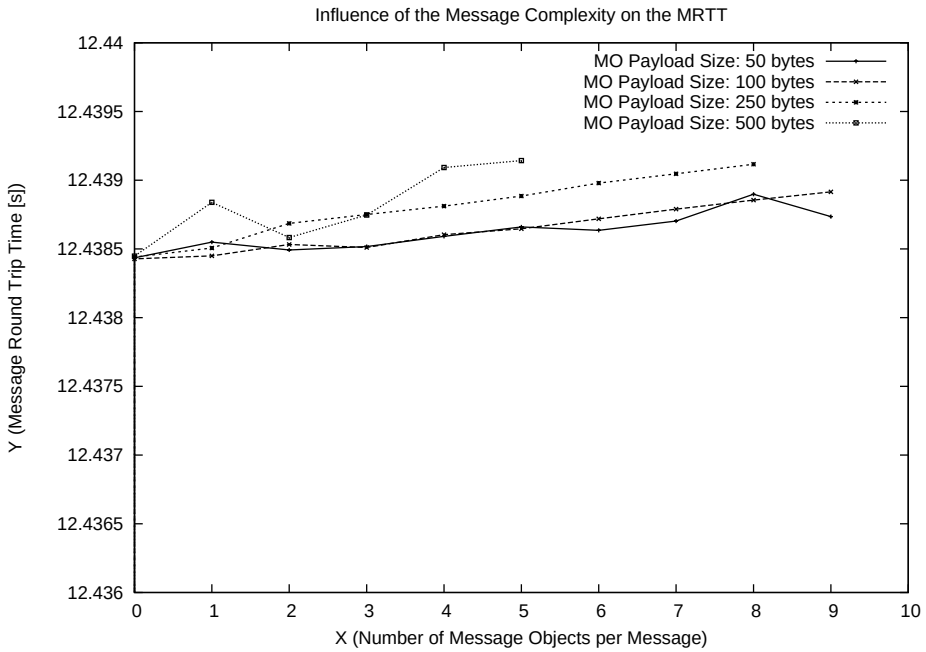


Figure 4.3: Influences of the message size and number of MObjects on the MRTT.

The test log file produces the output shown in Fig. 4.3. Instead of the `Message::AddObject()` function described above another slightly different imple-

mentation had to be used due to an unexpected behaviour for large messages. This modified `Message::AddObject()` function is frequently used during all tests and performs well. However, it can only be used for a limited number of MObjects. Nevertheless some conclusions can be drawn from Fig. 4.3. It plots the message round trip time against the number of MObjects per message. Because the used variant of the `Message::AddObject()` function does not support a dynamic adaptation of the payload size during the meassurement, the payload size in this test is constant. Each of the four test iterations has been executed with a constant payload size. As the number of MObjects increases the overall message length increases as well resulting in a bigger MRTT. This is best shown by the plot with a payload size of 500 bytes.

Chapter 5

Conclusion and Outlook

5.1 Conclusion

For a network of independent, loosely-coupled computers the middleware is the most appropriate design to transform this network into a distributed system. This software-based solution is typically implemented as an additional layer between the transport layer and the application layer in the network protocol stack.

Various already existing middleware solutions offer a large spectrum of different services. Traditional designs like CORBA and DCOM try to provide as many features as possible, which decreases performance. As a result newer middlewares offer less features but are more specialized in one or the other direction. Examples are ZeroC's Ice or TAO, which offer real-time support, or Jini, which provides an innovative concept in sharing data with JavaSpaces.

The middleware DANAOS - developed in this thesis - is specialized in another way, which has not been done before. Both the sharing of data by a distributed shared memory and fast communication via message passing is supported. The distributed shared memory is founded on an approved algorithm ensuring memory coherency and mutual exclusion. Due to the modular design the actual physical distribution of the shared memory data is for one thing realized by the Windows' file mapping concept and for another thing realized by the DANAOS message passing. The message passing system itself is based on the client/server concept and is implemented by using Windows' asynchronous I/O mechanism, called I/O completion port. This mechanism is particularly suited to process a large number of active network sockets in parallel and fits perfectly in the design of the DANAOS Broker. Finally the DANAOS Interface constitutes the API to the programmer of the application. It provides a set of easy-to-use functions like publishing messages or sharing large data blocks. These functions are specially tailored for the requirements of the image processing modules which run on the ARGOS machines and provide maximum flexibility at the same time. That means that the function signatures often allow instances

of class `Message` to be passed to the function. These objects can be easily extended and manipulated by calling one of the member functions.

The middleware has been tested on the same machines which will later be used in the ARGOS scenario. Its performance has been examined when the number of concurrently running modules or the message size increase. If these parameters are set to realistic values as they occur when sending administrative messages DANAOS performs well. In particular the achieved message round trip times and the message processing rate fulfil the near real-time criteria required by ARGOS.

During the early design phase several interesting questions arose. For instance the DANAOS broker needs to handle multiple sockets at the same time. The first approach was to simply create one thread per socket. Then one socket after the other could be served with a round-robin scheduling procedure. After some additional research the high-performance I/O Completion Port concept has been preferred.

The first idea of the distributed shared memory design was also quite simple. After a short period of time and some research it was obvious that the implementation of a self-written DSM algorithm would slow down the implementation process significantly due to the complex handling of read and write faults. For this reason one well-designed existing algorithm has been chosen and adapted for the needs of DANAOS.

5.2 Outlook

Both the implementation of a message passing system for multiple clients and the implementation of a distributed shared memory is a challenging and complex task.

One interesting aspect for further evaluation is the performance of the distributed shared memory when a varying number of read and write faults occur and how often a page has to be copied between hosts. The results could reveal several possibilities to optimize the distributed shared memory. The load level of the message queues could give further hints at shortages in the message routing. As they are a nucleus component of DANAOS it should be possible to achieve further improvements when additional performance data of these data structures are available.

For the long term diagnostic tools would be a useful extension of the middleware. Organic computing offers automatic reconfigurability, which can be used to re-arrange the processing order of the modules, and self-healing mechanisms. Because a flight campaign is costly these diagnostic tools could detect or even prevent malfunctions of running image processing modules and this way could possibly prevent the failure of flight campaigns.

Moreover extra tests and application scenarios can be constructed. For instance via an additional wireless interface the distributed system could be extended to the station on the ground or to more than one aircraft to combine the acquired image data of several aircrafts. Scalability would be the key issue in this scenario.

Bibliography

[Birrell and Nelson 1984] BIRRELL, Andrew D. ; NELSON, Bruce J.: Implementing remote procedure calls. In: *ACM Trans. Comput. Syst.* 2 (1984), Nr. 1, S. 39–59. http://dx.doi.org/http://doi.acm.org/10.1145/2080.357392. – DOI http://doi.acm.org/10.1145/2080.357392. – ISSN 0734–2071

[Chockler et al. 2000] CHOCKLER, Gregory V. ; DOLEV, Danny ; FRIEDMAN, Roy ; VITENBERG, Roman: Implementing a caching service a distributed COBRA objects. In: *Middleware '00: IFIP/ACM International Conference on Distributed systems platforms*. Secaucus, NJ, USA : Springer-Verlag New York, Inc., 2000. – ISBN 3–540–67352–0, S. 1–23

[Coulouris et al. 2005] COULOURIS, George ; DOLLIMORE, Jean ; KINDBERG, Tim ; LIMITED, Pearson E. (ed.): *Distributed Systems - Concepts and Design*. Addison-Wesley Publishers Limited, 2005

[Fleisch and Hyde 1998] FLEISCH, B.D. ; HYDE, R.L.: High Performance Distributed Objects Using Distributed Shared Memory and Remote Method Invocation. In: *Hawaii International Conference on System Sciences* 7 (1998), S. 574. http://dx.doi.org/http://doi.ieeecomputersociety.org/10.1109/HICSS.1998.649255. – DOI http://doi.ieeecomputersociety.org/10.1109/HICSS.1998.649255. – ISSN 1060–3425

[ISO 1998] ISO: *Information technology - Open Distributed Processing - Reference model: Overview*. 12 1998

[ITU 1994] ITU: *X.200 : Information technology - Open Systems Interconnection - Basic Reference Model: The basic model*. July 1994

[Kessler and Livny 1989] KESSLER, R. E. ; LIVNY, M.: An analysis of distributed shared memory algorithms. In: *Proc. th International Conference on Distributed Computing Systems*, 1989, S. 498–505

[Kurz et al. 2008] KURZ, Franz ; EBNER, V. ; ROSENBAUM, Dominik ; THOMAS, Ulrike ; REINARTZ, Peter: Near Real Time Processing of DSM from Airborne Digital Camera System for Disaster Monitoring. (2008)

[Kurz et al. 2007] KURZ, Franz ; MUELLER, Rupert ; STEPHANI, Manfred ; REINARTZ, Peter ; SCHROEDER, Manfred: Calibration of a Wide-Angle Digital Camera System for Near Real Time Scenarios. (2007). http://elib.dlr.de/48842/

[Laukien 2005] LAUKIEN, Marc: *Comment on shared memory in Ice*. http://www.zeroc.com/forums/comments/294-what-feature-would-you-like-see-most-ice-8.html#post5763. Version: 2005

[Li and Hudak 1989] LI, Kai ; HUDAK, Paul: Memory coherence in shared virtual memory systems. In: *ACM Trans. Comput. Syst.* 7 (1989), Nr. 4, S. 321–359. http://dx.doi.org/http://doi.acm.org/10.1145/75104.75105. – DOI http://doi.acm.org/10.1145/75104.75105. – ISSN 0734–2071

[MSDN 2008] MSDN: *I/O Completion Ports*. http://msdn.microsoft.com/en-us/library/aa365198(VS.85).aspx. Version: 2008

[Neuman 1994] NEUMAN, B. C.: Scale in distributed systems. In: *Readings in Distributed Computing Systems*, IEEE Computer Society Press, 1994, S. 463–489

[Rosenbaum et al. 2008] ROSENBAUM, Dominik ; CHARMETTE, Baptiste ; KURZ, Franz ; SURI, Sahil ; THOMAS, Ulrike ; REINARTZ, Peter: Automatic Traffic Monitoring from an Airborne Wide Angle Camera System. In: *ISPRS 2008 (21. Congress)*, 2008

[Russinovich 2006] RUSSINOVICH, Mark: *Inside I/O Completion Ports*. http://technet.microsoft.com/en-us/sysinternals/bb963891.aspx. Version: November 2006

[Schill and Springer 2007] SCHILL, Andreas ; SPRINGER, Thomas ; SPRINGER (ed.): *Verteilte Systeme*. Springer-Verlag Berlin Heidelberg, 2007

[Tanenbaum and van Steen 2002] TANENBAUM, Andrew S. ; STEEN, Maarten van ; HOLM, Toni D. (ed.): *Distributed Systems - Principles and Paradigms*. Prentice Hall, Inc., 2002

[Tay and Ananda 1990] TAY, B. H. ; ANANDA, A. L.: A survey of remote procedure calls. In: *SIGOPS Oper. Syst. Rev.* 24 (1990), Nr. 3, S. 68–79. http://dx.doi.org/http://doi.acm.org/10.1145/382244.382832. – DOI http://doi.acm.org/10.1145/382244.382832. – ISSN 0163–5980

[Thomas et al. 2008a] THOMAS, Ulrike ; KURZ, Franz ; ROSENBAUM, Dominik ; MUELLER, Rupert ; REINARTZ, Peter: GPU-based Orthorectification of Digital Airborne Camera Images in Real Time. (2008)

[Thomas et al. 2008b] THOMAS, Ulrike ; ROSENBAUM, Dominik ; KURZ, Franz ; SURI, Sahil ; REINARTZ, Peter: A new Software / Hardware Architecture for Real Time

Image Processing of Wide Area Airborne Camera Images. In: *Journal of Real Time Image Processing* (2008)

[Waldo 1999] WALDO, Jim: The Jini architecture for network-centric computing. In: *Commun. ACM* 42 (1999), Nr. 7, S. 76–82. http://dx.doi.org/http://doi.acm.org/10.1145/306549.306582. – DOI http://doi.acm.org/10.1145/306549.306582. – ISSN 0001–0782

[Wessels and Fomenkov 2003] WESSELS, Duane ; FOMENKOV, Marina: Wow, That's a Lot of Packets. In: - 1 (2003), S. 1–9

[ZeroC 2008a] ZEROC: *Differences between Ice and CORBA*. http://www.zeroc.com/iceVsCorba.html. Version: 2008

[ZeroC 2008b] ZEROC: *Ice Performance*. http://www.zeroc.com/performance/index.html. Version: 2008

List of Abbreviations

API	application programming interface
ARGOS	**a**irborne wide a**r**ea hi**g**h altitude m**o**nitoring **s**ystem
CORBA	Common Object Request Broker Architecture
DANAOS	**D**istributed Middlew**a**re for a **N**ear Re**a**l Time M**o**nitoring **S**ystem
DCOM	Distributed Component Object Model
DDE	Dynamic Data Exchange
DEM	Digital Elavation Model
DI	DANAOS Interface
DII	Dynamic Interface Invocation
DSI	Dynamic Skeleton Interface
DSM	Distributed shared memory
DSM	digital surface models
FQDN	Fully Qualified Domain Name
GIOP	General Inter-ORB Protocol
GPL	General Public License
HTTP	hypertext transfer protocol
Ice	Internet Communications Engine
IDE	Integrated Development Environment
IDL	interface definition language
IIOP	Internet Inter-ORB Protocol

IOCP	I/O Completion Port
IPC	interprocess communication
MOM	message oriented middleware
MRTT	Message Round Trip Time
MTU	Maximum Transmission Unit
OMB	Object Management Group
ORB	Object Request Broker
PDU	protocol data units
RMI	remote method invocation
RPC	remote procedure calls
RT SIG	Real-Time Special Interest Group
SAP	service access point
Slice	Specification Language for Ice
TAO	The ACE ORB
VDO	virtual distributed object

List of Figures

Appendix A

Class Documentation

A.1 Danaos::DanaosInterface Class Reference

Provides function to use DANAOS. Provides functions to communicate and share data with local and remote modules via DANAOS.

`#include <DanaosInterface.h>`

Public Member Functions

- int **Initialize** (void)
- SOCKADDR_IN **GetMyId** (void)
- char * **GetRecvBuffer** (void)
- int **RegModule** (char *label)
- int **DeregModule** (void)
- void **Subscribe** (char *)
- void **Unsubscribe** (char *)
- void **Publish** (char *, char *)
- void **Broadcast** (Message *msg_bcast)
- char * **GetMyLabel** (void)
- bool **CheckLabelFormat** (char **label)
- void **CheckMessageQueue** (void)

- bool **PrepareSendMessage** (char ∗dst_label, char ∗payload, Message ∗msg_send)

- bool **AsynchronousSend** (Message ∗msg_send)
- bool **AsynchronousSend** (char ∗raw_bytes)
- char **GetNextMessage** (void)
- int **SynchronousSend** (Message ∗msg_send, Message ∗∗msg_recv)
- int DSMWriteRequest (char ∗dst_host_name, SIZE_T allocation_size, char ∗∗dst_addr, MEMORY_ID ∗new_memory_id)

  Maps view of a distributed shared memory data block of size allocation_size on host host_name.

- int **DSMReadRequest** (char ∗src_host_name, SIZE_T src_size, char ∗∗src_addr, MEMORY_ID memory_id)
- int **DSMWrite** (char ∗dst_host, char ∗dst_addr, SIZE_T buffer_size, char ∗src_-addr, MEMORY_ID mem_id)
- int **DSMRead** (char ∗dst_addr, char ∗src_host, char ∗src_addr, SIZE_T buffer_-size, MEMORY_ID mem_id)

Private Member Functions

- void **GetDomainName** (char ∗global_label, char ∗domain_name)
- char ∗ **AppendHostname** (char ∗src_label1, char ∗src_label2)
- int DSMMapViewOfRemoteFile (char ∗host_name, SIZE_T allocation_size, char ∗∗p_remote_base_addr, MEMORY_ID mem_id)

  Maps view of a shared memory data block, located on a remote host.

Private Attributes

- CSocketHandler ∗ **cs_handler**
- MessageQueue ∗ **di_msg_queue**
- SharedMemoryAllocator ∗ **sm_accessor**
- HANDLE **mq_update**

- HANDLE **shutdown_event**
- boost::mutex **di_msg_queue_mutex**
- char **label** [LABEL_LENGTH]
- char **domain_name** [LABEL_LENGTH]
- SOCKADDR_IN **my_id**
- char **recv_buffer** [BUFFER_SIZE]

A.1.1 Detailed Description

Provides function to use DANAOS. Provides functions to communicate and share data with local and remote modules via DANAOS.

A.1.2 Member Function Documentation

A.1.2.1 int Danaos::DanaosInterface::DSMWriteRequest (char * *dst_host_name*, SIZE_T *allocation_size*, char ** *dst_addr*, MEMORY_ID * *new_memory_id*)

Maps view of a distributed shared memory data block of size allocation_size on host host_name.

Parameters:

char *host_name specifies host where data shall physically be stored.

Returns:

If the function succeeds, the return value is zero.

The documentation for this class was generated from the following file:

- Module/DanaosInterface.h

A.2 Danaos::Message Class Reference

Represents a message for communication between modules. A message object holds all fields of the message as attributes and provides functions to easily read and modify them. Message between modules and between brokers are represented by message objects. For sending a ready-made message it provides a serialization function and also for parsing a received buffer from a socket, it provides a function for parsing.

```
#include <Message.h>
```

Public Member Functions

- Message (void)

  Constructs a message object for received message data.

- Message (unsigned)

  Constructs a message object for sending.

- ∼Message (void)

  Destructs a message object.

- SOCKADDR_IN GetUpdateId (void)

  Retrieves a GLOBAL ID from a message object.

- int GetBufferSize (void)

  Calculates buffer size for a new message object. Must be called before Serialize() function to calculate the correct size for the sending buffer. The sending buffer is provided by either a CSocketHandler object in the DANAOS Interface or by a ModuleContext object in the broker.

- void SetPriority (char priority)

  Sets the priority of this message The specified priority must be a value of the interval [0x01..0x3F]. The higher the value, the higher the priority.

- char GetPriority (void)

  Gets the priority of this message The higher the value, the higher the priority.

- bool SetServiceName (char ∗)

  Sets the service name which either stores a service name of a subscription or a module label.

- bool SetDestinationAddressLabel (char ∗)

  Sets the module label of the destination.

- bool **SetSourceAddressLabel** (char ∗)
- void **AddObject** (int mo_type, char ∗label)
- void **AddObject** (int mo_type, int id)
- void **AddObject** (int mo_type, double id)
- void **AddObject** (int mo_type, unsigned short id)
- void AddBrokerUpdate (SOCKADDR_IN id, char ∗n, int type)

  Adds subscription or name service information to a message object. Adds one of two different kinds of a broker update to a message object. 1. Adds a label of a newly registered or recently removed module with its corresponding SOCKADDR_IN to a message object. 2. Adds a service name with the GLOBAL ID of the module which wants to subscribe/unsubscribe to/from a service.

- char GetType (void)

  Gets the message type.

- char ∗ GetServiceName (void)

  Gets a service name. The returned service name identifies a service a module can (un)subscribe to/from.

- char ∗ GetDestinationLabel (void)

  Gets the label of the destination module. The destination label is used by the owning broker to determine the destination module.

- char ∗ GetSourceLabel (void)

  Gets the label of the source module. The returned source label identifies a module which has sent a message.

- int GetPayloadLength (void)

  Gets the number of message objects in a message.

- MObject ∗ GetMObject (int index)

  Gets a pointer to a message object.

- void Serialize (char ∗send_buffer)

  Transforms an object of class message into a byte stream. Must be called with a pointer to an already allocated byte buffer which has the correct size to hold the whole message object. When the function returns, send_buffer stores the message which is ready to be sent.

- void Parse (char ∗recv_buffer)

  Transforms a byte stream into an object of class message. Must be called with a pointer to an already allocated byte buffer which has the correct size to hold the whole message object. When the function returns, this message object stores all parameters and message objects. The members can accessed by the appropriate functions.

Private Attributes

- char type

  Type of the message.

- unsigned short buffer_size

  length of the whole message buffer in bytes.

- char priority

  Priority of the message.

- char ∗ service_name

  Pointer to the name of the service which a module wants to publish or subscribe to.

- char ∗ src_label

  Pointer to the label of the source module.

- SOCKADDR_IN update_id

  IP address and port of a {NAME|SUB}-update.

- int header_length

  Header length in bytes. The header of an Object is not included.

- int payload_length

  Number of objects in this message.

- char service_name_length

  Length of service name or of destination module label.

- char src_label_length

  Length of source module label.

- MObject ∗ payload [10]

  Array of pointers to message objects. Each message object contains a header and payload.

A.2.1 Detailed Description

Represents a message for communication between modules. A message object holds all fields of the message as attributes and provides functions to easily read and modify them. Message between modules and between brokers are represented by message objects. For sending a ready-made message it provides a serialization function and also for parsing a received buffer from a socket, it provides a function for parsing.

A.2.2 Member Function Documentation

A.2.2.1 void Danaos::Message::SetPriority (char *priority*)

Sets the priority of this message The specified priority must be a value of the interval [0x01..0x3F]. The higher the value, the higher the priority.

Parameters:

char priority this message is set to.

A.2.2.2 char Danaos::Message::GetPriority (void)

Gets the priority of this message The higher the value, the higher the priority.

Returns:

The priority of this message.

A.2.2.3 void Danaos::Message::AddBrokerUpdate (SOCKADDR_IN *id*, char * *n*, int *type*)

Adds subscription or name service information to a message object. Adds one of two different kinds of a broker update to a message object. 1. Adds a label of a newly registered or recently removed module with its corresponding SOCKADDR_IN to a message object. 2. Adds a service name with the GLOBAL ID of the module which wants to subscribe/unsubscribe to/from a service.

Parameters:

SOCKADDR_IN id of module which wants to (un)subscribe or (de)register.

char* n Points either to a service name or a new module label.

A.2.2.4 char Danaos::Message::GetType (void)

Gets the message type.

Returns:

the type of the message.

A.2.2.5 char* Danaos::Message::GetServiceName (void)

Gets a service name. The returned service name identifies a service a module can (un)subscribe to/from.

Returns:

the pointer to a character array which stores the label of the destination of this message.

A.2.2.6 char* Danaos::Message::GetDestinationLabel (void)

Gets the label of the destination module. The destination label is used by the owning broker to determine the destination module.

Returns:

the label of the destination module

A.2.2.7 char* Danaos::Message::GetSourceLabel (void)

Gets the label of the source module. The returned source label identifies a module which has sent a message.

Returns:

the pointer to a character array which stores the label of the source of this message.

A.2.2.8 int Danaos::Message::GetPayloadLength (void)

Gets the number of message objects in a message.

Returns:

the number of message objects in a message.

A.2.2.9 MObject* Danaos::Message::GetMObject (int *index*)

Gets a pointer to a message object.

Parameters:

int index specifies the object with this index in an array of message objects

Returns:

a message object.

A.2.2.10 void Danaos::Message::Serialize (char * *send_buffer*)

Transforms an object of class message into a byte stream. Must be called with a pointer to an already allocated byte buffer which has the correct size to hold the whole message object. When the function returns, send_buffer stores the message which is ready to be sent.

Parameters:

char* send_buffer points to the first byte of the buffer of the serialized message object.

A.2.2.11 void Danaos::Message::Parse (char * *recv_buffer*)

Transforms a byte stream into an object of class message. Must be called with a pointer to an already allocated byte buffer which has the correct size to hold the whole message object. When the function returns, this message object stores all parameters and message objects. The members can accessed by the appropriate functions.

Parameters:

char* recv_buffer points to the first byte of the buffer of the received message.

A.2.3 Member Data Documentation

A.2.3.1 char* Danaos::Message::service_name [private]

Pointer to the name of the service which a module wants to publish or subscribe to.

It is also used to store the label of the destination address. Name of a service to subscribe to.

The documentation for this class was generated from the following file:

- DanaosLib/Message.h