DYNACORE Project

DYNACORE Final Report
Plasma Physics Prototype

Ref: D1.1
Version: DRAFT **Date: Sept 2000**

CEC Project: RE 4005 (RE)
Telematics for Research
Telematics Applications Programme

To this report contributed:

	Institution	
W. Lourens	UU	Editor
B.U.Nideröst	UU	
A. Taal	UU	
B.A. Andree	UU	
H. Blom	UU	
E. van der Meer	UU	
A. Gerritsen	UU	
P. van Haren	UU	
C.T.A.M. de Laat	UU	
M. Korten	FZJ	
G. Kemmerling	FZJ	
C. Fuchs	FZJ	

Contents

Preface

In the following chapters we will describe the DYNACORE Plasma Physics prototype (PPP). To this end we will pay attention to:

- The experimental physics environment and user community,
- The existing electronic network situation between the co-operating institutes and at the local and remote sites,
- The local (host) experimental facilities and procedures globally,
- The remote facilities and mission,
- Outlines of the DYNACORE architecture, already described in earlier deliverables,
- A description of the target experimental set-up to be controlled remotely,
- A description of the control prototype,
- The architecture for remote data retrieval and analysis,
- Implementations of the data retrieval system
- Security issues
- Performance measurements,
- Audio and video connectivity,
- Dissemination results

For reasons outlined in the sequel the emphasis in the PPP was laid at the data retrieval part, the control aspects were unfortunately given less attention. As for the data retrieval part we can speak of a marketable product, whereas the control part is still a prototype.

> The contents of the chapters on “Performance measurements”, “Network performance aspects” and “Audio and video connections” don’t relate directly to commitments for the Dynacore work packages, but is included to complete the setting of the project.

Plasma Physics communities.

The start of the DYNACORE project coincided with the actual outset of the formal collaboration between plasma physics research institutes in Germany(-Rhineland-Westfalia), the Netherlands and Belgium.

This effects to plasma physics research to be carried out in the framework of the Euregio Cluster TEC (Trilateral Euregio Cluster) at the Institut fuer Plasma Physik (IPP) located in the Forschungs Zentrum Juelich (FZJ) in Juelich, Germany. The partners will accommodate their experimental set-ups at IPP, which houses the experimental plasma fusion device TEXTOR-94[1].

German and Belgium plasma physics groups already were used to carry out their experiments at the IPP-site, but the Dutch group had its own facilities in Nieuwegein (near Utrecht), which facilities ceased to end, effectively end 1998. The Dutch group (FOM group, see also “Remote partner and validation site”) however already co-operated with IPP for a long time in mutual experiments. The Dutch group moved part of its equipment to the IPP site (viz. the so called Pulsed Radar Reflector Diagnostic, to be described further on: “The pulsed Radar Reflector Diagnostic”). In this sense the FOM group was an ideal target for remote operations, and to adapt the DYNACORE architecture to their requirements and establish a platform for validation of the products.

There are however several circumstances that hamper the remote operations of large experimental facilities, like big accelerators, tokamaks and reactors:

- Experimental facilities are usually not designed for routine operations. Which means that always some technicians and scientists have to be on-site to prepare and maintain the set-ups. Usually is it is quite arbitrary which part of the scientists have to be on-site and which part can stay in the home institute. Sometimes it is more efficient to move the greater part of the staff to the experimental site. In this case not any architectural design for remote operation can offer enough facilities to prevent the removal of the majority of the scientific staff from the home institute and remote operation is a very scarcely used phenomenon.

- Even very complicated experiments, as are conducted in the plasma physics research, need a kind of “feel and look” from the scientists point of view. Normally all big experiments are conducted more or less remotely, since radiation and other hazards prevent the scientist to come to close to a working set-up (a tokamak). But it is mostly not too difficult to observe things closely, not only by closed circuit (TV) surveillance, but also by eye. It is rather difficult to include this “feel and look” in remote operations, but one can try.

- Remote operations require at least sufficient capacity of the interconnecting networks. If requirements to this end are not met sufficiently, one can forget about remote operations at all, because the scientists are not interested. For this reason in the DYNACORE project we paid much attention to the behaviour of the (international) connectivity between the collaborating institutes.

[1] TEXTOR-94: see http://www..fz-juelich.de/ipp/

From the arguments described above one can extrapolate that remote control of an experimental set-up whenever feasible will only be used reluctantly. Usually it will not be the prime option for a scientist (or boards of scientists that govern these large facilities). One can observe this fact with rather all the large facilities of physics research. (e.g. CERN, JET, etc. in Europe, but also in the USA, FERMI lab, etc).

Yet another point is the remote analysis of the data that originate from the experiments. There are two reasons for favouring at least the remote processing of data.

1) There does not exist so much as a notion amongst scientists about local and/or remote data. The physical whereabouts of data are always more or less remote, so it does not matter if it is stored and/or analysed locally (at the site) or remotely. However in some scientific communities (e.g. CERN) there is a strong feeling about a centralised place for archiving of raw (experimental) data. This should always be done at the experimental site. Maintenance for a longer period can be guaranteed better in this way.

2) Usually there are never enough computer facilities locally, at the experiment's site. So one has to divert the computational workload anyway to the home institutes. In communities with a centralised storage of the raw data this will be accomplished via data replication. Processed data will usually not be included in the centralised storage, but can be kept at the (remote) home institutions. This could give rise to the so-called computational grids[2].

The last observations have lead us to put the emphasis on remote processing of experimental data, and to the distributed storage and retrieval of data. The remote operation was implemented in the DYNACORE plasma prototype, but mainly as a demonstration of the possibilities of the proposed architecture, as a kind of template, more than as a routine tool.

Wherever scientific communities collaborate in large experimental facilities it is obvious that the participants start from different positions, especially with respect to experimental procedures, but certainly also for data formats, etc. If the collaboration exists for many years (as is e.g. the case for the CERN communities) then gradually a common pattern for data formats, data reduction and analysis tools will evolve. This is not a fast process, but takes several years. Since the plasma physics communities co-operating in the TEC collaboration are only recently coupled, there doesn't exist at this moment a common attitude towards data formatting, data storage etc. With the exception of the user community of JET[3]

[2] The grid, blueprint for a new computing infrastructure, I. Forster and C. Kesselman, Morgan Kaufman publ. inc. 1999

[3] V. Schmidt, The development of the JET Control and data acquisition system, IEEE Trans. Nucl.Sc. 45 (1998), 2026

V. Schmidt, The development of the JET Control and data acquisition system, in C.E. Vandoni, Xth IEEE Real-Time Conference, Beaune, 1997, 25-30

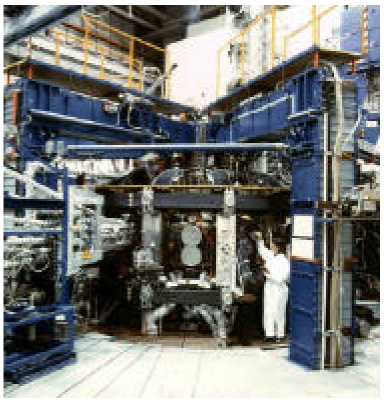


Figure 1 TEXTOR '94,Tokamak and auxiliary equipment

Introduction

The generation and behaviour of plasma in a fusion device and its interaction with surrounding materials is studied by observing several phenomena that will accompany a plasma discharge. These phenomena are recorded by means of so called *Diagnostics*. These are instruments that comprise complex electronic equipment, coupled to various sensors. The generation of the plasma is also governed by electronic systems that control different parameters of the fusion device, the *Tokamak*, and of auxiliary equipment. Ideally there are two separated channels that are used to (see also Figure 3):

1. Control (monitor, read and set) the adjustable parameters of the electronic systems,

2. Perform the reading of data that are collected in the instrument during an experiment.

The first issue is dealt with in the chapter " The DataViewer."; the second will be treated in chapters "Data retrieval architecture "; "The DataManager, ObjectManager and Launcher"; and "Performance measurements".

The experimental fusion device. Tokamak TEXTOR '94

At IPP's premises a large research facility is available in the tokamak TEXTOR-94. (Figure 1)

In TEXOR-94 high temperature plasmas can be generated in a pulsed fashion (see Figure 2, the drawing is not to scale). The periods between pulses, including the pulse itself, are divided in 4 intervals as depicted in the figure. In the first interval (T_1, T_2) decisions about settings of the tokamak and diagnostics have to be made, in the second interval, the pre-pulse (T_2, T_3) the new settings have to be applied, the system is armed. During the pulse (T_3, T_4) data has to be acquired and stored. Data will be analysed subsequently in the fourth interval, the post-pulse (T_4, T_5). After this the cycle can be repeated.

(T_3, T_4) is in the order of seconds, while the other intervals range up to minutes. The whole cycle constitutes a state machine; its states and transitions have to be incorporated in the (remote) control. At least the onset of the pre-pulse, pulse and post-pulse have to be communicated (broadcasted) to all diagnostics participating in a particular experiment. This communication will be no matter for the *remote* control, but will be dealt with locally by a suitable circuitry.

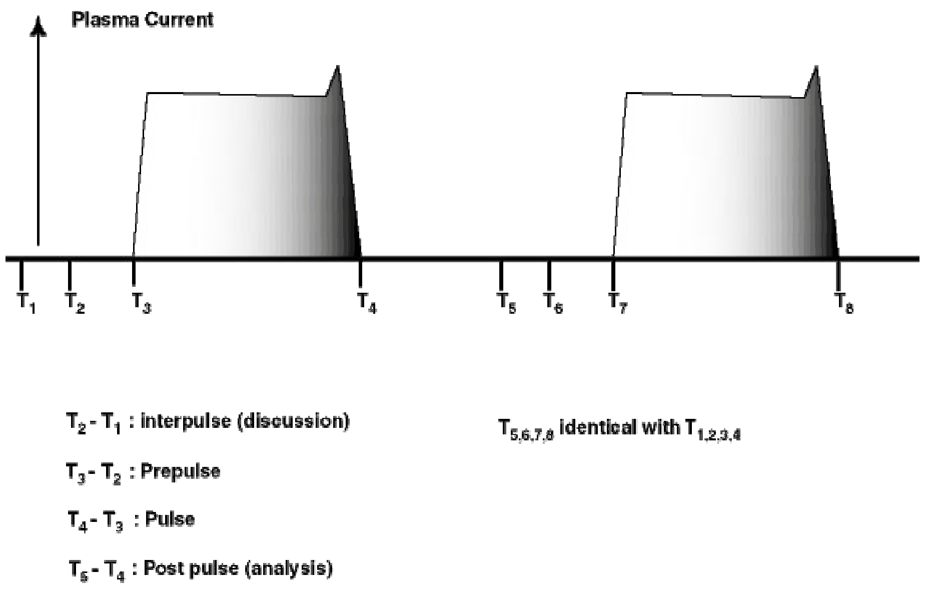


Figure 2: Pulsed operation of the tokamak and auxiliary equipment (diagnostics)

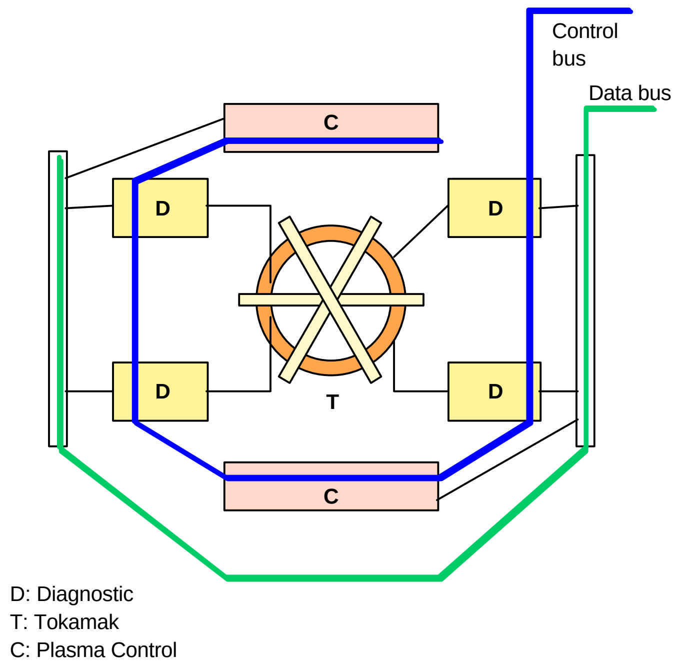


Figure 3 Data and control paths in fusion experiments (see also Figure 6)

So TEXTOR-94 is surrounded by diagnostics as depicted schematically in

Figure 3. The task of operating the large research facility is to control and monitor both the state of the tokamak and of the diagnostics.

In this context IPP (TEXTOR-94) functions as the local (large) experimental facility. Collaborating institutes could either operate their instruments at the IPP site (locally) or do it remotely. In the DYNACORE project IPP (FZJ) is a partner in the development of the teleoperation system (DYNACORE System). One of the TEC partners, FOM-Rijnhuizen, functions as the remote site, where the prototype has be implemented and tested.

The use of the DYNACORE System in the plasma physics environment

Traditionally the research facilities (TEXTOR-94 and surrounding diagnostics) are operated at the site of IPP. In the context of TEC at least one of the larger diagnostics will be operated remotely from the remote site at Nieuwegein in the Netherlands. The Dynacore prototype will be designed and implemented to facilitate this intention. The global requirements for this remote control are:

- Operation of the diagnostic. (This diagnostic, Pulsed Radar Reflector, will be described in the chapter: “Experiment to be controlled remotely”). Operation means in this respect total remote control of the settings of the instrument and complete control over the data acquired during a shot (or in offline tests).
- Remote access to status and some settings of the tokamak and supplying instrumentation.
- Remote access to some of the data of other diagnostics
- Audio / video connectivity in order to facilitate deliberations between scientists and engineers at the remote site and local operators, scientists and technicians.

The last issue will be covered in the description of the recommended AV-applications in chapter: “Audio and video connections”. The first requirement will be covered in the description of the diagnostic Pulsed Radar Reflector and its remote operation in the sequel. The second and third requirement will be covered in principle by the description of the DYNACORE System architecture that will be used for this purpose and will be described in following paragraphs.

Remote partner and validation site

The FOM Institute for Plasma Physics ‘Rijnhuizen’[4] is one of the research institutes of the Foundation for Fundamental Research on Matter, FOM. The institute is usually referred to as ‘Rijnhuizen’, the name of an 18th-century mansion and of the surrounding estate, which forms the institute’s premises since its foundation in 1959.

FOM is a government-supported organization with an annual budget of about 150 million DGL (approximately 65 M$), which co-ordinates and stimulates most of the fundamental physics research in the Netherlands. The funding of FOM is provided for a large part by the Netherlands Organization for Scientific Research (NWO). NWO is the central Dutch organization in the field of fundamental and strategic scientific research.

A major mission of Rijnhuizen is to contribute to the European research programme on nuclear fusion, a programme that is co-ordinated and financially supported by Euratom via an Association Agreement. The Association Euratom-FOM is also responsible for the Dutch contribution to the European Fusion Development Agreement (EFDA[5]) signed 30 March 1999. The EFDA Agreement is part of a long-term programme of co-operation covering all the activities in the field of controlled thermonuclear fusion by magnetic confinement in the European Union and in the Swiss Confederation.

Its aim is to provide a framework for implementing research; development and design work in preparation for the possible construction of an experimental fusion reactor. The Dutch participation in the development of reactor technology is carried out by the Nuclear Research and Consultancy Group NRG[6].

The institute has the disposal of a versatile computer and network infrastructure, comprising of primarily UNIX-based systems. These are used, clustered and stand-alone, in the remote participation.

[4] http://www.rijnh.nl/
[5] http://europa.eu.int/comm/research/fusion/efda.html
[6] http://www.nrg-nl.com/general/index.html

Network situation

Introduction

The remote operation of experiments at –experimental- , fusion devices requires appropriate bandwidth between collaborating institutes. In the course of the project REMOT[7], preceding DYNACORE, these requirements were tentatively formulated as ought to be at least equivalent to normal Ethernet conditions (10Mbit/s), which exist at the partners' institutes. Since we have decided that the connections between these partners should use the offered Internet facilities, these requirements should at least be met by the available capacity of the backbone TEN-155 infrastructure (Figure 4).

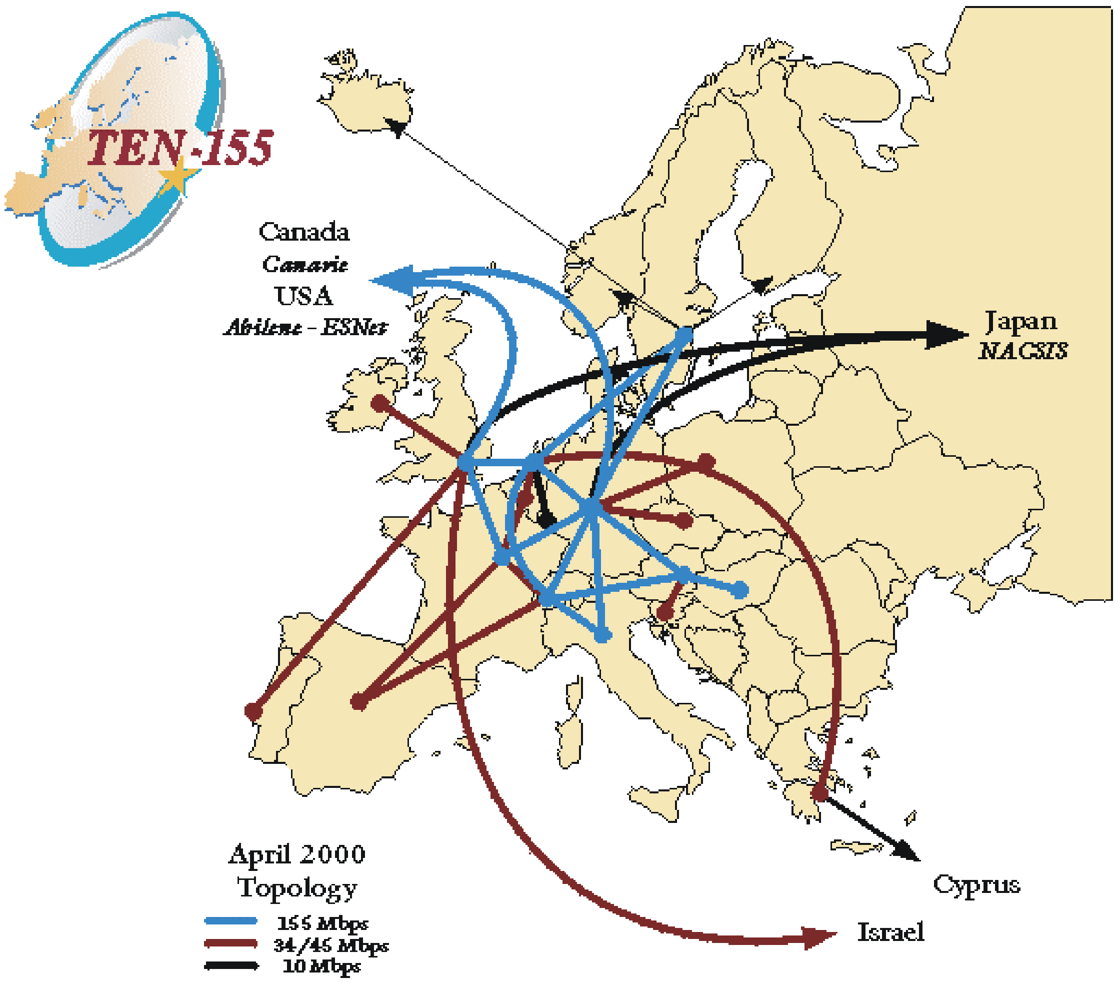


Figure 4

Network connections

The TEN-155 backbone is a part of the total network infrastructure that is utilised in the final stages of the DYNACORE project.

[7] REMOT: TAP (in Research) -EU program, RE 1008

In the Netherlands both Utrecht University (UU) as the FOM Institute Rijnhuizen (FOM) are coupled to the Amsterdam PoP, at this moment by nominally 155, respectively 34 Mbit/s connections (Surfnet). UU has a 100 Mbit/s Ethernet infrastructure, whereas FOM has locally a 10 Mbit/s Ethernet LAN. Forschungszentrum Juelich (FZJ-ZAM) is coupled via DFN to the PoP at Frankfurt by nominally at least 155 Mbit/s via Cologne and Aix-la Chapelle. At the site of FZJ the Institute of Plasma Physics (IPP) is coupled via nominally 100 Mbit/s FDDI rings to the DFN backbone and in IPP itself several FDDI rings are used, but the LAN is normal Ethernet (10Mbit/s).

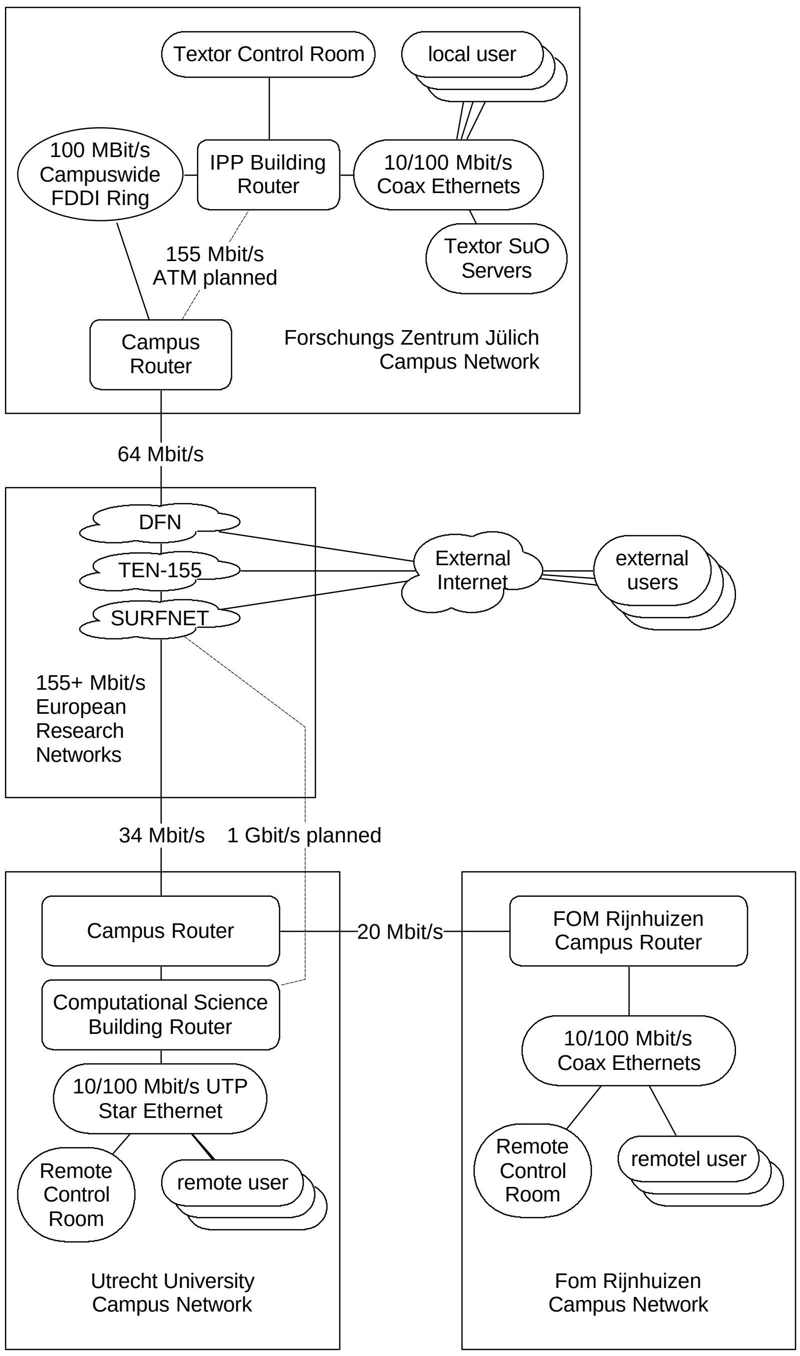

Textor Control Room
local user
100 MBit/s Campuswide FDDI Ring
IPP Building Router
10/100 Mbit/s Coax Ethernets
Textor SuO Servers
155 Mbit/s ATM planned
Campus Router
Forschungs Zentrum Jülich Campus Network
64 Mbit/s
DFN
TEN-155
SURFNET
External Internet
external users
155+ Mbit/s European Research Networks
34 Mbit/s
1 Gbit/s planned
Campus Router
20 Mbit/s
FOM Rijnhuizen Campus Router
Computational Science Building Router
10/100 Mbit/s UTP Star Ethernet
10/100 Mbit/s Coax Ethernets
Remote Control Room
remote user
Remote Control Room
remotel user
Utrecht University Campus Network
Fom Rijnhuizen Campus Network

Figure 5 Network topology exploited in the DYNACORE (PP) project

In testing all these connections that have to transport “messages and data” in the normal mode of operation we used the scheme as presented in Figure 5. Measurements were carried in the last part of the DYNACORE project (august ’99 until recent). The performance measurements will be presented in the chapter “Network performance aspects” and are also presented elsewhere[8].

[8] http://www.phys.uu.nl/~wwwfi/rtpl/home.html

DYNACORE system architecture

Introduction

Planning and executing plasma physics experiments evolves along a rather complicated operational schema. There are different viewpoints, connected to the different actors in the scene. There is the staff which is planning the set-up, the experimental procedures and first of all the goals of the experiments to be carried out. There is the operational crew of the fusion device, which is responsible for the operational status of the apparatus. This status is closely coupled to the character of the experiments to be performed. There are the scientists of the collaborating institutes, partly represented in the global staff, as far as the overall planning is concerned, but mainly responsible for their own (diagnostic) experiments, together with dedicated technicians. Planning and scheduling of the different tasks will finally result in the actual experimental phase. At this phase the DYNACORE architecture is aimed at.

The architecture to be described is mainly meant to support the experimentalists, including the technicians, who constructed the instrument and maintain it. The User Requirements Specification and Validation report [D3.1, D7.1], represent roughly their interests. This is understandable, since only they use the facilities directly, either locally, but also remotely. The operation of the fusion device itself is too delicate to be performed remotely, be it is obvious that the architecture could be used in the local conduction of the operations.

The deliberations of the staff could be supported by some parts of the architecture's implementation as if they were management support tools, as are used in commercially operated corporations.

The DYNACORE architecture is based on the three-tier concept. This means that client and server applications have been separated by a middle layer, in our case CORBA, originated from OMG[9] standards. This approach serves several goals: one is e.g. distribution (also in the topological sense) of services and data; another is the use of a heterogeneously composed environment. The latter is very adequate for the plasma physics community, since its equipment and software possesses historically and functionally a great many-sidedness. Yet another goal is to implement security at an appropriate level.

The scientists and technicians operating the peripheral instruments, around the fusion device, situated part remotely, and part locally, have the following global requirements:

- Information concerning their past and present (raw-)data.
- Information of the status of their equipment and the ongoing experiment.
- Information regarding past and present status of the fusion device.

[9] OMG group, "CORBA services specification", chapter 3. Available on the Internet via http://www.omg.org/cgi-bin/doc?formal/97-12-10

- Information about related experiments, run by different scientific groups at the same fusion device. (Tokamak).
- Means to analyse theirs and others data
- Information about related experiments performed in the past in other institutes (archives)
- Means to control and monitor and set experimental parameters during the experiment (remotely)
- Means to synchronise their experiment with other ongoing experiments, but certainly with the operation of the tokamak
- Means to keep track of their doing (logbook, etc.)

During the DYNACORE project it turned out that the scientists had their priorities in the realisation of all these requirements. These priorities are reflected in the above list, meaning that the scientists' highest priority is given at the top of the list. In developing the architecture and establishing the implementations, the emphasis was on those parts with highest priority, implying that for some parts rather extensive implementations were prepared, - viz. data retrieval -, for other parts only provisional solutions were worked out, - viz. remote control of experimental parameters-.

In the considerations leading to the development of a PP-DYNACORE prototype from the general architecture and the above-described requirements, it seemed self-evident to investigate the information streams in a (plasma) physics experiment. An abstraction of these streams is depicted in Figure 6. In the discussion with the experimentalist it turned out that not all information has the same priority. It became clear that the concept of a (technical) / settings database is rather abstract, though useful. It is usually directly implemented in the front-end's electronics. For this reason we did not implement stream "6" and "7" of Figure 6 in the prototype. Another consideration is the use of a middle layer between the database for acquired data and the experimental set-up. From symmetry reasons one should include it in the architecture, so we introduced it in performance measurements (described in sect.) of a data supplier to a data store (object database). Since the community uses their own implementation in the form of coherent files for their data storage directly coupled to the experiments, it seemed not necessary to implement it in the prototype. So the layer "11" from Figure 6 is not (yet) implemented in the DynaDemo (the prototype). Since the scientists wanted to refer to each others data the streams indicated by "4" and "5" (bi-directional) and the belonging interfaces "8" and "9" are very important in the validation of the prototype, so the emphasis in the next chapters will be on this subject. Finally the remote control of an experiment, this could be accomplished via streams "1" and "2" via layer "11". In practise experimentalists choose for local control, but they realise that this could not last for ever, so we constructed a very light control application via the CORBA layer, which is functioning quite well and is described in more detail in "The Pulsed Radar Reflector Control" In this application we don't use the settings database concept, but approach the experimental set-up via the CORBA layer, directly. Since remote control of delicate experiments cannot do without security provisions, we decided that the middle tier should implement these. But since available ORBs don't still incorporate this at a sufficient level we worked out a scheme for security services as described in chapter "Security"

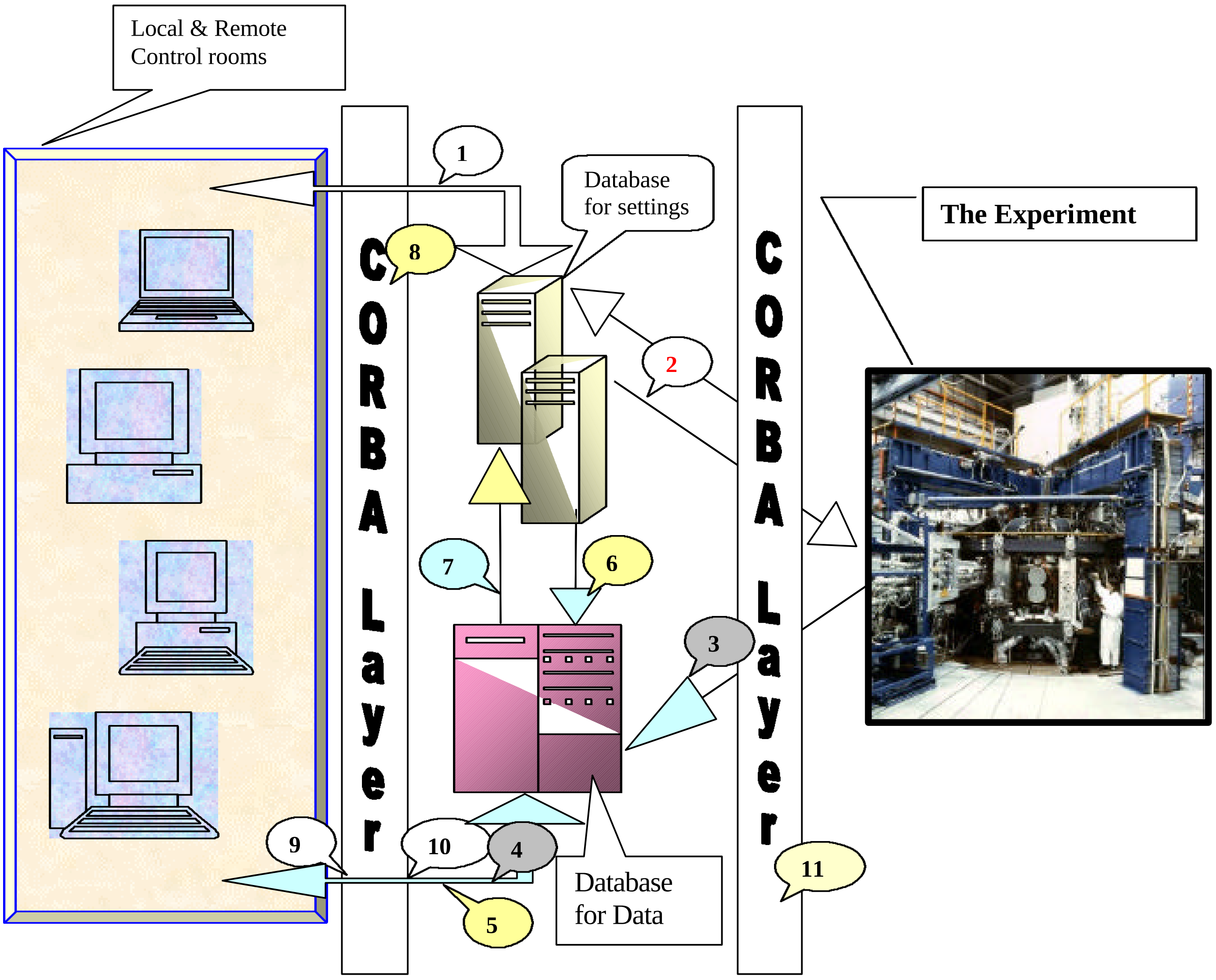


Figure 6 Information streams in a fusion experiment. An abstraction of the real situation.

1. Feeding the settings database with new settings for the instrument(s), responses included.

2. The configuration stream from the settings database to all involved experiment equipment, responses included.

3. The storage of acquired data from the data recorders into the data database. This database in principle has infinite storage capacity. This (raw) data, once measured, is untouchable.

4. Retrieval of acquired data from the pulse database by operators, diagnosticians and other scientists for e.g. analysis.

5. The storage of computed or interpreted values by scientists into the data database.

6. The automatic storage of (all, or selected) settings into the data database.

7. The automatic retrieval of (all, or selected) settings stored in the data database by the settings database.

8. The CORBA middleware layer that is between the scientists and the databases. This layer hides many implementation details of the databases from the users and provides uniform and controlled access to the database services.

9. The Application Programmers Interface (API) at the client side of the CORBA layer.

10. The API at the data database server side of the CORBA layer.

11. The CORBA layer that is between the databases and the experiment.

Data retrieval architecture

Introduction

Some kind of a measurement database will always be part of a fusion experiment. This database is used to store all the data created by diagnostics and auxiliary equipment during a well-defined period in time (seconds to minutes, this period is sometimes called a shot). As a past design this database is implemented as a collection of files having some kind of internal coherence, usually in the form of a directory structure. Every diagnostic that participates in a shot generates its own file during the first few minutes after the shot. Typical information stored in this file is the content of ADC buffers, which are filled during a shot, and the set-up parameters of the diagnostic. After a diagnostic has created its data file, it is moved to a central file server. The file name and location indicate which diagnostic generated it, and which shot it belongs to. Also more advanced systems are in use[ref].

A standard diagnostic (electronic) front-end consists of a number of CAMAC crates with many ADC channels (or equivalents). One ADC channel typically generates 10 to 100 kB of data per shot. During the years, the number of channels has grown steadily, and some newer channels use faster ADC's that generate more data per channel per shot. There are also new diagnostics being developed. These use e.g. video camera's PC platforms instead of CAMAC crates or high speed ADC's that produce tens of MB of data per shot[10].

The volume of the data generated per shot, together with the need to store this data in the short time between the shots puts high performance demands on the file server. Also the large data volumes of the new diagnostics and the non-CAMAC diagnostics cannot be integrated (without many difficulties) in the original architecture. Finally, for remote operation, the data in the database must be accessible from the virtual control rooms to allow for quick analysis directly after the shot. The original architecture provides nearly no hooks to allow for this.

Because of the shortcomings, it has been decided that the DYNACORE plasma physics prototype would incorporate investigations to another way to implement a measurement database. This database should be accessible for storage from a large number of computer platforms, allowing new diagnostics to be added to the system easily.

The database requirements can be summarised as follows:

- The database must be able to store signals, which have a size of several MB.
- The database must be able to store hundreds of MB of data per shot.
- All the data generated during a shot need to be stored in the database within a few minutes after the shot.
- The database must store on the average 30 shots per operation day, 80 days a year, adding up to TB of data per year.

[10] M. Korten et al, "Upgrade of the TEXTOR '94 Data Acquisition Systems for Plasma Diagnostics," *Proceedings 17th IEEE/NPSS*, Vol. 2, pp 803 – 806, October 1998

- The database needs to be accessible from many different computer platforms, locally and in virtual control rooms.

- A hard requirement has been defined as follows: the measurement database must be able to store 500 MB of data within 1 minute.

CORBA

CORBA[11] is a open standard for middleware. Using the standardised IIOP protocol, it can work on the existing Internet (IP) infrastructure. There are implementations of CORBA available for many computer platforms. The standardisation guarantees that these implementations can interact with each other. This makes CORBA an ideal candidate to provide for an architecture of a dynamically configurable (remote) access to data either for storage or retrieval in a heterogeneous environment.

The Dynacore architecture defines data managers with CORBA interfaces. These data managers have direct access to the measurement database. Data clients, for example analysis programs, use the data managers via their CORBA interface to store and retrieve objects in their database. The data managers provide not only platform independence, but also a way to introduce security into the database, even if the real database, underneath the data manager, does not implement it. Additionally, the data managers shield the actual database implementation from the clients. This allows us to change to a new database implementation without reprogramming any data clients. We only have to implement the data manager's CORBA interface for a new database type. Specialised data manager will be discussed in the sequel.

[11] CORBA homepage: http://www.corba.org

Database

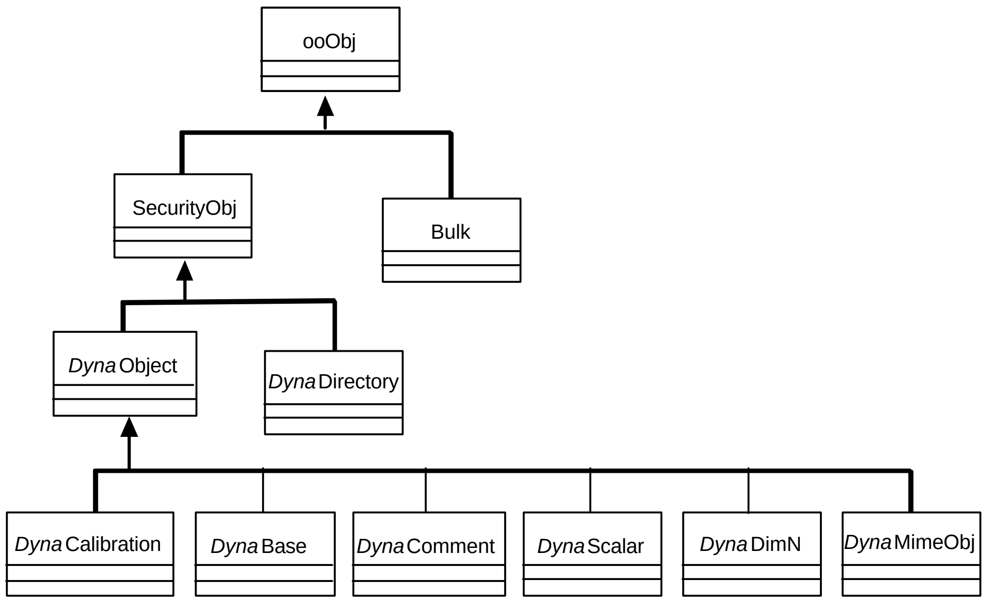


Figure 7 Inheritance tree of the measurement database classes

As an example we investigated the abilities and performance of an object database. We selected for this purpose Objectivity/DB[12].

The object-oriented database is used to store a predefined set of data classes. Our database model defines these classes.. These are displayed in Figure 7.

Storage Hierarchy

Objectivity/DB provides a storage hierarchy based on a federated database, which contains a number of databases. Each of these databases in turn contains a number of containers. Our persistent objects are stored in these containers.

In the proposed architecture, all measurement data (perhaps sometimes supplemented with settings of the diagnostic) is put together in one federated database. For every diagnostic, a new database is created in this federated database. Every diagnostic that participates in a certain shot stores the measurement data that belongs to that shot in a new container in its own database. The name of this container is the unique shot number.

For easy data access, every container has a *Dyna*Directory object, which holds references to the objects that the container holds. Additionally, database users can logically group measurement objects together in modules and sub modules, which are represented by sub directory entries ("pathnames") in the *Dyna*Directory object. This way, modules and sub modules resemble a Unix-like directory structure, which is stored in a flat Objectivity container.

[12] see :http://www.objectivity.com

One important remark must be made with regard to transporting the measurement objects via CORBA. It is theoretically possible to create a CORBA *interface* to every *persistent class* in the database. This would, however, complicate the design of the data manager and database clients considerably. It would also add a lot of network traffic overhead to our architecture. Our data manager, therefore, provides methods that pass data objects as CORBA's Interface Definition Language (IDL) *structures*.

Database Distribution

Objectivity/DB allows for the distribution of a federated database over multiple computers. Each computer can hold one or more databases of the federation. For our architecture this means that every diagnostic can have its own data storage machine. However, several diagnostics that create only moderate amounts of data might share one machine.

Users that access Objectivity databases from their desktop don't need to be running on a computer containing any database. They can access the federation via a private data manager. Objectivity uses internally the standard NFS protocol to access remote data, but it can also use its own proprietary protocol, called AMS. Using AMS improves the performance of Objectivity. In our performance tests, we used CORBA IIOP to communicate between database client and data manager. The data manager had its database locally.

Figure 8 shows a typical set-up of a distributed architecture. We have used Objectivity's database distribution functionalities in order to distribute our federation – and, therefore, the total load on the measurement database – over multiple computer systems. In addition, we could run data managers on many computers, not necessarily the ones that hold the database, while every data manager can access both local and remote data via an access to one federated database. However, from e.g. performance considerations one could decide to use more than one data manager in a client application to address data from other diagnostics (remote data) and implement distribution in this way via CORBA. This issue has to be studied to a greater extent in a next phase of future projects.

The central class in the model is the class *Dyna*Object. Its subclasses are instantiated as the measurements objects, e.g. *Dyna*Scalar, *Dyna*Dim*N* and *Dyna*MimeObj. These are wrapper objects for a specific type of measurement data generated during a fusion experiment (e.g. at Textor 94). A measurement object contains a reference to an object of the class Bulk, which contains the raw data. The Objectivity database allows us to put this referenced data directly into the database.

The measurement objects hold references to objects of other classes in the database. For example, a measurement object has reference to *Dyna*Base objects, which contain information on the measurement bases and to a *Dyna*Calibration object, which holds calibration information. Finally, a user can add a comment to a measurement by setting a reference in the measurement object to a *Dyna*Comment.

Splitting the information about data and the raw data itself speeds up the browsing of the contents of the database. To view the properties of data, only the measurement object needs to be retrieved. Opening the full Bulk object, which can contain megabytes of data, would cause too much overhead. Using references to *Dyna*Comment, *Dyna*Calibration and *Dyna*Base objects allows clients to reuse these objects, such that many measurements that use the same calibration, for example, can reference the same *Dyna*Calibration object. This saves database space and provides users with extra information on the origin of the data. When, for example, a calibration turns out to be wrong, all measurements that are influenced by this calibration can be found easily. This is a consequence of implementing bi-directional associations for e.g. the *Dyna*Calibration object.

Since there are special references for comments, calibrations and measurements bases, smart database browsers can be built that use these references to enhance data viewing. This is a very useful feature in a multi-user environment. It keeps together all the information in the database that is necessary to interpret a measurement. In case a user has additional information on a certain measurement, like a special calibration function, this information can be put into a *Dyna*Comment, and, if necessary, parsed by a specialized database client. In the case that a *Dyna*Comment contains this type of information, the *Dyna*-Comment should obey certain syntax rules, in order to give a handle to how this information should be used. A *Dyna*Comment can have a reference to another *Dyna*Comment for this purpose. Our data browsers do not yet implement this feature.

The *DynaObject* class inherits via the SecurityObj class from ooObj. OoObj is a class provided by the Objectivity database framework. Inheriting from ooObj makes a class persistent. This means that its attributes, which must be of special Objectivity types, can be stored in a database automatically. The SecurityObj class adds security attributes like the user ID and group ID of the owner of an object to all subclasses. It also has attributes that hold an object’s access rights. Inserting the security class allows us to create data managers that provide a Unix-like security architecture, while all information necessary to implement is stored in the measurement database itself, together with the objects to which the information belongs. Also this functionality has not yet been implemented, however the IDL interface is provided with the necessary hooks. (See also the chapter “Security”)

Measurement data

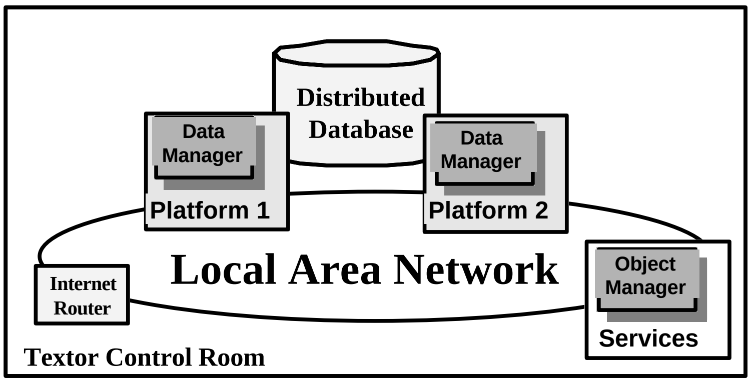


Figure 8 Typical set-up for a distributed heterogeneous database

The prototype also provides for an *object manager*, which acts a central starting point for all database clients. When a data manager is started, it registers at the object manager. The object manager then assigns the available data managers to clients, thereby distributing the database load over more computers. This object manager has at first only been implemented in its most simple form. For the early performance test we included the functionality in a dedicated data manager. Later on we designed the ObjectManager in a more elaborated form as will be described in a next chapter.

When a database client contacts a data manager and asks it for an object from the database, the data manager first has to load the object from the database into its memory. After that, it can fill IDL structure and send this structure to the database client. In this scheme, a database object must be sent twice over the network in the local control room, once from the database that stored the object to the data manager that serves the client and once from that data manager to the Internet router. Running the data manager on the machine that contains the data object to be transferred can save one transfer. Since certain clients (for example diagnostics themselves) mainly use objects that are located in the same database on the same physical machine, it makes sense to assign data managers to these clients that are located on those specific machines. In the proposed architecture, the object manager has knowledge about such clients and is able to assign them the most network-efficient data managers.

Moreover since the existing practise at fusion experiments involves still more traditional ways of storing data (Files), we constructed from the most elaborated design specialised DataManagers to make the integration of new and old as smooth as possible. These (simpler) DataManagers fulfil also another purpose since collaborations, involving different external groups, usually have no other means to inspect each other's data than looking at each other's screen (X-term capability). In our approach it is possible that with the same client application (e.g. a Data Viewer) one can inspect each other's data, provided the ObjectManager knows the exact location of the data and can address a proper Launcher, DataManager combination.

The DataManager, ObjectManager and Launcher

Introduction

The *DataManager* is “the middleware” between a client and a database. A client can be a user (scientist), who wants to read data for analysis, or an instrument that delivers data in a raw format for storage. Via an interface, the DataManager provides access to the database in a generic way. This means that the interface does not reveal any information about the DataManager actually stores the data. This makes it possible to store the data in various types of databases, like flat files, a relational database or in an object-oriented database.

A DataManager is designed to service only one client at a time. Therefore, each client receives its own personal DataManager. So, if multiple clients need to access the same database simultaneously, an equal number of DataManagers needs to be running. To realise this a *Launcher* application is designed.

When a client requests a specific DataManager type, a Launcher will start this type of DataManager, which then can be used by a client. When a client is finished it can notify the DataManager that it is no longer needed. The DataManager will then shutdown and release its systems-resources. If a client fails to notify the DataManager or, if the connection between client and DataManager is lost, the DataManager will auto-shutdown after a predefined timeout period.

Because various databases can exists at various places, it would be difficult for a client to know the precise location of all these databases and know the address of the particular Launcher at each of these locations. Therefore, the location of each database and address of each corresponding Launcher is held by one application, called *ObjectManager*.

This makes everything much easier for clients. They now only have to know (or register) the address of this ObjectManager. The clients can then request a list of all available databases from the ObjectManager. When a client wants to access a specific database in the reach of the ObjectManager, it can ask the ObjectManager for the address of a DataManager for this database. The ObjectManager will contact the appropriate Launcher ad return the address of the newly started DataManager to the client, which then can contact this DataManager. The communications between client, ObjectManager, Launcher and DataManager will take place via CORBA[13]. For this purpose interfaces of the specific “modules” have to be defined more in the framework of CORBA, which means in IDL[14].

[13] CORBA homepage: http://www.corba.org

[14] IDL: Interface Definition Language, a inalienable property of the CORBA specs.

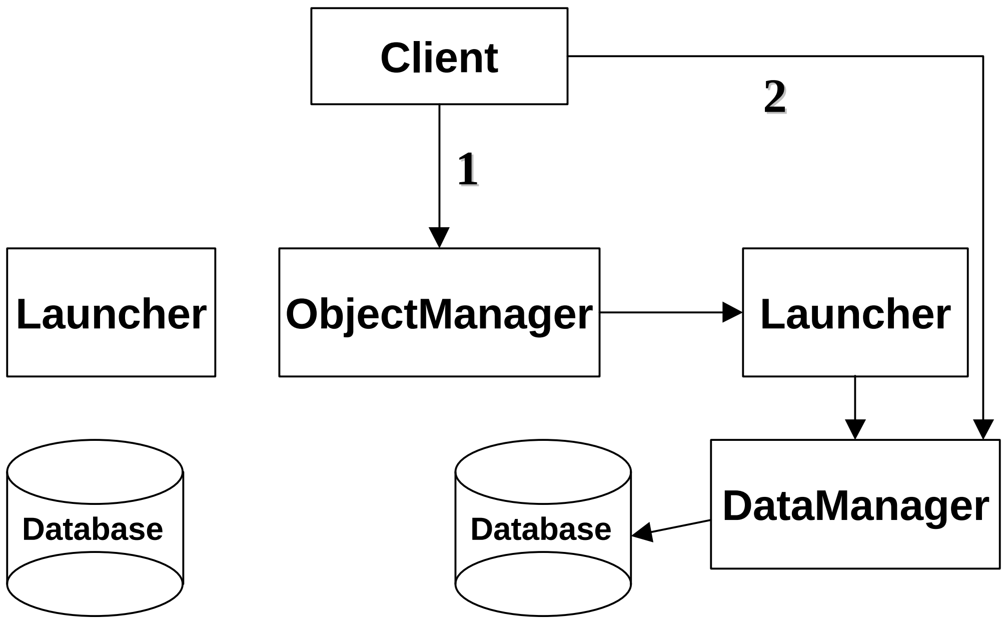


Figure 9 The ObjectManager, Launcher and DataManager at work

Interface Launcher.

Accessing the Launcher proceeds through CORBA, where at the server side a C++ ORB (omniORB 2.8.0 for C++) is employed. The Launcher's interface has only one method.

```
boolean launch( out string ior )
```

This starts a new DataManager and returns *true* or *false* to notify the caller if the operation succeeded. If the launch-operation was successful, the string "ior" contains the address of the newly launched DataManager. Else "ior" is NULL.

Interface ObjectManager.

Accessing the ObjectManager proceeds also through CORBA, where at the server side a C++ ORB (omniORB 2.8.0) is employed. The ObjectManager's interface contains two methods.

```
boolean GetList(
      in  OmListReqStruct ListRequest,
      out OmListStruct    ListResult )
raises( Error ) ;

boolean GetDataManager(
      in  OmConReqStruct  ConnectionRe-
     quest,
      out OmConStruct     ConnectionRe-
     sult )
raises( Error ) ;
```

A client first calls **`GetList`** to get a list of all available DataManagers. After it received a list, it can get the address of specific DataManagers. The struct-objects used to sent and receive the request are defined in the following way:

```
struct OmListStruct
{
  OmResultType  Result    ;
  string        strResult ;
  StringSeq     Names     ;
  LongSeq       IDs       ;
  long          iSize     ;
} ;

struct OmListReqStruct
{
  CryptoSeq Key     ;
} ;

struct OmConStruct
{
  OmResultType Result    ;
  string       strResult ;
  string       IOR       ;
  } ;
 struct OmConReqStruct
{
  CryptoSeq Key       ;
  long      iID       ;
} ;
```

Without going into each detail, it is interesting to note that with each request a key is passed to prove to the ObjectManager that the client in indeed authorised to perform a specific action. Also interesting is that the return-struct always contains a value and a string which can be used to present to the user in case of a failure.

Because everything is packed into structs, it makes the design flexible, because future extensions and changes to the function of the ObjectManager will not require a complete rewrite of the whole architecture around it, but only a change in those members actually handling the structs.

Interface DataManager

Accessing the DataManager proceeds also through CORBA, with at the server side a C++ omniORB 2.8.0 ORB. The operations of the CORBA interface can be divided into *transaction* specific operations and *data* specific operations. These operations are listed below. The comments are specified to the DataManager implementation for the DOM4[15] database type, but are virtually identical in terms of interface to other implementations[16].

First a common note: all interface-members have a parameter *CryptoSeq signature*, which is used to pass a security key to the DataManager, so that it knows the client is authorised to perform the action.

Transaction specific operations

```
void start( in CryptoSeq signature)
```

Starts a new transaction with the DataManager. If a transaction is already active, an Error-exception will be raised.

```
void commit( in CryptoSeq signature )
```

Commits the changes made to the database during the current transaction. This stops the current transaction. If no transaction is currently active, an Error-exception will be raised.

```
void commitAndHold( in CryptoSeq signature )
```

Commits the changes made to the database during the current transaction, but does not end the current transaction. If no transaction is currently active, an Error-exception will be raised.

```
void abort( in CryptoSeq signature )
```

Aborts the changes made to the database in the current transaction and ends the current transaction. If no transaction is currently active, an Error-exception will be raised.

Data specific operations

```
void store( in any object, in string path )
```

Creates a new object in the database at location path. If an entry with the same path exists, an exception will be raised. If the type of the object is not a legal type known to the DataManager, an exception will also be raised.

```
void update( in any object, in string path,
     in boolean headerOnly, in string info,
     in CryptoSeq signature )
```

Updates the contents of an existing object with the data provided. If the entry specified by path does not exists an exception is raised. If the type of the object is not a legal type known to the DataManager, an exception will also be raised. Finally, also when the type of the object sent is different from the existing object, an exception will be raised.

[15] DOM4 is the way in which the FOM group stores their data. It is basically a structured file system, which emulates a database behaviour.

[16] We have constructed also a DataManager for the file type used at IPP and in the control of the TEXTOR 94 itself, the so-called RT2 file system; also for the old FOM data type RTF.

```
RevInfoSeq getHistory( in string path,
     in CryptoSeq signature )
```

Retrieves the history of the object indicated by `path` as a sequence of `RevInfo` structures. If the object is not found, an exception is raised.

Also, note that not all Database types support reminding of the history. In such case, an exception is raised. DOM4 does not support history reminding.

```
ObjectHeader getHeader( in string path,
     in CryptoSeq signature )
```

Retrieves only the basic object header of the object indicated by `path`. If the object is not found, an exception is raised.

```
any getProperties( in string path, in CryptoSeq sig-
     nature )
```

Retrieves all properties of the object specified by `path`. If the object is not found, an exception is raised. This function will return an object of the correct type with all property attributes set, but with empty content.

```
any getData( in string path, in CryptoSeq signature )
```

Retrieves the object specified by `path`. If the object is not found, an exception is raised.

```
void rm( in string path, in CryptoSeq signature )
```

Removes the object specified by path. If the object is not found, an exception is raised.

```
void link( in string source, in string destination,
     in CryptoSeq signature )
```

Creates an entry that binds an object to a symbolic name `destination`. The object specified by `source` will be used as the path of the object that will be linked. If the object is not found, an exception is raised. In addition, if an object with name `destination` already exists an exception will be raised too.

Note that not all database types support linking. If linking is not supported, and exception will be raised. DOM4 does not support linking.

```
StringSeq list( in string path,
     in CryptoSeq signature )
```

Returns a list of strings with the full path of all objects at a given location path. If the object is not found, an exception is raised. This function doe not recursively list the content of directories. Also note that the path of directories in the list will end with a '/' character so they can easily be distinguished from other objects.

```
void lock( in string path, in CryptoSeq signature )
```

Locks the object specified by `path` for exclusive use by the client. If the object is not found, an exception is raised.

Note that not all database types support locking. If locking is not supported, and exception will be raised. DOM4 does not support locking.

```
void unlock( in string path, in CryptoSeq signature )
```

Unlocks the object specified by `path`. See also lock. If the object is not found or wasn't locked, an exception is raised.

Note that not all database types support locking. If locking is not supported, and exception will be raised. DOM4 does not support locking.

```
oneway void shutdown( in CryptoSeq signature )
```

If the client no longer needs the DataManager it can tell it to shutdown. This will shut down that DataManager and abort any currently active transaction.

```
void keepAlive( in CryptoSeq signature )
```

To prevent a DataManager from wasting valuable system-resources in case of a client-disconnect where the client didn't call `shutdown`, the DataManager will auto-shutdown after a specific period of time in which it was not used (default one hour). If a client wants to prevent this, it should call this function one in a while to reset the internal timer in the DataManager.

```
const unsigned long maxIdleTime = 3600 ;
```

```
readonly attribute unsigned long idleTime ;
```

Both attributes are used to control the auto-shutdown behaviour of the DataManager. See also `shutdown` and `keepAlive`. `MaxIdleTime` describes the maximum time the DataManager should remain idle before it should auto-shutdown.

The `idleTime` is the internal timer that tells how long the DataManager hasn't been used. To manually reset it, use `keepAlive.`

Access rights and the DataManager.

The following operations are relevant for security issues.

```
PolicySeq getPolicies( in string path,
      in CryptoSeq signature )
```

Gets a list of policies that apply to an object. If the object is not found, an exception is raisedNote that not all database types support policies. In these cases, it is a good idea to fake it and return a dummy-value.

DOM4 does not support policies and the DOM4 DataManager therefore returns a policy that indicates that everyone is able to read the specified object.

```
void setPolicies( in string path, in Policy policy,
      in CryptoSeq signature )
```

Sets a policy for an object specified by `path`. If the object is not found, an exception is raised.

Note that not all database types support policies. DOM4 does not support policies and the DOM4 DataManager therefore returns raises an exception.

Extended Interface

Problem with the standard **getData** member is that CORBA Any objects require a lot of system resources, especially CPU resources, when packed and unpacked. In case of large objects, like DimN-objects (one or multiple dimension array-objects), this method can be very time-consuming. To provide optimal read performance in all circumstances the interface is extended with members for specific data types.

```
DimNFloat64 getDim1Data( in string path,
     in ulong npoints, in ulong interval,
     in Interpolation how, in CryptoSeq signature )
```

Retrieves expanded data of an 1-dimensional object indicated by `path.` If the object is not found, an exception is raised. Allows retrieval of only a part of the data providing the index of the first point to read together with an interval and the total number of points to be retrieved. The data in the interval is interpolated in the way specified by the `Interpolation` argument. `None` does no interpolation and returns only the first data point of each interval. `Average` will return the average of all points in the interval. `MinMax` will return both the minimum and the maximum value for each interval. Therefore, in `MinMax` mode twice the number of requested points is returned.

```
DimNFloat64 getDim2Data(
     in string path, in unsigned long x_first,
     in unsigned long x_npoints,
     in unsigned long x_interval,
     in unsigned long y_first,
     in unsigned long y_npoints,
     in unsigned long y_interval,
     in Interpolation how,
     in CryptoSeq signature )
```

Retrieves expanded data of a 2-dimensional object indicated by `path`. If the object is not found, an exception is raised. Allows retrieval of only a part of the data providing, for both the x and y range, the index of the first point to read together with an interval and the total number of points to be retrieved. The data in the interval is interpolated in the way specified by the `Interpolation` argument. `None` does no interpolation and returns only the first data point of each interval. `Average` will return the average of all points in the interval. `MinMax` will return both the minimum and the maximum value for each interval. Therefore, in `MinMax` mode, 4 times (2x2) the number of requested points is returned.

These members behave identical to the **getData** member, except that they do not require the wrapping (on server side) and unwrapping (on client side) of the CORBA Any object.

Security

Introduction

The architecture for the DYNACORE prototype is based upon the Common Object Request Broker Architecture (CORBA) as defined by the Object Management Group (OMG)[17] in [Obj96]. CORBA however shows a lack on an important issue: “Though security services are defined in CORBA services[18], they are not implemented within the employed CORBA implementations so far”. Since security mechanisms are essential for many remote actions, it is necessary to define and implement a security service for the DYNACORE prototype, which could be replaced by an intrinsic CORBA service at a proper time.

This chapter presents a description of the security service defined for and used within the DYNACORE (plasma physics) prototype (system). This security service acts as a CORBA component, which is on one hand responsible for the authentication of users that want to get access to the DYNACORE services offered by the prototype and on the other hand for management purposes (i.e. management of registered objects within the system, forwarding of object access rights to clients, etc.)[19].

To perform these tasks the security service uses well-defined mechanisms as the Data Encryption Standard (DES)[20] [46-93] and an authentication scheme similar to the security service Kerberos[21]. In this respect every client who wants to get access to the system first has to authenticate itself at the **authenticator** (the authentication service) with *username* and *password*. Encryption is foreseen in the security service; however, the data produced by plasma physics experiments often is by all means bulky, but isn't secret[22]. Overall encryption would lead to much overhead on the sender as well as on the receiver side. To this end we added a feature of partly encrypted messages. The encrypted part of the message, the so-called authentication value, is of small size and serves to definitely authenticate (and authorise) the sender of the message. Utilizing such authenticated messages results in an enormously reduction of overhead that accompanies security.

CORBA *object servers* in the DYNACORE prototype that want to offer their services only to authorised clients, have to register their *servant* objects at the **authenticator**. To this end such an object server needs to have a user account at the authenticator, because it first has to authenticate itself as a normal user and then, after granted authorisation, register its servant objects there. If an authorized client wants to utilize such a servant object, it has to request for a servant object *ticket* at the **authenticator**. With this ticket, that contains all the *encrypted* data the servant object needs to know about the client, the client is able to register at the servant object and to utilize its services.

[17] Object Management Group. The Common Object Request Broker: Architecture and Specification. OMG, July 1996.
[18] Object Management Group. CORBA services: Common Object Service Specification. OMG, March 1995.
[19] If we talk about clients we mean the software application that acts on behalf of a user.
[20] Federal Information Processing Standards Publication 46-2. Data Encryption Standard (DES). National Institute of Standards and Technology, December 1993.
[21] J.G. Steiner, C. Neumann, and J.I. Schiller. Kerberos: An Authentication Service for Open Network Systems. Project Athena, Massachusetts Institute of Technology, Cambridge, January 1988.
[22] Raw data is in itself not very meaningful to outsiders

Secure information transfer

Message encryption

The development of a network based security system requires a mechanism to send information within messages in a secure manner across the network. The most obvious thing to realize that (and the one used here) is sending these messages in an encrypted form. This requires, that the sender has a key to encrypt the message and the receiver a key to decrypt this message.

By using a private cryptosystem both the sender and receiver need the same key for encryption and decryption. Another possibility is to use a public cryptosystem, where the sender has one key to encrypt the message (the public key of the receiver) and the receiver has another key to decrypt the message (the receivers private key). In such a cryptosystem it ought to be impossible to use the keys interchangeably.

Both cryptosystems have their advantages and disadvantages. For example, how do two parties in a private cryptosystem come to an agreement about the key to use? Since the key to encrypt is public in a public cryptosystem, and could be published by a key distribution center, this problem doesn't exist here. On the other hand, a public cryptosystem needs very long keys and the algorithms to en- {or} decrypt messages are much more complex than in a private cryptosystem.

We decided to use the Data Encryption Standard (DES), which is a private crypto system. To avoid the problems concerning agreement over the key between sender and receiver, we designed the security system respecting to the following items:

- The DES key needed for authorisation is generated on the basis of the *password* of the user. This has two advantages: firstly there is no problem with agreement about the key, because only the user and the **authenticator** know the users password, secondly this procedure avoids the necessity of sending the (not encrypted) password across the network. If the **authenticator** concludes that the user has the correct key, he also must have the correct password.

- If two parties (users) want to intercommunicate, they get each their common DES key from the **authenticator**. The main assumption here is, that the **authenticator** is *reliable* (if not, it has to be exchanged with a reliable one). In this context the authenticator acts as a kind of key distribution center.

- To enhance security, the password generated DES key is only used within the primary authentication procedure. (Attackers, that want to break the key of a cryptosystem, now get as little crypto code as possible, this hampers breaking the key.) Let's suppose that an attacker breaks a key; he not only gets the DES key, but also the password of the user (since he is able to reverse the password DES key generation). To play it safe, the authenticator generates a *session* DES key for the further communication between user and authenticator.

With respect to future enhancements the security system does not depend solely on DES; it could be exchanged very easy by another private cryptosystem and with little modifications by a public cryptosystem. The decision to make use of DES is based on the following:

- DES is one of the free usable cryptosystem definitions. In difference to other cryptosystems, one doesn't have to pay a fee for using it.
- There are many existing implementations of DES and if one is missing for a particular language (environment), it is easy to implement.
- DES is a well-defined and standardized cryptosystem.

N.B.
In the following, we often use the terms *private* (DES) key, *session* (DES) key or *communication* (DES) key. The meaning of these terms is:

- The private key is the key generated from the users password.
- The session key is an authenticator-generated key for the succeeding communication between an authorised client (which acts on behalf of an user) and the authenticator.
- The communication key is an authenticator-generated key for the communication between two authorised parties.

Authentication by use of tickets

In real life presenting an identity card or something else that proves ones identity and is accepted by the party that requires the authentication often performs the procedure of authentication.

In computer based systems the identity of a user is normally checked by a non-ambiguous *username*, that can be public, and a *password*, that has to be a common secret to the user and the party that requires the authentication. This authentication combination (username and password) is stored in a database (e.g. a password file). In a system with many services (to be performed by *servant objects*), which require authentication of users with the same authentication combination, it is necessary that all these servant objects have the same copy of the database[23]. This can lead to problems:

- When a new servant object is added to the system. How it gets a copy of this database?
- If a new user is added to the system. How can all these database copies be updated consistently?

[23] Under the assumption that these servant objects can't share the same database.

There are some options to solve the problems. One is to use *one dedicated servant object*, which is responsible for the authentication, at all other servant objects. This servant object, the so-called **authenticator**, is the only servant object in the system, which holds a database with authentication information for every user. Whenever a new servant object is added to the system, that wants to restrict access only to authenticated users, it has to engage the **authenticator** with an *authentication service* by registering itself at the **authenticator,** while establishing a *session* key. The authentication service performed by the authenticator requires, that every user, that wants to utilize a dedicated servant object, first has to authenticate himself at the authenticator. If this is successful, the user has to specify the servant object that he wants to utilize, and that has to be registered in advance at the authenticator. The authenticator generates from the user's and servant object's information a *digital ticket*, which, after being obtained by the user, has to be used to register at the specified servant object. - Such a ticket is comparable with a normal 'bus ticket' -. The authenticator puts in it all the information, which a servant object needs to know from the user. Since this ticket is encrypted with the servant object's session key[24], the servant object knows that the authenticator, which is a reliable party, could only have generated this ticket.

The user himself can't read the contents of the ticket, since he doesn't know the key by which the ticket was encrypted. It isn't necessary that the user understands the contents of the ticket, because the only thing the user has to do with it, is to send this ticket unchanged to the target servant object.

Authentication within messages

A possibility to authenticate messages between communicating parties is to encrypt the whole message and precede it with the sender's identity in plain text. Doing so the receiver is able to identify the sender and get the belonging DES key to use for the decryption of the message. This is certainly the safest way to exchange messages, but it will also result in much overhead on both sides. If the parties exchange messages with secret contents only, there is no alternative than proceeding this way[25]

In the DYNACORE prototype many of the exchanged messages contain data, which do not have to be protected against reading by others. It is important however, that the message comes from a trustable party. Therefore the security service has to support both alternatives: the encryption of entire messages and the authentication of plain text messages. The authentication of plain text messages is supported by a small piece of encrypted code (the so called *authentication value*), which comes along with the message. The authentication value is encrypted with the *session* or *communication* key in use between the communicating parties and contains an integer number. Both parties (and only these parties) know the value of this number. The receiver checks the validity of the received message by decrypting the obtained authentication value and comparing it with the one expected. If these two values match, the received message has to come from the right party. Otherwise the receiver would reject the message.

[24] This is the DES key, which is only known by the authenticator and the dedicated object server (and its servant objects).
[25] Except to using a Secure Socket Layer (SSL), which is a variant of this.

Since it isn't very safe to use the same authentication value in consecutive messages[26], this authentication value will have to be modified by both parties from message to message. To this end, the sender and the receiver, share in addition to the authentication value a secret *modification rule*. Every time a message is exchanged, both sides apply this modification rule to their authentication value copy, so that they always have a new, but a matching authentication value.

Components in the security system

In the system there are the following conceptual components (cf. Figure 10):

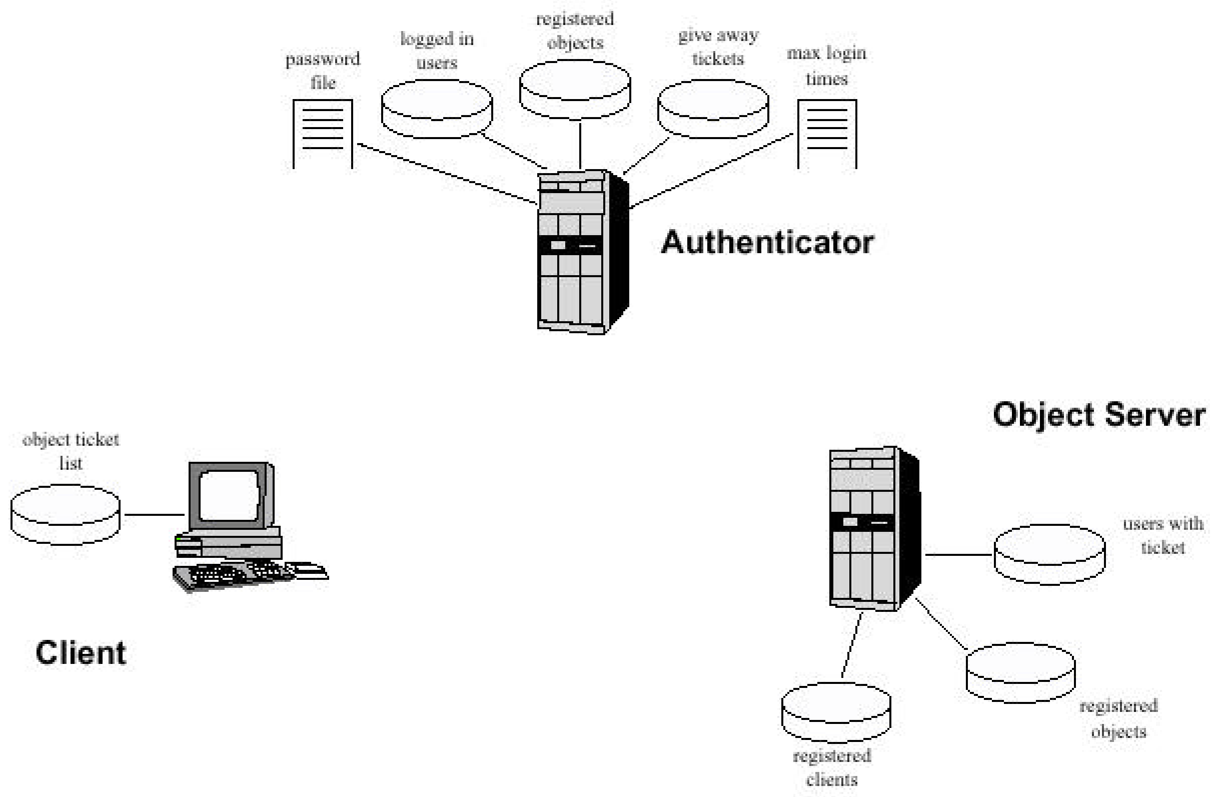


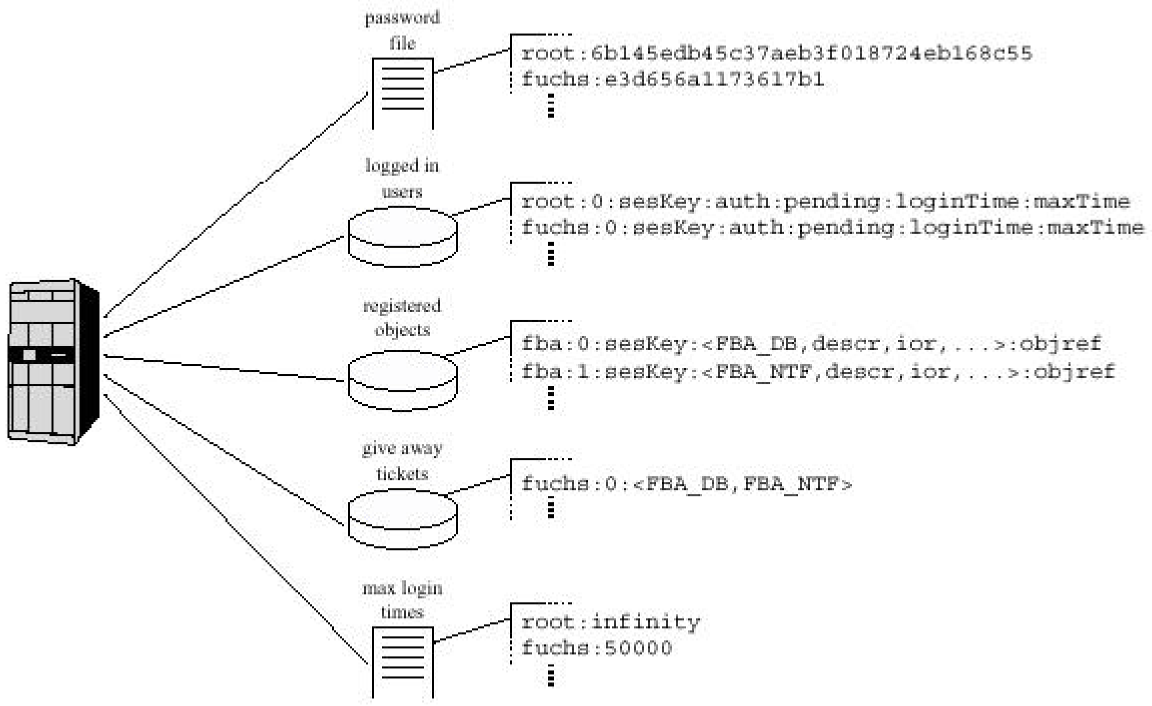
Figure 10 Security service components

- One Authenticator,
- Several Clients, acting on behalf of an user and
- Several CORBA Object Servers, which offer their servant objects via the CORBA middleware to authenticate clients.
- Several *Callback* Servers. A mixture of a client and servant objects. It doesn’t add extra services to the total system, but allows a client to obtain automatically updated status information via servant objects from an Object Server. The callback server is implemented in a Client by means of an additional servant object..

[26] An attacker could tap the connection line and use this authentication value for own messages to the receiver and to the sender.

The operation of these components will be described in detail in the following chapters.

The Authenticator



```
root:6b145edb45c37aeb3f018724eb168c55
fuchs:e3d656a1173617b1

root:0:sesKey:auth:pending:loginTime:maxTime
fuchs:0:sesKey:auth:pending:loginTime:maxTime

fba:0:sesKey:<FBA_DB,descr,ior,...>:objref
fba:1:sesKey:<FBA_NTF,descr,ior,...>:objref

fuchs:0:<FBA_DB,FBA_NTF>

root:infinity
fuchs:50000
```

Figure 11 Databases of the authenticator

The **authenticator** is the main component in the entire security system. It has to be started before any other component. The authenticator manages the user accounts and the registered servant objects. Without the authenticator no one can access anything remotely. To this end the authenticator has several long-term and short-term databases to store the necessary security information (cf. Figure 11):

- **Password file.** Within this file the authenticator stores the usernames and the passwords of the users, which are allowed to get access to the system.

- **List of authenticated clients.** To every point in time there can be several active authenticated (logged in) clients in the system. A client is a software application that acts on behalf of a user. The information concerning the security and the status of this client are held in this database.

- **List of registered servant objects.** Every registered servant object is accompanied by (much) information concerning the service it offers and how it is to be referenced. All this information and the dedicated security information is stored here.

- **List of servant object tickets.** For each client that requested for servant object tickets this database holds an entry with information about the servant objects, this particular client has tickets for. This information is essential for a servant object that wants to be notified when the particular client leaves the system.

- **Maximum login time file.** Apart from the security information, which is necessary for communication, there is another important information element: the *maximum login time*. When the **authenticator** is started, there is a default maximum login time for each user. If it is necessary to specify a different maximum login time for a specific user, one can do that by creating an entry in this file.

The password file

Like many other security systems do, this system manages the information about the users that are allowed to access the system via a password file. For every user, which has an account for the system, there is one entry containing the user's ID (*username* in plaintext) and his *password* encrypted with the superuser's DES key. Every time a user wants to login, the authenticator references this file to check the users access rights.

Example:

```
root:6bf2b61ed778921d
adminntf:6bf2b61ed778921d
fba:6bf2b61ed778921d # this a servant object
fuchs:6490fce06d1cf2fb
kemmerli:0686a02c3a297133
```

The list of authenticated clients

If a client wants to get access to the system and starts the login procedure to authenticate himself at the authenticator, the following information is generated that has to be stored within this database:

- The client's *username*,
- The *session id*, whereby the client's session together with the username can be definitely tied together,
- A DES *session key* for further communication between this particular client session and the authenticator,
- The *login time* that denotes the point in time when the client logged in,
- A *pending flag*, from which the authenticator, on the basis of the login time, is able to determine if the authentication procedure lasts to long or is already finalised,
- A *maximum login time* after which the client is kicked out of the system, and
- An *authentication value* that is used within the communication between the client and the authenticator to validate their mutual identity.

Example:

```
root:0:sesKey:loginTime:pending:maxTime:auth
fuchs:0:sesKey:loginTime:pending:maxTime:auth
```

The list of registered servant objects

CORBA object servers represent their servant objects. If a CORBA object server wants to offer a service in the form of a servant object, it first has to register this servant object at the **authenticator**. In this way the CORBA object server primarily makes its servant object public to the whole system and secondly it engages the **authenticator** for the authentication of each client that wants to utilize this servant object. To do this, the authenticator needs some information about the servant object:

- The *name* of the *object server*, that registers this servant object (i.e. name of the object's *owner*),
- The session id of the owner,
- A *base object information* element, which consists of the following informational elements:
  - The *name* of the *servant object*, that should be registered at the server,
  - A textual *description*, that describes the servant object in a concise way,
  - The *stringified IOR* of the servant object to be registered,
  - The *type* of the servant object,
  - The *owner* of the servant object,
  - The *group* this servant object belongs to and
  - The *access mode* of the servant object.
- A CORBA object reference (obtained from the stringified IOR) to call the servant object via CORBA.

Example:

```
        fba:0:sesKey:<FBA_DB,descr,ior,...>:objref
        fba:1:sesKey:<FBA_NTF,descr,ior,...>:objref
```

The list of servant object tickets

The *list of servant object tickets* is a security enhancement of the security service. It is not a standard, but an optional element of the **authenticator**. If a servant object wants to offer its services to clients, who are actually logged in, the authenticator has to notify all servant objects, a client has tickets for, on a client's *ticket request* and the client's *logout*. To do so, the authenticator needs to know for which servant objects this particular client has obtained tickets. To this end, the authenticator holds a *list of servant object tickets* which consists of the following elements:

- The *username* of the client, that has at least one ticket,
- Its *session id* to definite identify the client's session,

- A *list of servant object names* the specified client has gotten tickets for.

On a client’s ticket request, the authenticator makes an appropriate entry in the list and notifies the specified servant object about it. On client’s logout, the authenticator looks in the list for all servant objects the client has obtained tickets for. It notifies all these servant objects about the client's logout. Than it is left to the servant object's individual decision how to treat this client.

Example:

```
fuchs:0:<FBA_DB,FBA_NTF>
```

The maximum login time file

A security gap would be present if every client were allowed to stay logged in until oblivion. Because of that, the **authenticator** is initiated with a default *maximum login time*. This means the time after which every client should be automatically logged out. Using this static time for all clients leads to problems with servers (since every server at first acts as a *normal* client), that should run longer than this default time, or with clients that may not be allowed to stay logged in for even the default time. For his reason the **authenticator** maintains a table with an entry for every client that has a maximum login time different from the default one. Every entry consists of:

- The *username* and
- The *maximum login time*, that is different from the default one (otherwise it isn't necessary to make an entry for this user).

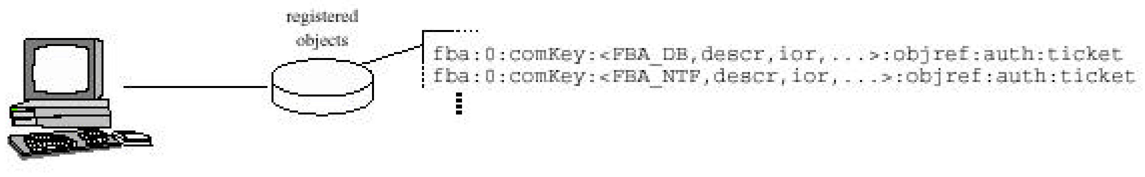
To allow a client to stay logged in until system shut down, the maximum login time in the particular entry is set to a *negative* value. This could be useful for object servers, which should last as long as the authenticator itself (which in principle runs until system shut down).

Example:

```
root -1 # until eternity
fba -1 # until eternity
kemmerli 100000 # maximum login time in seconds
fuchs 50000 # -"-
```

The Client

The client is a software application acting on behalf of a user. A client could be for example a world clock implementation that requests the local time of worldwide distributed CORBA time servers, but also an **authenticator** administration application that allows users with proper rights to add/delete user accounts or to change the *maximum login time* of any user. It could be a monitoring tool that presents a graphical view to the state of the security system[27]. If those clients wants to access additional services from servant objects residing on object servers, that offer their services only to authenticated clients, all those clients have one thing in common: they have to authenticate themselves at these servant objects[28].



```
fba:0:comKey:<FBA_DB,descr,ior,...>:objref:auth:ticket
fba:0:comKey:<FBA_NTF,descr,ior,...>:objref:auth:ticket
```

Figure 12 Client's database for the servant objects it is registered at

Due to the fact that a client often utilizes more than one servant object, every client has to hold (security) information about every servant object it is accessing, in a appropriate data structure, i.e. he needs a database for the servant objects he is registered at, as it is shown in Figure 12.

The client's list of servant object tickets

This database consists of the following elements:

- The *name* of the servant object owner (e.g. the Object Server),
- The *session id* of the servant object owner,
- A *base object information* element, that consists of the following elements:
    - The *name* of the servant object,
    - A textual *description*, describing the servant object in a short way,
    - The *stringified IOR* of the servant object,
    - The *type* of the servant object,
    - The *owner* of the servant object,
    - The *group* this servant object belongs to and

[27] There are many more possible client applications; this is only a small overview!
[28] How this will be done is not the object of this section, this will be explained in detail in the next section.

    - The *access mode* of the servant object.

- A *CORBA object reference* (gotten from the stringified IOR) to call the servant object via CORBA.

- An *authentication value*, that is used to authenticate the message exchange between this client and the servant object and

- A *servant object ticket* (obtained from the authenticator) that authenticates the client's request for registration at the servant object.

A client needs such an entry in his private database for every servant object from which he wants to utilize its services. If all the required information is complete, the client first has to register himself at the servant object with the (servant object) *ticket.* After having obtained permission he can use the service(s) of this object.

The CORBA Object Server and Servant Objects

A *CORBA object server* acts in principle as a client. It uses the main CORBA object in the system, the **authenticator**, to get access to the system and it can use other *servant objects* offered by other object servers. This means, that a CORBA object server basically has the same structure as every client does (see section The Client).

In addition to the client nature of a CORBA object server, there are services in the form of accessible CORBA servant objects within the server. The object server has to register the servant objects at the authenticator and engage the authenticator with the authentication of each client that wants to utilize this servant object.

The servant objects are working individually, but share the same base (under the supervision of the **authenticator**) security information, inherited from the object server (session key, authentication value and modification rule for communication with the **authenticator**). If a client wants to utilize such a servant object, it has to register at the servant object by calling its *registration method*. To authenticate the registration, the client needs a valid *servant object ticket* and an *authentication value*. The servant object accepts the registration request when the ticket was generated by the authenticator and when the authentication value, that comes along with the request, matches the one in the ticket.

All this information are managed by a servant object in the following databases (cf. Figure 13):

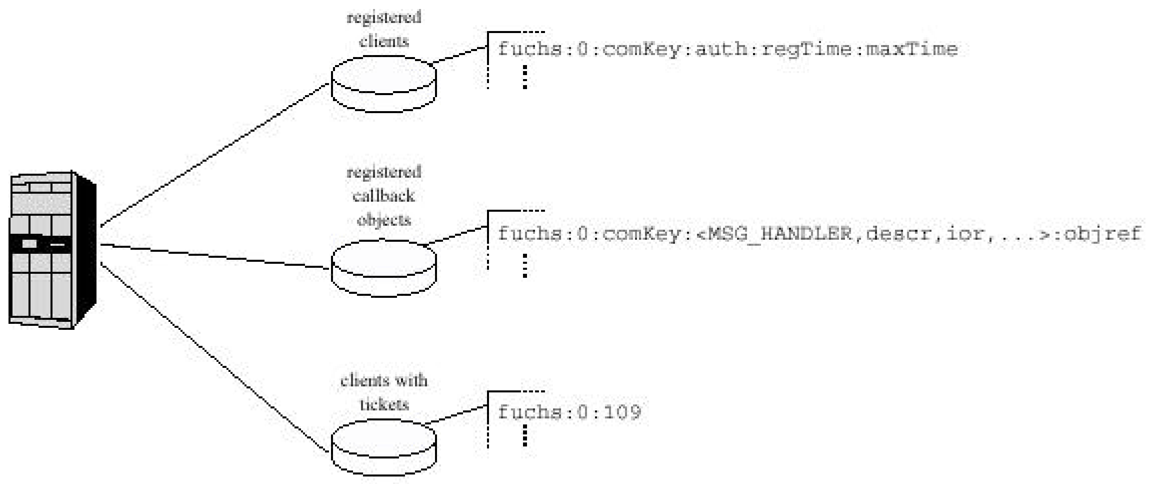

registered clients
fuchs:0:comKey:auth:regTime:maxTime
registered callback objects
fuchs:0:comKey:<MSG_HANDLER,descr,ior,...>:objref
clients with tickets
fuchs:0:109

Figure 13 Databases of a servant object

- **List of registered clients**. If a client wants to register at a servant object, it first has to get a ticket for this servant object from the authenticator. This ticket that is encrypted with the servant object session key, consists of the information necessary for service communication between the servant object and the client. After decrypting the ticket, the servant object extracts the following information from it an stores it in the database:

    - The client's *username*,

  - The client's *session id*,
  - The *communication key* for the communication between the servant object and the client,
  - The *authentication value* and the belonging modification rule for the communication between the servant object and the client,
  - The *time* the client tries to register and
  - The *maximum time* the client is allowed to stay registered.

- **List of registered callback objects.** The *list of registered callback objects* is not standard to every servant object. It is only needed if the service, this servant object furnishes, requires a callback to a callback object of a registered client (e.g. if this servant objects acts as a notifier about special events). To this end every callback client has to register a callback object at this servant object. The information that comes along with such a callback registration is stored in this table and consists of the following elements:
  - The client's *username*,
  - The client's *session id*,
  - The *communication key* for the communication between the servant object and the client,
  - A *base object information* element of the callback object, that consists of the following informational elements:
    - The *name* of the *callback object*, that should be registered at the servant object,
    - A textual *description*, that describes the callback object in a concise way,
    - The *stringified IOR* of the callback object to be registered,
    - The *type* of the callback object,
    - The *owner* of the callback object,
    - The *group* this callback object belongs to and
    - The *access mode* of the callback object.
  - *CORBA object reference* (obtained from the stringified IOR) to call the callback object via CORBA.

- **List of clients with tickets.** When the **authenticator** is initiated with the optional feature of notifying registered servant objects about the clients that have requested tickets for this servant object, and the servant object wants to make use of this mechanism as an security enhancement, the servant object has to store the information about these clients in a table. If the **authenticator** notifies an servant object about an issued ticket, the notification message contains the following elements:

  - The *name* of the client,
  - The *session id* of the client and
  - A *digest* of the given servant object ticket. This is the count of the set bits within the ticket.

  If then a client wants to register at this servant object, the servant object looks in this list for a matching client entry. If this entry exists, the servant object compares the ticket, which came along with the client's registration request, and the ticket’s digest. If this matches, the registration request maybe granted, otherwise rejected. There are two possible reasons for a missing match:

  - The client has never requested for a ticket for this servant object. Such a registration request should always be rejected, because the requesting client is obviously an attacker.
  - The client has requested a ticket, but either the maximum login time at the **authenticator** is expired (i.e. the client was logged out by the authenticator) or the client has logged out on its own. If the client is logged out, normally the authenticator notifies all the servant objects, the client has tickets for. Normally, the servant objects remove the client entry from their lists upon receiving such a notification.

The Security Service Protocol

As mentioned above, common to all parties in the system is the necessity of phasing the security protocol, without putting severe restrictions about periods a party can offer or use a service. The first thing to do is to become an authenticated member of the system by performing a *login procedure*. Being successful in this respect, carrying out the remaining protocol phases depend on the kind of component to engage. In principle there are, apart from the **authenticator**, three possible targets (cf. Figure 10):

- **Client.** A client represents the simplest component within the system. It only utilizes the servant objects offered by CORBA servers in the system. In a CORBA system without security functionality, such a client would only need the interface definition of the objects it want to utilize[29] and an IOR of the object to call the servant object on the server side via the middleware.
- **Object Server and its Servant Objects.** From the standpoint of the **authenticator**, an object server is a simple client with additional service functionality. In a CORBA system, initially without security functionality, such a CORBA server would be one of the main components, because of the services, via the servant objects, it can offer to clients in the system.
- **Callback Server.** A callback server is a mixture of a client and an object server. On the one hand, it utilizes servant objects from (real) CORBA servers, and on the other hand, it implements a servant object (the so called *callback method*), which can be used from the server side to realize the offered service, as it is shown in Figure 14. A callback server is more client than server, because it doesn't offer a service to the system, but implements a servant object to utilize servant objects from the servers in the system.

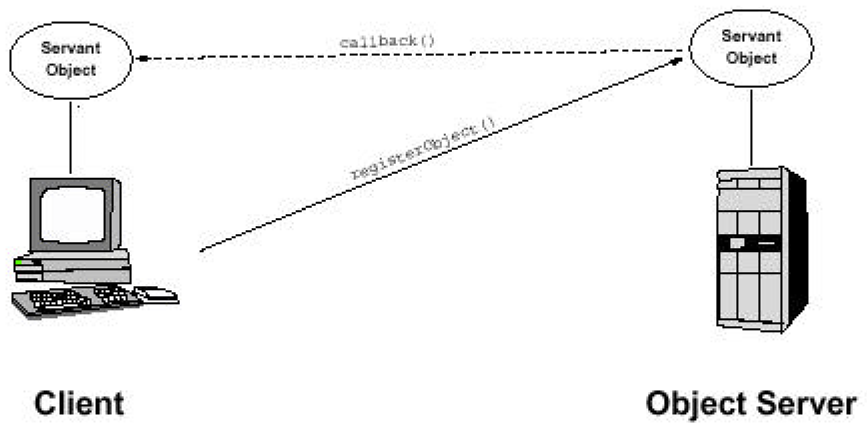


Figure 14 Client as callback server.

[29] In the form of an CORBA IDL-file, from which it has to generate the client stub.

An example of such a callback client could be a monitoring tool, which offers the user an actual view of the state of some devices in the system. Since the user expects that the state given by the monitoring tool is always up to date, it is necessary to update the monitor representation every time, a device changes state. There are two possibilities for the implementation of such a (monitor) client:

1) Poll the servant object, that gives the actual state of the device in certain intervals (this could result in high server and network load), or

2) Implement a servant object on the client side, which can be used by the servant object on server side to notify the client about state changes of the device.

The second alternative is precisely the one used in a callback server.

The next sections give detailed descriptions of the security protocol phases, which have to be carried out by the all mentioned system components in common and the ones, which are type dependent.

Common protocol phases

Every party that wants to access the system, controlled by a security service, has to authenticate itself at the main system component, the **authenticator**, by performing a *login procedure*. In analogy to that, the parties also have to perform a logout procedure, if they want to leave the system[30].

Login

The *login procedure* is implemented as a three-way-handshake mechanism and is divided in the following two parts (see Figure 15):

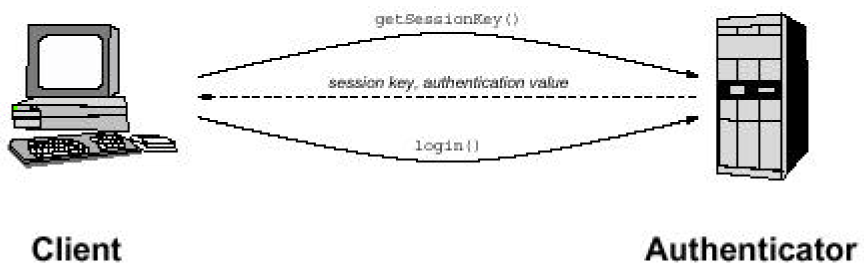


Figure 15 Client's login procedure at authenticator.

`getSessionKey():`

(C) The client, who acts on behalf of a user, sends its username to the authenticator.

[30] If the party doesn't support this, the authenticator in a kind of lightweight garbage collecting will do this.

(A) The authenticator looks in its *password file* for a matching entry and, if one exists, extracts the users password.

(A) Since this password is encrypted with the authenticator's private key, the authenticator decrypts this password and generates the belonging users private key from it.

(A) Since it is possible that *one* user with *one* username tries to open more than one session at the same time, it is necessary to distinguish the user's different sessions. To this end, the authenticator generates a *session id* for this particular session, so that in future requests to this session could be identified in a unique way by username and session id.

(A) As explained in the section Message encryption, further communication between the client and the authenticator will take place by using a *session key*. The authenticator creates this session key randomly.

(A) Most messages between the authenticator and the client only need to be authenticated, but not encrypted as a whole. To this end the authenticator generates an *authentication value* and a belonging modification rule for the further communication between client and authenticator.

(A) Since the client's login procedure isn't finalised at this point, the authenticator sets the *loginPending* flag to true. This indicates that the client still is in the process to validate his login.

(A) Now, the authenticator encrypts the generated session key with the private key of the user, and the authentication value and its modification rule with the generated session key. These two pieces of information are sent back to the client.

(C) The client application prompts the user for his password when it gets back these two pieces of encrypted information; it then generates the users *private key* from it.

(C) With the private key the client decrypts the obtained *session key*.

(C) With the session key the client decrypts the obtained authentication value and the belonging modification rule.

`login():`

(C) The client modifies the authentication value with respect to the modification rule.

(C) He encrypts the authentication value with the session key and sends this together with *username* and *session id* to the authenticator.

(A) The authenticator encrypts the authentication value and compares the result with its copy. If these two match, it sets the *loginPending* flag to false, which indicates a successful completed login procedure. Otherwise, the authenticator sends a signal back to the client that indicates, that the authentication value was wrong[31], and releases all resources allocated for this session.

Logout

The *logout* procedure is the simplest method in this security service. If a party wants to logout at the authenticator it only has to call the logout() method together with its username, session id and a valid authentication value at the authenticator as it is shown in Figure 16. The authenticator checks the authentication value, and (if it is correct) releases all resources for this particular user session.

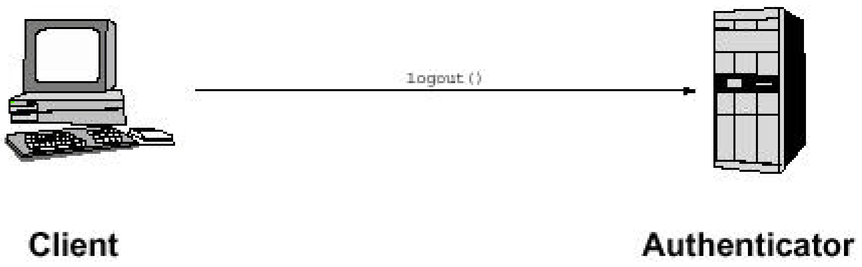


Figure 16 Client's logout procedure at authenticator.

Client dependent protocol phases

A CORBA client is an application that utilises servant objects offered by CORBA servers in a way similar to using functions of local libraries. If a client wants to utilise servant objects of servers within this security system, it has to be registered at this servant object. The procedure to register at such a servant object is similar to the login procedure at the authenticator, with the exception, that the servant object doesn't have to proof the authentication of clients on its own, but engages the **authenticator** with this job.

For this reason, the authenticated client first has to request for a servant object ticket at the **authenticator**. If the client has sufficient access rights to utilize the particular servant object, it will get such a ticket; otherwise the **authenticator** will reject the request. The only thing the client can do with the ticket is to send it to the servant object, requesting it to register the client as a party allowed to utilize this servant object. Since the ticket was generated by the authenticator and contains all the information the servant object needs about the client, the registration will normally be accepted. The ticket is completely encrypted with the object server’s session key, so that the object server and its servant objects only can decrypt it. Only the particular client can use it (i.e. no other client can take this ticket and exchange the client’s information with its settings).

The registration at a servant object will be carried out in two phases:

- Requesting a servant object ticket at the authenticator and
- Registering with this ticket at the servant object.

[31] This normally points to a typo in the password.

This and the analog unregistration from the servant object will be explained in the following subsections.

Register at a servant object

As mentioned above, the client first has to request for a servant object ticket at the **authenticator**, before it can request the servant object for registration. The question is, how can a client know, which servant objects are registered at the **authenticator**. One possibility is to request a servant object ticket with the hope that this servant object is registered at the **authenticator**. Since this is a gamble, the **authenticator** (as every object register should) provides a **`getRegisteredObjects()`** method which returns with a CORBA sequence of information of the registered servant objects (name, short description, IOR, etc.)[32]. In this sequence, the client can look for the servant object he wants to utilise, or for an alternative servant object if the one, he primarily wanted to register at, isn't available. With this information, the client is able to start the registration procedure (cf. Figure 17 and Figure 18):

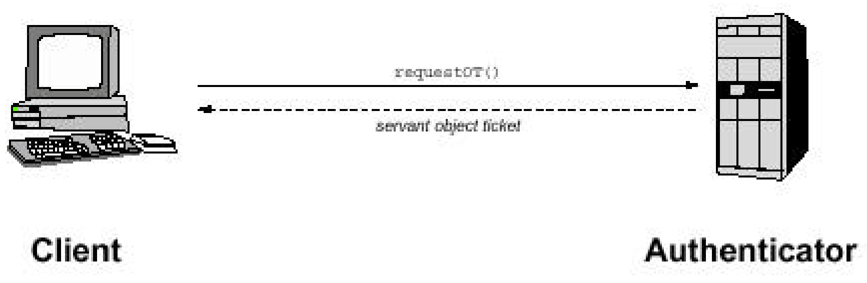


Figure 17 Client's request for an servant object ticket at authenticator.

`requestOT():`

(C) The client sends the servant object name together with his username, session id and a valid authentication value to the authenticator.

(A) The authenticator verifies the authentication value with respect to the username and session id.

(A) It randomly generates a *communication key*, an *authentication value* and the belonging *modification rule* for the communication between the requesting client and the target servant object.

(A) Since the servant object ticket has to be encrypted with the servant object session key, the authenticator gets this key from its list of registered servant objects (cf. The list of registered servant objects).

(A) The servant object ticket is generated by putting the client's username, session id, communication key and authentication value (with the belonging modification rule) into a CORBA sequence, and encrypts the whole sequence with the servant object's session key.

[32] The getRegisteredObject() method is the only method that could be called unauthenticated.

(A) Since the client also need to know about the communication key and the authentication value, the authenticator gets the clients session key from the list of authenticated clients (cf. The list of authenticated clients) and encrypts the communication key and the authentication value with this key.

(A) If this is supported by the servant object, the authenticator sends a ticket digest of the encrypted ticket to the servant object to notify it about the given ticket.

(A) Finally it sends the ticket, the communication key and the authentication value back to the requesting client.

(C) The client decrypts the communication key and the authentication value and stores them together with the untouched ticket in the list of servant objects; it is registered at (cf. The client's list of servant object tickets).

`registerClient():`

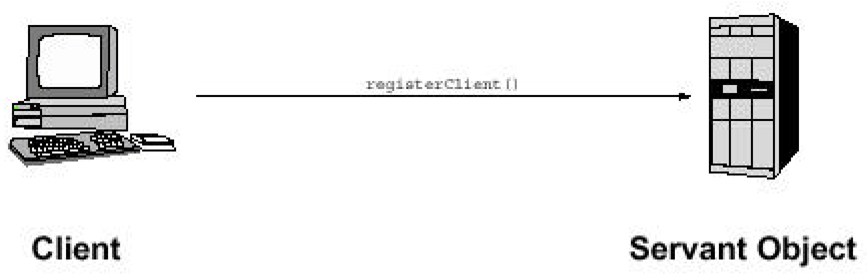


Figure 18 Clients registration at a servant object.

(C) The client uses the servant object's IOR to get a reference to the servant object.

(C) It sends the encrypted servant object ticket together with a valid authentication value to the servant object.

(O) If the servant object supports this, it validates the obtained, encrypted ticket with respect to the message digest obtained earlier from the authenticator on client's ticket request.

(O) The servant object decrypts the obtained ticket with its session key and extracts the client's username, session id, communication key and authentication value from it, validates the additionally sent authentication value with respect to the one in the ticket and stores this information in the list of registered clients.

If the client no longer uses the servant object or wants to leave the system, it has to unregister from the servant object, so that it can release the held resources for this client[33] (cf. Figure 19).

[33] If the client doesn't do this, the servant object in a kind of lightweight garbage collecting should do this.

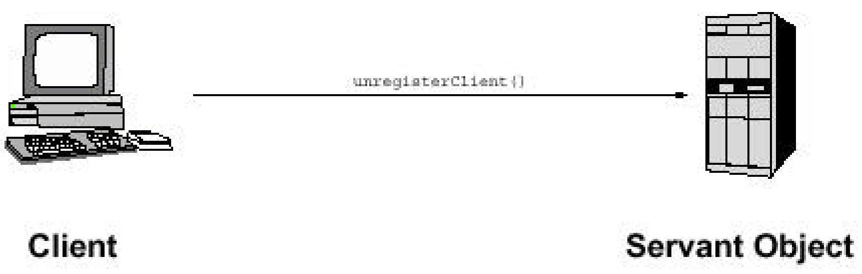


Figure 19 Clients unregistration at a servant object.

`unregisterClient():`

(C) The client sends its username, session id and a valid authentication value to the servant object.

(O) The servant object validates the authentication value, releases all the resources of this client, and returns with no value, if the authentication value is correct. Otherwise it sends a signal to client.

Object server dependent protocol phases

Basically, every object server is a client and has to carry out the phases to login at the **authenticator** as described in the forgoing sections. If an object server needs to utilise servant objects offered by other object servers, it also has to register itself at these servant objects. The thing that makes an object server different from an normal client is the services it offers in the form of servant objects, that can be utilized by registered clients as shown in Figure 20.

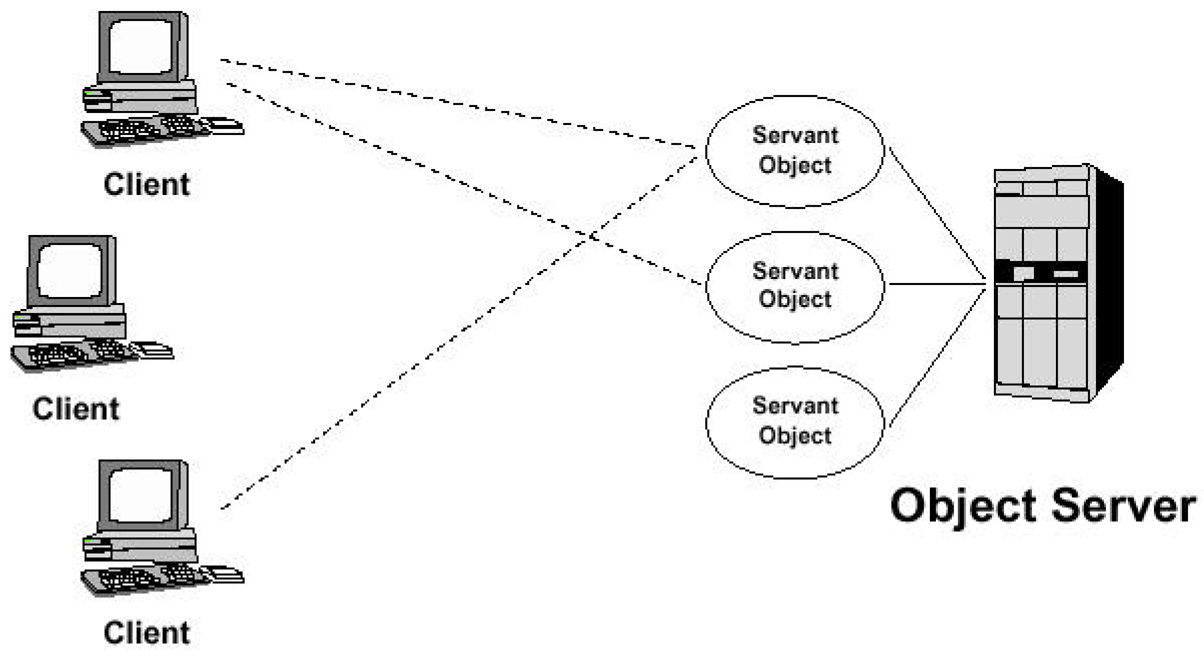


Figure 20 Object server that offers servant objects to clients.

In the security service an object server doesn't authenticate the clients by itself, but it engages the **authenticator** with this job. For this reason an object server has to register all its servant objects at the **authenticator**. A servant object is the implementation of a CORBA object and is accessible over the network. In a CORBA system without security functionality it would suffice to a client to have the interface description and the IOR of this servant object in order to utilise it. By using the security system, the client has to proof, that he has the required rights to utilise this servant object. But before any client can utilise a servant object, the respective object server has to register it at the authenticator. This is done as described in the following subsections.

Registering a servant object at the authenticator

Assuming, the implementation of a servant object to register already exists, and the Basic Object Adapter (BOA) is ready, the object server can start the registration procedure at the **authenticator** by calling the authenticator method

`registerObject():`

(OS) The object server creates a basic object information element, which contains the following elements:

- The name of the servant object that should be registered,
- A textual description that describes the servant object in a concise way,
- The stringified IOR of the servant object,
- The type of the servant object,
- The owner of the servant object (this is the username of the object server),
- The group of the servant object and
- The access mode of the servant object.

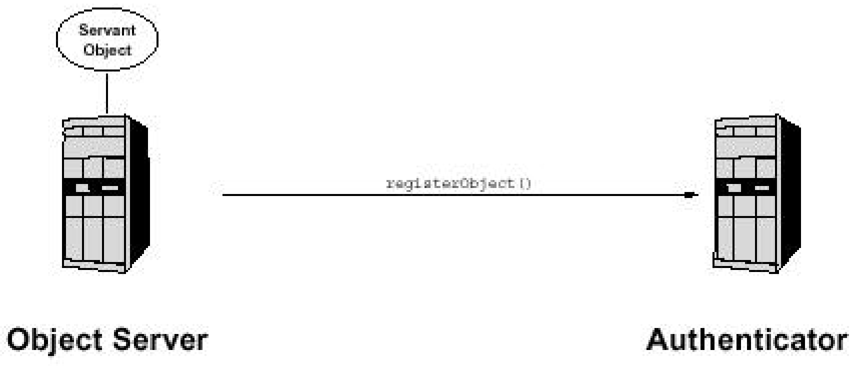


Figure 21 Registration of a servant object.

(OS) The object server sends this information together with its username, session id and an valid authentication value via a call to registerObject() to the **authenticator** (cf. Figure 21).

(A) The authenticator verifies the authentication value with respect to the username and session id of the object server.

(A) It extracts the servant object dependant information from the given basic object information element and puts these together with some object server inherited information in its list of registered servant objects.

At this point, the servant object is registered at the authenticator and can be utilized by clients that request tickets for it.

unregisterObject

If the Object Server decides to stop serving, because it want to leave the system, or no longer wants to offer a servant object to clients in the system, it has to unregister this servant object at the authenticator. This is done by a call to the unregisterObject() method at the authenticator. This method has the same parameters as registerObject(). Even the contents of the parameters are the same with the exception of the authentication value, that must have a different value (since it was modified with the modification rule before).

Utilizing Servant Objects

Once an authorised client has requested for a servant object ticket of its choice, and is registered at the servant object, this client can utilise the servant object. How such a servant object should be utilised is however not specified by the security service, but this is up to the programmer of the object server. If the programmer wants full security support, each servant object requires a registration (table) of authorised clients, and every method of the servant object must contain at least the authentication value as parameter (which is to be validated by the servant object). It is also up to the programmer to implement full message encryption or only authenticated messages.

It is also possible, that the programmer decides to offer some services to authenticated clients only, and other (not secure) services to the rest of world. This requires, that he includes a method in the servant objects to publish the IORs for these kind of services, instead of engaging the **authenticator** with this job.

The DataViewer.

Introduction

The *DataViewer* is the client-software used to view the data from a database (or equivalent files) made accessible by a DataManager. The DataViewer is a Java applet that can run as a standalone application in a so-called appletviewer or in a browser. It requires however a Java 1.1.x compatible virtual machine. All the images here were made with IBM Java 1.1.7 running on Windows 2000 Professional (= Windows NT 5.0 Workstation).

The Dataviewer application

When a user (scientist) has started the applet, in a browser, or as a standalone applet with in an appletviewer, an image as displayed in Figure 22 will be presented to him.

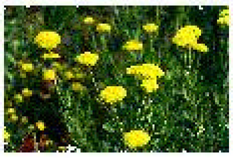


Figure 22 Start Image of the DataViewer

Clicking on this image will initiate the start-up procedure of the actual viewer. When the DataViewer is started, clicking again on this image will hide, or re-show the DataViewer.

During the initialisation of the DataViewer, the user will first be asked to identify him or herself by logging in. See chapter “Security” and the section on “Interface DataManager”.

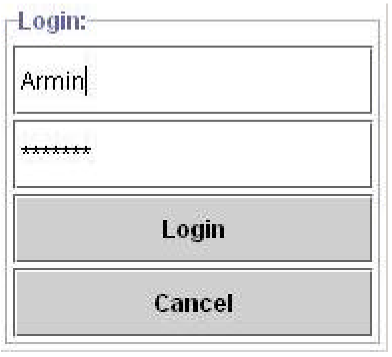
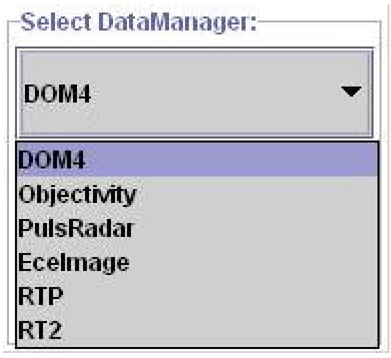


Figure 23 Login window

This enables the use of policies. One can for instance imagine that not all users have access to all data.



If the user decides not to login by pressing cancel, or if the logging fails, one can retry by simply clicking again on the image displayed in Figure 22.

Figure 24 Available databases

If the login succeeds, the DataViewer will contact the ObjectManager with the login-result to get a list of all available databases. It will then present the list of all DataManagers, available to the user, in a dialog with a combo-box. The user can then select one of them to connect to. See Figure 24.

If the user has made his selection, the DataViewer will contact the ObjectManager again, and request the IOR-string of the selected DataManager.

When the DataViewer has received this IOR-string, it is ready to request the DataManager for the selected database.

The DataViewer will then contact this DataManager, load all available plugins (more on this later) and start the actual viewer-GUI. The result is presented in Figure 25.

The DataViewer is by default of size 800x600, but can be resized to any preferred size. In addition, the split-pane can be repositioned, if needed.

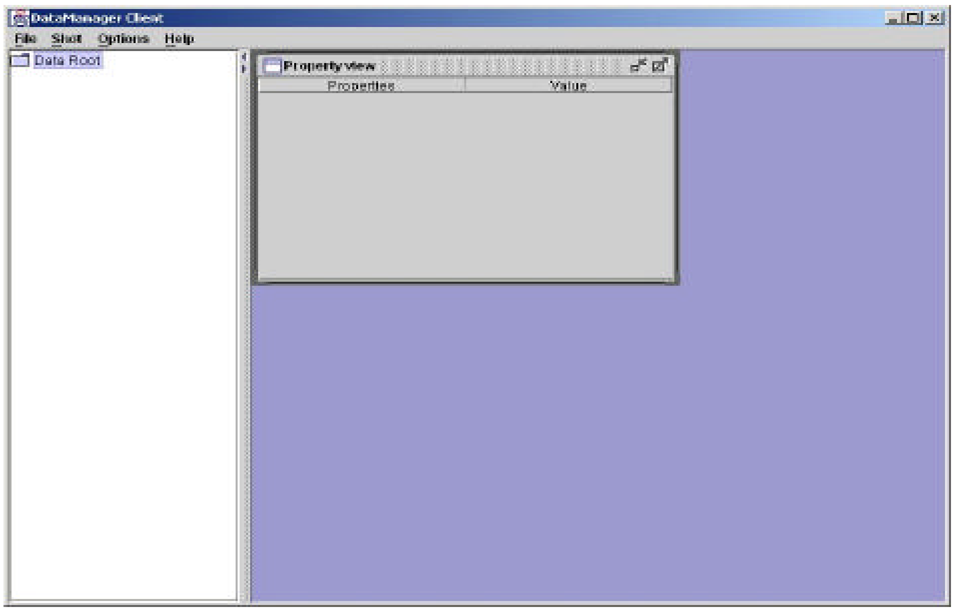


Figure 25 DataViewer window with split panes and empty property view.

Clicking on the item *Data Root* will result in the DataViewer contacting the DataManager to get the contents of the database and display the hierarchy. The user can then browse through the whole database (unless of course its user-rights are insufficient.) See Figure 26.

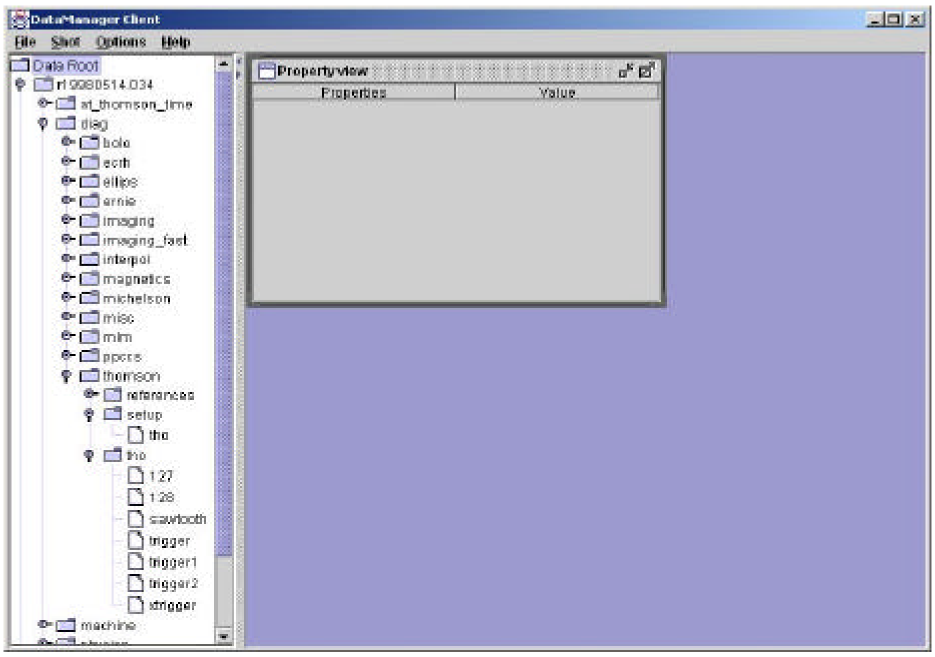
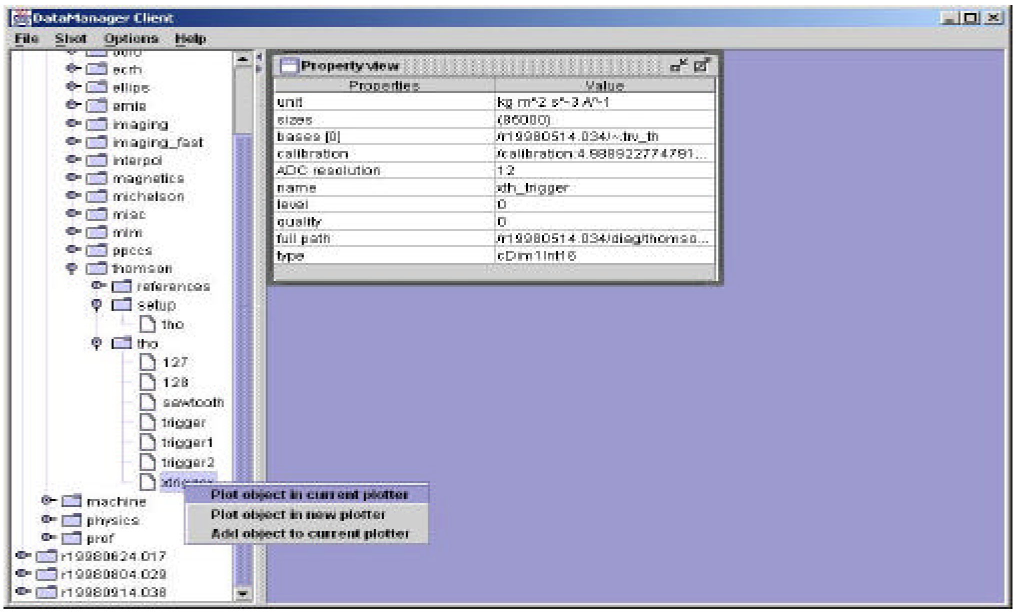


Figure 26 DataViewer window with unfolded content tree

If a user wants to have more information about an object in the database, he/she will only have to click on the item. The DataViewer will then retrieve the properties of this object and display them in the *Property view*. It can also graphically display the item, provided that a plug-in is available. See Figure 27.



Properties	Value
unit	kg m^2 s^-3 A^-1
sizes	(86000)
bases [0]	/r19980514.034/~:trv_th
calibration	/calibration:4.888922774781...
ADC resolution	12
name	xth_trigger
level	0
quality	0
full path	/r19980514.034/diag/thomso...
type	cDim1Int16

Figure 27 DataViewer window with unfolded content tree and property view containing the pointed to information.

Each data type has a (graphical display) plug-in that is able to handle the particular data type. Users can write additional plug-ins for new data types. Depending on the object, one is viewing; the plug-in may have additional options available. See Figure 28.

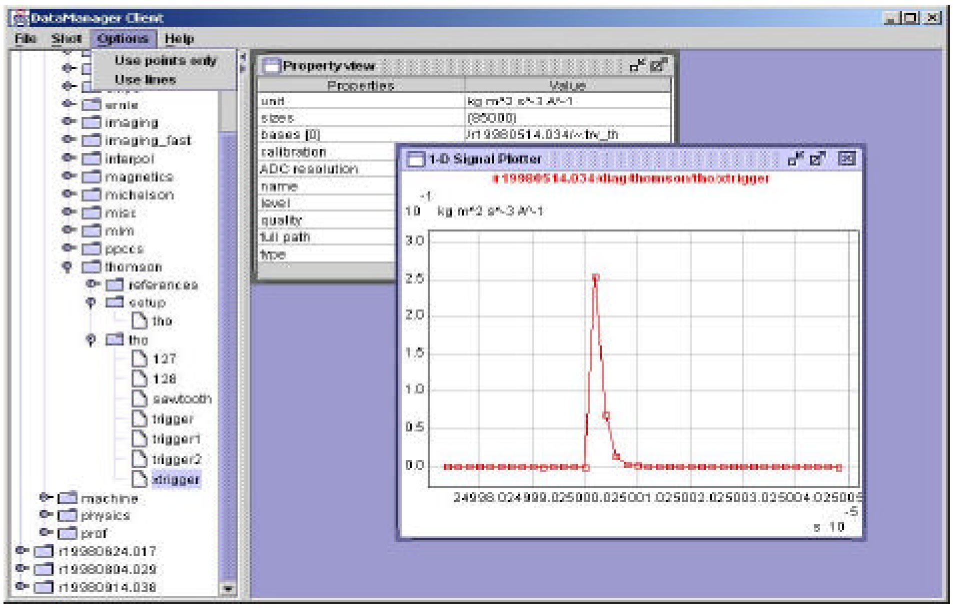


Figure 28 DataViewer window with graphical plug-in (1-D) to view particular data.

Of course, it is possible to view various objects at the same time. It is also possible to combine several objects in one viewer, for instance for comparison. See Figure 29.

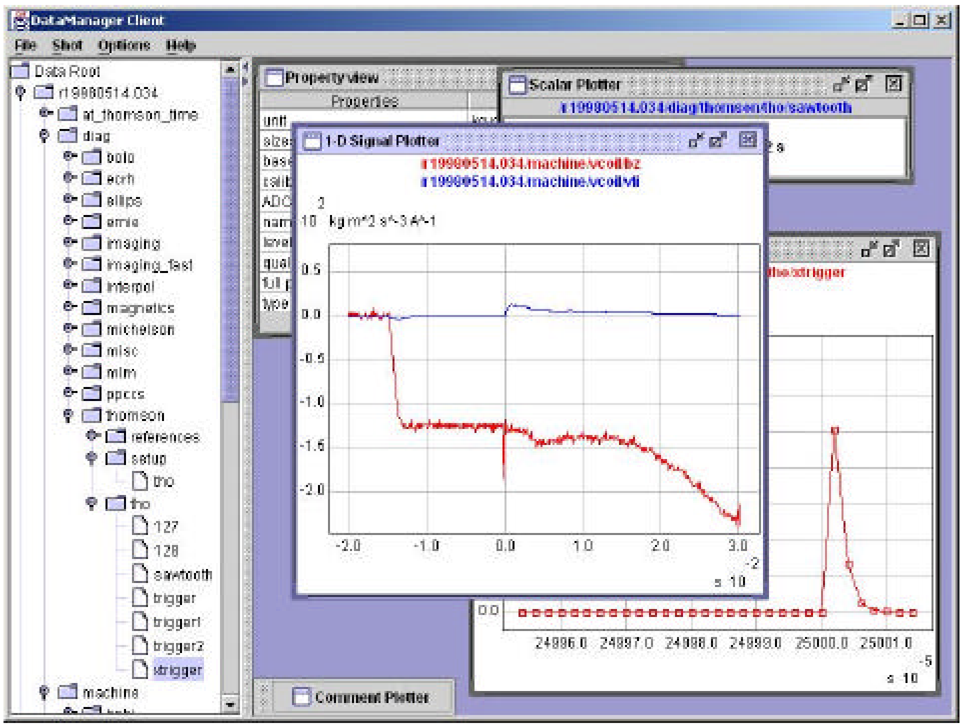


Figure 29 DataViewer window with two graphical displays simultaneously open.

Note that with the 1D (as for instance displayed in Figure 28 and Figure 29) and 2D (for instance intensity plots) plug-ins, it is possible to zoom in and out. Zooming *in* is performed by selecting a part of the graph (by using the pointing device) and zooming *out* by simply right-button clicking. The DataViewer will exclusively retrieve the data needed to create the graph. This can speedup viewing considerably (efficient use of network capacity), since the pointed to data could consist out of millions of points.

Finally by choosing the *Close* menu-item in the *File* menu one can hide the viewer as explained earlier. The DataViewer-applet is automatically destroyed when the browser leaves the page, or, in case of standalone use, when the user closes the appletviewer.

The DataViewer – Architecture

DataViewer structure

The DataViewer was build with flexibility in mind. This becomes clear, when viewing the global structure of the viewer, see Figure 30.

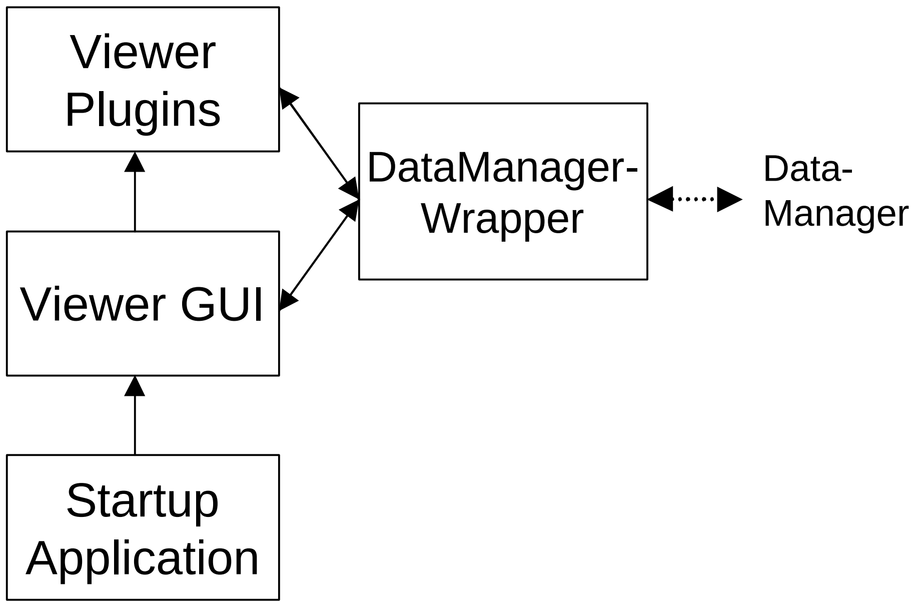


Figure 30 DataViewer structure

One can observe that we have decomposed the DataViewer in four parts:

- The Application Start-up code
- The Viewer Graphical User Interface
- The plugins
- The DataManagerWrapper

The DataViewer Start-up Code.

In the current version the DataViewer will first contact a Security Manager (Authentica-tor), then contact an ObjectManager to get a list of all available DataManagers and finally contact that DataManager and start the GUI.

However, one can imagine that in other situations more or less steps are required / wanted before the DataManager is contacted and the GUI displayed. For development purposes, a specials *lite* client was created, which is identical to the actual DataViewer, except for the fact that it contacts a DataManager directly, to allow for the testing of DataManagers. The flexible design, which separates the actual Viewer GUI from the initial start-up-code made, facilitates this.

Another example is that the same Viewer GUI could be used in developing both an applet-version for use in web pages and a standalone Java-application. Only the Start-up-code had to be different.

The DataViewer GUI.

The DataViewer GUI consists of a set of Java-classes based on the Java SWING library. What the Viewer GUI does is nothing more than contacting the Data-Manager to get all the available items and list them in a tree-like fashion (Figure 31). When a user (scientist) selects an item, it will display its properties in a special property window and when the user decides to view the item, it will start the appropriate plugin, which will then handle all further actions.

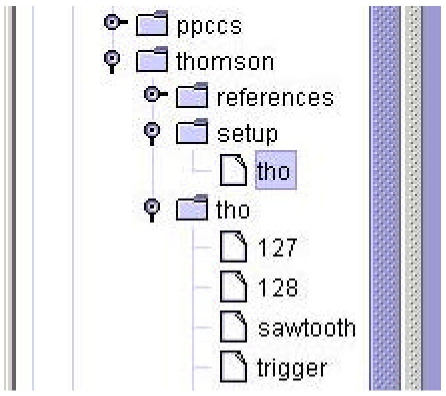


Figure 31 DataViewer, expanded tree

The Plugins.

Because a database can contain many different object types, the DataViewer should be able to view just as many as there are present. However, the Viewer GUI does not have to accommodate all data-types by itself, but uses plugins instead. This prevents the Viewer GUI from getting too complex and difficult to maintain. This also means that the functionality of the DataViewer can be augmented by simply adding or changing plugins. In the case of different applications, one could also opt for different plugins for the same data-type.

Moreover, if in the future, additional data-types are added to a database, the viewer can be easily extended to support the new types.

Writing a Java-plugin for the DataViewer involves the following steps:

1) Define a class with a unique name, which extends the abstract class DataPlotter:

```
class MyClass extends DataPlotter
 {
 }
```

2) Implement the abstract member functions of the DataPlotter class:

JMenu getMenu()

Should return a JMenu, which can be used to configure your DataPlotter. May return null, if you don't provide a menu.

`void clear()`

Perform whatever actions will clear your plotter.

`void setDataSource(DataManagerWrapper dm)`

Tells your plotter where from to get its data. The DataManagerWrapper class is described later.

`void addSignal(ObjectHeader oh)`

If possible, add the signal corresponding to the ObjectHeader to your plotter. Your implementation should retrieve the data itself, using the data source provided by setDataSource().

`boolean canPlot(ObjectType t)`

Should indicate, whether or not your potter is capable of plotting an object of the provided type.

`ObjectType[] getPlotableTypes()`

Should return an array of all the types your plotter is capable of handling. Don't return an empty array here, as your plotter will only be asked to plot data of the types you return.

3) Add the name of your class to the list in the 'plugin.list' file. This file will be read by the Start-up code, and all classes in it will be automatically registered with the Viewer GUI.

The DataManagerWrapper.

The DataManager-interface has certain aspects, like security, which would make the viewer, in having to deal with them, very complex and not very flexible. Any change in the DataManager’s interface, or any change of its aspects, like the security, would require rewriting large amounts of code. Therefore, a DataManagerWrapper-class was put in between.

The DataManagerWrapper is, as the name already indicates, a wrapper around the actual DataManager-interface. This wrapper shields the Viewer GUI and its plugins from the actual CORBA DataManager interface.

The DataManagerWrapper provides the same calls as the DataManager-interface implements, but then simplified, by stripping all to the GUI and plugins irrelevant aspects. This design makes programming less complicated, since all these aspects of the DataManager-interface are now hidden from the Viewer GUI and plugins.

All communication between a DataManager and the DataViewer occur via this wrapper, which also results in a very flexible design, since now any change in the DataManager-interface will only require an equivalent change in the code of the wrapper-class.

Experiment to be controlled remotely

The nature of diagnostic systems ranges from simple magnet field sensors to complete interferometers, from current reading devices to complete tomographs. The majority of the signals from the sensors have to be digitised and undergo analogue-to-digital conversion after proper amplification and filtering. The electronic systems are there to perform i.e. all these tasks. The settings of the electronics have to be adjusted remotely because of radiation hazards etc. The same holds for the instruments that control the generation of the plasma. (See Figure 3). Ideally control and data acquisition should be carried out along well-separated channels, but in practise this is not always the case. The PP DYNACORE architecture is developed

The pulsed Radar Reflector Diagnostic[34]

In pulsed radar reflectometry, short microwave pulses in the order of 1 ns are launched into the plasma by means of antennas. Depending on the radar frequency (channels) and the plasma parameters, the pulse is reflected by a critical density layer and received again (also in an antenna) by the diagnostic equipment. The basic quantity that is measured by the pulsed radar diagnostic is *the flight time* of the microwave pulse between transmission and detection.
The number of independent channels is ten and two variable frequency channels are added to the system. The two variable frequency channels can be used in combination with two fixed frequency channels to perform correlation measurements and to study MHD modes in the plasma. The pulse repetition frequency is 2 MHz for the ten channels. The flight time is recorded with an accuracy of 70 ps, corresponding to a spatial resolution of 1 cm when reflected from a metal mirror. The accuracy can be further improved to 35 ps. One of the drawbacks of the pulsed radar technique is the fact that fluctuations and shallow density gradients give additional pulse broadening, which has an effect on the flight time measurement. The chosen pulse length of 1 ns is a compromise between the accuracy of the time of flight measurement and the pulse broadening. In the present system, the flight time is measured between the 50% level of the leading edge of the transmitted and received pulse. By clocking also the 50% level of the falling edge a measurement of the pulse width could be obtained. This would provide additional information on the density gradient.

[34] http://ns2.rijnh.nl/n3/n1/n3/f1234.htm; C.A.J. Hugenholtz et al., Fast pulsed radar reflectometry for the Textor Tokamak, Rev. Sci. Instr. 70 (1999) 1034

System layout

The distance from the antennas to the radar set-up itself is about 10 meters. Two echo pulses will be received from each transmitted radar pulse. The first one is a start pulse traveling via a bypass and the second one is the stop pulse reflected at the critical density layer. The minimum time between the two pulses is 4 ns, which is determined by the constant fraction discriminator (CFD) in the video section of the set-up (Figure 24). The longest time delay is obtained from reflections at the far wall when the density rises to near the critical density (n_c). The time delay with plasma densities near n_c is about 6 ns longer than the time delay without plasma. Calibration is performed using the time of flight system (TOF) of the reflected pulse at the far wall. The bypass consists of two 10 dB directional couplers, an attenuator and a short section of wave-guide. The position of the bypass must be chosen in such a way that the start and stop pulse coincide with the 20 ns LO (oscillator)-pulse. The two (bypass and signal) pulses will start and stop a time-of-flight counter (TOF), developed at Rijnhuizen using eight parallel-gated counters. The data produced by the TOF-counter is fed to a data acquisition system built in VME. The data handling, storage etc. will be described in the next section.

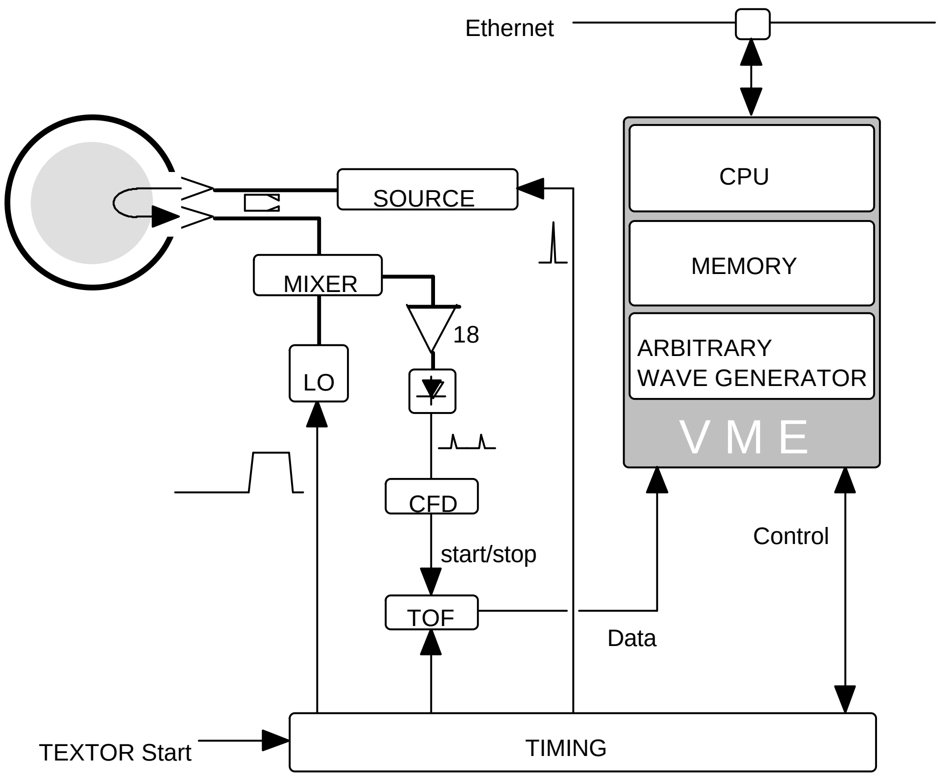


Figure 32: Timing, control and data storage

Software for control, data handling, and storage

The pulsed radar diagnostic will be regarded as a subsystem of TEXTOR. This means that the diagnostic should be synchronized with the TEXTOR control system. Four timing stages are distinguished:

- inter-pulse (diagnostic can be used),

- pre-pulse (all devices are initialised and ready to accept a pulse),
- start-pulse (radar start),
- post-pulse (the data in the memory module can be stored in a database).

There is a possibility to work stand alone for test purposes. The data acquisition of the pulsed radar diagnostic is (whenever possible) built up from commercially available components like a Solaris v2.4 operating under UNIX with a VME-bus system (Figure 25). The embedded controller[35] clocks the data via the RS485 input into a dual-port memory. A VSB VME-bus is the connection between the modules. In this way the memory-module acts as a memory extension of the UNIX system. The pulsed radar controller and the arbitrary waveform generator (for calibration and test purposes) are developed by FOM-Nieuwegein.

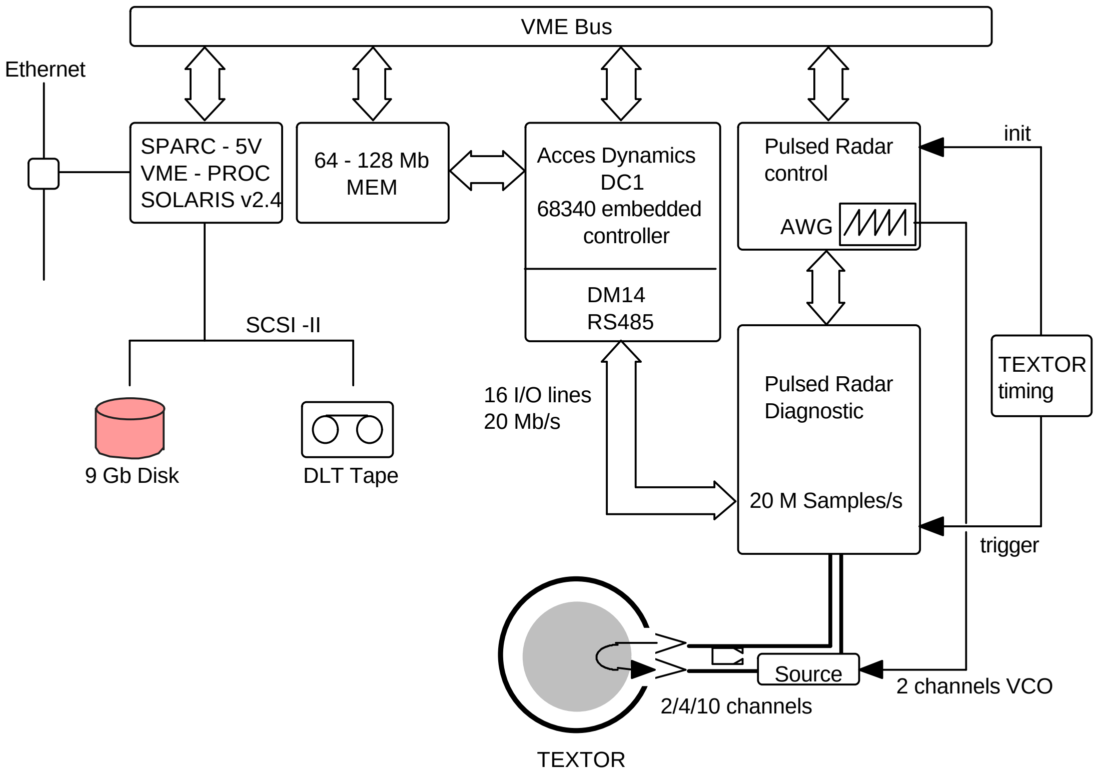


Figure 33: Control, data handling and storage systems

[35] Acces Dynamics, DC1.

The Pulsed Radar Reflector Control

The *PulsedRadar Viewer* is a viewer application for the exemplary “Diagnostics Controller”. This controller is an application, which sets and monitors all the settings used for a specific experiment that is performed on the Textor-94 tokamak. Since there can be various sorts of experiments, there can be various types of PulsedRadar viewers. This document describes an example of such a viewer (controller).

Interface Pulsed Radar Reflector viewer

Since the exact interface depends of the type of experiment there is no need to go into all the full details of the interface and only the basics of a typical “Controller” interface will be described.

```
          enum Event_Type
          {
           UNKNOWN_Event,
           STOP,
           START,
           ARM,
           TRIG,
           DATA,
           HALT,
           Nr_Events
     };
```

The controller typically uses a set of events, which are used to set and monitor the status of the experiment. These would usually contain events like in our example.

```
     enum State_Type
          {
           UNKNOWN_State,
           NoPulse,
           InterPulse,
           PrePulse,
           ActivePulse,
           PostPulse,
           Abort,
           Nr_States
          };
```

The controller reports the status to the viewer during the experiment. Therefore, the interface will typically contain an enumeration of state-types.

```
        struct Config_Type
    {
            long       pre_trigger;
            shortmode;
            shortinterlace;
            long       start_time;
            long       nr_samples;
            long       step;
            long       end_time;
    };
```

The interface will also contain a structure, which defines the configuration. The client uses this to set a new configuration for a new experiment. The exact content of course varies from experiment to experiment.

```
    struct Status_Type
    {
            State_Type  state;
            string      shotnr;
            long        progress;
            long        pre_trigger;
            long        samples;
            long        dc1_status;
            long        awg_status;
            short mode;
            short interlace;
    };
```

Besides general status about the current experiment more advanced status-information is returned to enable the client to view the exact results of the experiment and enables the user (scientist) to actively control the experiment remote.

Besides these structures there are a few other structures, which define the interface and GUI of the client. All these structures can be read and set during the experiment.

An example of how to set the *Config_Type* structure with Java or C++ would for instance be:

```
CurrentConfig = PrsObject.config()  ;    // get configuration

PrsObject.config( NewConfig ) ;          // set configuration
```

Because everything is packed into structures and enumerations, the interface is very flexible. Although the exact contents of these structures and enumeration's may change, all clients will be of a similar form, which enables fast development of such viewers and controllers.

Next to this set of enumeration's and structures, the interface will typically contain a set of functions of which these are the most important.

```
long generate_event(in Event_Type event) ;
```

This function is used to generate an event. This function enables the client to control the Pulsed radar controller.

```
Config_Type check_config( in Config_Type conf );
```

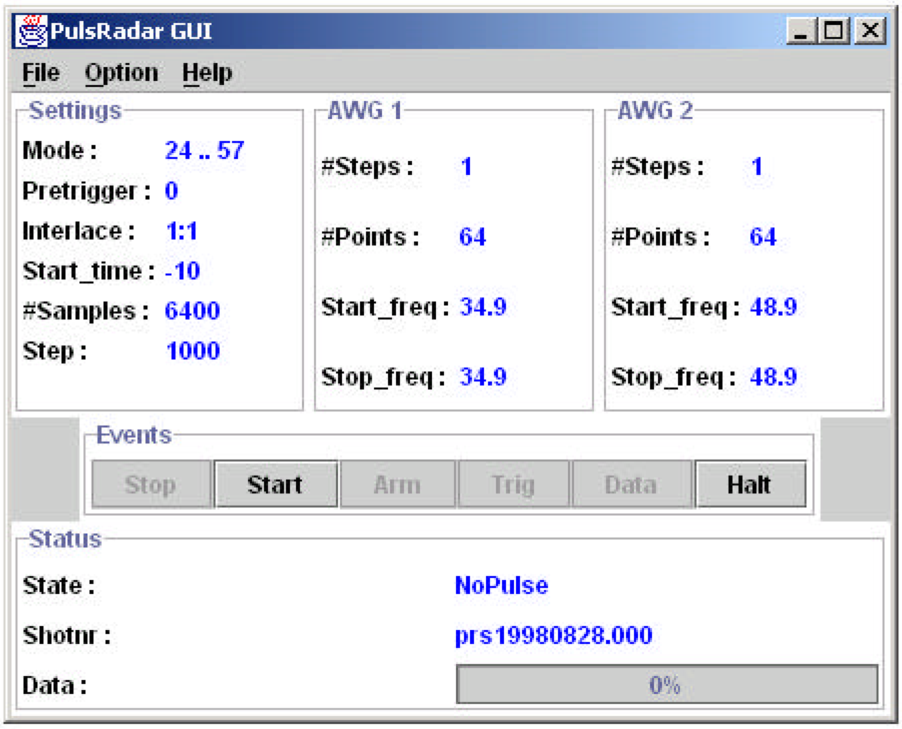
This function is as the name says, used to check a new configuration. Some settings might be theoretically possible, but not supported by the actual equipment that perform the experiment. In addition, conflicting settings might occur. This function enables the client to check such settings before actually trying to set them (which of course will fail if the settings are not valid). It will automatically correct incorrect or conflicting settings.

```
oneway void shutdown();
```

This function will shutdown the remote controller if the experiment is finished.

An example interface.

An example of an interface is presented in Figure 34.



Settings
Mode : 24 .. 57
Pretrigger : 0
Interlace : 1:1
Start_time : -10
#Samples : 6400
Step : 1000

AWG 1
#Steps : 1
#Points : 64
Start_freq : 34.9
Stop_freq : 34.9

AWG 2
#Steps : 1
#Points : 64
Start_freq : 48.9
Stop_freq : 48.9

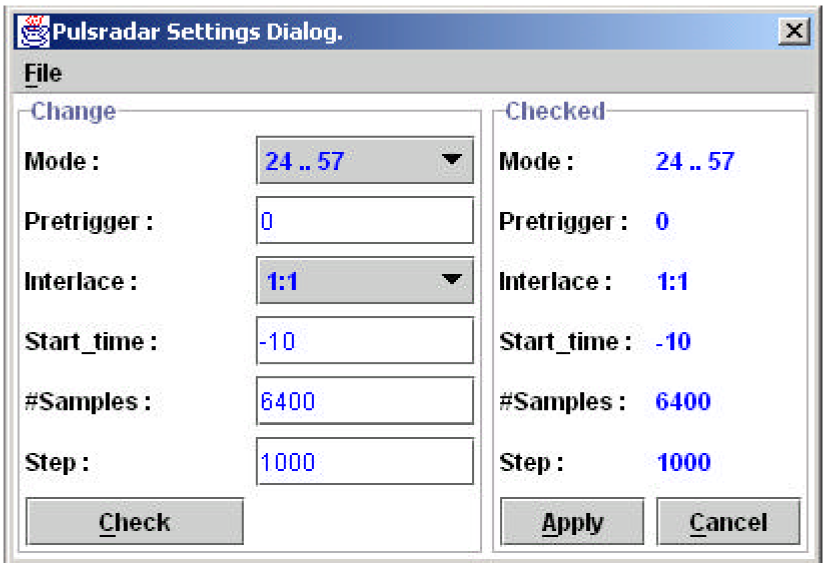
Status
State : NoPulse
Shotnr : prs19980828.000
Data : 0%

Figure 34 GUI for controlling the Pulsed Radar Reflector, either locally or remotely.

The interface shows all the current settings in three panels (Settings, AWG1 and AWG2) and an event-bar, which shows all the events and enables the user to create them too.

On the bottom, the status is being displayed.

A typical settings-screen to set the options can be found in the *Option* menu. See Figure 35.



	Change		Checked
Mode :	24 .. 57	Mode :	24 .. 57
Pretrigger :	0	Pretrigger :	0
Interlace :	1:1	Interlace :	1:1
Start_time :	-10	Start_time :	-10
#Samples :	6400	#Samples :	6400
Step :	1000	Step :	1000

Figure 35 Extra pane (options) to change settings with check.

Note the *Check* button, which is used to validate the new settings.

Performance measurements

We measured the performance of our distributed database architecture in order to see if it can meet the high performance requirements mentioned before. For this, we have used the GigaCluster set-up as shown in Figure 36[36].

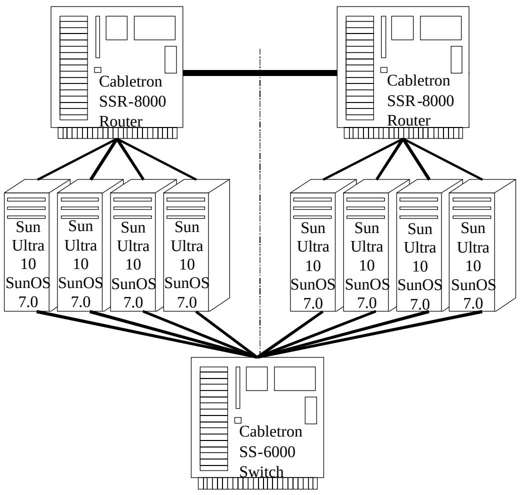


Figure 36 The GigaCluster measurement set-up

The GigaCluster set-up consists of eight Sun-Ultra-10 workstations running SunOS 5.7 and two Cabletron SSR-8000 Smart Switch Routers. The SUNs are grouped in two clusters of four computers. All computers in a cluster are interconnected via a Cabletron router in a switched 1 Gigabit/sec fibre network.

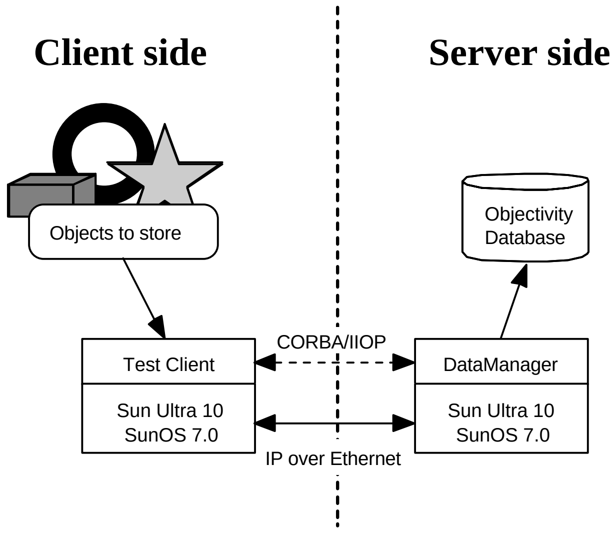


Figure 37 Actual set-up for the performance measurements.

[36] SUN-Ultra-10 GigaCluster project overview and status:http://www.phys.uu.nl/~niderost/gigacluster. This reference is given for a complete overview on the available hardware and is not directly of concern for this report.

The two routers are also interconnected via a 100 Mbit/sec fibre network. Finally, all computers are also interconnected via a 100 Mbit/sec link using a Cabletron switch.

For the performance measurements described in the sequel we used only a part of the cluster. To this end a client (data producer) and a server (data storage) were implemented according to the scheme above, each on one computer in the cluster.

Direct versus CORBA:ANY parameter passing

In our first test, we have run a database with a data manager on a computer of one of the clusters, and a database client on a computer on the other cluster. The measurement was performed with two different CORBA interfaces. Using the first (fast) interface, data was sent as is from the client to the server. Using the second (generic) interface, data was packed into a CORBA:ANY object before transport, and after the transport, this object was unpacked again by the server before storage. The fast interface looks very complicated, since it needs separate methods for storage of every type of data objects. The generic interface is much simpler, but the data packing might influence the performance of the system significantly. The measured times are given in Table 1.

CORBA interface	Client time	Server time
Fast interface	98.48 ± 0.14 sec	70.55 ± 0.11 sec
Generic interface	483.9 ± 0.5 sec	289.7 ± 0.3 sec

Table 1:Time to set up a transaction, store 324 data objects, each consisting of a header and 10^6 bytes of raw data, in a single directory, and commit the transaction. The client time is the total time as seen from the client. The server time is the time spent in the data manager routines at the server.[37]

The errors given are the internal errors in the results of the measurement series, taken with only minimal processes running on the computer, and one active user. Systematic errors depending on the software environment can have much larger influences. Clearly, the time necessary to pack data into a CORBA:ANY and unpack it again adds a considerable overhead. This is true for the server as well as for the client, as can be seen from the measurements of the time spent in the server routines during the previous test (see Table 1 again).

Filling a database

In order to achieve higher performance, we have used only the fast interface in further testing. In the next test, we measured again the time necessary to store 324 objects with 10^6 bytes of raw data. We repeated the measurements 20 times, while we reused the database until it was full. Every time the database was full, we emptied the database and continued our measurement. The result is depicted in Figure 38.

We grouped the 20 measurement results into four series (1 to 6, 7 to 12, 13 to 18 and 19 & 20), since we had to empty the database after every 6 measurements[38]. The number on the x-axis of the graph is the number of the measurement within its series.

[37] These are the routines that implement the CORBA interface. Only the time used to unpack and store the data is included, not the time spent in the IP-stack or in the CORBA IIOP protocol.

The measurements show that the time to store data in a database is slightly dependent on the size of the database. Maximum time is about 10 % above the average. The time does not increase linearly with increasing database size, but shows a characteristic peak just below a 1 GB database size. Perhaps that this is a result of the algorithm used by Objectivity/DB to increase to database file stepwise. We repeated this experiment with different object sizes. The same characteristic appeared, and it turned out that it depends on the amount of data in the database, not the number of objects.

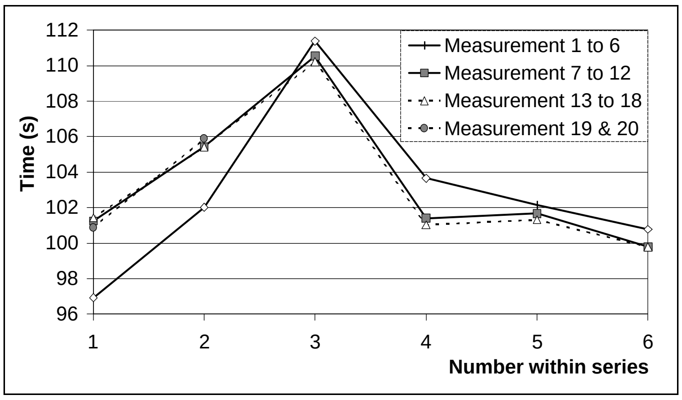


Figure 38 Repeatedly storing data in the same database. The database is emptied after the 6th, 12th and 18th measurement.

One more remark should be made here. The first measurement series started with a completely new database, while the other three reused the database after it was emptied. This difference might explain the difference between the corresponding graphs. The physical file size of the database on the hard disk was small in the first case, but it remained 2 GB after emptying a full (2 GB) database.

Dependency on number and size of objects

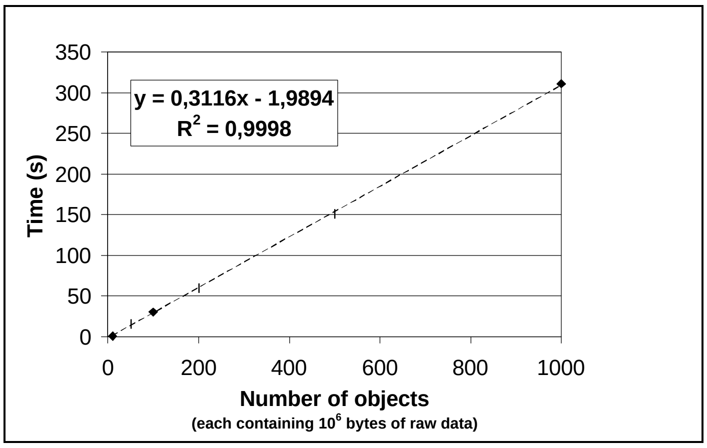


Figure 39 Time needed to store objects containing 10^6 of raw data as a function of the number of objects stored. (R^2 is correlation coefficient squared)

[38] On our test platform, the maximum database size is 2^{31}-1 bytes, or 2 GB. Since we store 320'000'000 bytes per measurement, we hit the database limit during the 7th measurement. This number is not the maximum storage capacity of our architecture, since a federated database can contain many databases.

The following two measurements show the dependency of the performance on the object size and the number of objects stored. They both measure the time at the client involved in storing a number of data objects into an empty database. In the first case, the size of the objects was fixed to 10^6 bytes of raw data, and the number of objects stored in one transaction was varied (Figure 39), in the second case, the number of objects was fixed to 324, and the size was varied (Figure 40).

Both measurements fit well to a linear function y = a x + b. The offset b can be understood as a non-linearity for small number of objects or object sizes respectively. The a-values indicate a storage speed of 3.2 x 10^6 and 3.4 x 10^6 B/s respectively. This is about one third of the raw data storage speed of the hard disk used (10 to 11 MB/s).

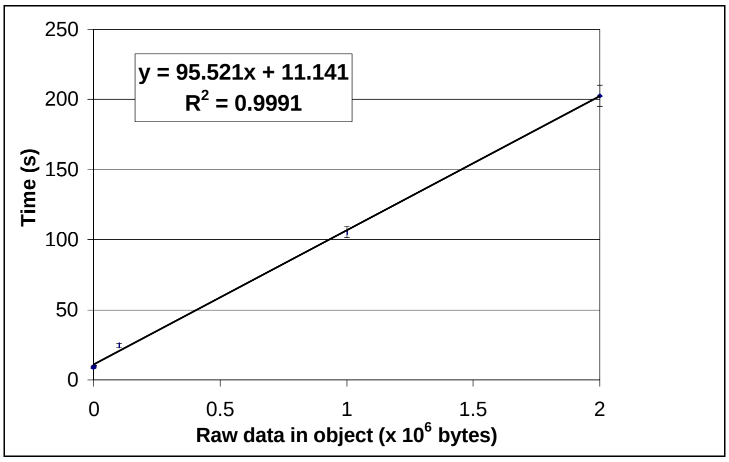


Figure 40 Time to store 324 objects as a function of object size.

To achieve the performance requirements, a storage speed of 500 x 10^6 / 60 = 8,3 x 10^6 B/s is needed. This can easily be achieved using 3 SUNs in parallel (3 x 3.2 x 10^6 = 9.6 x 10^6 B/s).

Finally we measured the time needed to store data using different networks. The results are displayed in Table 2.

Network type	Client time	Network time
Client and server on same machine	89.20 sec	0 sec
10 Mbit/sec UTP	350.77 sec	259 sec
100 Mbit/sec UTP (using SS-6000)	98.51 sec	25.9 sec
1 Gbit/sec & 100 Mbit/sec fiber network (client and server on different clusters)	95.76 sec	25.9 sec
1 Gbit/sec fiber-optic (client and server on same cluster)	91.64 sec	2.59 sec

Table 2 The time measured at the client to start a transaction, store 324 objects containing 10^6 bytes of raw data each and finish the transaction using different networks, and the theoretical minimal time to transfer the amount of data without any overhead over the network.

Considering that the time spent in the data unpack and storage routines always amounts to about 70 s, the times measured for a client and a server on the same machine and for a client and a server interconnected via a 1 Gbit/s network can be explained when it is assumed that about 20 s are spent in the IP-stack routines. The three other times can be explained considering the network limitation, where the network overhead varies from negligible for the 1 Gbit/sec & 100 Mbit/s fibre network to 10 % for the 10 Mbit/s and 100 Mbit/sec switched UTP network Using these figures, it can be seen that at least a 100 Mbit/s switched UTP network is necessary to achieve the performance goal using 3 SUNs in parallel. A 100 Mbit/s shared network would not have enough bandwidth to meet the requirements.

Conclusions

We have designed a database model that is very flexible. It can store any measurement object that is created currently at the Textor ’94 experiment, and we assume it is flexible enough to be able to store any new type of measurement data that will be created in the future.

The database model is embedded in a distributed database architecture using Objectivity and CORBA. The architecture has been optimised for performance, since high performance is of utmost importance in this project.

We have measured the performance of our prototype architecture on a state-of-the-art computer cluster. We used different network configurations to emulate a real-world scenario. Our measurements showed that the prototype architecture can meet the high performance requirements of a Textor ’94 measurement database using SUN Ultra-10 workstations in parallel as database servers together with a 100 Mbit/s switched network.

Performance measurements on a distributed database

There are many tests possible that measure the performance of the demonstrator described above. How useful they are depends mainly on for what purpose they are done. The measurements described in this section are meant to show if the distribution mechanisms work. Results of other measurements, which show the performance of a single DataManager object using a single database, were described before.

Distribution over multiple SUN-Ultra-10 computers

The following measurements have been performed on four computers that are part of the GigaCluster which was described before. The GigaCluster consists of eight SUN-Ultra-10 computers, running the Solaris 7 operating system, Objectivity/DB version 5.1 and the SUN workshop compiler version 4.2. The same computers have been used under the similar conditions for previous measurements, mentioned above. For this test the computers are interconnected using a 10 Mbit/s Ethernet network.

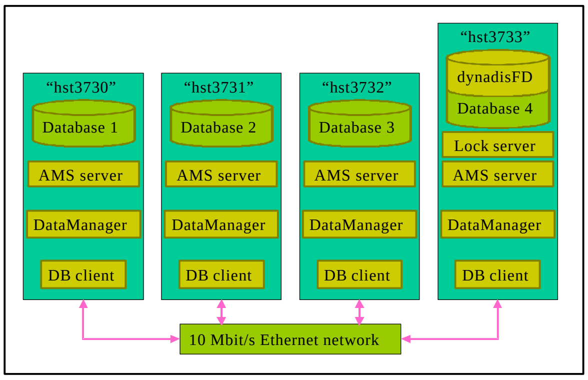


Figure 41 Set-up for the distributed database performance measurements on the SUN-Ultra-10 GigaCluster.

Figure 41 shows the measurement set-up. The measurement database is distributed over four computers. Every computer has one database file, one AMS server, a DataManager, and a database (DB) client that stores data in the database. The database client uses the DataManager that is running on the same computer, and stores data in the local database file. This is the most optimal situation, as it does neither use the AMS servers nor the network to store data. However, the DataManager objects still need the network to connect to the lock server, and to resolve the references to the databases. Initially the DataManager objects only know the location of the federated database.

Using this set-up, the time a database client needs to store 500 MB of raw signal data has been measured. The measurement has been repeated four times, first with only one database client, running on computer "hst3733", then with two clients, three clients, and finally with four clients. The results are shown in Figure 42.

As can be seen from the figure, the parallelisation of the data storage works very well. The graph showing the total processor time indicates that there is only a few seconds overhead associated with the database distribution over two or three computers. Distributing the database over four computers yields a larger overhead, and a much larger uncertainty in the total processor time. The exact reason for this effect is unclear, but it seems plausible that the (shared) network reaches a limit. For example, it might become overloaded and drop packets. This would result in TCP/IP time-outs, which in turn cause a lot of overhead and uncertainty in the total processor time.

In the present set-up the average computer time shows that three computers in parallel are able to meet the performance requirement of storing 500 MB of measurement data within

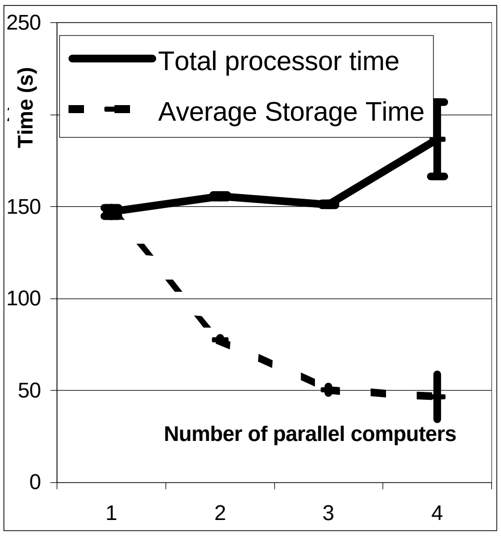


Figure 42 The average time it takes to store 500 MB of raw data, using 1, 2, 3 respectively 4 computers in parallel. The total processing time of all participating computers together is also shown. Every point has been measured multiple times. The error bars show the standard deviation in the results of the repeated measurements. Only the one for 4 computer-case stands out, the others are too small to be visible in the graph.

1 minute. Four computers in parallel should further reduce the time, but the uncertainty in the storage time will become larger. The figure indicates that the requirement of storing 500 MB within 1 minute on 4 computers is not met always. In this test situation distribution of the data storage over three computers seems to be optimal, but would not be exemplary for other configurations.

Performance of a 700 MHz Athlon computer running Windows NT 4

The computer industry increases the performance of their architectures at an incredible pace. The tests of the previous paragraph were performed on computers that are over a year old. To have a view of what a single commodity computer can achieve nowadays, another test has been done. It was carried out on a computer with an Asus K7M motherboard[39] running the Microsoft Windows NT Server 4.0[40]. The CPU was a 700 MHz Athlon-processor[41]. The measurement database was stored on an 18.2 GB Quantum Atlas 10K SCSI hard disk[42]. The test repeated the one in the previous paragraph, but now only for a single DataManager using a single computer. The time necessary to store 500 MB of raw signal was 99 ± 6 seconds in this case.

This measurement indicates that today, two commodity computers working in parallel can achieve the performance goal of storing 500 MB of measurement data within one minute. If the machines would work completely in parallel, it would take them approximately 100 / 2 = 50 seconds to store 500 MB of measurement data. The locking mechanism will increase this time slightly, but, looking at the distribution overhead on the SUN cluster, this overhead would not amount to more than 10 seconds. It is also to be expect ed that in the near future, one single commodity computer will be able to achieve the performance goal all by itself.

[39] Asus K7M motherboard product description:
http://www.asus.com/Products/Motherboard/slota/k7m/index.html
[40] Microsoft Windows NT server 4.0 product description available on the Internet:
http://www.microsoft.com/ntserver/default.asp
[41] AMD Athlon processor product description available on the Internet:
http://krypton.amd.com/products/cpg/athlon/
[42] Quantum Atlas 10K product description available on the Internet:
http://www.quantum.com/products/hdd/atlas_10k/atlas_10k_overview.htm

Network performance aspects

Network performance measurements IPP - FOM – UU

Features

The network performance between the sites at IPP, FOM and UU (see also chapter " Network situation" for the positions in the network and the next paragraph) is measured with a package called RTPL (Remote Throughput Ping Load). The intention of this package is to do periodic net performance measurements between a set of hosts, which can be specified by the user. From a control host, these measurements are started at the participating hosts with a remote shell command. The following tests are executed:

- The *throughput* between each host pair, using the netperf[43] command.
- The *roundtrip times* between each host pair, using the ping command[44,45].
- The *load* of each host with the uptime command. The *load* is measured to be able to relate net performance decrease to eventual machine load.

The measurements are performed at Unix workstations by executing Perl scripts. The crontab utility is used to start the tests periodically at the so called control host. The Perl script at the control host starts the net performance measurements at the test hosts with remote or secure shells. The results are collected at the control host and stored in ZIP compressed data files. The presentation of the data is Web based: A JAVA Applet is used to load the ZIP compressed data files, JavaScript is used to generate dynamically several views to the data in the form of HTML tables. JavaScript directly calls Applet methods to obtain the required data. The Applet can also be used to present a plot of the data, displayed in the tables.

The following data files, also recent ones, of more general interest, can be viewed via the Web in the mean time: [http://www.phys.uu.nl/~blom/doc/net_test/ipp_fom_uu/]
Since these data do not particularly focus on the connectivity between IPP and FOM, they are mentioned here just for completeness. This "archive' contains

- data of the last 7 days.
- For each week of the last half year a data file is available.
- The week's mean values from the last year.
- The day's mean values from the last year.

[43] Netperf Home Page: http://www.netperf.org/netperf/NetperfPage.html
[44] R. L. A. Cottrel, C. A. Logg, and D. E. Martin, "What is the internet doing? performance and reliability monitoring for the HEP community", Computer Physics Communications, vol.110, pp.142--148, May 1998
[45] URL: http://www-iepm.slac.stanford.edu/pinger/

- The mean values, calculated at the periodic measurement times, for the days of the week, averaged during a quarter. The data are stored during a year.
- The mean values, calculated at the measurement times, for the workdays of the week, averaged during a month. The data are stored during a year.

Measurements IPP -- FOM – UU, earlier measurements, August '99

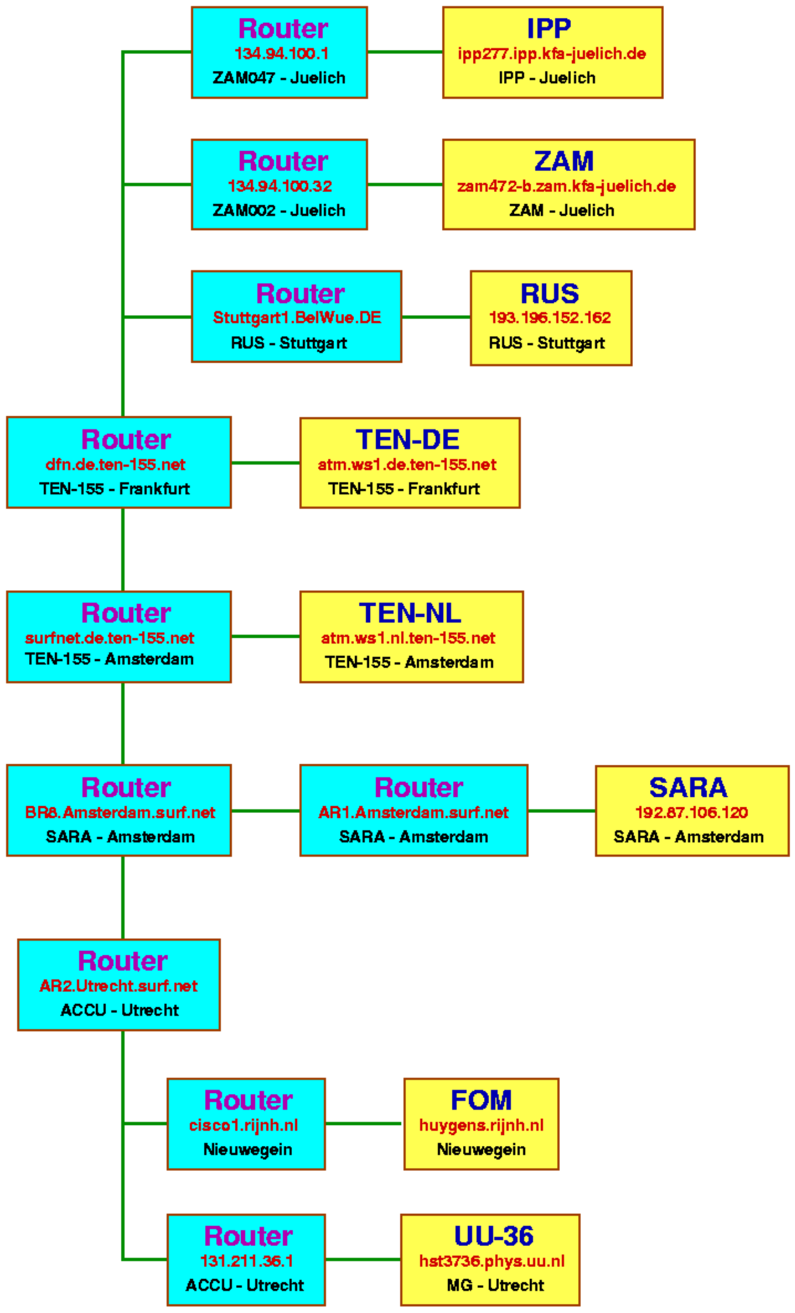


Figure 43 Topology used in the tests, described in the text

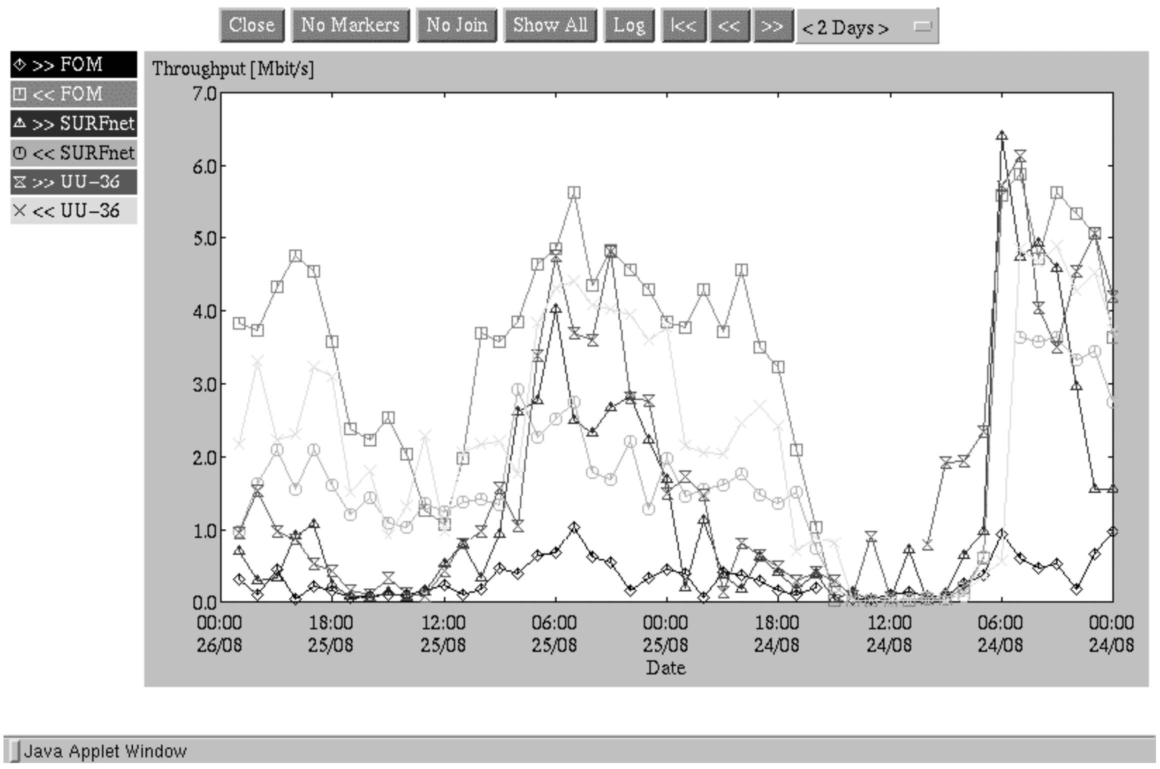


Figure 44 Throughput data between the IPP site and sites at the FOM and UU for two workdays in August. The test direction is specified in the plot labels. The site entitled "SURFnet" is positioned close to the router in Utrecht.

With these features of the package the network performance was monitored between the following sites participating in the DYNACORE initiative:

- Institute for Plasma Physics (IPP) in the Science Centre Jülich (FZJ), Germany (DYNACORE partner).
- FOM-Institute for Plasma Physics Rijnhuizen, Nieuwegein, the Netherlands.
- Institute of Computational Physics, Faculty of Physics & Astronomy, Utrecht University, Utrecht, the Netherlands (DYNACORE partner).

To check if possible bad performance could have been caused by congestion in the network at the FZJ, two other sites at the FZJ were included in the measurements:

- A host at the ZAM department, close to the (DFN) router.
- A host at the ZELAS department at another region of the FZJ.

The August '99 data, here presented, are the most representative for the start of the tests for the DYNACORE deployment. Before, but also after August, not all connections were available. So to evaluate the situation with respect to the connection 'IPP – FOM' this was the most relevant period. In the comparison of the results we focused on the situation to and from IPP. During day time there is a considerable net performance loss at the connections between IPP and FOM and between FOM and UU. In Figure 44 the throughput measurements from the site at IPP to sites at the FOM and UU during two workdays in August are displayed.

The results of the measurements with these sites at the same days as the throughput data from Figure 44 are shown in Figure 45

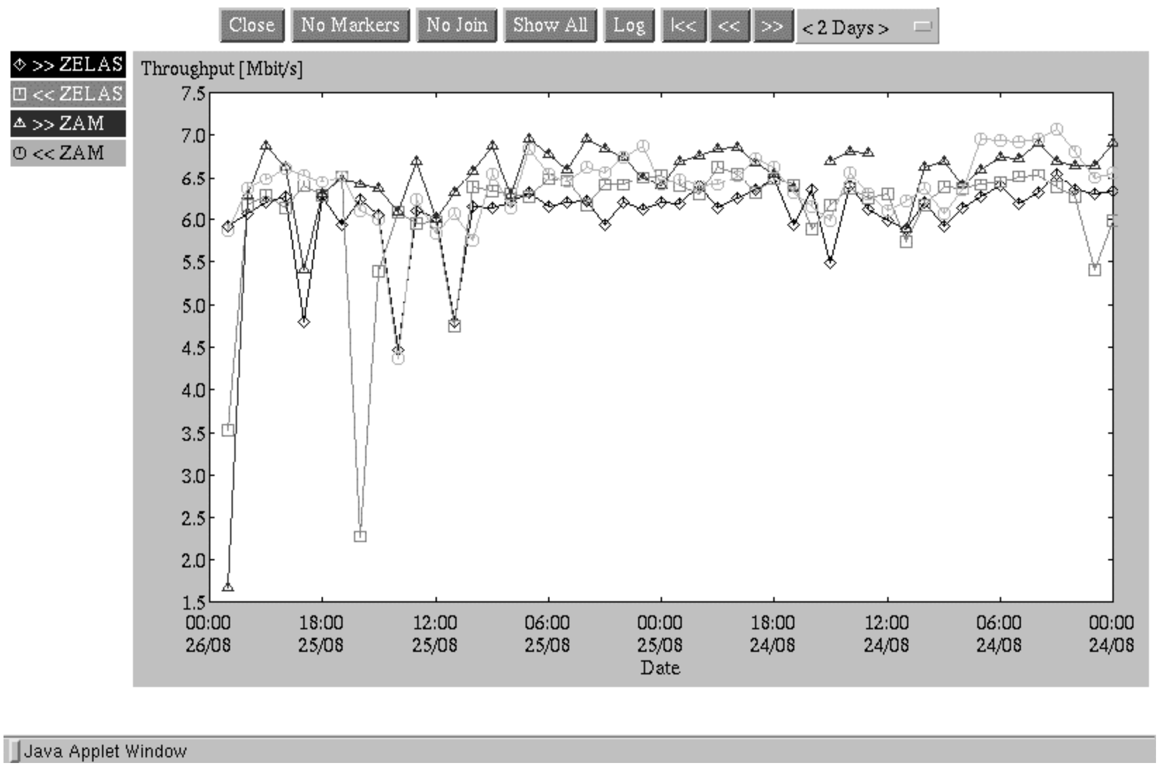


Figure 45 Throughput data between the sites IPP, ZELAS and ZAM at the Forschungszentrum Jülich for two workdays in August.

Figure 45 shows clearly, with the exception of some accidental dips, that there is no congestion at the FZJ network: the throughput values during daytime and night time are not much different.

The general conclusion from the measurements was that at the time of the first network performance measurements the international connectivity was not very stable, but improving. The local networks at FZJ, FOM and UU were rather stable with some exceptions that have been cured in the mean time.

Measurements IPP -- FOM – UU, later measurements, Jan – April 2000

In this paragraph the results of the network performance monitor between IPP - FOM and Utrecht will be reviewed. The attention will be focussed on the results in the first thirteen weeks of the year 2000. Moreover the improvements during the complete observation period (22-8-1999 until 2-4-2000) are discussed. Because bandwidth, and not so much availability, is the limiting factor in the connections, mainly throughput results are presented here.

Sites

The results of the throughput measurements between the following sites will be compared:

Connection	BW [Mbit/s]	Weeks
ZAM<=>UU-36	100	34 (99) - 14 (00)
IPP<=>FOM	10	34 (99) - 05 (00)
TEN-DE<=>TEN-NL	100	37 (99) - 49 (99)

The participating sites where placed at the following locations:

Site	Location
ZAM	ZAM Department, Jülich, Germany.
IPP	IPP Department, Jülich, Germany.
TEN-NL	Dutch PoP TEN-155 network, Amsterdam, Netherlands.
TEN-DE	German PoP TEN-155 network, Frankfurt, Germany.
FOM	FOM Institute Rijnhuizen, Nieuwegein, Netherlands.
UU-36	Computational Physics Uni. Utrecht, Utrecht, Netherlands.

For the connections between these sites the results concerning performance and availability are presented in the following sections.

Time throughput averages

In this section the throughput average values, calculated at the hours of the days for the bi-directional connections ZAM <=> UU-36 and IPP <=> FOM will be compared. There are mean values calculated for working days (Mon - Fri) and for in the weekend (Sat - Sun). The results are obtained for the first thirteen weeks of 2000. This implies that the mean value of a workday (weekend day) is the result of averaging 65 (26) throughput values. Figure 46 presents the hourly throughput values during working days and Figure 47 shows the corresponding values in the weekend.

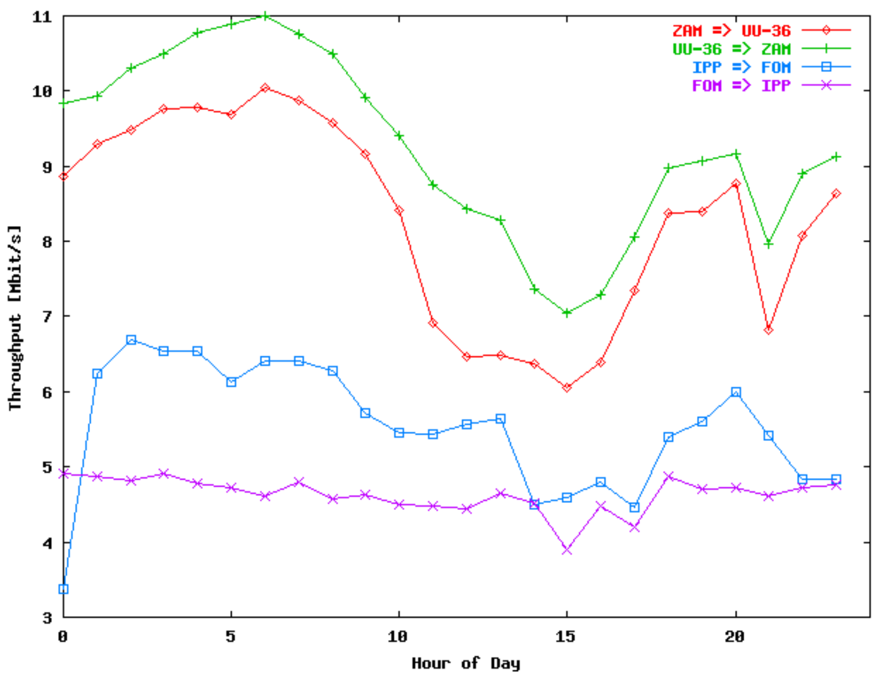


Figure 46 mean workday throughputs in the network between IPP and FOM

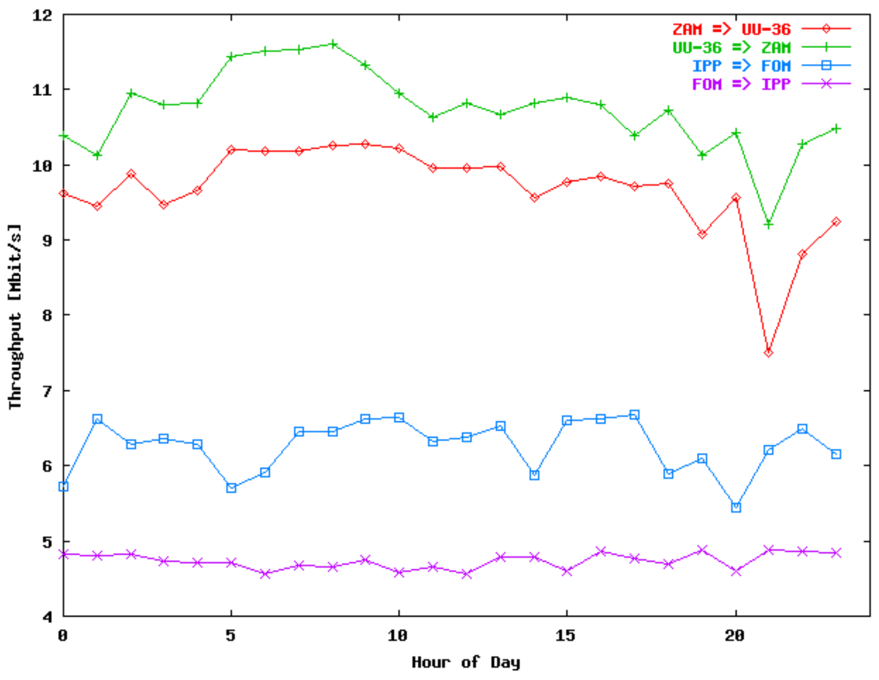


Figure 47 mean weekend throughputs in the network between IPP and FOM

From both figures the following conclusions can be drawn:

- The striking behaviour is the clear performance decrease at working days between 08h - 18h. This is especially true for the connections between the sites with 100 Mbit/s interfaces (ZAM <=> UU-36), but also the connections between the sites with 10 Mbit/s interfaces show a decrease in performance. During the weekend the performance difference between day and night is not so very significant.

- We computed the ratio between the minimum throughput during daytime and the maximum throughput at night for working days. The table below contains this ratio for the various connections (in the values the non-typical performance decreases for IPP => FOM at 00h and for ZAM <=> UU-36 around 20h have been ignored)

Connection	Min-Tput / Max-Tput
ZAM => UU-36	0.60
UU-36 => ZAM	0.64
IPP => FOM	0.67
FOM => IPP	0.79

- With the exception of FOM => IPP all ratios are about the same value. The explanation for this may be that with congestion at a router, the queuing protocols, sliding window adjustment, etc. are responsible that a proportional part of the received packages will be sent to the next hop. This implies that the bandwidth to the next hop will be related to the incoming bandwidth. This mechanism breaks down when packets are lost due to heavily congestion at the router. Therefore, these results were less clear found in earlier throughput measurements where the performance of the network was worse.

- The performance decrease at 00h for the IPP => FOM connection is typical for this connection. The result is unknown, but probably local to the IPP. May be backup activity or other regular service jobs, generating local traffic may be the cause. The load of the IPP host at that moment is not larger than otherwise, so it is not a performance feature. The reason that we do not find it in the reverse situation, FOM => IPP, may be due to the overall lower bandwidth of that connection.

- The performance decrease around 20h for the connection ZAM <=> UU-36 is not clear. However, other results show that the cause is probably situated in the Utrecht University network. The performance diminution is found for all days of the week.

Throughput histograms

In this paragraph histograms from throughput counting are presented for the connections ZAM <=> UU-36. The bin counts are given as percentage from the total # of observations. The results are obtained for the first thirteen weeks of the year 2000, but only at working days (Mon - Fri)

The following histograms are presented: Figure 48 and Figure 49 show the histograms for connection ZAM => UU-36 and v.v. UU-36 => ZAM during working hours (08h - 18h);

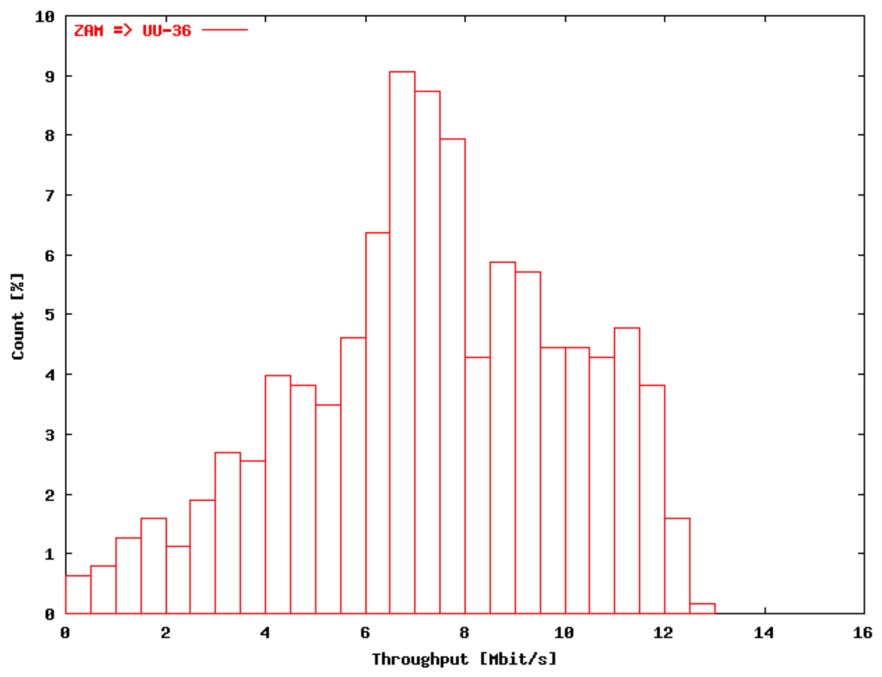


Figure 48 Histogram of the throughput distribution during working days (08h - 18h) ZAM – UU

Figure 50 and Figure 51 display the histograms for connection ZAM => UU-36 and the reverse, UU-36 => ZAM, during the evening and night (18h - 24h; 00h - 08h).

The results lead to the following conclusions:

- In the evening and night the higher throughput bins are more frequently represented than during workday, as may be expected.
- As expected, during workday the lower bins (Tput 5 Mbit/s) are, due to congestion, more filled than during the evening and night.
- For the connection UU-36 => ZAM there exists more heavily congestion (Tput 1Mbit/s) than for the ZAM => UU-36 connection.

- With the exception of ZAM => UU-36 at nighttimes, all histograms show a clear maximum (shifted to a larger bin at night compared to the working hours). The distribution of the higher throughput bins shows a shape similar to a Poisson distribution, probably due to router -> queue algorithms, while there exists a relative flat shape for the lower bins. In this area more incident driven protocols may play a role, like for instance packet retransmission.

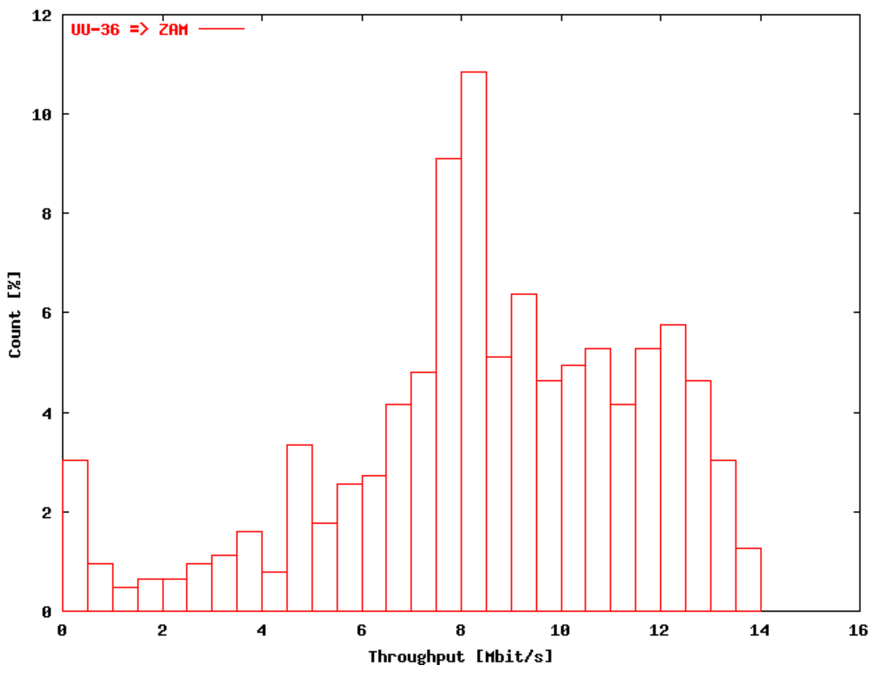


Figure 49 Histogram of the throughput distribution during working days (08h - 18h) UU – ZAM

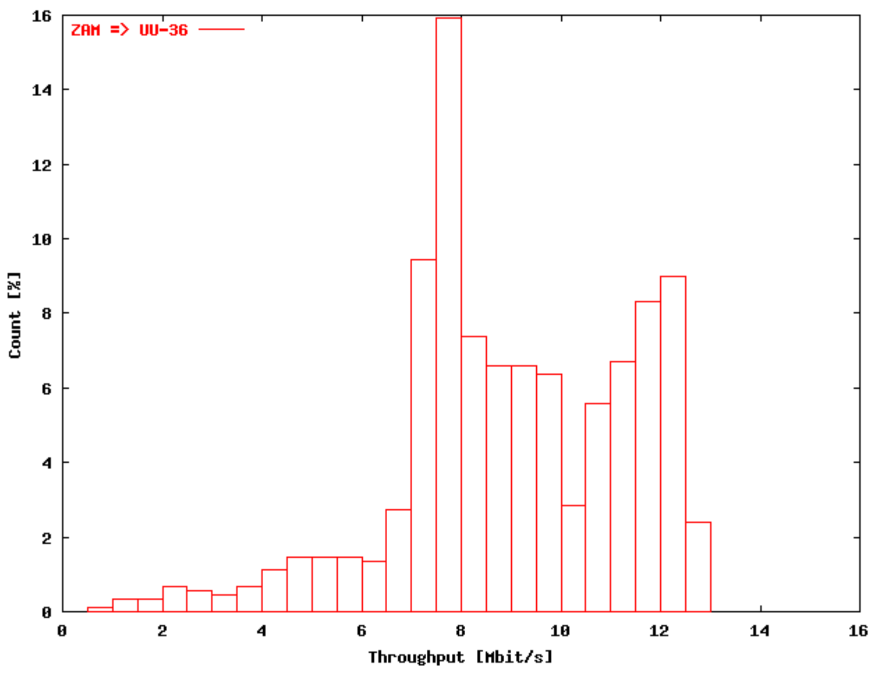


Figure 50 Histogram of the throughput distribution during nights of working days working days (18h - 24h; 00h - 08h) ZAM - UU

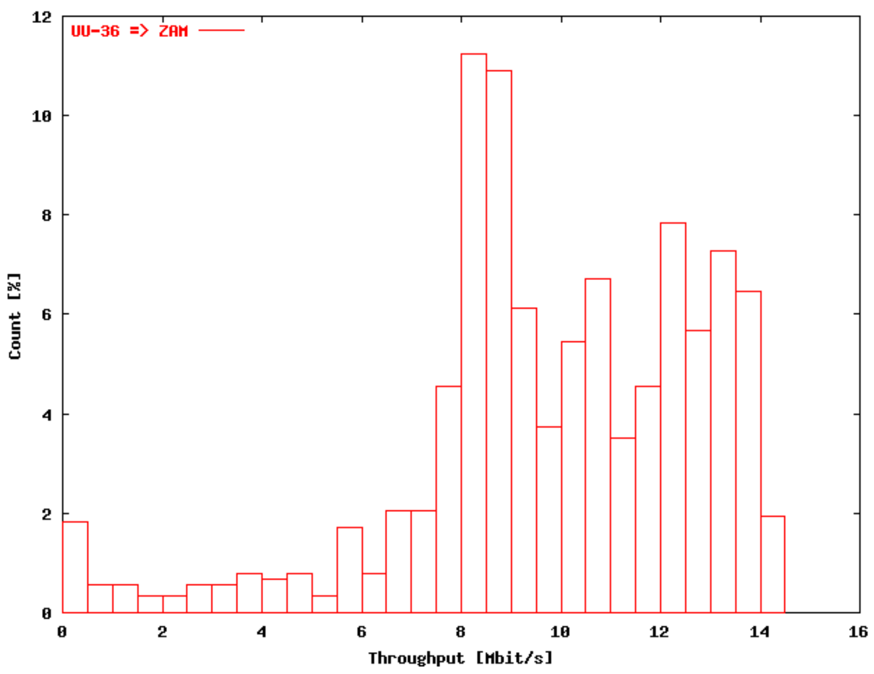


Figure 51 Histogram of the throughput distribution during nights of working days working days (18h - 24h; 00h - 08h) UU - ZAM

Overview time throughput averages

In this section we give an overview of the throughput average values, calculated at the hours, from all available workday data for a particular connection. The data are presented in the form of 3D plots, where the x-axis represents the hour and the y-axis the week of year (1999 and 2000). The plots for the following connections at workdays (Mon - Fri) are shown:

TEN-DE <=> *TEN-NL*

During a couple of weeks at the last half of 1999, hosts at the Frankfurt and the Amsterdam PoP of the TEN-155 network were added to be able to see the influence of router tuning in the throughput performance measurements. Figure 52 displays the performance of the TEN-DE => TEN-NL connection and Figure 53 the reverse connection. In both plots the data are averaged over the workdays of one week.

The following conclusions can be given:

- Both plots clearly show the performance improvements due to the router tuning.
- In fact there were two stages in the tuning: after large improvements around week 42 (1999), there was also a tuning around week 48 where the performance during daytime was improved. Meanwhile also some high performance peaks especially for TEN-DE => TEN-NL) were flattened.

ZAM <=> UU-36

Figure 54 presents the hourly throughput values for the connection ZAM => UU-36 and Figure 55 for the reverse connection. In these plots the data are averaged over the workdays of two weeks.

The following conclusions can be given:

- The same conclusions are valid as for the TEN-DE <=> TEN-NL connections.
- Around week 52 1999 there is a maximum for all hours. This is caused by the traditional low seasonal traffic, especially in the Netherlands, in that period.

In begin of 2000 there was a further improvement of the performance.

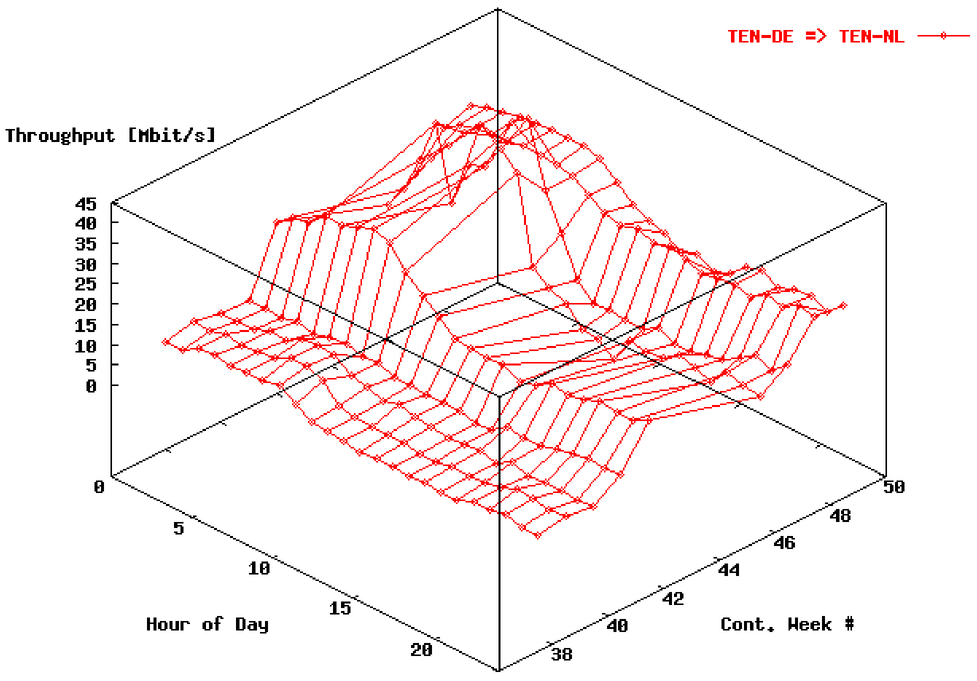


Figure 52 Performance of the network backbone between Germany and the Netherlands

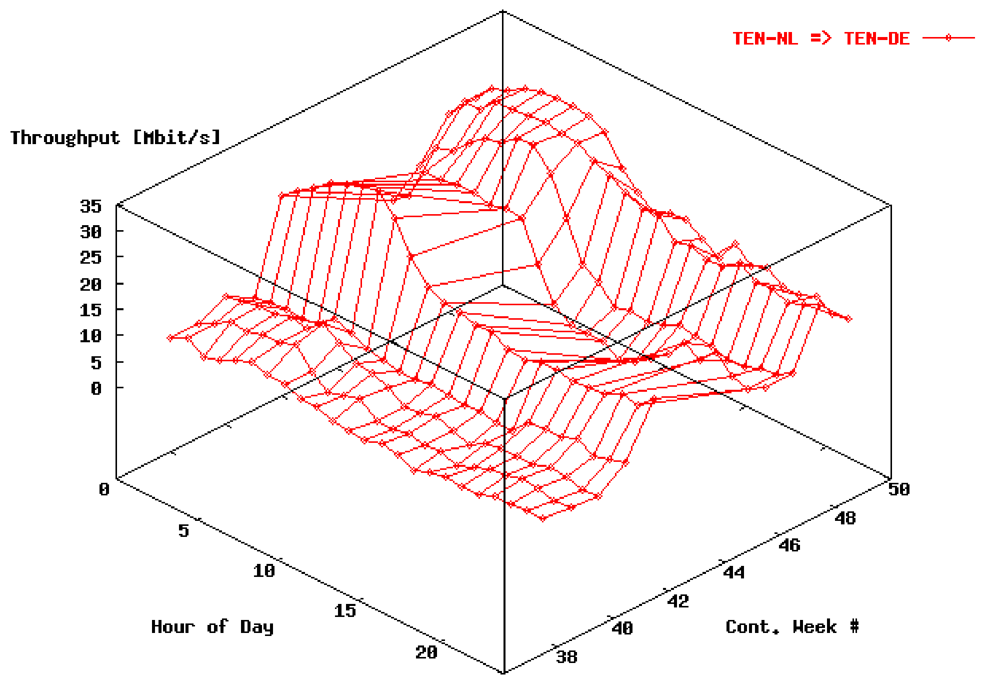


Figure 53 Performance of the network backbone between the Netherlands and Germany

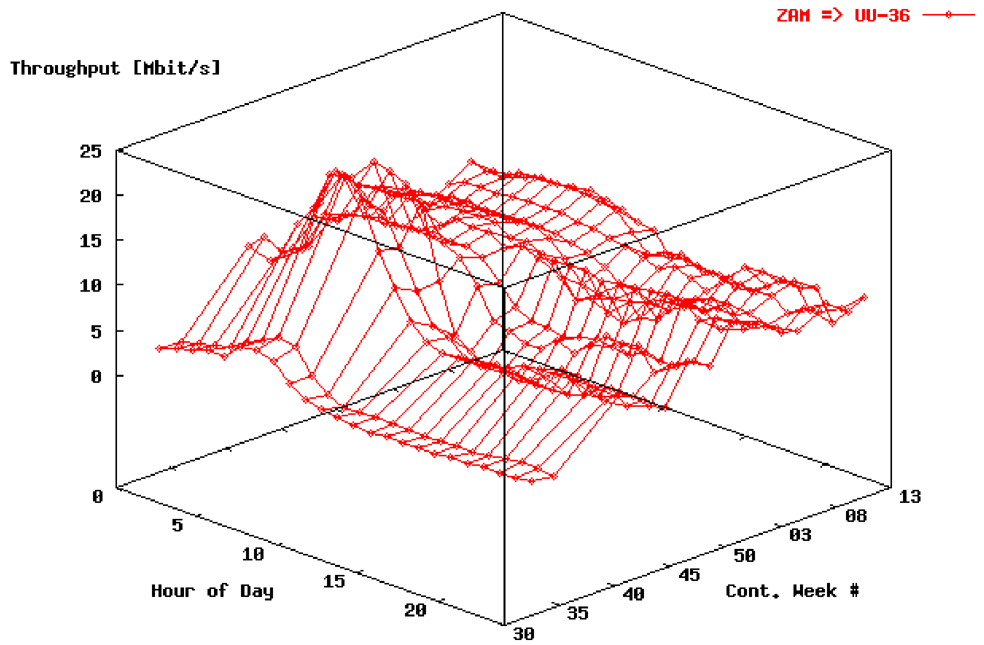


Figure 54 Performance of the network between ZAM-FZJ Germany and UU, the Netherlands.

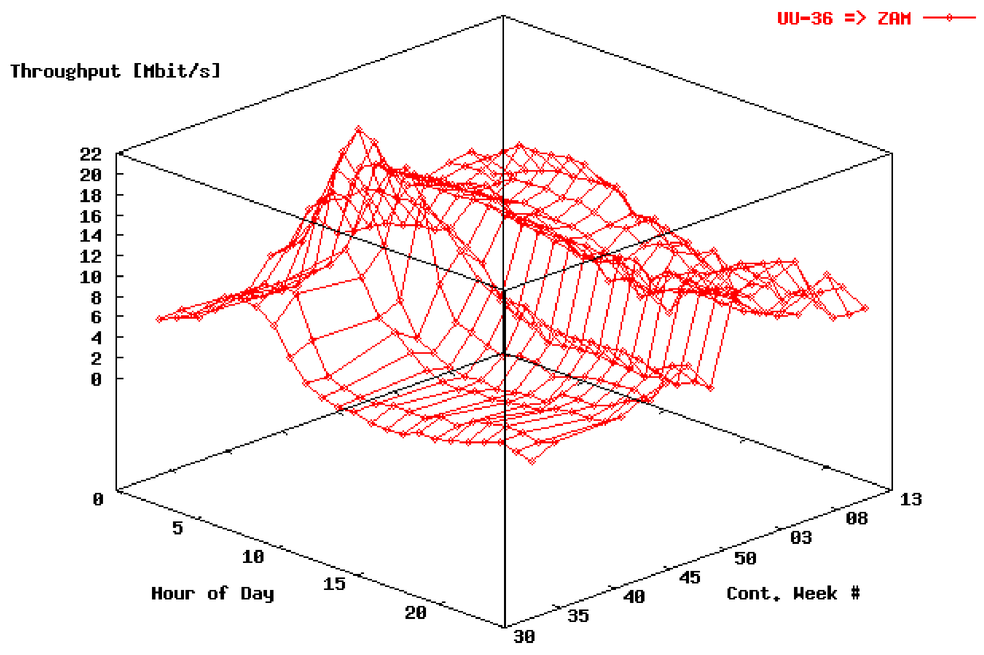


Figure 55 Performance of the network between UU, the Netherlands and ZAM-FZJ Germany.

Bad performance events

Table 3 Events with Tput < 0.5 Mbit/s for the connection ZAM <=> UU-36, vv

Date	Time	Site	Site	Tput	Ping			lost
dd/mm/yyyy	hh:mm:ss	1	2	Mbit/s	min[us]	avg[us]	max[us]	[%]
29/03/2000	17:00:08	UU-36	ZAM	0.32	31.200	57.075	85.500	5.000
29/03/2000	15:00:04	UU-36	ZAM	0.21	29.300	45.431	96.500	7.500
29/03/2000	13:00:02	UU-36	ZAM	0.02	23.500	46.862	144.000	2.500
29/03/2000	12:00:07	UU-36	ZAM	0.03	19.500	27.762	44.500	2.500
29/03/2000	11:00:03	UU-36	ZAM	0.07	30.500	60.181	81.500	5.000
29/03/2000	10:00:03	UU-36	ZAM	0.02	19.100	34.594	63.000	5.000
29/03/2000	09:00:03	UU-36	ZAM	0.08	18.300	23.492	45.000	2.500
29/03/2000	08:00:03	UU-36	ZAM	0.01	17.800	22.600	44.100	2.500
29/03/2000	07:00:04	UU-36	ZAM	0.02	16.500	20.341	112.000	2.500
29/03/2000	06:00:03	UU-36	ZAM	0.15	17.500	20.051	32.300	2.500
29/03/2000	05:00:03	UU-36	ZAM	0.02	17.300	18.611	20.100	0.000
29/03/2000	04:00:04	UU-36	ZAM	0.02	17.200	18.624	21.300	0.000
29/03/2000	03:00:03	UU-36	ZAM	0.01	17.000	19.868	61.000	0.000
29/03/2000	02:00:03	UU-36	ZAM	0.02	17.500	23.264	33.300	5.000
29/03/2000	01:00:03	UU-36	ZAM	0.14	17.400	20.060	52.600	7.500
29/03/2000	00:00:04	UU-36	ZAM	0.01	17.200	19.697	24.700	0.000
28/03/2000	23:00:02	UU-36	ZAM	0.03	17.100	19.224	22.900	2.500
28/03/2000	22:00:01	UU-36	ZAM	0.00	19.100	25.067	35.000	5.000
28/03/2000	21:00:02	UU-36	ZAM	0.00	17.500	20.013	28.800	0.000
28/03/2000	20:00:03	UU-36	ZAM	0.00	17.400	21.742	30.800	5.000
28/03/2000	19:00:02	UU-36	ZAM	0.01	17.700	19.561	23.200	0.000
28/03/2000	18:00:03	UU-36	ZAM	0.01	17.800	19.263	23.700	0.000
28/03/2000	17:00:01	UU-36	ZAM	0.02	18.300	25.151	53.900	7.500
28/03/2000	15:00:02	UU-36	ZAM	0.23	98.500	268.350	448.000	20.000
27/03/2000	18:00:04	UU-36	ZAM	0.34	18.300	20.792	23.800	0.000
22/03/2000	18:00:01	UU-36	ZAM	0.16	25.600	35.103	47.300	0.000
22/03/2000	12:00:06	UU-36	ZAM	***	83.000	121.276	152.000	0.000
20/03/2000	16:00:13	UU-36	ZAM	***	25.200	27.197	33.400	0.000
16/03/2000	16:00:06	UU-36	ZAM	0.10	25.300	31.614	44.700	5.000
16/03/2000	16:00:06	ZAM	UU-36	0.08	25.000	32.243	59.000	2.500
16/03/2000	15:00:05	UU-36	ZAM	0.00	25.100	27.881	34.700	2.500
16/03/2000	15:00:05	ZAM	UU-36	0.00	25.000	28.314	36.000	7.500
16/03/2000	14:00:12	UU-36	ZAM	0.02	24.400	27.229	37.300	0.000
16/03/2000	13:00:08	UU-36	ZAM	0.12	19.100	22.389	31.000	2.500
16/03/2000	12:00:02	UU-36	ZAM	0.04	18.700	21.543	29.000	2.500
16/03/2000	11:00:05	UU-36	ZAM	0.01	18.400	22.418	46.300	0.000
16/03/2000	10:00:02	UU-36	ZAM	0.48	334.000	422.105	505.000	0.000
16/03/2000	10:00:02	ZAM	UU-36	0.21	448.000	527.811	629.000	2.500
14/02/2000	15:00:04	UU-36	ZAM	0.28	44.200	47.850	51.300	10.000
08/02/2000	12:00:07	ZAM	UU-36	0.40	30.000	36.270	43.000	2.500
28/01/2000	09:00:05	UU-36	ZAM	0.46	30.500	36.554	43.300	2.500
24/01/2000	17:00:06	ZAM	UU-36	***	766.000	850.094	919.000	15.000
23/01/2000	14:00:06	ZAM	UU-36	0.03	25.000	52.667	83.000	5.000
20/01/2000	10:00:07	UU-36	ZAM	***	23.000	25.126	30.400	0.000
13/01/2000	16:00:07	UU-36	ZAM	***	17.500	19.463	23.300	0.000

Table 3 shows the monitor parameters for all events where Tput < 0.5 Mbit/s, which is a arbitrairy number, for the first thirteen weeks of 2000. Only the events for the ZAM <=> UU-36 connections are listed.

The following conclusions can be derived from this table:

- There are no structural performance decreases (collapses).
- The performance diminutions are clustered at the same dates. They are probably caused by network problems. This is especially the case for the events during the night.
- The most events are registered for the connection UU-36 => ZAM. They can also be observed as local maxima in the histograms for the corresponding bins.

Table 4 Failures in the network listed according date / time for the last part of the reported period (end April 2000) for the connection FOM – IPP, vv

Date	Time	Site	Site	Tput		Ping		lost
dd/mm/yyyy	hh:mm:ss	1	2	[Mbit/s]	min[us]	avg[us]	max[us]	[%]
03/05/2000	12:30:05	FOM	IPP	0.08	21.511	31.684	100.014	2.500
02/05/2000	14:30:05	FOM	IPP	0.45	26.727	32.291	39.207	5.000
01/05/2000	01:30:04	FOM	IPP	***	94.804	95.665	96.626	0.000
01/05/2000	01:30:04	IPP	FOM	0.03	93.600	94.621	95.746	0.000
30/04/2000	14:30:05	FOM	IPP	***	94.999	95.503	96.045	67.500
30/04/2000	04:30:04	FOM	IPP	***	18.284	19.087	20.518	0.000
30/04/2000	04:30:04	IPP	FOM	0.31	17.550	18.313	19.540	0.000
29/04/2000	03:30:06	IPP	FOM	***	17.550	24.973	219.375	15.000
25/04/2000	15:30:04	FOM	IPP	0.35	25.996	31.196	35.238	10.000
14/04/2000	15:30:05	IPP	FOM	0.05	26.325	80.163	254.475	22.500
14/04/2000	14:30:05	FOM	IPP	0.20	28.613	36.875	41.483	0.000
14/04/2000	13:30:05	FOM	IPP	0.05	30.705	103.175	267.399	27.500
14/04/2000	13:30:05	IPP	FOM	0.14	31.200	114.903	240.342	30.000
13/04/2000	08:30:04	IPP	FOM	0.03	23.400	84.153	373.724	15.000
12/04/2000	14:30:05	FOM	IPP	0.46	30.107	49.902	137.668	10.000
11/04/2000	12:30:06	FOM	IPP	0.21	30.782	160.100	508.345	37.500
11/04/2000	12:30:06	IPP	FOM	0.16	25.350	108.707	253.500	22.500

Table 4 shows the events with Tput < 0.5 Mbit/s, for the connection IPP – FOM directly. Since the first week of April this connection was monitored once again. There is still not much statistics.

Overall Conclusions

- The connection Jülich - FOM / UU performs quite satisfactory. There are no structural performance decreases.

- The required bandwidth of 10 Mbit/s can only be obtained during the night. However, improvements in the TEN-155 network in the near future may help to improve this picture.

- The initial problems in the video connections between IPP and FOM were one of the reasons to start the measurements on network performance. Some of the problems could be allotted to the unpredictable behaviour of the international connections (TEN-155). These problems were brought to the attention of national network providers (Surfnet, DFN) and were subject to discussions in the international research network associations (Dante, Terena). Since then some improvements could be noticed, but the situation is still not clear.

- Locally the situation seems stable enough, but since we can expect in the future a nominal bandwidth of > 100 Mbits/s, the local infrastructure has to be upgraded to this figure preferentially.

- ISDN does not seem a right solution, since we are expecting to use at least 10 Mbits/s capacity in the future. Moreover ISDN is a fading technology which, perhaps in a not to far future, can be substituted by the service providers. Using the facility offered by the international research networks (IP based) seems still the right way to go.

- Quality of service is still a research topic. QoS is closely related to the solution of authentication, authorisation and accounting in IP environments. Also interworking of products from different vendors is still not solved at the momen

Audio and video connections

Introduction

In remote participation, one of the goals is to use videoconferencing in both the Remote Control Room but also for meetings. In order to meet the requirements, some alternatives have been investigated. In this chapter we will deal with the recommended architecture for video conferencing that emerged from the investigations, the one that uses the “Armada Cruiser” hardware, present at the validation sites for the DYNACORE (PP) prototype (The partners in the TEC collaboration). As will be explained in the following, this is still *the best solution for point-to-point, quality video conferencing at limited bandwidth*. We will describe the way in which the present tools and hardware can be used for multi-cast conferencing. This requires additional hardware at the validation sites (TEC), but could be tested during this DYNACORE project using the Surfnet facilities, present at the UU[46]. The AV application recommended is not directly integrated in the PP-DYNACORE prototype, but functions parallel with it.

We will also briefly comment on the “public domain” software solutions based on Mbone: VRVS, VIC and RAT, which could be considered as an alternative and is viz. used in the international community that participates at experiments at JET[47].

The VRVS package (vrvs.cern.ch) includes the VIC (currently version 2.8) and RAT (3.0.29) tools for video and audio respectively. We evaluated these tools using point-to-point connections. The use of a VRVS reflector offers multi-point facilities for these tools, it is dealt with in a next section.

Conferencing clients

Overview IP based clients

There are a number of IP based videoconferencing clients available. Here we can distinguish the hardware clients, such as the VCON Armada PC cards of which at all three partners of the DYNACORE project two systems are available, and software clients, such as NetMeeting and Cu-SeeMe.

Hardware clients nowadays offer a reasonable quality with quite low bandwidth demands (from 128 kbits/s up to 384 kbits/s). The available VCON systems all follow the H323 standard. This standard is now quite common and (should) guarantee interoperability.
Software clients, however, are a lot cheaper or cost nothing at all (i.e. NetMeeting, VIC + RAT). Some follow the H323 standard (NetMeeting, and Cu-SeeMe Pro) whereas others don’t (VDOPhone and VIC + RAT). The quality of software clients is still moderate but improving fast. The current state of these clients is that they can be used as a kind of telephone with low quality video. For the "remote participation" project however, software clients are not considered to be an option.

[46] http://contact.surfnet.nl/mcu/. Unfortunately only in Dutch
[47] see 21-st Symposium on Fusion Technology, Madrid, Sept 2000, Conference Contributions, p 335

In Figure 56, the remote video window of the VCON client is shown. The current VCON software (version 4.01) allows for automatic bandwidth adjustment and synchronisation of video and audio. Only 384 kilobit is needed for a good quality videoconference.
In addition to audio and video, the VCON systems support T.120 data sharing for chat and whiteboard. This standard also includes sharing programs.

Tests

From the tests of the last year we can conclude that hardware clients offer a very usable quality. Typically the following performance is measured:

- Video frame rate: 30 frames per second.
- Used bandwidth: 384 kilobit/second, excluding bandwidth for data (320 kilobit for video and 64 kilobit for audio).
- Video format: CIF, i.e. 352x288 pixels.
- Measured delay: approximately 0.5 seconds point-to-point for long distance connections. Not much difference is measured for European connections and connections from Europe to the U.S.
- Larger bandwidth settings up to 1500 kilobit yield a better video quality, i.e. smoother motion and less pixelisation. Less bandwidth settings of (at least) 128-kilobit still offer a usable connection, but for our applications the image is not smooth enough. With the current status of the hardware clients about 10 frames per second (in CIF format) can be transmitted with acceptable pixelisation.

We tested the VIC and RAT tools on two PC's on different VLANs, but inside the same building. The audio latency, using the RAT tool was up to four seconds, especially when at the same time the VIC tool was running. When only one microphone was un-muted, no VIC tools were running and no audio driver was using full duplex, the delay was about 1 second.

The VIC tool is much quicker although (in case of a point-to-point connection) selection of video device and IP port numbers must be done by the end-user. It is also possible to use a configuration file, but the use of these tools is far less simple than that of the well-known H.323 systems like NetMeeting and VCON MeetingPoint. The quality of VIC is comparable to that of NetMeeting.

Multipoint Servers

Most of the above systems only provide point-to-point connections. For the H323 standard “Multipoint Control Units” (MCU) are available. These devices provide the possibility to organise meetings with a larger number of participants at different locations.

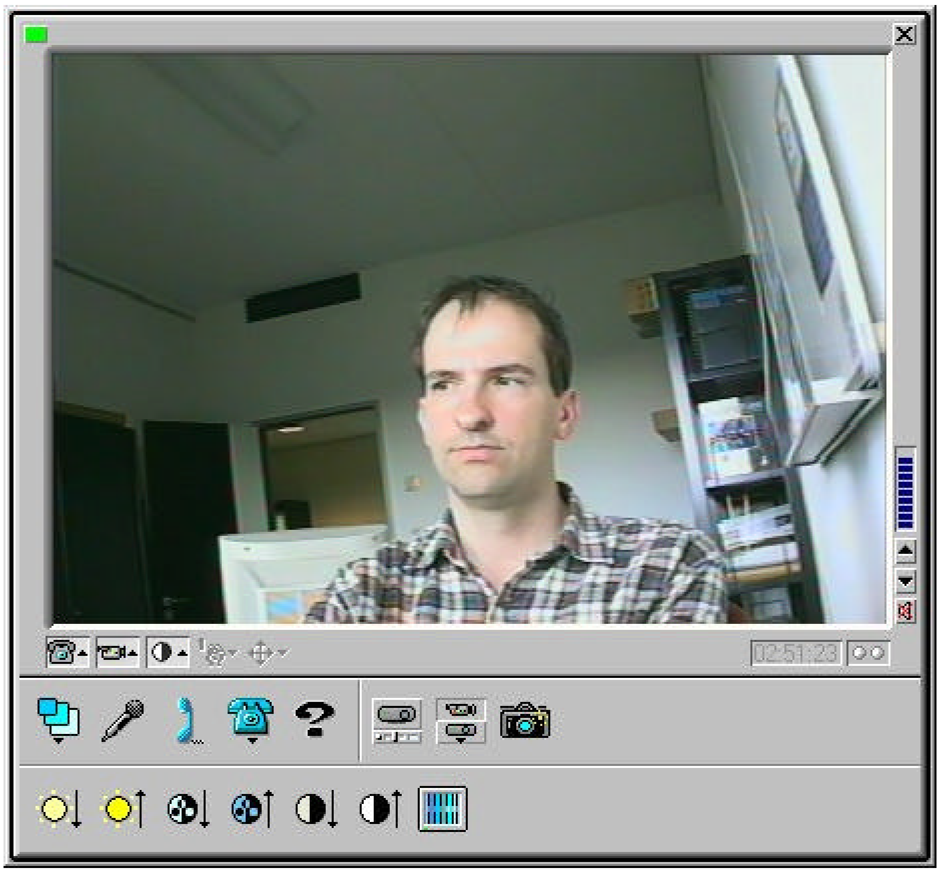


Figure 56 VCON client

We tested several MCU's from different vendors (PictureTel, RadVision and WhitePine). All of these MCU's still have some drawbacks for integration within the project. The WhitePine (software) MCU does not offer a quality that meets the high standards of our hardware clients. The PictureTel 330 (software) MCU works quite well but we were not able to use data sharing with our VCON clients. As far as we can conclude from the specifications the T.120 server of the PictureTel should be compatible with the VCON clients. For the RadVision (hardware) MCU, audio, video and data sharing works well, but there are some less attractive security aspects. The only way to prevent anybody from using this MCU is to predefine the IP addresses of allowed videoconferencing clients. Apart from this drawback, this MCU, which allows up to 9 client connections at 384 kilobit per second, works very well. The video and audio quality is almost equal to that of a point-to-point connection. The video is normally switched to the loudest speaker, but it is also possible to use chair control. In case of chair control, a WWW client can switch the video that is distributed to the participants.
As far as VRVS concerns: VRVS is a server for the well-known VIC/RAT tools. The end-to-end delay with RAT (audio) should be 1 second, not including transcoding in the server. This simply lies in the specification of the chosen CODEC for audio. With systems that work well with Netmeeting however, we measured much larger delays.

Recommendations

All parties need a system that is always switched on and always on auto answer (this way any party can conduct tests without having to ask the other parties). Of course, the audio of such a test system can be muted. It is preferable to point the camera at a moving object such as a clock, or out of a window. In this way the video quality can be checked at any time.
For our VCON systems, now a multicast option is available. We recommend testing this multicast option as an alternative to an MCU.
For data sharing we recommend using the T120 standard. For this standard Hardware whiteboards are available. These whiteboards just look like normal whiteboards, and copy its contents to the remote system. We would like to evaluate such systems.

MCU Recommendation

We recommend a RadVision MCU-323 with our VCON clients. The RadVision is a hardware MCU allowing 3 to 15 client connection. Using more clients can be achieved by stacking or cascading MCU's. In both ways a virtual MCU with more connections is constructed by combining two or more real MCU's. The tested MCU has the following specifications:
Software: MCU-323 version 1.5 (build 1.5.0.6) with OnLAN Configure 1.6.0 (build 1.6.0.19).
For H323 calls with a bandwidth of 384 kilobits per second as specified above under "Tests", 9 simultaneous connections are supported.
The tested MCU is a dedicated hardware device supporting up to 15 video calls and up to 24 audio only calls. The MCU comes with a software upload tool to upgrade the software from any windows 9x/NT machine. Our unit was configured software version 1.5 (build 1.5.0.6).

The tested MCU is still available as a "free-love" MCU, this means that people can connect to it when it is not used by SURFnet (its owner).

Information on when the MCU should be available and how to connect can be requested by e-mail: h.m.a.andree@phys.uu.nl

As the VIC and RAT tools offer a quality that doesn't match that of hardware H.323 systems by any means we do not investigate the use of a VRVS server yet. Software conferencing systems may be very promising in the future, but at the moment only hardware systems seem to offer the quality and that is needed in future Virtual Control Room collaborations.

Dissemination of results

Account of the progress in the Dynacore project was given at several occasions. We present below a list.

B.U. Nideröst et al., Objectivity / Corba Distributed database Performance on a Gigabit Sun-ultra-10 Cluster, IEEE Trans. on Nucl. Sc. 47 (2000) 313-318.

Information on the exact working of the Dynacore architecture, and the measurement of its performance. Also published in the 11th IEEE NPSS Real-Time Conference Record.

B.U. Nideröst et al., Objectivity / Corba Distributed database Performance on a Gigabit Sun-ultra-10 Cluster, 11th IEEE NPSS Real-Time Conference Record, Alphagraphics, Santa Fe (NM), 1999, pp. 442-445.

Information on the exact working of the Dynacore architecture, and the measurement of its performance. Also published in the IEEE Trans. on Nucl. Sc. 47

E.A. van der Meer et al., A distributed Plasma Physics experiment system using CORBA, in S.C. Schaller (ed.), 11th IEEE NPSS Real-Time Conference Record, Alphagraphics, Santa Fe (NM), 1999, pp. 438-441.

This article gives a global overview over the Dynacore project: its features, architecture and design decisions. It describes the ideas as of summer 1999, not the implementation that the demonstrator uses today.

G. Kemmerling et al., Development of an integrated data storage and retrieval system for TEC, 2nd IAEA Technical Committee Meeting on Control, Data Acquisition and Remote Participation on Fusion Research Conference Report, Lisbon, 1999, to appear.

This article gives a work in progress overview over the Dynacore project, as of summer 1999.

E.A. van der Meer et al, Combining Objectivity and CORBA: “A laborious marriage” at European Technical Forum Objectivity/DB, Munich Oct. 1999

An overview of matching the demands of the middleware to existing solutions of an object database.

M. Korten et al., Upgrading a TEXTOR Data Acquisition System for Remote Participation using Java and Corba, 2nd IAEA Technical Committee Meeting on Control, Data Acquisition and Remote Participation on Fusion Research Conference Report, Lisbon, 1999, to appear.

General info on the work in progress over the Dynacore project, as of summer 1999.

B.U. Nideröst et al. .A software architecture for remote participation at the TEXTOR-9 experiment,21-st Symposium on Fusion Technology, Sept 2000, Madrid. pp 336.

The contribution gives an overview of the Dynacore project and the obtained results. Special focus on the data viewer.

.

Since the Dynacore system architecture opens the way for experimentalists to retrieve each other's data we have opened the possibility under certain restrictions to use the DynaDemo at the location: http://hst3731.phys.uu.nl/dynademo/. One can find a description at this location together with the demo's, i.e. an applet for retrieving TEXTOR-94 status, which is not dealt with explicitly in this report, but which we added for convenience. The DataViewer deals with real data from present and past. The scientists of TEC have no objections for a free use of these data, so the data is not severely protected according the given security scheme described before, but the hooks are available in the software to do so. The control of the Pulsed Radar reflector diagnostic deals with a dummy experiment that we have implemented on one of the computers at the UU. The dummy mimics the real behaviour of a diagnostic, running at TEXTOR-94.