
Remote Participation Services

Report I

Werkgroep Fysische Informatica, faculty Physics
and Astronomy, Utrecht University

E.A. van der Meer,
B. Niderost,
A. Taal,
H Blom,
H.M.A. Andree,
C.T.A.M. de Laat,
W.Lourens

Utrecht, 9-12-99

Content

Content.....	1
Summary.....	5
Preface.....	6
Network performance measurements IPP - FOM - UU.....	7
Introduction.....	7
Features	8
Measurements IPP -- FOM -- UU.....	9
Conclusion.....	10
Overview videoconferencing systems	12
Introduction.....	12
Conferencing clients	12
Tests.....	12
Overview IP based clients	12
Multipoint Servers.....	13
Recommendations	13
MCU Recommendation.....	14
Measurement data	15
Introduction.....	15
CORBA.....	16
Database	16
Storage Hierarchy.....	17
Database Distribution	18
Data Manager	20
Interface DataManager.....	20
<i>Transaction specific operations</i>	20
<i>Data specific operations</i>	20
Access rights and the DataManager	22

Extended interface.....	22
Performance measurements	23
Direct versus CORBA:ANY parameter passing.....	25
Filling a database.....	25
Dependency on number and size of objects	26
Conclusions	27
Appendices	29
Syntax.....	30
Store	31
Any_2_object.....	34
getDynaObjectRef	37
CORBA Specification of the DataManager.....	39
IDL Interface Description DataManager	49
CORBA Object Diagrams.....	54
Objectivity Specification and Class design	56
DDL for Objectivity database layout	60
Objectivity class diagrams	67

Summary

Remote participation services rely on stable network connections between partners. Research was conducted in order to get information about the present status of national and international research network backbones as well as about the situation locally (IPP and FOM). Measurements indicate that wide area research networks are still not in a stable state. However improvements can be observed. In the near future it is expected that a bandwidth in the range of 10 - 20 Mbit/s can be obtained, even during the day. The situation in the local institutes (IPP and FOM-Rijnhuizen is already stable. It is recommended that in the local institutes the bandwidth will be eventually upgraded to 100 Mbit/s.

Remote monitoring of experiments cannot do without a video conferencing system. Also collaboration meetings will be conducted using this facility. At present, video conferencing (point-to-point) using hardware solutions (VCON) have shown to be adequate. Multi cast video conference is feasible if one implements certain standards and uses specific solutions. It is recommended where necessary to bring the hard- and software up to the required level.

Storing data via CORBA into an object database has proven to meet the requirements of 500 MB/min in a limited amount of servers (3). One has to take an implementation route via specific data types in CORBA (not the any type) and use an efficiently structured database. The complete design of the database used during the tests is given in Appendices. The specifications can be adapted easily to other implementations.

Preface

This document is the first report on issues related to the contract "Remote participation services" between IPP-FZJ and WFI-Utrecht University .

In the contract the main subjects to research are distinguished as:

- Performance aspects of an object database to be used in plasma physics experiments and related issues
- Interfaces between a CORBA architecture and the (object) database
- Interconnections via an electronic network (public and private)

In the following report all these issues will be treated in the sense of giving an overview of the present status and where possible give recommendations for improvements in the next steps in the contract.

The subjects treated in more detail are:

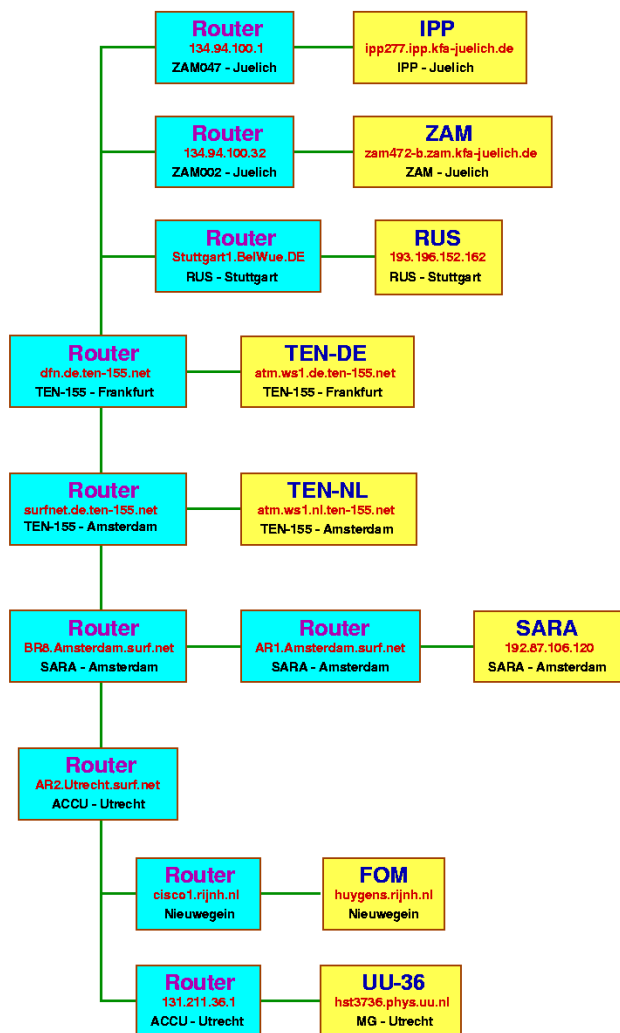
- Status of local (IPP and Rijnhuizen) and European research networks and some recommendations
- Interface between CORBA and Objectivity database. The DataManager. Status and suggested future tasks
- Performance test to be performed on the federation of the database. Aspects of distribution of data.
- Audio / video conferencing. Status and recommendations for improvement using existing hard- and software.

The starting point for the research is a requirement analysis for the interface between a client (user, scientist) that wants to either store data or retrieve data and the persistent data itself. The client that wants to store data can be an instrument, retrieval of data will be usually the business of a scientist or technician. The clearness of such an interface is of utmost importance for large scale implementations. The interface should give the scientist / technician a kind of template along which he can develop his software. The description of such an interface and the outlay of the data structures that will reside in some kind of persistent environment are given in an appendix of this report. The development sketched in chapters 'Measurement data' and 'Data Manager' starts from this analysis.

Network performance measurements IPP - FOM - UU

Introduction

In order to get an impression to what extent capabilities of the underlying network would influence those of the remote participation architecture, several performance measurements were carried out. At first a very coarse topology was defined.



- Figure 1 Topology for measuring network performance between IPP - UU and FOM. Measured is the response of workstations coupled directly to essential routers in the network. The measurements cover TEN-155 performance regionally.

Later on several measuring points (in local workstations) have been added. The scheme that was used for network performance measurements between IPP - UU, IPP - FOM(Rijnhuizen), and UU - FOM is indicated in the figure above.

The August '99 data, here presented, are the most representative in the period covered by this report. Before and after August not all connections were available. So to evaluate the situation with respect to the connection 'IPP - FOM' this is the most relevant period. We will reconsider to establish the connections once more when

we have the impression that the overall situation has improved. When more statistics are available we will discuss the (final) results in a next report.

The general idea from the measurements is that the international connectivity is unto now not very stable, but improving. The local networks at FZJ, FOM and UU are rather stable with some exceptions that have been cured in the mean time.

Features

The network performance between the sites at IPP, FOM and UU 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`¹ command.
- The *roundtrip times* between each host pair, using the `ping` command^{2,3}.
- 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

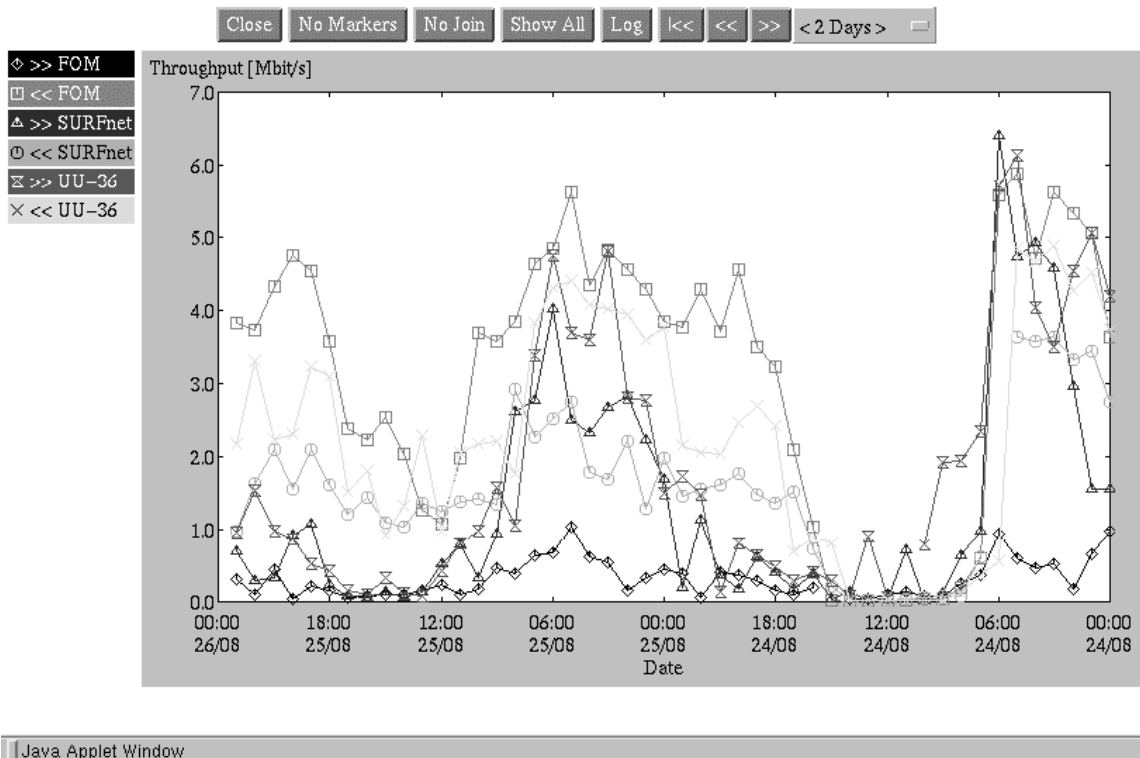
- data of the last 7 days.
- For each week of the last half year a data file is available.
- The week mean values from the last year.
- The day mean values from the last year.
- The mean values, calculated at the periodic measurement times, for the days of the week, averaged during a quarter. The data are stored during a year.

¹ Netperf Home Page: <http://www.netperf.org/netperf/NetperfPage.html>

² 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

³ URL: <http://www-iepm.slac.stanford.edu/pinger/>

- 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.



• Figure 2 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.

Measurements IPP -- FOM -- UU

In this version of the package the network performance was monitored between the following sites participating in the Dynacore⁴ initiative:

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

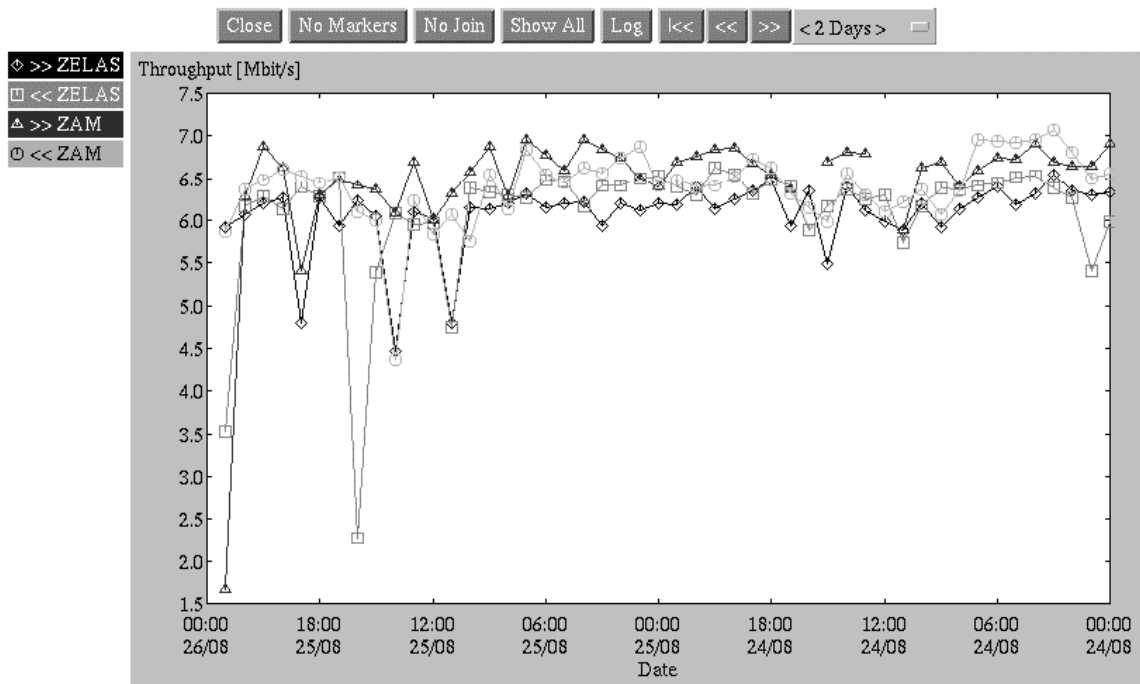
In the comparison of the results we will focus on the situation to and from IPP. During day time there is a considerable net performance loss at the connections between IPP and FOM and between FOM and UU. In Figure 2 the throughput measurements from the site at IPP to sites at the FOM and UU during two workdays in August are displayed.

To check if the bad performance was caused by congestion in the network at the Forschungszentrum Jülich, two other sites at the FZJ were included in the measurements:

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

⁴ Dynacore EU-TAP project

The results of the measurements with these sites at the same days as the throughput data from Figure 2 are shown in Figure 3



• Figure 3 Throughput data between the sites IPP, ZELAS and ZAM at the Forschungszentrum Jülich for two workdays in August.

Figure 3 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. For this reason the sites at ZELAS and ZAM have been replaced temporarily by hosts close to the routers of the provider (a/o. TEN-155). However at this moment the ZAM site again participates in the measurements.

A possible congestion source for the future may be the fact that the IPP network is only 10 Mbit/s. This may be insufficient for extensive data transfer and video conferencing. In principle the TEN-155 line from Frankfurt to Amsterdam has a much larger capacity: incidentally, between the TEN-155 site in Frankfurt and one of our workstations in Utrecht, throughput values up to 20 Mbit/s during night-time are reached. Without further investments the local IPP network may become a bottleneck in the future.

Conclusion

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 Mbit/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 moment.

Overview videoconferencing systems

Introduction

In remote participation, one of the goals is to use videoconferencing in both the Remote Control Room and for meetings. In order to achieve this goal, a lot of tests have been done. In the following document, the current status of videoconferencing techniques in general is outlined.

Conferencing clients

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 data bandwidth (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 kbit/s yield a better video quality, i.e. smoother motion and less pixelisation. Less bandwidth settings of (at least) 128-kbit/s still offer an 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.

Overview IP based clients

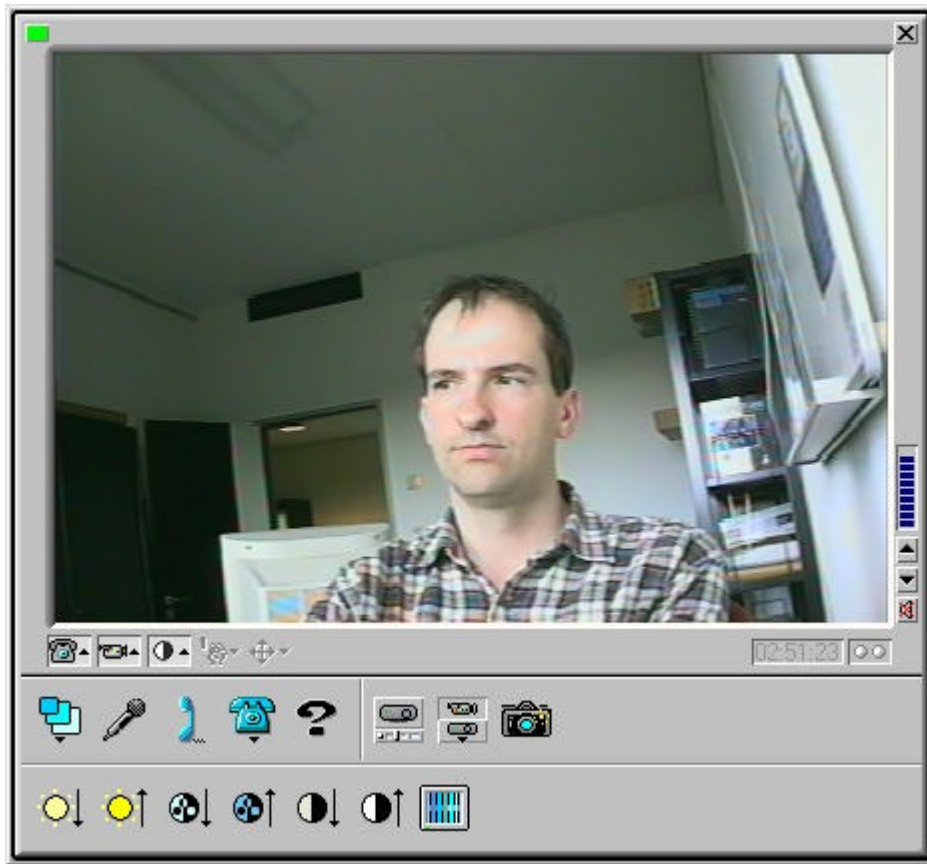
There are a number of IP based videoconferencing clients available. Here we can distinguish the hardware clients, such as the VCON Armada PC cards of which at all three partners of the Dynacore project two systems are available, and software clients, such as NetMeeting and Cu-SeeMe.

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 H.323 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 H.323 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, software clients are *not* considered to be an option.

Below, 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 kbit/s 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.



Multipoint Servers

Most of the above systems only provide point-to-point connections. For the H.323 standard “Multipoint Control Units” (MCU) are available. These devices provide the possibility to organise meetings with a larger number of participants at different locations.

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 kbit/s, 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.

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. However at this moment the available software does not yet meet our requirements entirely and we cannot evaluate its functionality properly. We will continue investigating this matter.

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

For multi-point conferencing we recommend a RadVision MCU-323. To our VCON clients this MCU behaves like a normal H.323 endpoint. The RadVision is a hardware MCU allowing 3 to 15 client connection, coupled via a LAN and H.323 protocol. Configured in this way both MCU and clients are “hardware”, the latter performing as good H.323 clients. 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 H.323 calls with a bandwidth of 384 kbit/s per second as specified above under "Tests", 9 simultaneous connections are supported.

Introduction

A measurement database will always be part of a Textor '94 experiment. This database is used to store all the data created by diagnostics and auxiliary equipment during a shot. As a past design this database is implemented as a collection of files on an Open VMS system. 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.

A standard diagnostic consists of a number of CAMAC crates with many ADC channels. 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 shot⁵.

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 prototype⁶ will incorporate its own 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 30 shots per operation day, 80 days a year, adding up to TB of data per year.
- The database needs to be accessible from many different computer platforms, locally and in virtual control rooms.

⁵ 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

⁶ This would be a prototype to be developed in the framework of the Dynacore project (EU-TAP-RE4005). For this prototype a definite choice of the database is not foreseen. Database could either be an existing one (TEXTOR type or DOM4-FOM-Rijnhuizen) or a commercial one like Objectivity, which is considered in this report.

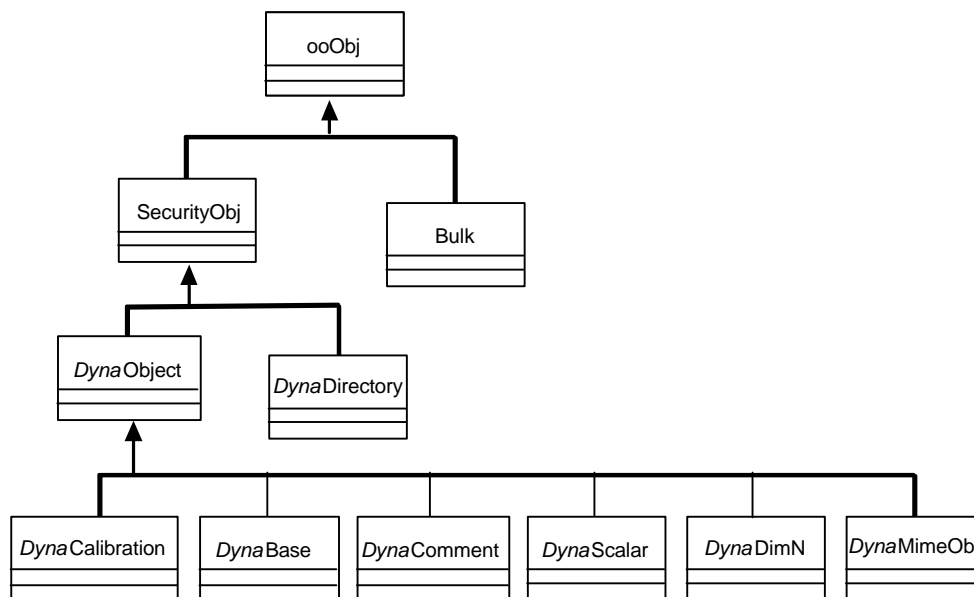
A hard requirement has been defined as follows: the measurement database must be able to store 500 MB of data within 1 minute.

CORBA

CORBA⁷ 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 architecture⁸ 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 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. A specialised data manager will be discussed in the section "Data Manager", it will be referred to as DataManager.

Database



• Figure 4 Inheritance tree of the measurement database classes

As a possible candidate for database implementation we investigated the abilities and performance of an object database.

The object-oriented database is used to store a predefined set of data classes. Our database model defines these classes. (See Appendix Objectivity Specification and Class design). They are displayed in Figure 4.

⁷ CORBA homepage: <http://www.corba.org>

⁸ With CORBA architecture is meant the architecture foreseen in the Dynacore project.

The central class in the model is the class *DynaObject*. Its subclasses are instantiated as the measurements objects, e.g. *DynaScalar*, *DynaDimN* and *DynaMimeObj*. These are wrapper objects for a specific type of measurement data generated at Textor 94. A measurement object contains a reference to an object of the class Bulk, which contains the raw data. The Objectivity⁹ 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 *DynaBase* objects, which contain information on the measurement bases and to a *DynaCalibration* 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 *DynaComment*.

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 *DynaComment*, *DynaCalibration* and *DynaBase* objects allows clients to reuse these objects, such that many measurements that use the same calibration, for example, can reference the same *DynaCalibration* 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 *DynaCalibration* 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 *DynaComment*, and, if necessary, parsed by a specialized database client. In the case that a *DynaComment* contains this type of information, the *DynaComment* should obey certain syntax rules, in order to give a handle to how this information should be used. A *DynaComment* can have a reference to another *DynaComment* for this purpose. Our data browsers¹⁰ do not yet use 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.

Storage Hierarchy

Objectivity 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.

⁹ Objectivity, Inc.: <http://www.objectivity.com>

¹⁰ To be described in a next report

For easy data access, every container has a *DynaDirectory* 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 *DynaDirectory* object. This way, modules and sub modules resemble a Unix-like directory structure, which is stored in a flat Objectivity container.

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*.

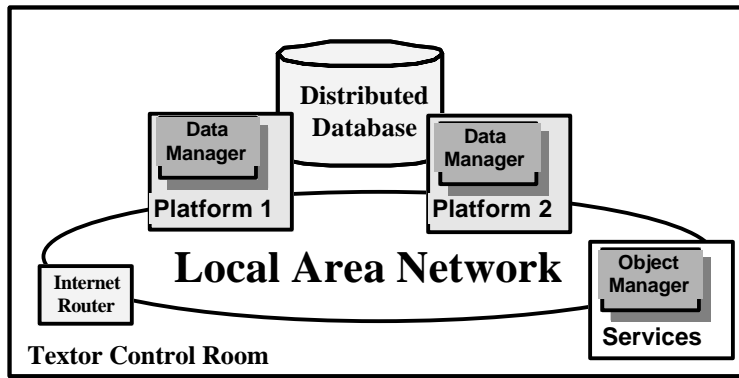
Database Distribution

Objectivity allows for the distribution of a federated database over multiple computers. Each computer can hold one or more databases of the federation¹¹. 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 5 shows a typical set-up of a distributed architecture. We are planning to use 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 the next phase of the project.

¹¹ A federation is mandatory for Objectivity, like a Master database in MS SQLServer. Under the umbrella of the federation, be it one, specific databases can be initialized. These databases can be addressed via the federation. It will be cumbersome to make more federations, as to keep data apart.



• Figure 5 Typical set-up for a distributed database

The prototype also provides for an *object manager*¹², 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 only been implemented at its most simple form. For the performance test we included the functionality in a dedicated data manager.

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.

¹² To be described in the next report

Data Manager

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 way the DataManager actually stores the data. It is possible that the data is stored in flat files, or in a relational database or in an object-oriented database.

Interface DataManager

Accessing the DataManager proceeds through CORBA, where at the DataManager side a C++ ORB (Object Request Broker, implemented in C++) is employed. The operations of the CORBA interface can be divided into *transaction* specific operations and *data* specific operations. These operations are listed below (for more information of the data types referred to in this paragraph see the Appendices “CORBA Specification of the DataManager” and “Objectivity Specification and Class design”).

Transaction specific operations

void start ()

Starts a new transaction with the DataManager. If a transaction is already active, an Error exception will be raised with type `NestedTransaction`.¹³

void commit ()

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 with type `NoTransaction` will be raised.

void commitAndHold ()

Commits the changes made to the database so far, but does not end the current transaction. If no transaction is currently active, an Error exception with type `NoTransaction` will be raised.

void abort ()

Aborts the changes made to the database in the current transaction. If no transaction is currently active, an Error exception with type `NoTransaction` will be raised.

Data specific operations

void store (in any object, in string path)

Creates a new object in the database and creates an entry which binds the object to a symbolic name, specified by `path`. The syntax of `path` should be a legal according to the grammar defined in ... If an entry with the same symbolic name already exists, an Error exception will be raised with type `ObjectExists`. Use **update ()** for updating existing objects. If the type of object is not a legal type known to the DataManager, an exception with type `InvalidType` will be raised. Revision information is supplied by adding a new `DynaRefInfo` object to the history of the object to be stored. The members `time` and `description` of the `DynaRefInfo` object are respectively set to, the creation time, and the string “Created”. For a detailed description of the implementation of this interface member see Appendix “Store”

¹³ IDL types and attributes are give in the font “Courier new 11” (`Invalid type`)
Corresponding DDL (Objectivity) objects in “Arial 11” (`DynaRevInfo`)

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

Updates the content of an existing object using the data provided. If the object indicated by `path` is not found an exception with type `NoSuchObject` will be raised. You can use the `headerOnly` to specify that you do not wish to update the bulk of the data contained in an object, but only the headers. If the type of the object you wish to update is not the same as the type of the object you're sending, or if the type of object you're sending is unknown, an exception with type `InvalidType` will be raised.

When an object is updated, revision information is stored by adding a new `DynaRefInfo` object to the history of the object. The `description` member of the `DynaRefInfo` object is set equal to the `info` argument.

```
RevInfoSeq    getHistory ( in string path )
```

Retrieves the history of the object indicated by `path` as a sequence of `RevInfo` structures, where the `RevInfo` structure is the CORBA counter part of the `DynaRefInfo` object. Every data object yields at least one `RevInfo` structure describing its creation. There can be more entries describing further revisions. If the object is not found, an exception with type `NoSuchObject` will be raised.

```
ObjectHeader    getHeader ( in string path )
```

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

```
any    getProperties ( in string path )
```

Retrieves all properties other than the bulk of data of an object. If the object indicated by `path` is not found an exception with type `NoSuchObject` will be raised. This will return an object of the correct type with all property attributes set, but with empty content.

```
any    getData ( in string path )
```

Retrieves an object from the database. If the object indicated by `path` is not found an exception with type `NoSuchObject` will be raised.

```
DimNFloat64    getDim1Data ( in string path, in ulong npoints, in  
ulong interval,in ulong how, in interpolation )
```

Retrieves expanded data of a 1-dimensional object. If the object indicated by `path` is not found an exception with type `NoSuchObject` will be raised. Allows retrieval of only a part of the data by 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 every 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.

```
void    rm ( in string path )
```

Deletes any object in the database. If the object indicated by `path` is not found an exception with type `NoSuchObject` will be raised. Data with attribute `level` equal to zero will never be deleted. Attempting to delete an object with level zero will raise an exception with type `PermissionDenied`.

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

Creates an entry that binds an object to a symbolic name. The `destination` will be used as the path for the new entry, and the `source` as the path of the object it is pointing to. The `source` must point to an existing object to prevent the possibility of dangling links. If an entry with the same path as the `destination` exists an exception with type `ObjectExists` will be raised, and the entry will not be created.

```
StrSeq  list ( in string path )
```

Returns a list with the full path of all objects at a given location. If the location pointed to by `path` is not found an exception with type `NoSuchObject` will be raised. Does not recursively list the content of directories. The path of directories in the list will end with a `'/'` character so they can easily be distinguished from objects.

```
void close ( )
```

Shuts down the `DataManager`. Any currently active transaction will be aborted.

Access rights and the DataManager

In the Dynacore project the authentication/authorisation issue has not yet crystallised into a clear-cut solution. In the present status a user may be authenticated/authorised by a third party (e.g. a login manager) and given access to the database by providing him with his 'private' `DataManager`. After the `DataManager` is launched, it produces an IOR(Interoperable Object Reference) string that is delivered to the user via the third party. Any user equipped with this IOR string can communicate with the `DataManager`, because the `DataManager` has no means at its disposal to authenticate the user it services. As long as of the `DataManager` is only known to the third-party-authenticated user, we may say that he has a private `DataManager`. But still access rights cannot be checked.

As soon as the authentication of a user through CORBA is provided for, the following operations become relevant:

```
PolicySeq  getPolicies ( in string path )
```

Gets a list of policies that apply to an object. If the object indicated by `path` is not found an exception with type `NoSuchObject` will be raised.

```
void    setPolicy ( in string path, in Policy policy )
```

The `DynaPolicy` object of data (`DynaObject`) contains information about the access rights. All the data specific operations have to deal with the `DynaPolicy` of a `DynaObject` and have to be extended when this issue has been settled. The return value of the `getPolicy` operation is a sequence of `Policy` structures, `PolicySeq`, where the `Policy` structure is the CORBA counterpart of the `DynaPolicy` object.

Extended interface

As mentioned above the use of CORBA Any objects to store large amount of data may be too time consuming under experimental circumstances. To provide for optimal storage performance the interface has

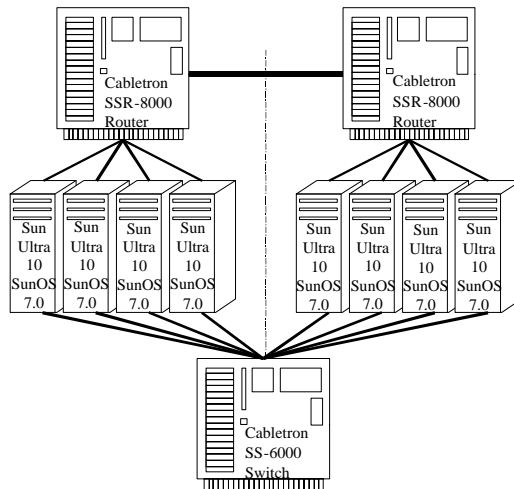
been extended with members to store separately those data types that may be very large in practice. These extra members are the following:

```
void storeMimeObj( in MimeObj mobj,in string path )
void storeDimNInt8( in DimNInt8 nint8,in string path )
void storeDimNInt16( in DimNInt16 nint16, in string path )
void storeDimNInt32( in DimNInt32 nint32, in string path )
void storeDimNUInt16( in DimNInt16 nuint16, in string path )
void storeDimNUInt32( in DimNUInt32 nuint32, in string path )
void storeDimNFloat32( in DimNInt32 nfloat32, in string path )
void storeDimNFloat64(in DimNFloat64 nfloat64, in string path )
```

These members have equal functionality as **store()**, except for the unwrapping of the CORBA Any object.

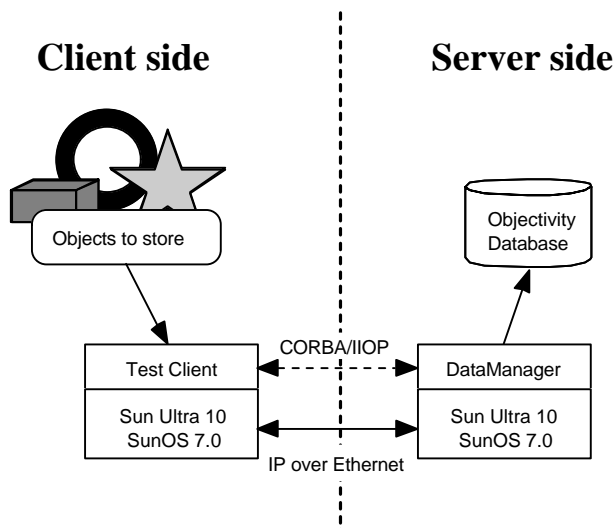
Performance measurements

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 setup as shown in Figure 6¹⁴.



• Figure 6 The Gigacluster measurement setup

The Gigacluster set-up consists of eight Sun-Ultra-10 workstations running SunOS 5.7 and two Cabletron SSR-8000 Smart Switch Routers. The SUN's 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 7 Actual set-up for the performance measurements.

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.

¹⁴ 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.

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.¹⁵

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

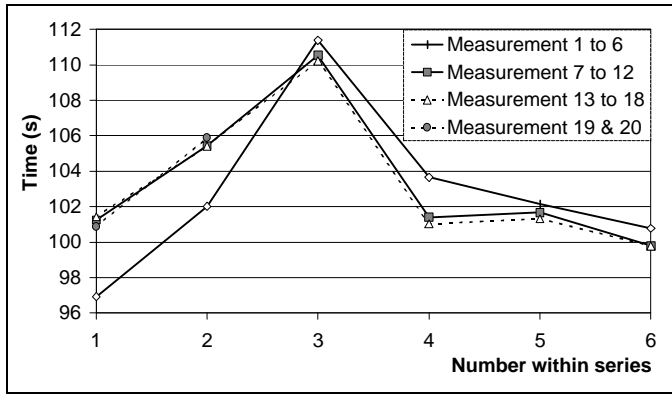
In order to achieve higher performance, we have used only the fast interface in further testing. In the next test, we measured again the time necessary to store 324 objects with 10^6 bytes of raw data. We repeated the measurements 20 times, while we reused the database until it was full. Every time the database was full, we emptied the database and continued our measurement. The result is depicted in Figure 8.

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 measurements¹⁶. The number on the x-axis of the graph is the number of the measurement within its series.

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 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.

¹⁵ 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.

¹⁶ 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.

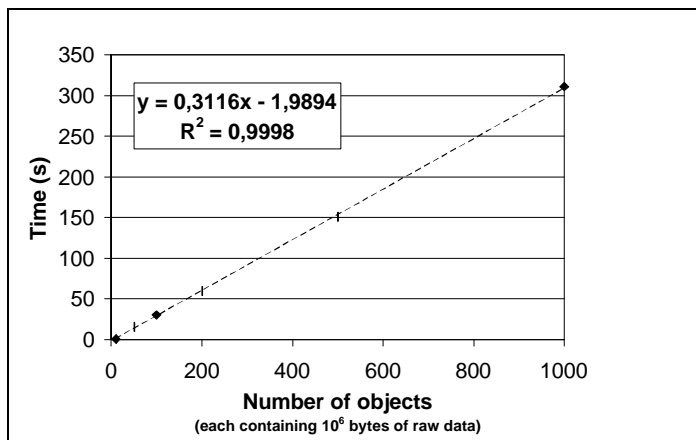


• Figure 8 Repeatedly storing data in the same database. The database is emptied after the 6th, 12th and 18th measurement.

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.

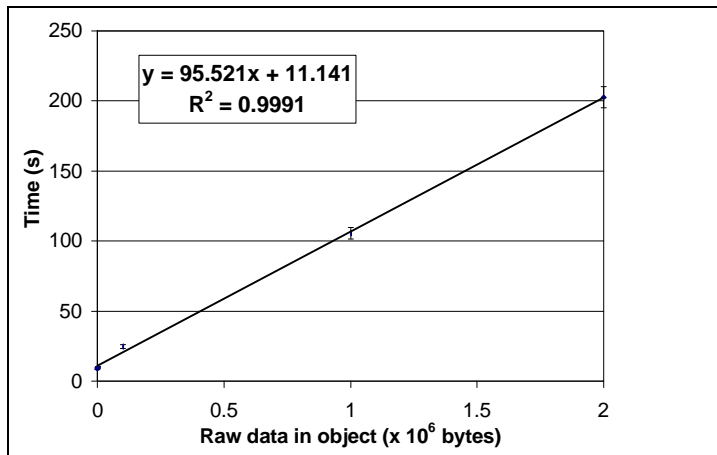
Dependency on number and size of objects

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 9), in the second case, the number of objects was fixed to 324, and the size was varied (Figure 10).



• Figure 9 Time needed to store objects containing 10^6 of raw data as a function of the number of objects stored. (R^2 is corr.. coeff. squared)

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 3.2×10^6 and 3.4×10^6 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 10 Time to store 324 objects as a function of object size.

To achieve the performance requirements, a storage speed of $500 \times 10^6 / 60 = 8,3 \times 10^6$ B/s is needed. This can easily be achieved using 3 SUN's in parallel ($3 \times 3.2 \times 10^6 = 9.6 \times 10^6$ 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 fiber 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 SUN's 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.

Appendices

Syntax

The grammar used is the following:

```
char      = {a | b | ... | z} | {A | B | ... | Z} | {+ | - | _}
digit     = {0 | 1 | ... | 9}
name      = char [name] | digit [name]
separator = /
subpath   = name separator [subpath]
path      = separator name separator name separator [subpath] [name]
```

Store

In the member store of the CORBA interface the expression ‘syntax(*path*) successful’ has the following meaning:

path equals: separator name separator name separator [subpath] name

Semantics

The first name in *path* is interpreted as the name of a container, while the second name in *path* is interpreted as the name of a database.

Store

void store(const CORBA::Any & any, const char* path)

For the meaning of the expressions ‘any_2_object(*any*) successful’ and ‘syntax (*path*) successful’, see section ‘Any_2_object’ and ‘Syntax’, respectively.

PRE: **TRUE**

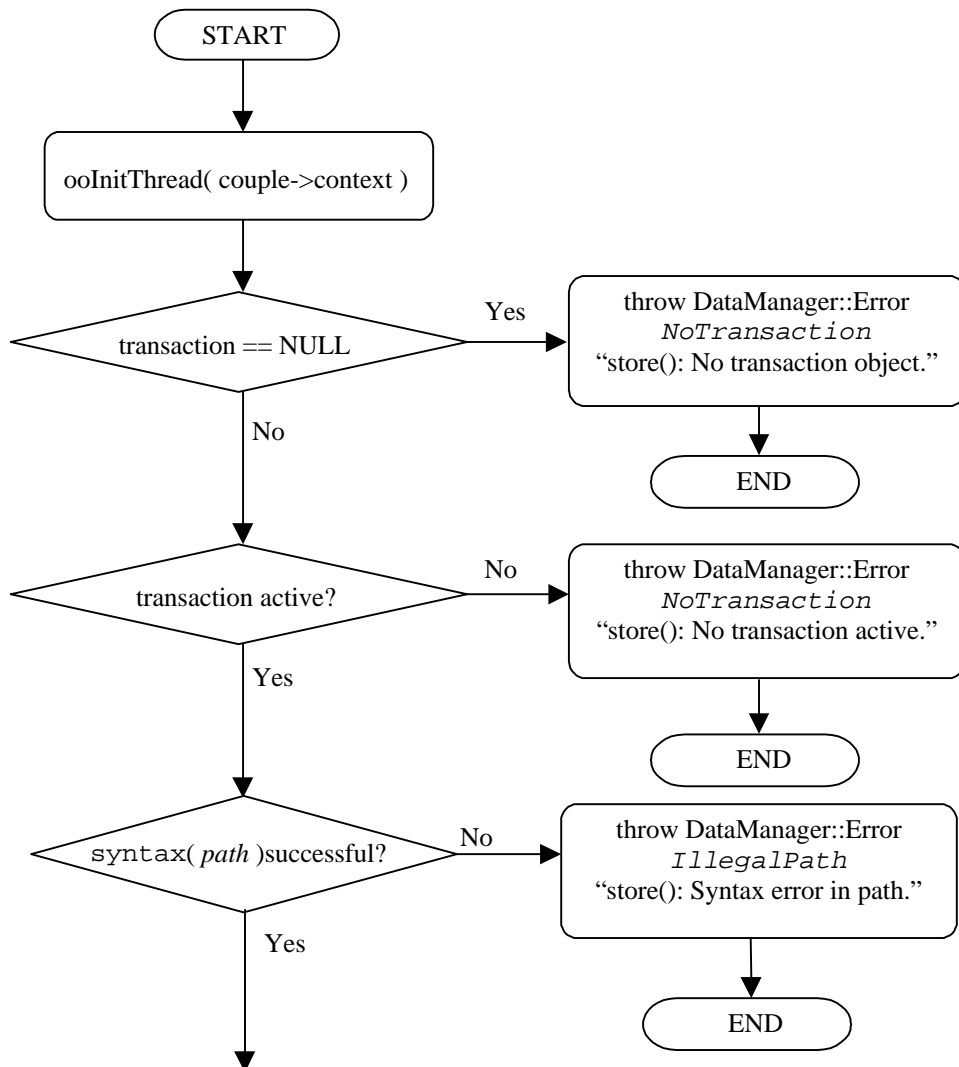
POST any_2_object(*any*) successful AND syntax(*path*) successful →
object in database.

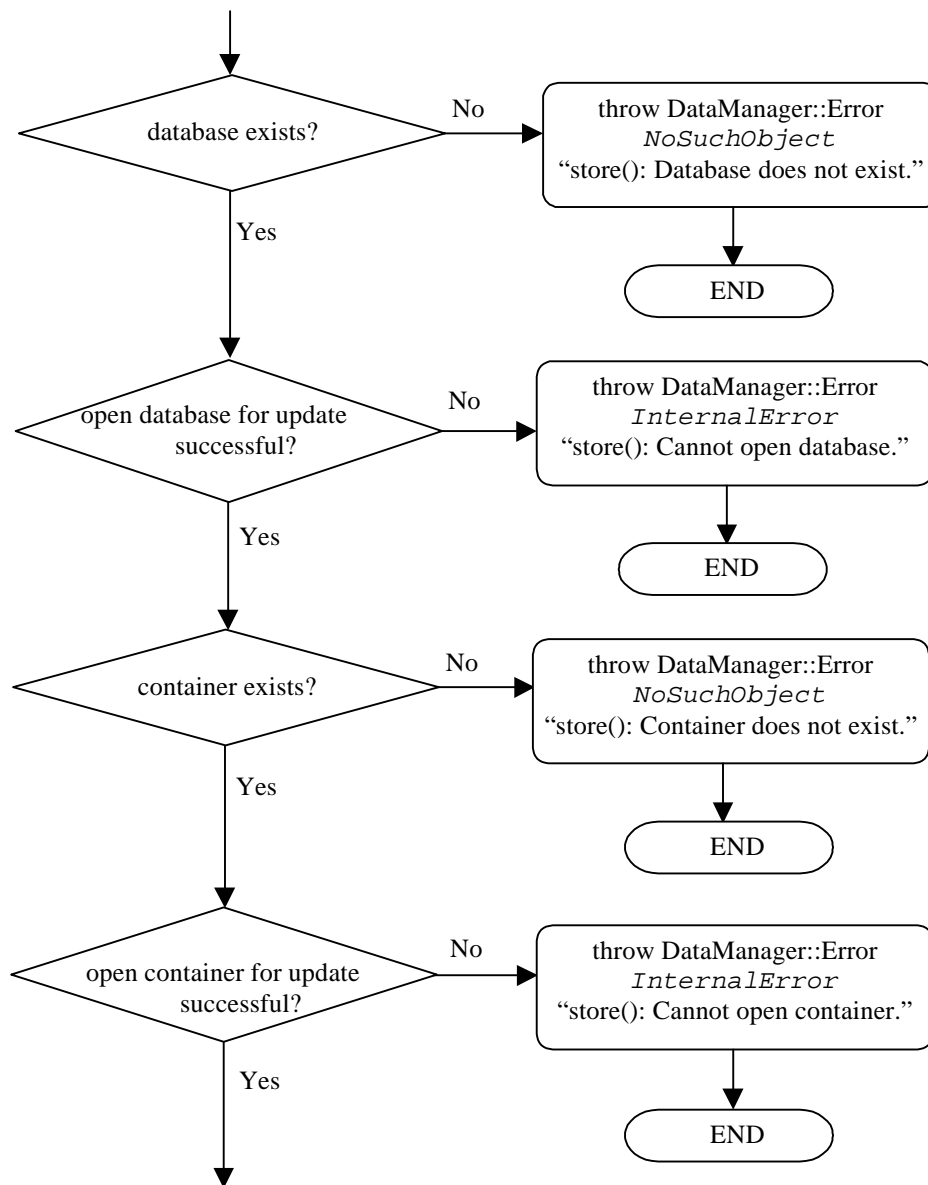
syntax(*path*) NOT successful →

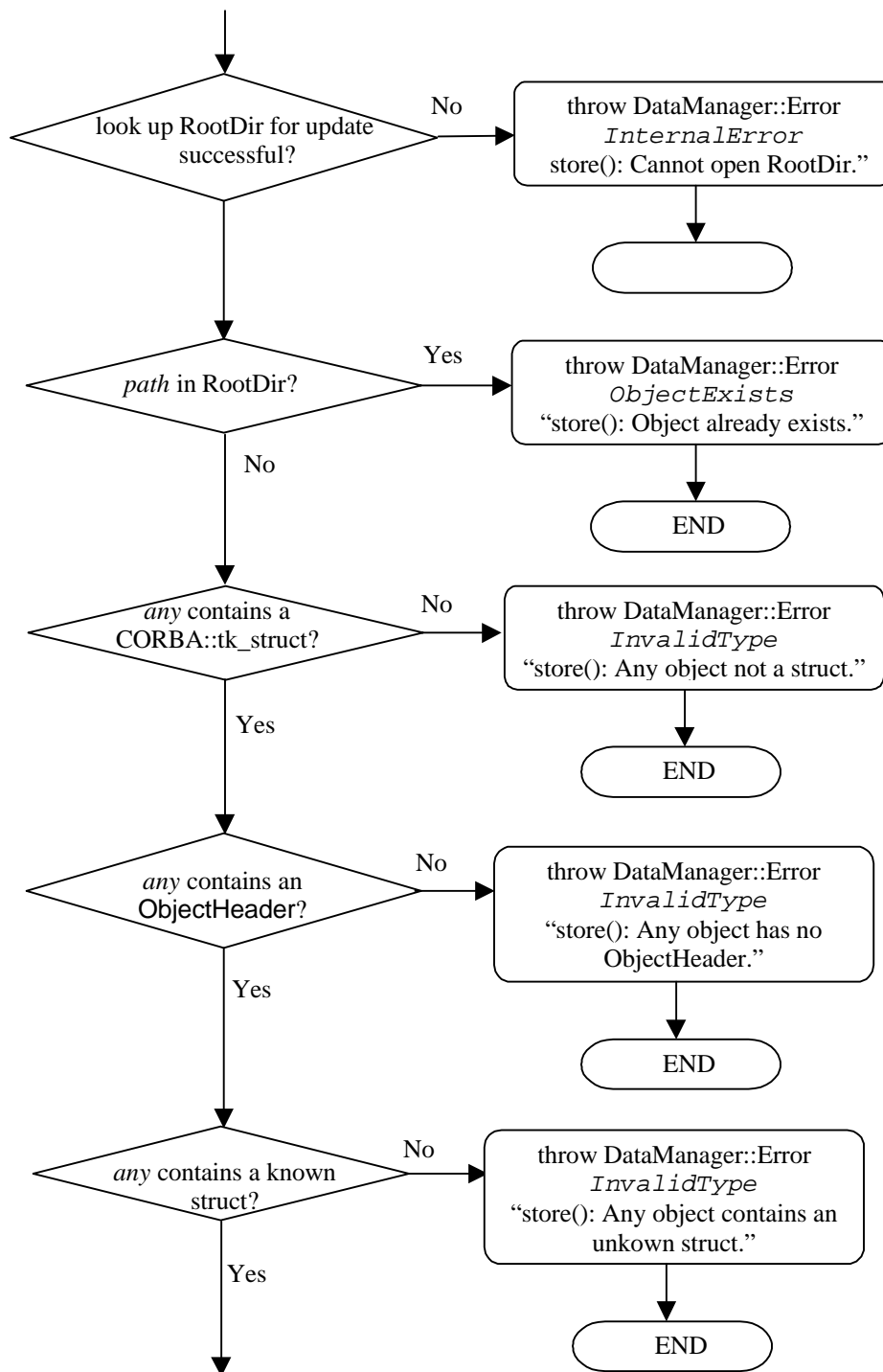
No change in database, exception thrown.

syntax(*path*) successful AND any_2_object(*any*) NOT successful →

No change in database, exception thrown.







Any_2_object

The expression ‘any_2_object(any) is successful’ means that in the conversion of the CORBA::Any object, any, to the corresponding DynaDimN object no exceptions have occurred.

Depending on the type of the CORBA structure contained in the Any object (cComment , cMimeObj, ..., or cDimNFloat64) the following conversions are distinguished:

cComment	→ any_2_DynaComment
cMimeObj	→ any_2_DynaMimeObj
cPolyCalibration	→ any_2_DynaPolyCalibration
cTableCalibration	→ any_2_DynaTableCalibration
cShortBase	→ any_2_DynaShortBase
cLongBase	→ any_2_DynaLongBase
cScalar	→ any_2_DynaScalar
cDimNInt8	→ any_Int8_2_DynaDimN
cDimNInt16	→ any_Int16_2_DynaDimN
cDimNInt32	→ any_Int32_2_DynaDimN
cDimNUInt16	→ any_UInt16_2_DynaDimN
cDimNUInt32	→ any_UInt32_2_DynaDimN
cDimNFloat32	→ any_Float32_2_DynaDimN
cDimNFloat64	→ any_Float64_2_DynaDimN

any_UInt32_2_DynaDimN

- Initialize DynaDimN object to be stored in database.
- Extract DimNUInt32 structure from Any object, if unsuccessful throw exception.
- Add references from ObjectHeader to DynaDimN object, if reference does not exist or reference is a cross-database or cross-container reference and cannot be obtained (due to lock on database/container by another thread), throw exception.
- Add name, level, quality from ObjectHeader to DynaDimN object.
- Add SiUnits to DynaDimN object.
- Copy sizes to DynaDimN object.
- Add DynaBase references to DynaDimN object, if reference does not exist or reference is not a DynaBase or reference is a cross-database or cross-container reference and cannot be obtained (due to lock on database/container by another thread), throw exception.
- Add DynaCalibration references to DynaDimN object, if reference does not exist or reference is not a DynaCalibration or reference is a cross-data or cross-container reference and cannot be obtained (due to lock on database/container by another thread), throw exception.
- Add adres to DynaDimN object.
- Initialize BulkUInt32 object.
- If the product of lengths in sizes not equal length of content, no copy of data.
- Copy content to BulkUInt32(data) object.

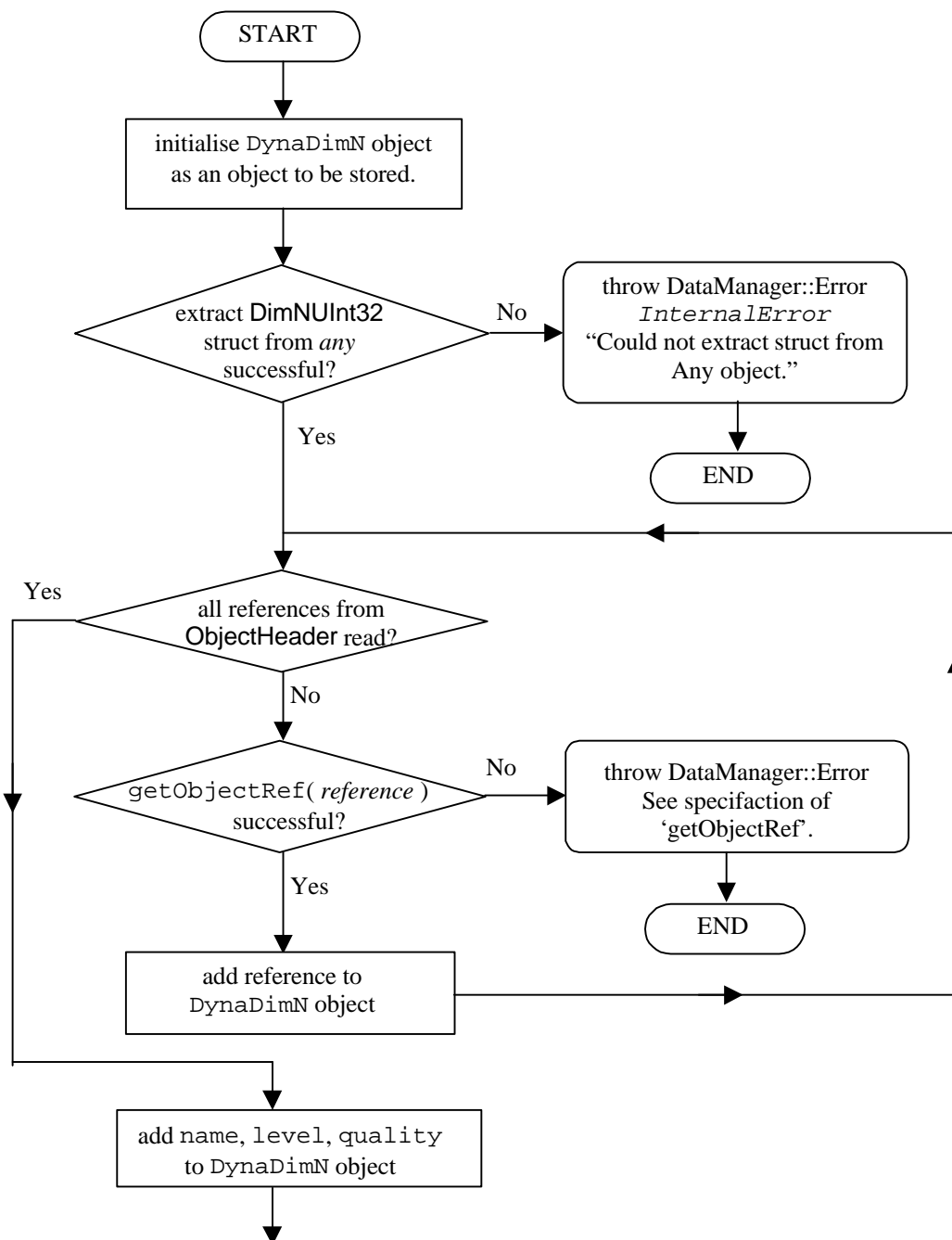
- Add BulkUInt32 object to DynaDimN object.

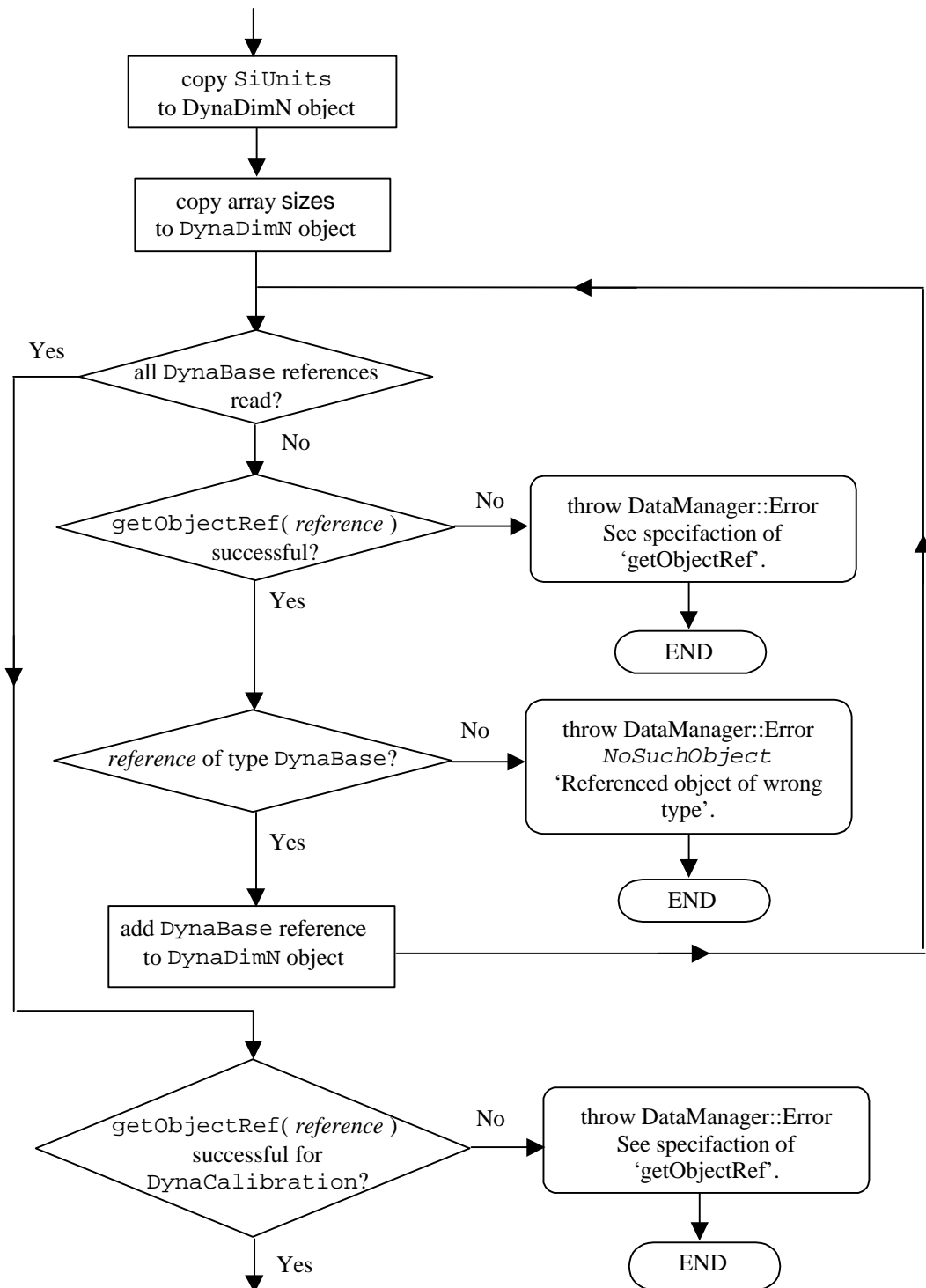
int any_UInt32_2_DynaDimN(const Any& any, ooHandle(DynaObject)& theObject)

For the meaning of the expression ‘getDynaObjectRef(*reference*) is successful’ see section ‘getDynaObjectRef’.

PRE: *any* contains a CORBA structure **AND** *theObject* is not NULL

POST: If no exception occurs *theObject* contains all information from the corresponding CORBA structure of type cDimNUInt32 contained in the Any object.





getDynaObjectRef

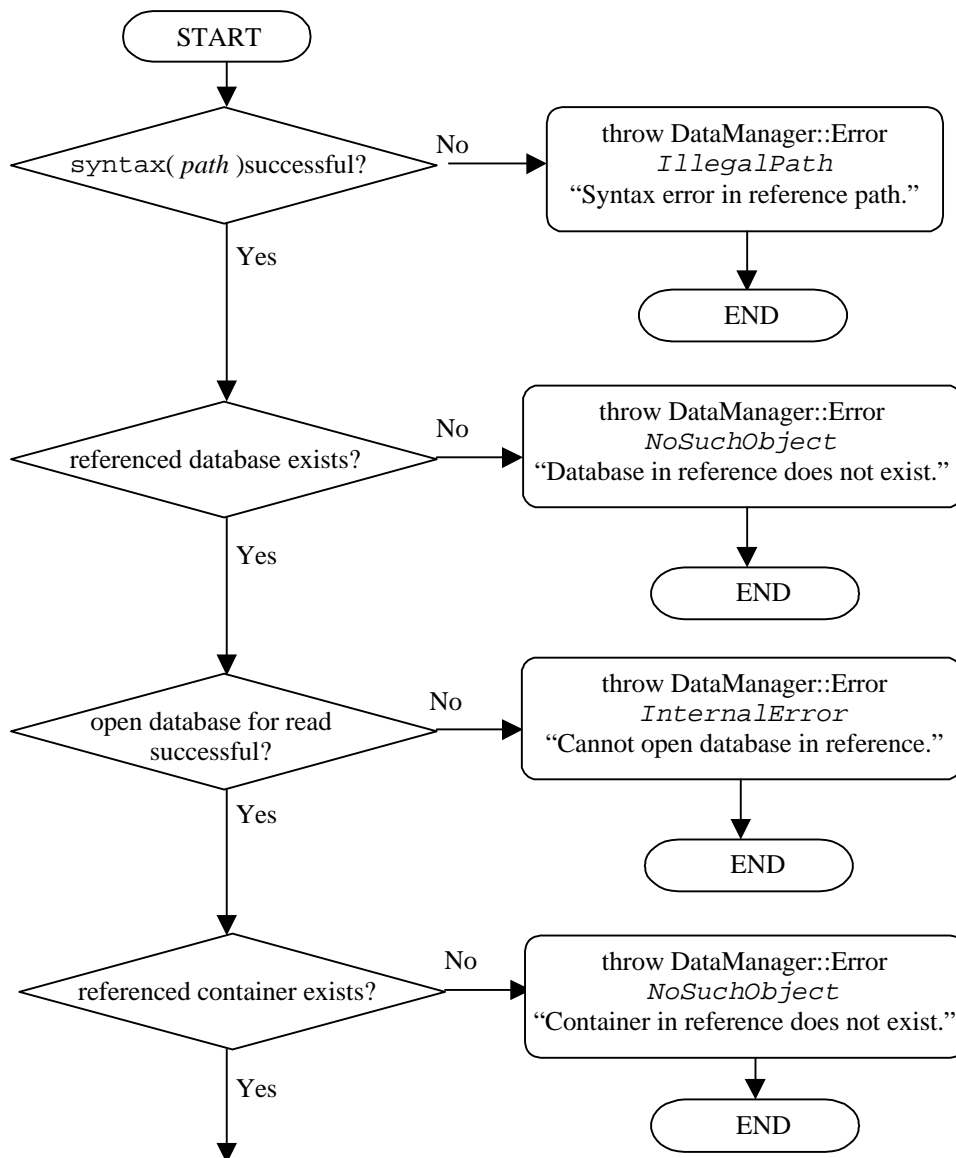
The expression ‘getDynaObjectRef(*reference*) is successful’ means that the method getDynaObjectRef proceeds without exception.

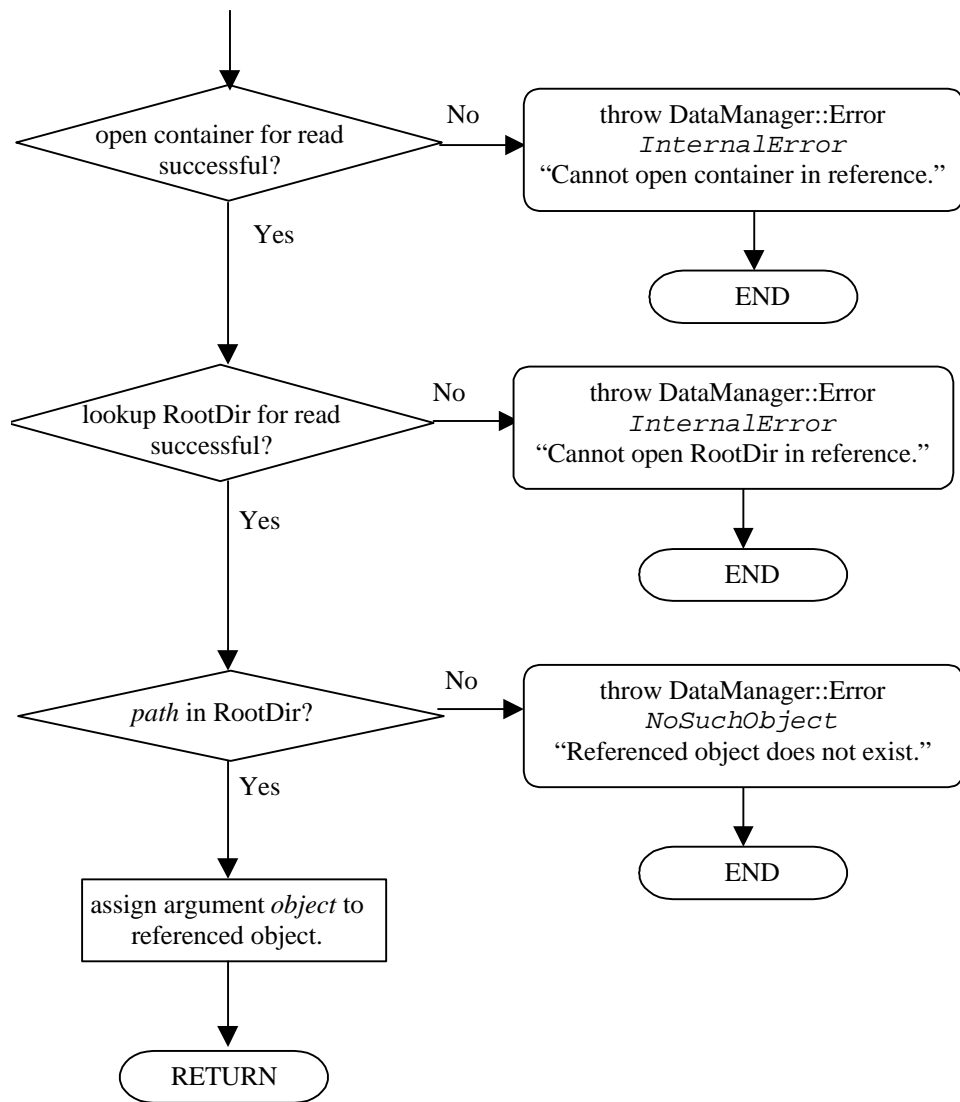
int getDynaObjectRef(const char* path, ooRef(DynaObject)& object)

For the expression ‘Syntax(*path*) successful’ see section Syntax.

PRE: **TRUE**

POST: If Syntax(*path*) successful **AND** database object referenced by *path* can be opened, argument *object* is assigned to the referenced database object.





CORBA Specification of the DataManager

This package contains all objects defined in the CORBA Interface Definition Language (IDL).

ObjectType

Enumeration of the possible types of objects.

Public Attributes:

cUnknown :
cComment :
cMimeObj :
cPolyCalibration :
cTableCalibration :
cShortBase :
cLongBase :
cScalar :
cDimNInt8 :
cDimNInt16 :
cDimNInt32 :
cDimNUInt16 :
cDimNUInt32 :
cDimNFloat32 :
cDimNFloat64 :

RevInfo

Information describing a revision of a data object.

Public Attributes:

time : long
Time of the revision, represented in the number of seconds since midnight, January 1st 1970 GMT.

username : string
Identity of the person making the modification.

description : string
Full description of the revision, including at least the reason for the modification, the type of modification that was made and optionally version/revision information about possible software that was involved.

SiUnits

Units of a data object in powers of the SI basic units.

Public Attributes:

kg : long

m : long

s : long

A : long

cd : long

mol : long

K : long

rad : long

sr : long

ObjectHeader

A header describing the properties of an object. All Dynacore objects have such a header, which can be retrieved separately using the DataManager.

Public Attributes:

name : string

Unique identifier of the object.

level : ulong

Storage level of the object. Level 0 is reserved for (raw) experiment data. If data is processed, and the results are stored, these results must have a level higher than the data used for the processing.

quality : ulong

Indicator that can be used to measure the quality of a data object. Could be used to mark an object as unreliable.

fullPath : string

The full path of the object in the database.

references : StringSeq

A list of paths of other objects that are somehow related to this object. If the list is empty, no objects are related to this object. Relationships can be used to tie together objects which are coupled, but are stored separately.

type : ObjectType

The type of this object.

ByteSeq

An IDL sequence of bytes.

ShortSeq

An IDL sequence of shorts.

UShortSeq

An IDL sequence of unsigned shorts.

LongSeq

An IDL sequence of longs.

ULongSeq

An IDL sequence of unsigned longs.

FloatSeq

An IDL sequence of floats.

DoubleSeq

An IDL sequence of doubles.

StringSeq

An IDL sequence of strings.

ulong

An IDL typedef of unsigned long.

ushort

An IDL typedef of unsigned short.

TimeStamp

A long representing a time value. Contains the number of seconds since January 1st 1970 GMT (Epoch).

RevInfoSeq

An IDL sequence of RevInfo structures.

Comment

A comment object which makes it possible to add comments on all levels of the database. Comments are free-form strings.

Public Attributes:

oh : ObjectHeader

A header describing the object.

content : string

The actual content of the comment. This can be formatted text like HTML.

MimeObj

An object used to store multimedia data. The data is stored as a sequence of bytes. The type can be derived by checking the mimetype attribute in the properties.

Public Attributes:

oh : ObjectHeader

A header describing the object.

mimetype : string

The mimetype of the object. This could be something like image/jpeg.

bytecount : ulong

Size of the object in bytes.

content : ByteSeq

Actual content of the object. The mimetype attribute can be used to interpret the content.

PolyCalibration

A calibration in the form of a polynomial. The most common type will be a first order (linear) calibration with two coefficients.

Public Attributes:

oh : ObjectHeader

A header describing the object.

coefficients : DoubleSeq

Coefficients of the calibration. The signal value can be derived from the value N by calculating $\text{coefficients}[0] + N * \text{coefficients}[1] + N^2 * \text{coefficients}[2]$ etc..

TableCalibration

A calibration in the form of a table which supplies the corresponding double precision value for each integer input value.

Public Attributes:

oh : ObjectHeader

A header describing the object.

table : DoubleSeq

Table with one double precision value for each value of the ADC output. The signal value corresponding to N will be table[N].

ShortBase

An object used to store base information for DimN objects in a short form, providing a start value and an interval value.

Public Attributes:

oh : ObjectHeader

A header describing the object.

unit : SiUnits

The unit of the object.

start : double

Start value of the base.

step : double

Interval value of the base.

LongBase

An object used to store base information for DimN objects in a long form with one base entry for each array entry.

Public Attributes:

oh : ObjectHeader

A header describing the object.

unit : SiUnits

The unit of the object.

data : DoubleSeq

A sequence of base values.

Scalar

A double precision floating point scalar.

Public Attributes:

oh : ObjectHeader

A header describing the object.

unit : SiUnits

The unit of the object.

time : double

The time the scalar was measured.

content : double

Value of the scalar.

DimNInt8

*An N-dimensional object containing 8-bit signed or unsigned integer data where ($N > 0$). The sizes and bases of each dimension are stored in two sequences of the same length. The corresponding data is stored in a one-dimensional sequence. You can find datapoint $obj[i][j][k]$ at index $(i*sizes[1] + j*sizes[0] + k)$ of the sequence.*

A flag is provided to determine whether the data is signed or unsigned. This flag is needed because CORBA does not distinguish signed and unsigned 8-bit values.

Public Attributes:

oh : ObjectHeader

A header describing the object.

unit : SiUnits

The unit of the object.

sizes : ULongSeq

Sequence of lengths of the data in each dimension.

bases : StringSeq

Sequence of paths of the base objects corresponding to each dimension of this object. If the paths are empty, no base information is present.

Path of the calibration object for this data object. If the path is empty, no calibration is present.

adcresolution : ushortcalibration : string

Resolution of the ADC used to measure this object.

sign : boolean

Flag describing whether the data should be interpreted as signed (true) or unsigned (false).

content : ByteSeq

Content of the data object.

DimNInt16

*An N-dimensional object containing 16-bit signed integer data where ($N > 0$). The sizes and bases of each dimension are stored in two sequences of the same length. The corresponding data is stored in a one-dimensional sequence. You can find datapoint $obj[i][j][k]$ at index $(i*sizes[1] + j*sizes[0] + k)$ of the sequence.*

Public Attributes:

oh : ObjectHeader

A header describing the object.

unit : SiUnits

The unit of the object.

sizes : ULongSeq

Sequence of lengths of the data in each dimension.

bases : StringSeq

Sequence of paths of the base objects corresponding to each dimension of this object. If the paths are empty, no base information is present.

calibration : string

Path of the calibration object for this data object. If the path is empty, no calibration is present.

adcresolution : ushort

Resolution of the ADC used to measure this object.

content : ShortSeq

Content of the data object.

DimNInt32

*An N-dimensional object containing 32-bit signed integer data where $(N > 0)$. The sizes and bases of each dimension are stored in two sequences of the same length. The corresponding data is stored in a one-dimensional sequence. You can find datapoint $obj[i][j][k]$ at index $(i*sizes[1] + j*sizes[0] + k)$ of the sequence.*

Public Attributes:

oh : ObjectHeader

A header describing the object.

unit : SiUnits

The unit of the object.

sizes : ULongSeq

Sequence of lengths of the data in each dimension.

bases : StringSeq

Sequence of paths of the base objects corresponding to each dimension of this object. If the paths are empty, no base information is present.

Path of the calibration object for this data object. If the path is empty, no calibration is present.

adcresolution : ushortcalibration : string

Resolution of the ADC used to measure this object.

content : LongSeq

Content of the data object.

DimNUInt16

*An N-dimensional object containing 16-bit unsigned integer data where ($N > 0$). The sizes and bases of each dimension are stored in two sequences of the same length. The corresponding data is stored in a one-dimensional sequence. You can find datapoint $obj[i][j][k]$ at index $(i * sizes[1] + j * sizes[0] + k)$ of the sequence.*

Public Attributes:

oh : ObjectHeader

A header describing the object.

unit : SiUnits

The unit of the object.

sizes : ULongSeq

Sequence of lengths of the data in each dimension.

bases : StringSeq

Sequence of paths of the base objects corresponding to each dimension of this object. If the paths are empty, no base information is present.

calibration : string

Path of the calibration object for this data object. If the path is empty, no calibration is present.

adcresolution : ushort

Resolution of the ADC used to measure this object.

content : UShortSeq

Content of the data object.

DimNUInt32

*An N-dimensional object containing 32-bit unsigned integer data where ($N > 0$). The sizes and bases of each dimension are stored in two sequences of the same length. The corresponding data is stored in a one-dimensional sequence. You can find datapoint $obj[i][j][k]$ at index $(i * sizes[1] + j * sizes[0] + k)$ of the sequence.*

Public Attributes:

oh : ObjectHeader

A header describing the object.

unit : SiUnits

The unit of the object.

sizes : ULongSeq

Sequence of lengths of the data in each dimension.

bases : StringSeq

Sequence of paths of the base objects corresponding to each dimension of this object. If the paths are empty, no base information is present.

calibration : string

Path of the calibration object for this data object. If the path is empty, no calibration is present.

adcresolution : ushort

Resolution of the ADC used to measure this object.

content : ULongSeq

Content of the data object.

DimNFloat32

*An N-dimensional object containing single precision floating point data where ($N > 0$). The sizes and bases of each dimension are stored in two sequences of the same length. The corresponding data is stored in a one-dimensional sequence. You can find datapoint $obj[i][j][k]$ at index $(i * sizes[1] + j * sizes[0] + k)$ of the sequence.*

Public Attributes:

oh : ObjectHeader

A header describing the object.

unit : SiUnits

The unit of the object.

sizes : ULongSeq

Sequence of lengths of the data in each dimension.

bases : StringSeq

Sequence of paths of the base objects corresponding to each dimension of this object. If the paths are empty, no base information is present.

Content of the data object.

content : FloatSeq

DimNFloat64

*An N-dimensional object containing double precision floating point data where $(N > 0)$. The sizes and bases of each dimension are stored in two sequences of the same length. The corresponding data is stored in a one-dimensional sequence. You can find datapoint $obj[i][j][k]$ at index $(i*sizes[1] + j*sizes[0] + k)$ of the sequence.*

Public Attributes:

oh : ObjectHeader

A header describing the object.

unit : SiUnits

The unit of the object.

sizes : ULongSeq

Sequence of lengths of the data in each dimension.

bases : StringSeq

Sequence of paths of the base objects corresponding to each dimension of this object. If the paths are empty, no base information is present.

content : DoubleSeq

Content of the data object.

IDL Interface Description DataManager

DataManager.idl

```
#include "DataObject.idl";

interface DataManager
{
    enum DmErrType
    {
        NoTransaction,
        NestedTransaction,
        PermissionDenied,
        IllegalPath,
        NoSuchObject,
        ObjectExists,
        LockTimeout,
        LockNotActive,
        InvalidType,
        InternalError
    };
    exception Error { DmErrType type; string message; };

    enum Interpolation
    {
        None,
        Average,
        MinMax
    };

    const unsigned long maxIdleTime = 3600;
    readonly attribute unsigned long idleTime;

    // Transaction operations
    void start()
        raises(Error);
    void commit()
        raises(Error);
    void abort()
        raises(Error);
    void commitAndHold()
        raises(Error);

    // Data object operations
    void store( in any obj, in string path )
        raises(Error);
    void update( in any obj, in string path, in boolean headerOnly, in string
info )
        raises(Error);
    RevInfoSeq getHistory( in string path )
        raises(Error);
    PolicySeq getPolicies( in string path )
        raises(Error);
    ObjectHeader getHeader( in string path )
        raises(Error);
    any getProperties( in string path )
        raises(Error);
    any getData( in string path )
        raises(Error);
}
```

```

DimNFloat64 getDim1Data( in string path, in ulong first, in ulong npoints,
                        in ulong interval, in Interpolation how )
    raises(Error);
void rm( in string path )
    raises(Error);
void lock( in string path )
    raises(Error);
void unlock( in string path )
    raises(Error);
void link( in string srcpath, in string dstpath )
    raises(Error);

// Directory operations
StringSeq list( in string path )
    raises(Error);

// Other operations
void keepAlive( )
    raises(Error);
oneway void shutdown();
};

```

DataObject.idl

```

//
// Typedefs
//
typedef sequence<octet> ByteSeq;

typedef sequence<short> ShortSeq;
typedef sequence<unsigned short> UShortSeq;
typedef sequence<long> LongSeq;
typedef sequence<unsigned long> ULongSeq;
typedef sequence<float> FloatSeq;
typedef sequence<double> DoubleSeq;
typedef sequence<string> StringSeq;

typedef unsigned long ulong;
typedef unsigned short ushort;
typedef long TimeStamp;

//
// Enumeration types
//
enum ObjectType
{
    cUnknown,
    cComment,
    cMimeObj,
    cPolyCalibration,
    cTableCalibration,
    cShortBase,
    cLongBase,
    cScalar,
    cDimNInt8,
    cDimNInt16,
    cDimNInt32,

```

```

    cDimNUInt16,
    cDimNUInt32,
    cDimNFloat32,
    cDimNFloat64
};

enum AccessMode
{
    cNone,
    cRead,
    cWrite,
    cReadWrite,
    cPol,
    cPolRead,
    cPolWrite,
    cPolReadWrite
};

//
// Structs used in the data objects
//

struct Policy
{
    ushort gid;
    AccessMode mode;
};

typedef sequence<Policy> PolicySeq;

struct RevInfo
{
    long time;
    string username;
    string description;
};

typedef sequence<RevInfo> RevInfoSeq;

struct SiUnits
{
    long kg, m, s, A, cd, mol, K, rad, sr;
};

struct ObjectHeader
{
    string name;
    ulong level;
    ulong quality;
    string fullPath;
    StringSeq references;
    ObjectType type;
};

//
// Structs for actual data objects
//

struct Comment
{
    ObjectHeader oh;
};

```

```

    string content;
};

struct MimeObj
{
    ObjectHeader oh;
    string mimetype;
    ulong bytecount;
    ByteSeq content;
};

struct PolyCalibration
{
    ObjectHeader oh;
    DoubleSeq coefficients;
};

struct TableCalibration
{
    ObjectHeader oh;
    DoubleSeq table;
};

struct ShortBase
{
    ObjectHeader oh;
    SiUnits unit;
    double start;
    double step;
};

struct LongBase
{
    ObjectHeader oh;
    SiUnits unit;
    DoubleSeq data;
};

struct Scalar
{
    ObjectHeader oh;
    SiUnits unit;
    double time;
    double content;
};

struct DimNInt8
{
    ObjectHeader oh;
    SiUnits unit;
    ULongSeq sizes;
    StringSeq bases;
    string calibration;
    ushort adcresolution;
    boolean sign;
    ByteSeq content;
};

struct DimNInt16
{
    ObjectHeader oh;

```

```

    SiUnits unit;
    ULongSeq sizes;
    StringSeq bases;
    string calibration;
    ushort adcreolution;
    ShortSeq content;
};

```

```

struct DimNInt32
{
    ObjectHeader oh;
    SiUnits unit;
    ULongSeq sizes;
    StringSeq bases;
    string calibration;
    ushort adcreolution;
    LongSeq content;
};

```

```

struct DimNUInt16
{
    ObjectHeader oh;
    SiUnits unit;
    ULongSeq sizes;
    StringSeq bases;
    string calibration;
    ushort adcreolution;
    UShortSeq content;
};

```

```

struct DimNUInt32
{
    ObjectHeader oh;
    SiUnits unit;
    ULongSeq sizes;
    StringSeq bases;
    string calibration;
    ushort adcreolution;
    ULongSeq content;
};

```

```

struct DimNFloat32
{
    ObjectHeader oh;
    SiUnits unit;
    ULongSeq sizes;
    StringSeq bases;
    FloatSeq content;
};

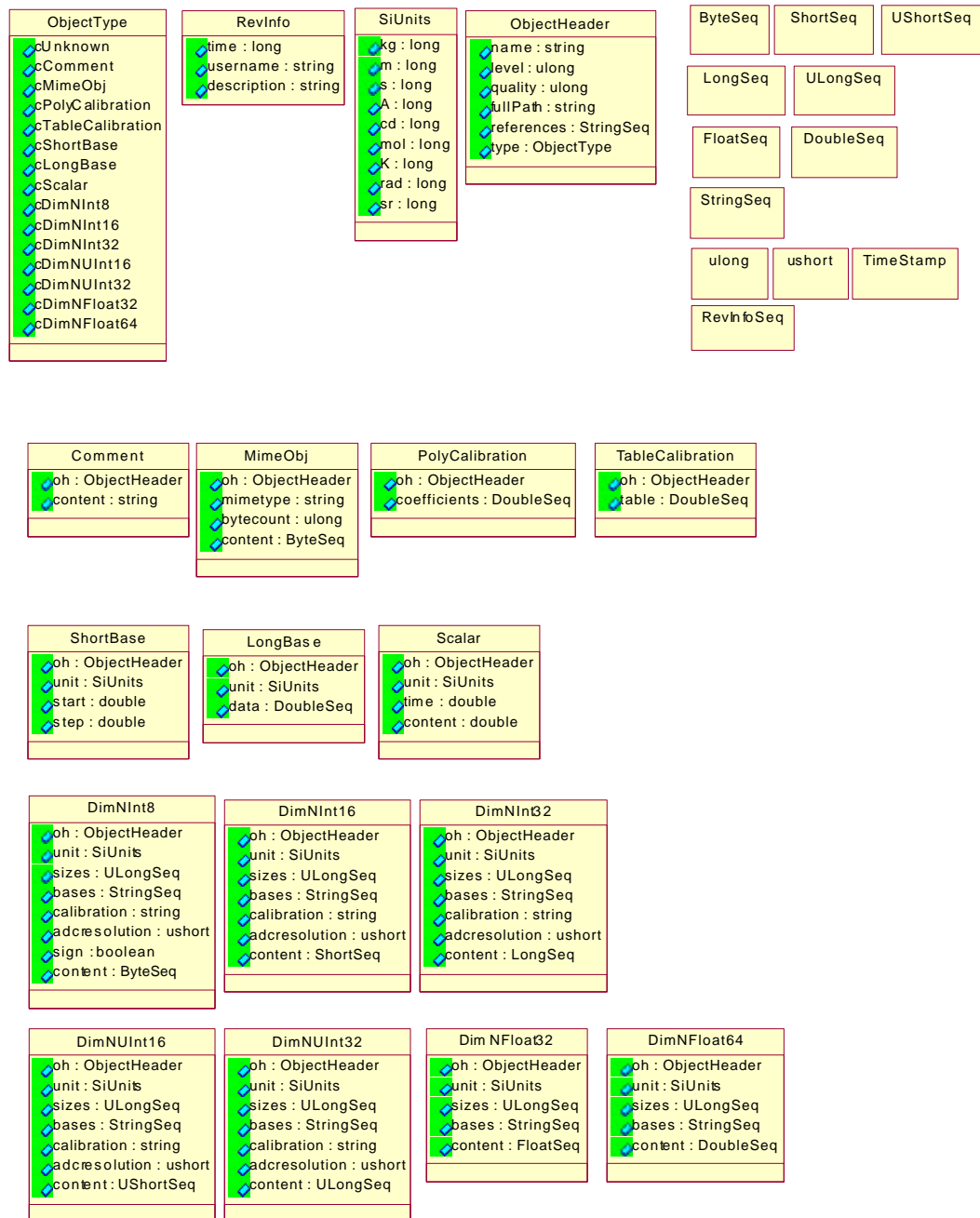
```

```

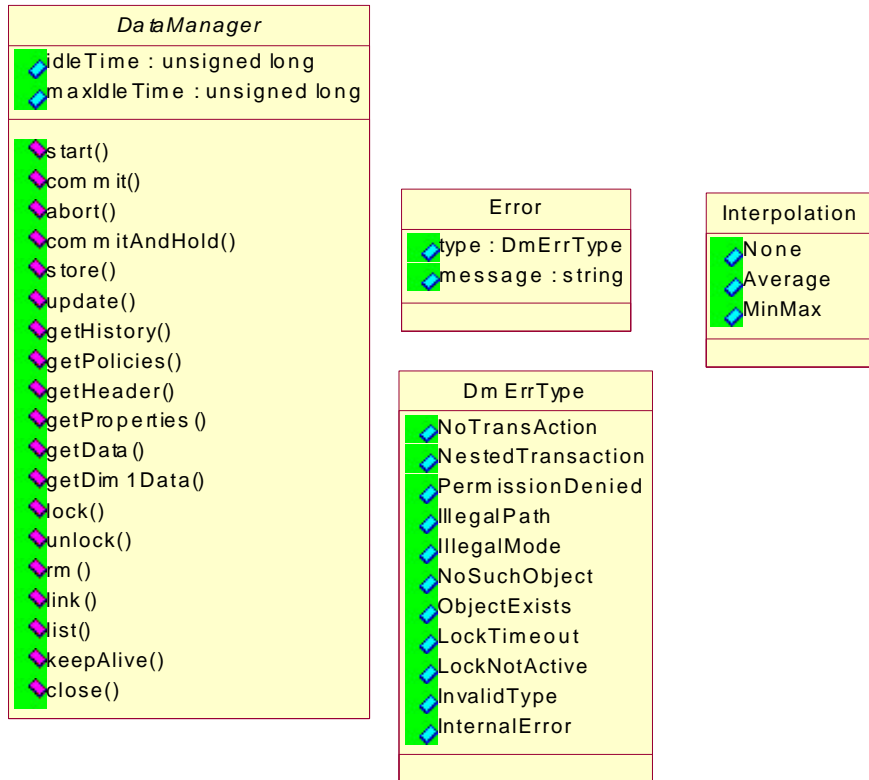
struct DimNFloat64
{
    ObjectHeader oh;
    SiUnits unit;
    ULongSeq sizes;
    StringSeq bases;
    DoubleSeq content;
};

```

CORBA Object Diagrams



• Figure 11 Description of the IDL data types of the CORBA interface of the DataManager



• Figure 12 Continuation of Figure 11

Objectivity Specification and Class design

This package contains all objects that will be implemented in Objectivity Data Definition Language (DDL)

DynaSiUnits

Expresses the physical units of measured values. The units are expressed as powers of SI units. This is convenient, since it allows for arithmetic on units.

Public Attributes:

kg : int32

m : int32

s : int32

A : int32

cd : int32

mol : int32

K : int32

rad : int32

sr : int32

DynaPolicy

Public Attributes:

gid : uint16

mode : DynaAccessMode

DynaRevInfo

Information describing the revision history of data objects. These structures are automatically created by the DataManager and cannot be modified by the user. The time and username attributes are automatically set by the DataManager, and the description is supplied by the user.

Public Attributes:

time : int32

Time of the revision, represented in the number of seconds since midnight, January 1st 1970 GMT.

username : char[256]

Identity of the person making the modification.

description : char[8192]

Full description of the revision, including at least the reason for the modification, the type of modification that was made and optionally version/revision information about possible software that was involved.

DynaObject

The base class for all Dynacore data objects. Each object can contain optional references to other objects.

Derived from ooObj

Public Attributes:

fullpath : ooString(256)

name : ooString(32)

Unique identifier for this object.

level : uint32

Analysis level of this object. Objects with level 0 contain raw data. Data is written back after analysis should always have a level attribute that is one higher than the level found in the source data.

quality : uint32

Attribute which can be used to assign a quality to the data.

references[] : ooRef(DynaObject)

Other objects which are somehow related to this object.

policies : ooVArray(DynaPolicy)

The access policies that apply to this data object.

history : ooVArray(DynaRevInfo)

The full revision history of this object.

DynaComment

An object used to store comments

Derived from DynaObject

Public Attributes:

content : ooVString

The text contained in the comment

DynaMimeObj

A container-like object used for storing multimedia data. The data is stored as an array of bytes.

Use the mime type attribute to determine the type.

Derived from DynaObject

Public Attributes:

mimetype : ooVString

content : ooVArray(uint8)

DynaCalibration

Calibration information used for ADC's. It is stored separately to make it easy to correct calibration information.

Derived from DynaObject

DynaPolyCalibration

A polynomial calibration. For a linear calibration, the number of coefficients would be two, a constant term and a first order term.

Derived from DynaCalibration

Public Attributes:

coefficients : ooVArray(float64)

DynaTableCalibration

A calibration lookup table.

Derived from DynaCalibration

Public Attributes:

data : ooVArray(float64)

DynaBase

An object used to store base information for DimN objects.

Derived from DynaObject

Public Attributes:

unit : DynaSiUnits

DynaShortBase

An object used to store base information in a short form

Derived from DynaBase

Public Attributes:

start : float64

step : float64

DynaLongBase

An object used to store base information in a long form (one value per sample).

Derived from DynaBase

Public Attributes:

data : ooVArray(float64)

DynaScalar

A double precision floating point scalar object.

Derived from DynaObject

Public Attributes:

unit : DynaSiUnits

time : int32

content : float64

DynaDimN

An N-dimensional array object.

Derived from DynaObject

Public Attributes:

unit : DynaSiUnits

sizes : ooVArray(uint32)

bases[] : ooRef(DynaBase)

cal : ooRef(DynaCalibration)

ad cres : uint16

content : ooRef(Bulk)

DDL for Objectivity database layout

DynaObject.ddl

```
#ifndef __DynaObject
#define __DynaObject
#endif

#include "DynaSiUnits.h"
#include "DynaPolicy.h"
#include "DynaRevInfo.h"

#include "Bulk.ddl"

declare(ooString,32);
declare(ooString,256);
declare(ooVArray,DynaPolicy);
declare(ooVArray,DynaRevInfo);

class DynaObject: public ooObj
{
public:
    ooString(256) fullpath;
    ooString(32) name;
    uint32 level;
    uint32 quality;
    ooRef(DynaObject) references[] <-> theReferences[] ;
    ooRef(DynaObject) theReferences[] <-> references[];
    ooVArray(DynaPolicy) policies;
    ooVArray(DynaRevInfo) history;
};

class DynaComment: public DynaObject
{
public:
    ooVString content;

public:
    DynaComment() {};
};

class DynaMimeObj: public DynaObject
{
public:
    ooVString mimetype;
    ooVArray(uint8) content;

public:
    DynaMimeObj() {};
};
```

```

class DynaCalibration: public DynaObject
{

public:
    ooRef(DynaDimN) theDim[] <-> cal;

public:
    DynaCalibration() {};

};

class DynaPolyCalibration: public DynaCalibration
{

public:
    ooVArray(float64) coefficients;

public:
    DynaPolyCalibration() {};

};

class DynaTableCalibration: public DynaCalibration
{

public:
    ooVArray(float64) table;

public:
    DynaTableCalibration() {};

};

class DynaBase: public DynaObject
{

public:
    DynaSiUnits unit;
    ooRef(DynaDimN) dimNs[] <-> bases[];

public:
    DynaBase() {};

};

class DynaShortBase: public DynaBase
{

public:
    float64 start;
    float64 step;

public:
    DynaShortBase() {};

};

```

```

class DynaLongBase: public DynaBase
{

public:
    ooVArray(float64) data;

public:
    DynaLongBase() {};

};

class DynaScalar: public DynaObject
{

public:
    DynaSiUnits unit;
    int32 time;
    float64 content;

public:
    DynaScalar() {};

};

class DynaDimN: public DynaObject
{

public:
    DynaSiUnits unit;
    ooVArray(uint32) sizes;
    ooRef(DynaBase) bases[] <-> dimNs[];
    ooRef(DynaCalibration) cal <-> theDim[];
    uint16 adres;
    ooRef(Bulk) content <-> theBulk : delete(propagate);

public:
    DynaDimN() {};

};

```

DynaDirectory.ddl

```

#ifndef __DynaDirectory
#define __DynaDirectory
#endif

#include "DynaDirEntry.h"

declare(ooVArray,DynaDirEntry);

class DynaDirectory: public ooObj
{

public:
    ooVArray(DynaDirEntry) list;

public:
    DynaDirectory() {};

```

```
};
```

Bulk.ddl

```
#ifndef __Bulk
#define __Bulk
#endif
```

```
declare(ooVArray,int8);
declare(ooVArray,int16);
declare(ooVArray,int32);
declare(ooVArray,uint8);
declare(ooVArray,uint16);
declare(ooVArray,uint32);
declare(ooVArray,float32);
declare(ooVArray,float64);
```

```
class Bulk: public ooObj
{
```

```
public:
    uint32 size;
    ooRef(DynaDimN) theBulk <-> content;
```

```
public:
    Bulk() {};
```

```
};
```

```
class BulkInt8: public Bulk
{
```

```
public:
    ooVArray(int8) data;
```

```
public:
    BulkInt8() {};
```

```
};
```

```
class BulkInt16: public Bulk
{
```

```
public:
    ooVArray(int16) data;
```

```
public:
    BulkInt16() {};
```

```
};
```

```
class BulkInt32: public Bulk
{
```

```
public:
    ooVArray(int32) data;
```

```

public:
    BulkInt32() {};

};

class BulkUInt8: public Bulk
{
public:
    ooVArray(uint8) data;

public:
    BulkUInt8() {};

};

class BulkUInt16: public Bulk
{
public:
    ooVArray(uint16) data;

public:
    BulkUInt16() {};

};

class BulkUInt32: public Bulk
{
public:
    ooVArray(uint32) data;

public:
    BulkUInt32() {};

};

class BulkFloat32: public Bulk
{
public:
    ooVArray(float32) data;

public:
    BulkFloat32() {};

};

class BulkFloat64: public Bulk
{
public:
    ooVArray(float64) data;

public:

```



```

        BulkFloat64() {}

};

DynaDirEntry.h
#ifndef __DynaDirEntry
#define __DynaDirEntry

#include "DynaObject.h"

class DynaDirEntry
{

public:
    char path[256];
    ooRef(DynaObject) ref;

public:
    DynaDirEntry() { };

};

#endif

```

```

DynaSiUnits
#ifndef __DynaSiUnits
#define __DynaSiUnits

class DynaSiUnits
{

public:
    int32 kg;
    int32 m;
    int32 s;
    int32 A;
    int32 cd;
    int32 mol;
    int32 K;
    int32 rad;
    int32 sr;

public:
    DynaSiUnits() { };

};

#endif

```

```

DynaPolicy
#ifndef __DynaPolicy
#define __DynaPolicy

#include "DynaAccessMode.h"

class DynaPolicy
{

```

```
public:
    uint16 gid;
    DynaAccessMode mode;
```

```
public:
    DynaPolicy() { };
```

```
};
```

```
#endif
```

DynaRevInfo

```
#ifndef __DynaRevInfo
#define __DynaRevInfo
```

```
class DynaRevInfo
{
```

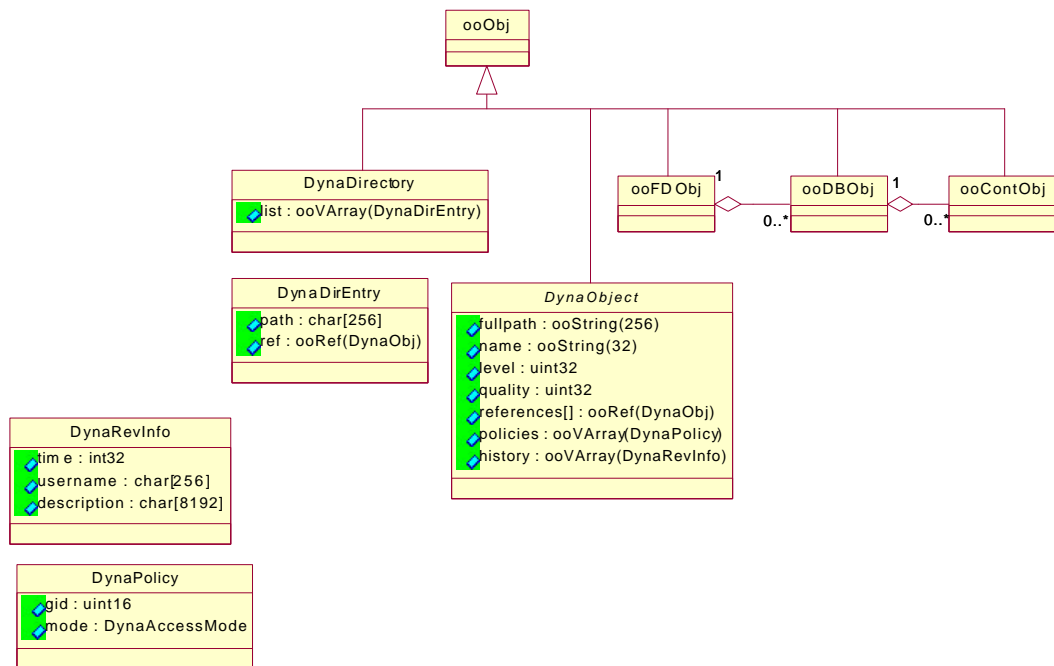
```
public:
    int32 time;
    char username[256];
    char description[8192];
```

```
public:
    DynaRevInfo() { };
```

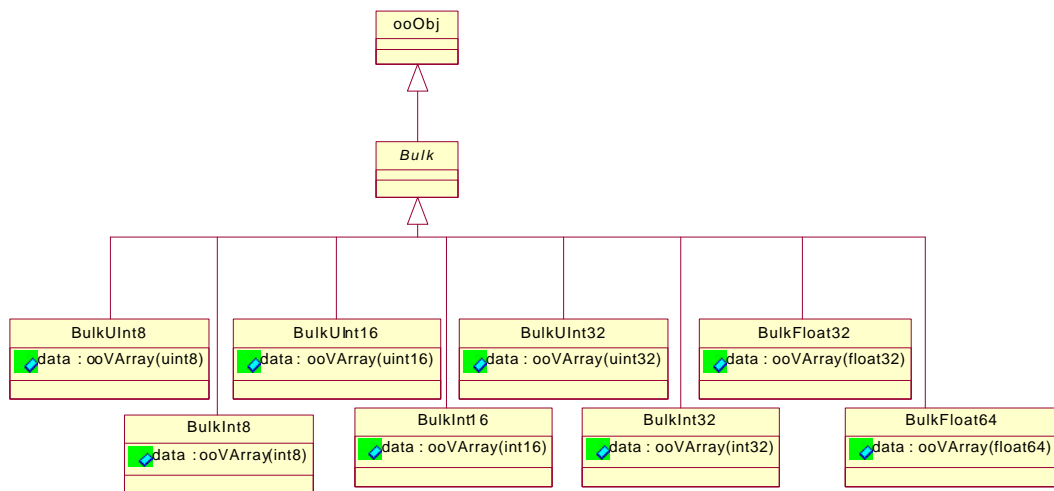
```
};
```

```
#endif
```

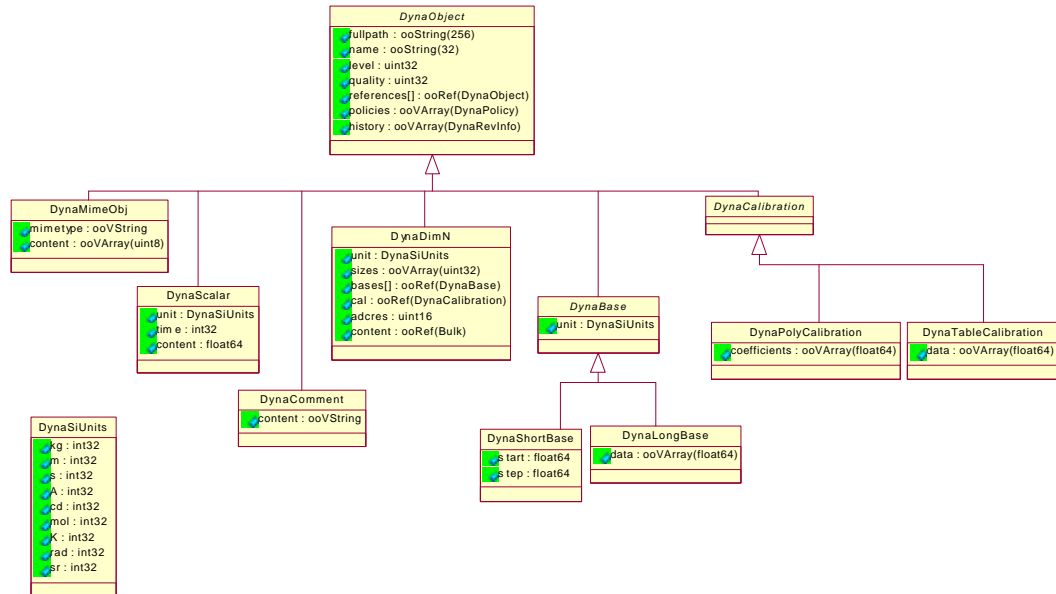
Objectivity class diagrams



• Figure 13 Hierarchy of top level classes of the DataManager



• Figure 14 Class hierarchy of raw data (BULK)



• Figure 15 Class hierarchy of measurement data