TELEMATICS APPLICATIONS
REMOT

Interfaces and Adaptations of the Plasma Physics Demonstrator

authors: Günter Kemmerling, Ernie van der Meer

Date: Monday, Friday 4 July 1997
Doc. ID: REMOT/C10 & C11/D6.3

Document type:	Research Report
Status:	last draft
Confidentiality:	Unclassified
Work Package allocation:	WP 6
Distribution:	PostScript, RTF, Word 6 HTML

Document history:

Version 1:
1st draft Friday 4 July 1997

Document abstract
This document presents the intended plasma physics demonstrator in the REMOT Project.

Project Sponsor
This report reflects work which is partially funded by the Commission of the European Communities (CEC) under the TELEMATICS APPLICATIONS programme.

Contents

1. Introduction

In document D6.2, a textual description of the soft- and hardware components of the plasma physics demonstrator as well as a definition of remote and local site was given. In order to couple these components to a complete teleoperation system, interfaces between them have to be defined and existing soft- and hardware have to be adapted. This task will be described in this document.

Because the CORBA standard for distribution will be used, it is most appropriate to describe the interfaces with the standardised *Interface Definition Language* (IDL). With IDL declarations, the interface between CORBA -client and -server implementations is defined. The IDL syntax is very similar to the C++ programming language. An *interface*-declaration corresponds to a C++ *class* declaration and the *attribute*-of an interface corresponds to the declaration of a class data member. In the server implementation an attribute can have two member functions. One to *get* the value of the attribute, and another to *set* the value of the attribute. Read-only attributes cannot be set.

The choice of the CORBA product is not yet fixed, because the OpenVMS operating system has to be supported, where no fully CORBA compliant implementation is currently available. ObjectBroker (DEC/BEA), which was foreseen for this platform, gave some trouble (e.g. complicated C++ mapping, lack of IIOP compatibility), wherefore it's no longer a candidate. It is expected to solve this problem by new versions of CORBA products (Orbix, DOME), which will appear in the next months. For other platforms, the free available omniORB (Oracle/Olivetti) has shown reasonable working.

The demonstrator will give the user the ability to handle the database and diagnostics remotely. For this task, there are four major aspects which have to be managed:

- secure access to the system
- synchronisation with the status of the TEXTOR machine
- control and set-up of diagnostics
- storage and retrieval of data

A mapping of these items to adequate interfaces will be described in the following sections. The design of the interfaces was done from an application specific point of view in order to concentrate first on the adaptation of the demonstrator components to the CORBA implementation. During the implementation phase, when more experience with CORBA is gathered, the interfaces will be refined and extended to a generic level, taking into account the elements of the REMOT architecture.

2. System access

First of all, the whole system must be protected against unauthorised access. Therefore user programs, which want to use the system, first have to contact a *LoginMgr* in order to make themselves known and to get authorisation for operations in the system (see figure 1).

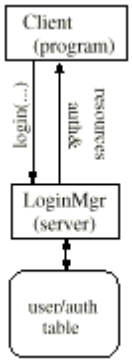

figure 1: User login

The server implementation of the *LoginMgr* is responsible for the distribution of proper authorisations. It has the following interface:

```
typedef struct
{
  string name;
  string location;
} resourceDescrpt;


interface LoginMgr
{
  void login(in string name, in string passwd);
  void logout(in string name, in string passwd);

  readonly attribute sequence<resourceDescrpt> resources;
};
```

The *login* and *logout* functions are used to enter or leave the system. By login, the *LoginMgr* compares the given username and password with an internal user/authorisation table and provides the client program with corresponding authorisation. There are currently different mechanisms under consideration, how the granting of authorisation can be managed. A description of this task will be included in a later document. Besides login/logout management, the *LoginMgr* has an attribute *resources*, which represents a list of other available servers in the system (Textor, diagnostics, database).

3. TEXTOR information server

From safety reasons, many set-up and control parameters of TEXTOR are outside the scope of remote control. Nevertheless, these set-up and status data are of importance for the data storage and the diagnostics, to be controlled by the demonstrator. All necessary information is available on an OpenVMS machine, which runs as a server and accesses the existing TEXTOR control system data.
To automate the retrieval of TEXTOR status information, users and diagnostics have to register at the TEXTOR server, which will broadcast messages in case of changes (see figure 2).

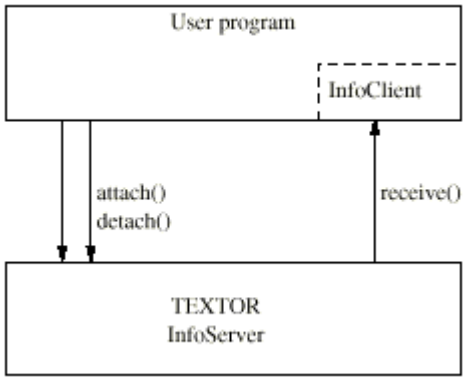

figure 2: Information broadcasting

The intended mechanism consists of two objects, *TextorInfoServer* and *InfoClient*. This last object has to be implemented by any client program that wants to be updated by the TEXTOR server. The following interfaces will be used:

```
interface InfoClient
{
  readonly attribute string name;

  void receive(in pulseState status);
};
```

and

```
interface TextorInfoServer
{
  void attach(in InfoClient);
  void detach(in InforClient);

  readonly attribute pulseState status;
  readonly attribute sequence<shotParam> params;
  readonly attribute sequence<cntrDescrpt> personnel;
};
```

All other Objects, which have to get TEXTOR information can be inherited from the *InfoClient* interface.

The *status*-attribute of the TextorInfoServer gives information about the current pulse-state of TEXTOR. It will have one of the following values (see also and D3.1-*User Requirements*).

```
typedef enum
{
  noPulse,
  prePulse,
  interPulse,
  activePulse,
  abortPulse,
  postPulse
} pulseState;
```

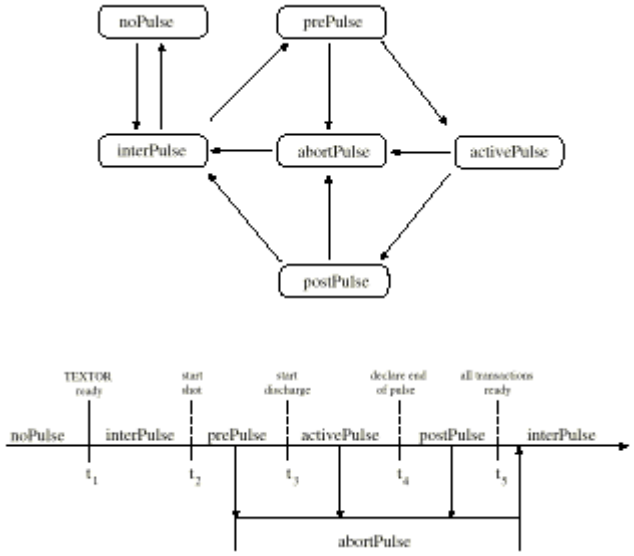

figure 3: pulse state diagram and transitions

The initial state of TEXTOR (switched off or maintenance) is described as *noPulse*. The experimental program begins with the *interPulse* state, which is also the stand-by state between two pulse cycles. In the *prePulse* state, set-up is done, instrumentation is armed and made ready for the next discharge. The discharge of TEXTOR takes place in *activePulse*. In the *postPulse* state all pulse related actions, like data storage, archiving will be terminated. *prePulse*, *activePulse* and *postPulse* states can, in case of error conditions, be aborted, which results in a *abortPulse* state.

The *params* attribute of the TextorInfoServer corresponds to the sequence of parameter of the last discharge at TEXTOR. The *shotParam* itself is defined as a data structure which contains the name of the Parameter, its value and the dimension of the value. The data format of the *paramValue*, which is declared as *any*, can be find out by analysing the connected CORBA typecode.

```
typedef string valueDim;
typedef string paramName;
typedef any paramValue;

typedef struct
{
  paramName name,
  paramValue val,
  valueDim dim
} shotParam;
```

The attribute *personnel* represents people working at TEXTOR control. It is a sequence of persons, which are described by their name and their responsibility (*cntrDescrpt*).

```
typedef string cntrName;
typedef enum
{
  machineOperator,
  dataOperator,
  radioFrequencyOperator,
  physicistOfTeam,
  engineerInCharge,
  physicistInCharge
} cntrType;

typedef struct
{
  cntrName name;
  cntrType function;
} cntrDescrpt;
```

As already mentioned, the server side has to run on an OpenVMS workstation in order to manage access to the existing TEXTOR control and databases in the local VMS cluster. One possibility to implement this interface is to adapt the attribute member functions to a script based query system, which is currently in use at TEXTOR.

4. Diagnostics

For the demonstrator two diagnostics will be controlled:

- the TEXTOR data logger and
- the pulsed radar system

Set-up and control of diagnostic instrumentation will be possible by the diagnostic server. Clients with appropriate authorisations will request actions on the server. The diagnostic implementation also contains a client side of the database, in order to store data (see figure 4). Furthermore TEXTOR itself behaves like a diagnostic, but it has a separate server implementation (see Ch. 3).

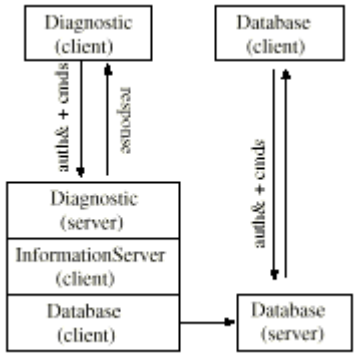

figure 4: Diagnostic and Database

It is foreseen to handle both diagnostics with the same interface description, which reflects the basic functionality of a diagnostic:

```
interface Diagnostic : InfoClient
{
  readonly attribute diagnDescrpt description;
  readonly attribute diagnState state;
  attribute diagnSetup setup;

  void start();
  void stop();
};
```

There are two readonly attributes. The first one, *description*, is set by the implementation and serves as an identifier for the diagnostic.

```
typedef string diagnName;
typedef enum
{
  mainDiagn,
  additionalDiagn,
} diagnType;

typedef struct
{
  diagnName name;
  diagnType function;
  cntrDescrpt responsible;
} diagnDescrpt;
```

The second, *state*, gives information about the current readout state, has still to be defined.
For the demonstration, the *setup* attribute will be nothing more than an index in a list of predefined set-ups for the specific diagnostic, which will be available at the client side.

The control of a diagnostic will be handled by two member functions (*start* and *stop*), which allow for an activation and deactivation of measurements. For the storage of data the Diagnostic will internally act as a client of the Database interface. As a result of the data acquisition (from the front electronics in the diagnostic system), data objects will be created and stored in the database. In case of performance problems, this mechanism can be changed at a later stage.
The experimental devices for the DataLogger-diagnostic are driven by an interface-card on an Alpha-station, using library functions provided by the supplier. To implement the diagnostic interface, library functions of the current data acquisition system will be used in the server member functions in order to give the right functionality.

5. Storage and retrieval of data

The current TEXTOR database system is divided into technical and experimental database. For the teleoperation system this tasks will be combined in one object database, whereas the commercial product Objectivity is foreseen.

The database will act as a object factory. Authorised clients can create different data objects, set their attributes and store them in the database. For the design of data objects, inheritance is used (see figure 5).

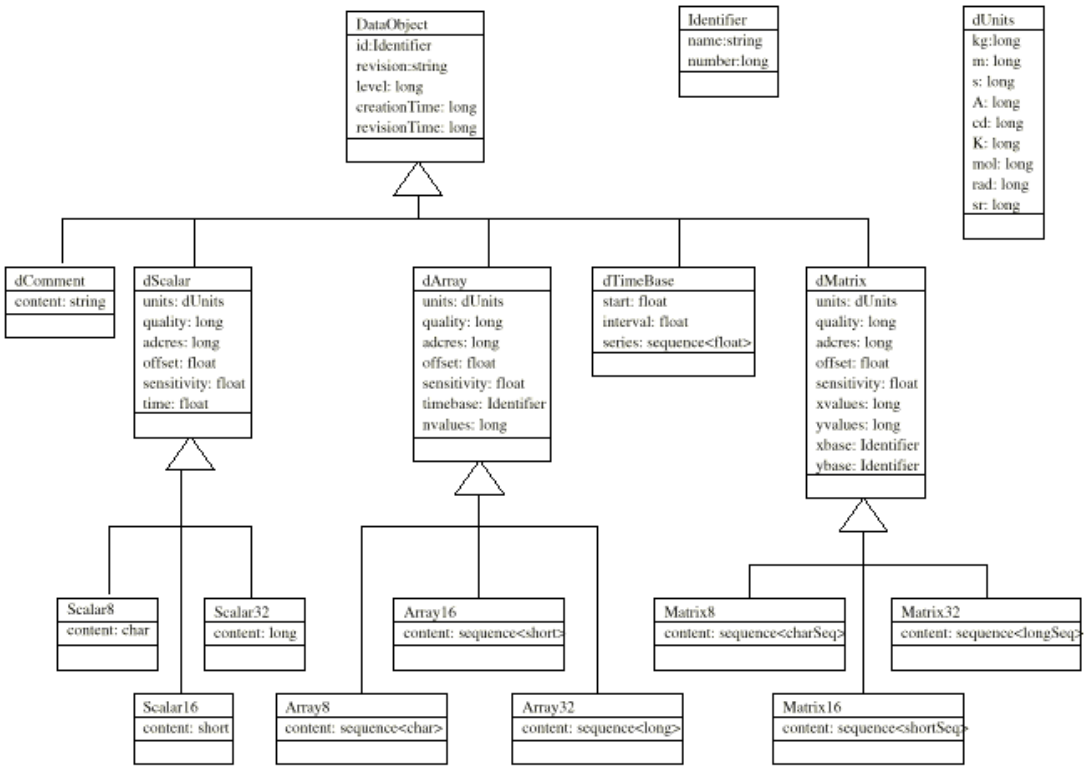

figure 5:Dataobject, inheritance scheme

There is one basic object (*DataObject*) from which a lot of other data objects are inherited, reflecting all possible data types of a diagnostic.

```
interface Database
{
  Object createData(in Identifier id, in type tp);
  boolean destroyData(in Identifier id);

  boolean storeData(in Object);
  Object retrieveData(in Identifier id);
};
```

The internals of the database are hidden from the users by the database server. The functionality of the database server will be provided by using the Objectivity programming interface. All data objects have their counterpart in Objectivity, defined through Objectivity's *Data Definition Language*. The storage location in Objectivity's

hierarchy (FederatedDatabase/Database/Container) is defined by the Identifier of the data object. This can also be used for the data retrieval.
Special attention has to be paid by storage of data from different diagnostics, because the level of distribution in Objectivity is the Database. In order to get the best performance in data transfer the most appropriate way for storage is to build a one-to-one relationship between a Diagnostic and a Objectivity Database (see). With this choice it will be possible to run a Diagnostic and a Database on the same machine, which will avoid network transfer of data. The composition of different Diagnostic-databases is then handled by database methods of Objectivity.

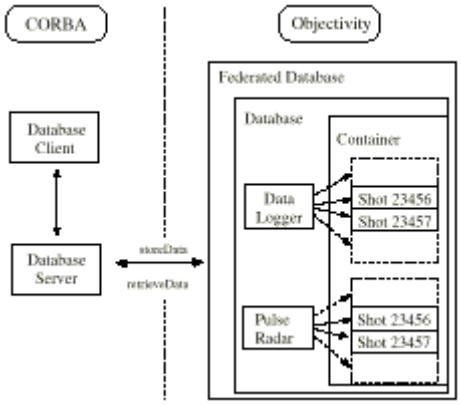

figure 6: database organisation

In a later stage, the concept of the database has to be extended to cover also the storage and retrieval of set-ups etc.