PAUL SCHERRER INSTITUT

SLS-TME-TA-2000-0162

October, 2000

CORBA Objects for SLS Subjects

Michael Böge, Jan Chrin

Paul Scherrer Institut
CH-5232 Villigen PSI
Switzerland

Presented at the Third International Workshop on Personal Computer and Particle Accelerator Control Systems (PCaPAC 2000), 9-12 October 2000, DESY, Hamburg, Germany

CORBA OBJECTS FOR SLS SUBJECTS

M. Böge, J. Chrin
Paul Scherrer Institut, 5232 Villigen PSI, Switzerland

E-mail: {Michael.Boege, Jan.Chrin}@psi.ch

Abstract

An object orientated client-server model based on the Common Object Request Broker Architecture (CORBA) has been developed for use in building high level software applications at the Swiss Light Source (SLS). A suite of CORBA server objects provides a convenient and uniform interface to several software packages required by beam dynamics applications. These include the TRACY accelerator physics package, the Common DEVice (CDEV) controls library, the Oracle database management system and a message and alarm logging facility. The framework facilitates rapid and reliable development of applications as client programs are sheltered from the intricate details of the many application program interfaces (APIs), while server objects are extensively tested through their use in different applications. The software and hardware components of the client-server model are presented together with a first account of their use in the commissioning of the SLS booster synchrotron.

1 INTRODUCTION

The SLS (Swiss Light Source) [1] is a synchrotron light source located at the Paul Scherrer Institute (PSI) in Switzerland. Scheduled for operation in August 2001, the SLS will deliver synchrotron radiation of high brilliance to experiments from a variety of disciplines. The SLS complex itself consists of three main accelerator facilities. A 100 MeV linear accelerator supplies electrons to a booster synchrotron which accelerates the electron beam to the full operating energy of 2.4 GeV before injection into the storage ring. Several high-level beam dynamics applications are required for the commissioning and operation of the SLS accelerator complex. These applications share a number of generic tasks that can be developed, through object oriented methodology, as reusable components. Common functions include:

- accelerator device control and monitor,
- data calibration and analysis,
- accelerator modelling,
- database operations and
- error message logging.

Traditionally, the choice of programming language and operating system has been motivated by the availability of associated libraries and extensions, often leading to a compromise. The advent of the Common Object Request Broker Architecture (CORBA) [2], however, allows for the development of a heterogenous distributed system in which objects, implemented in any language, on whatever platform, are able to interoperate. The advantage of using CORBA in a distributed client-server model for beam dynamics applications was recognized in earlier work [3] and its deployment at the SLS was motivated by requirements for flexibility, modularity and reusability [4]. Shared services can be developed as CORBA server objects in any language for which an appropriate application programmer interface (API) exists. In practice, APIs developed in C++ are preferred for optimal performance. The criteria for selecting the client language is consequently narrowed, in many cases, to aspects concerning the graphical user interface (GUI). Optimal use can thus be made of the GUI components provided by specialized toolkits such as the Tcl/Tk package [5], with the powerful BLT extension, or the Java Foundation Classes (JFC) [6]. Such a CORBA based client-server model further provides for the rapid development of applications as client programs are liberated from the complexities of the many APIs and can thus be kept comparatively short in length and be easily maintained.

The fundamental components of the CORBA framework are introduced and the CORBA objects developed for use in beam dynamics applications at the SLS are presented. An account of their use in the commissioning of the booster synchrotron accelerator is then given.

2 THE CORBA FRAMEWORK

CORBA, as with any new software technology, is not without its share of new terms and concepts. The most basic of these are recapitulated here. Their number should not, however, deter from the relative ease with which clients are able to invoke remote operations. (This is well illustrated by the code listings presented in section 3.) The basic CORBA capabilities are combined with standard CORBA services to provide the framework for the development of the CORBA server objects. These run on a single dedicated server machine providing a system that is distributed but not scattered!

2.1 The CORBA Reference Model

CORBA's most fundamental component is the **Object Request Broker (ORB)** whose task is to facilitate communication between objects. Given an **Interoperable Object Reference (IOR)**, the ORB is able to locate target objects and transmit parameters and return values to and from remote method invocations.

The interface to a CORBA object is specified using the CORBA **Interface Definition Language (IDL)**. An IDL compiler translates the IDL definition into an application programming language, such as C++ and Java, generating IDL stubs and skeletons that respectively provide the framework for client-side and server-side proxy code. Compilation of applications incorporating IDL stubs provides a strongly-typed **Static Invocation Interface (SII)**. Conversely, a more flexible communication mechanism can be established through the use of the **Dynamic Invocation Interface (DII)** and the **Dynamic Skeleton Interface (DSI)** allowing objects to be created without prior knowledge of the IDL interface.

Requests and responses between objects are delivered in a standard format defined by the **Internet Inter-ORB Protocol (IIOP)**, a communications protocol which adheres to the CORBA **General Inter-ORB Protocol (GIOP)** specification and, as such, acts as the base for CORBA interoperability on the internet.

Requests are marshalled in a platform independent format, by the client stub (or in the DII), and unmarshalled on the server-side into a platform specific format by the IDL skeleton (or in the DSI) and the object adaptor. In its mandatory version, the **Portable Object Adaptor (POA)** provides CORBA objects with a common set of methods for accessing ORB functions, ranging from user authentication to object activation and object persistence.

The ORB core is implemented as a runtime library linked into client-server applications. Fig. 1 shows the components of the CORBA architectural model.

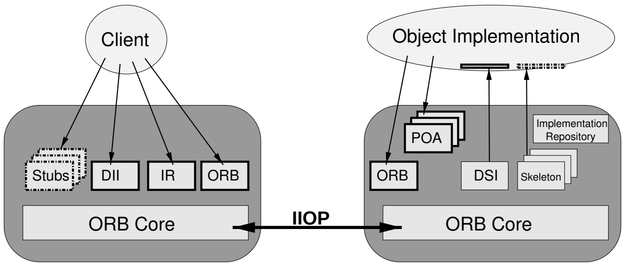

Figure 1: The CORBA client and server architecture.

2.2 Standard CORBA Objects

In addition to the basic ORB capabilities, a number of standard CORBA services, which are implementations of specific interfaces defined by the Object Management Group (OMG) [2], have been integrated to provide additional functionality to the CORBA applications.

The CORBA **Interface Repository (IR)** is a server object that disseminates the IDL interface of CORBA servers to clients enabling them to form and execute requests on the associated servers. The IR is typically used when establishing a dynamic DII interface to a server. The IOR for a server is obtained from the CORBA **Naming Service** that maps names to object references. CORBA servers register their object reference with the CORBA Naming Service at server start-up. An **Implementation Repository** is used by the POA for the reactivation of servers.

The CORBA **Event Service** provides services for the creation and management of CORBA event channels that may be used by CORBA supplier/consumer clients to progagate events asynchronously on a push or pull basis. Event channels are created and registered with the CORBA Naming Service allowing clients to obtain object references in the usual manner. Communication is anonymous in that the supplier does not require knowledge of the receiving consumers. The CORBA Event Service has been usefully employed in the monitoring of hardware devices and in the distribution of recalibrated data to client consumers.

2.3 Hardware and Software Platforms

The CORBA server machine, referred to as the "Model Server", is a dual-processor PC running Linux Red Hat version 6.0. It contains two 600 MHz Intel Pentium III processors with 512 Kilobyte memory cache per CPU and 1 Gigabyte of RAM. The use of Linux coupled with the GNU C++ compiler (egcs) minimizes vendor dependency and increases portability of applications. A second identical server is permanently available to provide redundancy.

The principal CORBA product employed is the recursively named MICO, '*M*ICO *I*s *CO*rba' [7], a fully compliant C++ implementation of the CORBA 2.3 standard, available free of charge under the GNU public license terms. In addition to the given IDL to C++ mapping, a Tcl extension [8] provides CORBA client and server functionality to Tcl scripts through the DII and DSI interfaces, respectively. MICO does not, however, provide an IDL to Java compiler. The Java-based JavaORB CORBA package [9] is presently used to negotiate language mapping to Java, although a second Java ORB, namely JacORB [10], is also under consideration and may ultimately be preferred[1].

[1] Extensive interoperability tests between JavaORB and MICO revealed a misinterpretation of data when passing certain primitive types over IIOP version 1.0. The mismatch could be alleviated through a realignment of the binary layout of the IDL data types. JacORB, on the other hand, specifically claims to interoperate with MICO. Both JacORB and MICO also support the Secure Sockets Layer (SSL) protocol which further adds to their appeal.

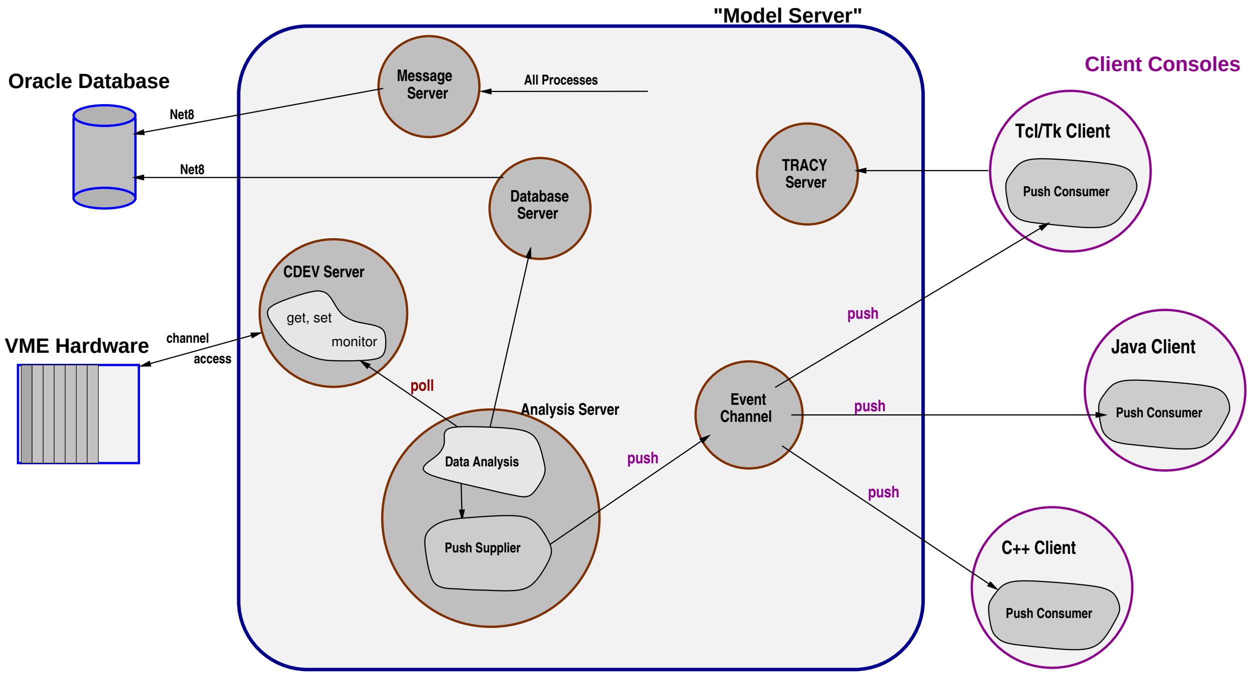

Figure 2: The architectural software model as used, in part, in the retrieval, analysis and distribution of controls data.

Client applications run on Linux PC consoles, typically equipped with a 450 MHz CPU and 256 Megabytes of RAM. The Red Hat Package Manager (RPM) is used for the distribution of the required software.

3 CORBA OBJECTS FOR SLS SUBJECTS

A number of *CORBA* application *Objects* have been developed *for* use by *SLS* Beam Dynamics *Subjects*. These objects have been built using the SII representation providing predefined interfaces to several APIs. A description of the objects is accompanied with listings of code that demonstrate their use. Remote method invocations essentially take on the welcoming appearance of a local function call, e.g. `object-->operation(arguments)` in C++. Preliminaries such as ORB initialization and object reference handles are not shown. Fig. 2 displays the components of the software architectural model. An example of an IDL source file is presented in section 3.5 which describes the interface to the message logger server.

3.1 *The CORBA TRACY Interface*

The TRACY [11] server provides access to the TRACY based model of the SLS transfer lines, booster synchrotron and storage ring. This includes methods to retrieve optics parameters for given machine settings and to perform model based corrections. The CORBA interface to TRACY provides access to the same machine model as used in offline simulations, permitting tested procedures to be employed for the optimization of the accelerator *online*. Table 1 shows a listing of a method that retrieves the optics parameters and corrector magnet values, as derived from an orbit correction procedure based on the orbit data associated with the vertical corrector magnets. The returned data are stored in a Tcl list which is unfolded through a Tcl procedure.

Table 1: Retrieval of optics parameters from the TRACY server in Tcl.

```
\\ $Object Operation Input Input Output
$GetLinOptI GetLinOptBCodDat -1 "vcm" ElemVcInput
\\ Procedure to unfold Tcl list and display data
$DisplayData $ElemVcInput
```

3.2 *The CORBA CDEV Interface*

The SLS accelerator device control system [12] is based on the Experimental Physics and Industrial Controls System (EPICS) [13] whose communications protocol is channel access [14]. The Common DEVice (CDEV) C++ class library (version 1.7.3) [15] provides the API to channel access, supplying functionality for both synchronous and asynchronous interaction with the control system. The CDEV interface takes the form of a device/message paradigm whereby a message constructed from a 'verb' and 'attribute' is routed to a 'device' through channel access. The CORBA CDEV server responds to CDEV type verbs such as 'set', 'get', 'monitorOn' to respectively download set-points, readback device attribute values and monitor selected channels. A change in value of a monitored channel invokes the CDEV callback function, wherein the new data value is both stored in memory and supplied to a CORBA event channel. Clients can either retrieve data from memory through the invocation of a CORBA method or be informed of updated values by subscribing as a consumer to the appropriate event channel. The software architecture allows several clients to connect to the CDEV server. However only a single monitor per channel need ever be enabled, thereby reducing the load on the SLS network and the low-level hardware. The CORBA CDEV interface handles a variety of EPICS records, including analogue and digital input/output records, waveforms and arrays. Use is also made of CDEV collections which serve to present related devices as a logical software entity.

Table 2: Retrieval of Slow Controls data from a Java client.

```
\\ Create holder for slow controls (SC) objects
BdCdevSeqHolder SeqCdevScObj= new BdCdevSeqHolder();
\\ Fill sequence of SC objects from config file
CdevRef.BdCdevScObj("ConfigFile", SeqCdevScObj);
\\ Send CDEV "get" message
CdevRef.BdCdevVerb("get", SeqCdevScObj);
\\ Local function prints results
BdCdevPrint (SeqCdevScObj);
```

Table 2 shows a Java client performing remote method invocations on the CDEV server. A sequence of ‘slow control objects’, consisting of CDEV device names, attributes, datatypes and values is first loaded with pertinent data from a configuration file or database. A subsequent method invocation employs the CDEV ‘get’ verb to retrieve the corresponding set of controls data. A local function then prints the corresponding set of results.

Data from the real-time control system, however, typically requires recalibration and/or analysis before being displayed by the client. Such tasks are assigned to the Analysis Server which distributes recalibrated data to interested clients through the CORBA Event Channel interface, as illustrated in Fig. 2.

3.3 The Event Channel Interface

Event data is propagated through the Event Service system in the form of the IDL type *any*. The *any* type is a container for either built-in or user-defined IDL data types, and further holds a TypeCode that acts as a run-time identifier of the prevailing data type. Dynamic run-time type safety is however sacrificed for static compile-type safety and any type mismatch escapes undetected at compile time. The possibility of such an occurrence is however greatly minimized through usage, under the auspicies of type *any*, of a single user-defined type that encapsulates a sequence of ‘slow control objects’. Thus, by design, all data propagated through the event channels of different domains, is of the same type, leading to a less complex and more robust transmission medium. Since event filtering is not part of the Event Service specification, network traffic is minimized by connecting a single supplier to a given event channel; clients interested in events from multiple sources thus register with multiple event channels. The use of event channels to handle event deliveries relieves the clients from the communication details and the need to establish callbacks. Table 3 shows how a push consumer is implemented in Tcl. The fact that the client is required to register as an event consumer is a consequence of the many event delivery models (i.e. push, pull, push/pull, pull/push.)

Table 3: Implementation of a push consumer in Tcl.

```
class Consumer_impl {
  inherit CosEventComm::PushConsumer
  public method Consumer_impl { }
  public method push { data } {
  \\ Invoke Tcl procedure to display data
  DisplayEventData $data
  }
  public method disconnect_push_consumer {
  \\ Action on disconnect
  }
}
\\ Instantiate Consumer class
set Consumer [Consumer_impl #auto ]
\\ Activate Consumer class
$poa activate_object $Consumer
$mgr activate
\\ Register as event channel consumer
set name {{ "SLS:Bd:EventChannel" kind " "}}
\\ Resolve event channel
set CosEventCh [$obj resolve $name]
set CosAdmin [$CosEventCh for_consumers]
set Supplier [$CosAdmin obtain_push_supplier]
$Supplier connect_push_consumer [$Consumer _this]
```

3.4 The CORBA Oracle Interface

The CORBA database server provides access to the Oracle8i database management system [16] by utilizing the Oracle Call Interface (OCI8 for Oracle8) through the C++ Oracle Template Library (OTL) [17]. The OTL templates are expanded into direct database API function calls which are transmitted to the Oracle database server using the Oracle Net8 network protocol. An high-level CORBA interface to Oracle is particularly appropriate as the overhead of building and then parsing textual commands is small compared to the cost of locating and reading the data from the Oracle instance.

The CORBA application objects developed typically handle three fundamental data types: a database connection, a Standard Query Language (SQL) statement and the corresponding results returned following execution of the statement. A connection to the database is first created by specifying the name of the database and the corresponding password. An SQL handle is then initialized with the SQL command that is to be executed. Once executed, for SQL queries returning data, a result set is delivered from which individual results may be requested. A description of the columns appearing in the SQL statement is also returned by quering standard Oracle database tables. The results are delivered in a separate class that can be interrogated when, for example, the typedef of a column is unknown.

Table 4: Database access from a Java client.

```
\\ Holder for results set from SQL query
BdDbSqlRetSeqHolder DbSqlRetSeq =
                        new BdDbSqlRetSeqHolder();
\\ Holder for column description
BdDbHeaderSeqHolder DbHeaderSeq =
                        new BdDbHeaderSeqHolder();
CORBA::String sqlStatement, stat;
\\ Prepare SQL statement (default database)
sqlStatement="Select sum(val),avg(val) from Table";
\\ Remote method invocation
stat= Dbref.BdDbOtlSelect
        (sqlStatement, DbHeaderSeq, DbSqlRetSeq);
\\ Print error message if any
if(stat!="") System.out.println("sqlStat:  " +stat);
\\ Print selected results from SQL query
System.out.println(DbSqlRetSeq.value[0].retDouble[0]
+"=Sum & Avg="+DbSqlRetSeq.value[0].retDouble(1));
\\ Print all rows from SQL query
BdDbPrint(DbHeaderSeq, DbSqlRetSeq);
```

The CORBA implementation provides easy access to the Oracle database as illustrated in Table 4 which lists a Java client performing a database retrieval operation. Methods that permit database modification operations have also been provided. Data arrays are mediated through OTL datatypes that map into Oracle BLOBs (Binary Large OBjects) or Oracle LONG RAWs.

The database server, for the most part, has been used to access static data though the storage of real-time data is foreseen.

3.5 *The CORBA Message Logger Interface*

All client and server processes are able to report messages and alarms to a dedicated CORBA message server. The server employs the UNIX syslog message logging facility incorporating a variety of priority levels, as indicated by the IDL specification shown in Table 5. Syslog entries are further transposed into SQL INSERT operations for immediate entry into the database via the Oracle SQL*Plus client. Messages entered follow a standard format, as illustrated in Table 6, and are viewed either through a Tcl/Tk based browser polling the log files or the native database browser which retrieves data from the corresponding database table.

Table 5: The IDL for the message server.

```
module MessageLogger {
  enum PriorityType{ emerg, alert, crit, err,
                     warning, notice, info, debug };
  interface Sys {
   short SystemLog(in string IdStr, in PriorityType
                    priority, in string LogMsg) };
};
```

Table 6: C++ code listing for error message logging.

```
\\ char *IdStr="processname@hostname[pid]"
char *IdStr=Identifier(argv[0]);
char LogMsg[MAXSIZE];
\\ Prepare Error Message
sprintf(LogMsg,"%s %d %s",__METHOD__,__LINE__,
                              "Invalid BPM data");
\\ Send Error Message with high severity "err"
LogRef-->SystemLog(IdStr,(PriorityType) err,LogMsg);
```

4 SLS BOOSTER COMMISSIONING

A number of beam dynamics applications have been developed for the commissioning of the booster synchrotron [4], making ample use of the CORBA application objects provided. During the three months of booster commissioning (July-September 2000), the model server recorded an uptime of 100%. The server processors are administered through the use of UNIX shell scripts which have sufficient modularity and flexibility to enable/disenable single or groups of servers and to understand and account for their inter-dependencies. Server objects, through their use in different client processes, were extensively tested and in some cases revealed certain anomalies in the underlying APIs. A crucial test of the software components presented itself whenever significant hardware problems occurred. The capability with which the message server could react to a multitude of messages was examined. Whilst some hundreds of error messages per second could be handled, the frequency with which error messages were reported was optimized to avoid unnecessary repetition. The situation also permitted a better understanding of the responses from the CDEV API suggesting, in some cases, a more appropriate action to be initiated.

The deployment of several event-based applications placed a stringent examination of the reliability of the MICO Event Service. Limits for the event queue lengths and timeouts were encountered which, in the worst case, would lead to a crash of the event daemon. Whilst such a scenario is easily avoided through a better match of supply and demand of events, a maximum queue length and an improvement to the error exception handling within the event daemon has since been afforded.

In general, following the initial period of implementation, operator intervention was minimal with clients able to interact spontaneously with the many servers and to display their event data. The CORBA objects are to be further tested and optimized through their use in applications currently under development for the commissioning of the SLS storage ring, beginning in January 2001.

5 CONCLUSION

An object oriented client-server model in which dedicated CORBA objects provide essential services to SLS subjects has been presented. Several beam dynamics applications developed for use in the commissioning of the SLS booster synchrotron provided an extensive test of the model components and demonstrated the framework to be both reliable and stable.

6 REFERENCES

[1] M. Böge *et al.*, "The Swiss Light Source Accelerator Complex: An Overview", Proc. 6th European Particle Acc. Conf. (EPAC'98), 22-26 June 1998, Stockholm, Sweden, Editors: S. Myers *et al.*, p. 623

[2] OMG, CORBA, http://www.omg.org/

[3] M. Böge, J. Chrin, "A CORBA Based Client-Server Model for Beam Dynamics Applications at the SLS", Proc. 7th Int. Conf. on Accelerator and Large Experimental Physics Control Systems (ICALEPCS'99), 4-8 Oct. 1999, Trieste, Italy, Editors: D. Bulfone, A. Daneels, p. 555

[4] M. Böge, J. Chrin, M. Muñoz, A. Streun, "Development of Beam Dynamics Applications Within a CORBA Framework at the SLS", Proc. 7th European Particle Acc. Conf. (EPAC 2000), 26-30 June 2000, Vienna, Austria, p. 1354

[5] J.K. Ousterhout, "Tcl and the TK Toolkit", Pub: Addison-Wesley; Tk/BLT, http://www.tcltk.com/blt/

[6] Java Foundation Classes, http://java.sun.com/products/jfc/

[7] A. Puder, K. Römer, "Mico: An Open Source CORBA Implementation", Pub: Morgan Kaufmann Publishers; MICO, http://www.mico.org/

[8] F. Pilhofer, "Combat, CORBA Scripting with Tcl", http://www.informatik.uni-frankfurt.de/~fp/Tcl/Combat/

[9] JavaORB, http://www.multimania.com/dogweb/details_java orb.html

[10] JacORB, http://www.inf.fu-berlin.de/~brose/jacorb

[11] J. Bengtsson, "TRACY-2 User's Manual", SLS Internal Document (1997); M. Böge, "Update on TRACY-2 Documentation", SLS Internal Note, SLS-TME-TA-1999-0002 (1999), http://slsbd.psi.ch/pub/slsnotes/

[12] S. Hunt *et al.*, "The Control and Data Acquisition System of the Swiss Light Source", Proc. 7th Int. Conf. on Accelerator and Large Experimental Physics Control Systems (ICALEPCS'99), 4-8 Oct. 1999, Trieste, Italy, Editors: D. Bulfone, A. Daneels, p. 615

[13] L. Dalesio *et al.,* "The Experimental and Industrial Controls System Architecture: Past, Present and Future", Proc. 4th Int. Conf. on Accelerator and Large Experimental Physics Control Systems (ICALEPCS'93), Berlin, Germany, Oct. 18-23, 1993, Editors: W. Busse, M.C. Crowley-Milling, Nucl. Instr. Meth. A352 (1994) 179; EPICS, http://www.aps.anl.gov/epics/

[14] J. Hill, "Channel Access: A Software Bus for the LAACS", Proc. Int. Conf. on Accelerator and Large Experimental Physics Control Systems (ICALEPCS'89), Vancouver, Canada, Oct. 30-Nov. 3, 1989, Editors: D.P. Gurd, M.C. Crowley-Milling, Nucl. Instr. Meth. A293 (1990) 352

[15] J. Chen *et al.*, "CDEV: An Object-Oriented Class Library for Developing Device Control Applications", Proc. 5th Int. Conf. on Accelerator and Large Experimental Physics Control Systems (ICALEPCS'95), Chicago, USA, Oct. 30-Nov. 3, 1995, Editors: M.C. Crowley-Milling, P. Lucas, P. Schoessow; CDEV, http://www.jlab.org/cdev/

[16] Oracle, http://www.oracle.com/

[17] S. Kuchin, "Oracle and Odbc Template Library (Version 3.0) Programmer's Guide", http://home.sprynet.com/~skuchin/

7 ADDENDUM

Since presenting this paper to the conference, extensive interoperability tests between MICO and several Java-based ORBs have been carried out (see section 2.3). The Java ORBacus 4 package (http://www.ooc.com/) was verified to interoperate *fully* with MICO and has thus been chosen to provide the IDL mapping to Java. Java-based client applications developed for SLS booster commissioning have since been adapted and tested.

November 2000.