# DIM, a Portable, Light Weight Package for Information Publishing, Data Transfer and Inter-process Communication

*C. Gaspar*[1] , *M. Dönszelmann*[1] , *Ph. Charpentier*[1]

CERN, European Organisation for Nuclear Research, CH 1211 Geneva 23, Switzerland

**Abstract**

The real-time systems of HEP experiments are presently highly distributed, possibly on heterogeneous CPU's. In many applications, there is an important need to make information available to a large number of other processes in a transparent way. For this purpose the "RPC-like" systems are not suitable, since most of them rely on polling from the client and one-to-one connections. DIM is a very powerful alternative to those systems. It provides a named space for processes to publish information (Publishers) and a very simple API for processes willing to use this information (Subscribers). It fully handles error recovery at the Publisher and Subscriber level, without additional software in the application. DIM is available on a large variety of platforms and operating systems with C and C++ bindings. It is presently used in several HEP experiments, while it was developed in the DELPHI experiment and is maintained at CERN. We shall present its capabilities and examples of its use in HEP experiments in domains ranging from simple data publishing to event transfer, process control or communication layer for an Experiment Control Package (SMI++). We shall also present prospectives for using it as communications layer for future experiment's control systems.

Keywords: asynchronous communications, heterogeneous distributed systems

## 1 Introduction

The DIM (Distributed Information Management)[1] system was designed in order to provide a communication layer to all process involved with the different tasks of Delphi's Online System. The Online System is responsible for Data Acquisition, Trigger, Control, Monitoring, User Interfacing, etc. Delphi's Online System is composed of about 500 processes distributed over around 50 different machines running mainly VMS and the OS9 Operating Systems. In order to fulfill their tasks these processes need to communicate efficiently and reliably across the different machines. DIM Provides transparent distributed communications to all the processes involved in the different activities of the Online System.

Even though DIM was developed for Delphi, it was designed in a generic fashion so that it could be used widely in other platforms and for other applications. It is currently being used by other experiments: NA50, L3, the L3 Cosmics sub-experiment and BaBar.

## 2 Design Requirements

Since the purpose of the DIM System is to provide a communication layer to all processes involved in all types of activities of the complete Online System its design had to take into account the requirements and constraints of the different areas. For example: for the Data Acquisition System efficiency (transfer speed) was an important issue, for the Control System reliability was the most important requirement, for the Monitoring System the difficulty is the handling of large amounts of

data, while for User Interface purposes it is necessary to have access to all the information available in the experiment. In order to accomplish this task DIM was designed according to the following requirements:

- Efficient Communication Mechanism
  Online, real time, systems have particular requirements for what concerns the communication mechanism:
  - Asynchronous Communications
    In order to react quickly to condition changes and error situations the communication mechanism should allow for asynchronous communications. A process should be not have to poll at regular intervals in order to find out that something changed, it should be informed when it changes.
  - One-to-many Communications
    In many cases when something happens in the system more then one processes have to be notified. This is, for instance, the case of User Interfaces since many users may be viewing the same part of the system.

  For DELPHI's purposes an efficient communication mechanism has to respect these requirements without major overheads in terms of time and bandwidth.
- Uniformity
  The DIM system should be capable of handling all exchanges within the online system, all processes involved with control, monitoring, processing or display should use the same communication system. A homogeneous system is much easier to program and to maintain.
- Transparency
  An important goal for a distributed communication system is transparency. No matter where a process runs, it should be able to communicate with any other process in the system, using a single mechanism that is independent of where the processes are located.
  - Run-Time
    The DIM system should allow processes to communicate with each other without knowing in which machine they run and should allow processes to move freely from one machine to another. All communications should be transparently re-established. This allows for an easier recovery from error situations and also for balancing the load between the different machines.
  - Coding-Time
    Distributed applications are often very difficult to program. When coding a distributed application the user should not be concerned with machine boundaries. The communication system should provide a location-transparent interface, i.e.: the code should be the same if the communicating processes reside in the same machine or on different machines of different types in different parts of the world.
    The programming interface should hide from the user all communication issues and reduce to the minimum the necessity for additional user code.
- Reliability and Robustness
  In an environment with many processes, processors and networks, it often happens that a process, a processor or a network link breaks down. The loss of one of these items should not perturb the rest of the application. DIM should provide for automatic recovery from error situations or the migration of processes from one machine to another.
- Wide-area Transparency
  DELPHI is an international collaboration, any necessary information should be available to the outside world, using the same system.

By fulfilling such requirements, a communication system can greatly improve the development and performance of the complete system. It provides a decoupling layer between software modules, which makes coding, maintenance and upgrading of the complete application easier and improves efficiency and reliability at run-time.

## 3 Communication Mechanism Considerations

When designing a communication system, the choice of the communication mechanism to be used is an important issue. Distributed applications are often based on remote procedure calls (RPC)[2]. In the RPC mechanism the client sends a message containing the name of a routine to be executed and its parameters to a server, the server executes the routine and sends a message back containing the result. This implies that the communication is point-to-point and synchronous, since the client always waits until the routine finishes execution, as shown in the diagram of Figure 1.

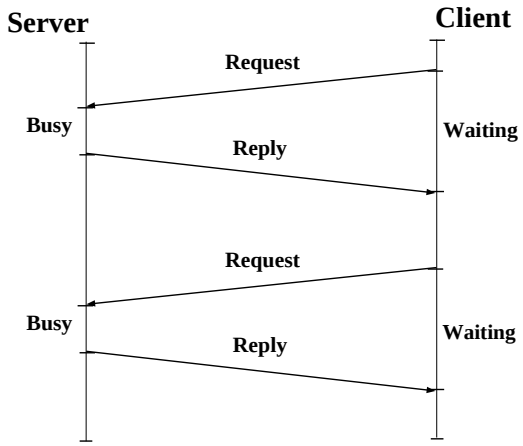


**Figure 1:** RPC Mechanism

Many of today's applications use CORBA (Common Object Request Broker Architecture)[3], CORBA is in some way an extension of the RPC mechanism to the object world since it allows the invocation of a method on a remote object. CORBA offers several advantages over RPC like dynamic facilities as well and more transparency (the client object doesn't have to know where the remote object is running) but, as RPC, at the base it implements a synchronous, one-to-one protocol.

For online applications like DELPHI (but probably also for others such as industrial control or real-time applications) the RPC mechanism is very heavy and ill suited. In most of these applications important tasks are the monitoring of parameters (at regular time intervals) and reacting asynchronously to changing conditions; furthermore these tasks are normally repetitive and have to be executed indefinitely.

The solution that seemed the best in this case is for clients to declare interest in a service provided by a server only once (at start-up), and then get updates at regular time intervals or when the conditions change. I.e., a protocol allowing asynchronous and one-to-many communications, as depicted in Figure 2.

This mechanism, interrupt-driven, as opposed to RPC's polling approach involves twice less messages sent over the network, i.e. it is faster and saves in network bandwidth. It has also the advantages of allowing parallelism (since the client does not have to wait for the server to reply

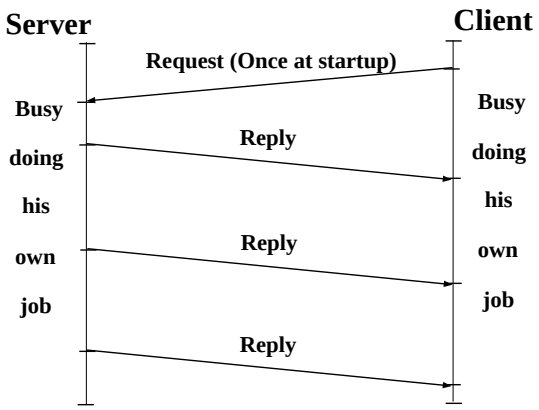


**Figure 2:** DIM Mechanism

and so can be busy with other tasks) and of allowing multiple clients to receive updates in parallel. This approach, together with the possibility of sending commands to servers (more RPC-like), are the main features of the DIM communication mechanism.

## 4 Design Philosophy

DIM, like most communication systems, is based on the client/server paradigm.

The basic concept in the DIM approach is the concept of 'service'. Servers provide services to clients. A service is normally a set of data (of any type or size) and it is recognized by a name ('named services'). The name space for services is free.

Four commonly used types of services have been identified:

- ONCE-ONLY: The client requests information.
- TIMED: The client requests the information to be updated at regular time intervals.
- MONITORED: The client requests the information to be updated whenever it changes (availability depends on whether the server provides it).
- COMMAND: The client sends a command to the server.

The TIMED and MONITORED services are only requested once by the client (normally at start-up). The service will then be updated automatically by the server.

When using MONITORED services, the server will update the information sent to all clients whenever it changes, thus making sure the data are coherent over all the clients of a certain service. The updating mechanism can be of two types, implemented either by executing a client call-back routine or by updating a client buffer with the new set of data, or both. In fact this last type works as if the clients maintain a copy of the server's data in cache, the cache coherence being assured by the server.

In order to allow for the required transparency (i.e., a client does not need to know where a server is running) as well as to allow for easy recovery from crashes and migration of servers, a name server was introduced.

Servers 'publish' their services by registering them with the name server (once, normally at start-up).

Clients 'subscribe' to services by asking the name server which server provides the service and then contacting the server directly, providing the type of service and the type of update as pa-

rameters.

The name server keeps an up-to-date directory of all the servers and services available in the system.

Whenever one of the processes (a server or even the name server) in the system crashes or dies all processes connected to it will be notified and will reconnect as soon as it comes back to life. This feature not only allows for easy recovery, but allows for the easy migration of a server from one machine to another (by stopping it in the first machine and starting it in the second one), and so for the possibility of balancing the machine load of the different workstations.

The interaction between Servers, Clients and the Name Server can best be seen in the Data Flow Diagram (DFD) of Figure 3.

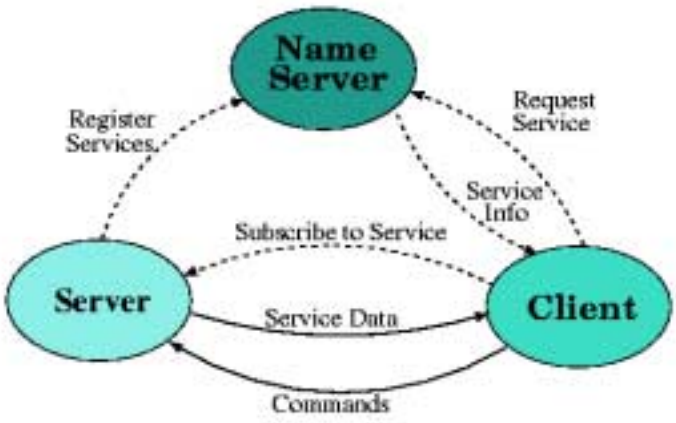


**Figure 3:** DIM's Main Data Flow Diagram

This DFD shows the control and data flow among the basic components of a DIM system, the Name Server receives service registration messages from servers and service requests from clients. Once a client obtains the "Service Info", i.e. the Service co-ordinates, from the Name Server it can then subscribe to services or send commands directly to the Server. If a client sends a "Request Service" for a service that is not (yet) know to the Name Server a negative "Service Info" is sent back to the client but the request stays queued in the Name Server and when the Service is made available a new "Service Info" is then sent to the client and the client proceeds to connecting to the Server.

## 5 DIM Implementation

### 5.0.1 DIM Servers

DIM Servers are processes that have information to provide. In order to become a server a process has to declare any services it can provide and any commands it is willing to accept and then send this information to the Name Server.

A Service is declared by specifying a name for the service, the address and size of the data that composes this service and the format of the data (in order to convert between different platforms).

In C++ the class DimService implements service handling. The format and size are implicit for basic data types (integers, floats, strings, etc.), for complex data types like structures the user has to provide format and size description.

```
//Example of a server publishing two services

#include <dis.hh>

int main()
{
    int run = 0;
    struct test{ int a; float b; char arr[10]; } var;
    DimService runNumber("DELPHI/RUN_NUMBER", run);
    DimService structVar("DELPHI/TEST/VAR", "I:1;F:1;C",
        &var, sizeof(var));
    DimServer::start("RUN_INFO");
//  ...
}
```

Commands are declared by specifying a name for the command, the command format, and a callback routine to be called when the command is received.

In C++ the class DimCommand implements command handling by using the virtual method commandHandler.

```
//Example of a server that receives a command

#include <dis.hh>

class Command: public DimCommand
{
    void commandHandler()
    {
        cout << "Received : " << getString() << "\n";
    }
    public:
        Command() : DimCommand("DELPHI/TEST/CMND","C");
};

int main()
{
    Command cmnd;
    DimServer::start("TEST");
    while(1)
        pause();
}
```

After declaring all services and commands provided, another method has to be called in order to start serving client requests - DimServer::start(). This static method of class DimServer will send the full list of services and commands to the name server and set-up all the necessary mechanisms in order for client requests to be handled transparently.

From then on the server can go back to its normal task, any client requests and service updates will be handled automatically by the DIM system.

The server can however force the update of a certain service whenever necessary (a condition changed or an error was found) as in the following example.

```
//Example of a server publishing and updating a service

#include <dis.hh>

int main()
{
    int run = 0;
    DimService runNumber("DELPHI/RUN_NUMBER",run);
    DimServer::start("RUN_INFO");
    while(1)
    {
//      ...
        run++;
        runNumber.updateService();
    }
}
```

### 5.0.2 DIM Clients

DIM clients are processes that need the available information in order to accomplish their tasks, that being display, monitoring or processing.

A process becomes a client by specifying the service name it is interested in and an optional update interval (it will anyway receive services on change). The client can also specify a constant to be used as service data whenever the service is or becomes unavailable.

```
//Example of a client requesting a service every 5 seconds
//(and/or on change). -1 is received if the service fails

#include <dic.hh>

int main()
{
    DimInfo runNumber("DELPHI/RUN_NUMBER",5,-1);
//  ...
    cout << "Run Number " << runNumber.getInt();
}
```

A client can also specify a callback routine to be called when new service data arrives.

```
//Example of a client requesting a service on change with a callback.
//-1 is returned be getInt() if the service fails.

#include <dic.hh>

class RunNumber : public DimInfo
{
    void infoHandler()
    {
        cout << "Run Number " << getInt() << "\n";
    }
    public :
```

```
        RunNumber() : DimInfo("DELPHI/RUN_NUMBER",-1) {};
};

int main()
{
    RunNumber runNumber;
    while(1)
        pause();
}
```

From then on the client can go on with its work, any service data will automatically be stored and/or the call-back executed whenever a service is received.

A process can at any time send a command to a server by specifying the command name and the command data.

```
//Example of a client sending a command

#include <dic.hh>

int main()
{
    DimClient::sendCommand("DELPHI/TEST/CMND","DO_IT");
}
```

The internal steps involved in becoming a client, either by requesting a service or sending a command are the following: first the client asks the Name Server where is the service available, the name server replies the server's co-ordinates, The client contacts then directly the server and formulates its request (including the update type). The server will then take care of updating the service whenever necessary. If the service is not available the Name server will reply negatively, the constant data are put into the service data path and an update is triggered but the request stays queued in the name server, whenever the service becomes available the client will be notified and the connection established.

If the Name server was not available the client will retry at random (between limits) intervals, until the Name Server is back up.

Any process can at the same time be a DIM Server and a DIM client.

### 5.0.3 The Name Server

The name server keeps an up-to-date list of all the servers and services in the system, it receives registration messages from servers and service requests from clients. Servers also send watch-dog messages at regular intervals so that the Name Server can be assured that they are functioning. If a server fails sending watch-dog messages the Name Server marks its services as not available and any client requests for them will be queued until they became ok again.

The name server will send a kill signal to any process that tries to register a service that is already registered. Services must be unique.

If the Name Server dies all previously established connections between servers and clients will keep working. When it comes back up all servers re-register all their services (they have been trying at random intervals) and all the clients re-request the services they are waiting for and all connections are then established.

The Name Server can start on a different machine, the servers and clients try to find it in a predefined list of possible machines (normally an environment variable).

The service directory in the Name Server is implemented as a Hash Table.

The Name Server is also a DIM server, in the sense that it provides DIM services. It publishes as DIM services information related to the servers and services available on the system, this information can be accessed by any process. In C++ the class DimBrowser can be used to access this information. It provides: list of services per server, list of clients of a server and in general list of servers and of services available (wildcards allowed).

## 6 Monitoring and Debugging

The behavior of complex distributed applications can be very difficult to understand without the help of a dedicated tool. The DIM mechanism, unlike others, makes it very easy to build a display tool: all services are published and so accessible from anywhere.

The DIM System provides a tool DID, the Distributed Information Display that allows the visualization of the processes involved in the application. DID provides information on the servers and services available in the system at a given moment and on the clients using them. The contents of a particular service can also be visualized. Figure 4 shows an example of the use of DID within the DELPHI environment.

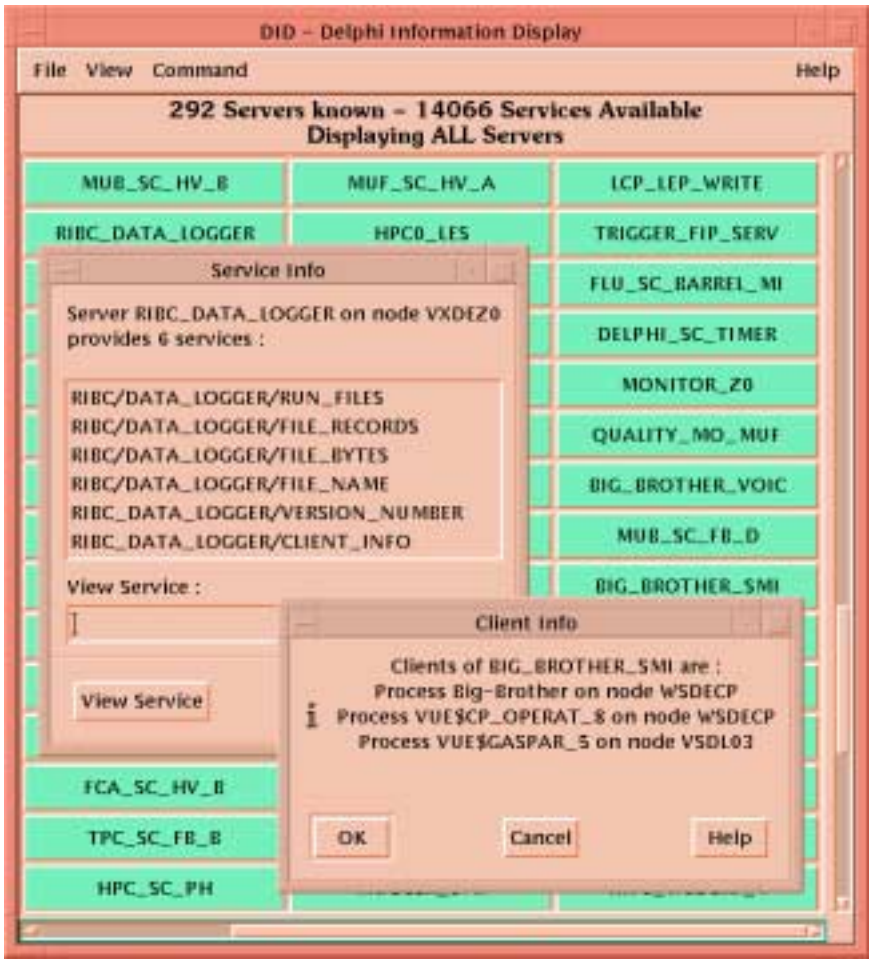


**Figure 4:** DID - The Distributed Information Display

DID obtains most of its information from the Name Server, the list of servers available, where they run and which services they provide. In order to obtain more detailed information like which clients are connected to a particular server, what is the format of a particular service or to get the contents of a service that particular server is contacted.

The information available as DIM services can also be accessed through WWW (the World Wide Web) through a DIM-WWW gateway[4]. The WWW page can be written in HTML with specific DIM tags containing the service name. The DIM tags are translated when the page is loaded.

## 7 Availability and Performance

DIM is available in mixed environments comprising the Operating Systems: VMS (VAX and ALPHA), UNIX flavors (HP-UX, IBM-AIX, SUN-OS, SUN-Solaris, DEC-OSF, Linux), Windows NT, and some Real-time Operating Systems (OS9, LynxOs and VxWorks).

When exchanging data between these machines the data formats and data representation in the different machines have to be taken into account. All differences (byte ordering, structure alignment, floating point representation, etc) are transparently handled by the DIM library.

Unlike other protocols DIM does not systematically convert data from machine format to network format and back on the receiving end. We have taken into account that in a large distributed system there is a high probability that several machines are of the same type in which case there is nothing to do. In DIM the data is only marshaled if the two machines involved are different.

In order to measure DIM's performance it has been compared with the performance of raw TCP/IP, i.e., the fastest possible TCP/IP transfer. The machines used are ALPHA Stations, 250 MHz running the VMS operating system connected through ethernet.

The test consists on sending a buffer from a client to a server and back to the client. In DIM's case the transfer is implemented using standard DIM calls, i.e., using the full functionality of DIM. What we call raw TCP/IP are two very simple programs containing a loop consisting of only read buffer, write buffer for the server and write buffer, read buffer for the client.

Two quantities were measured: The throughput, i.e., The quantity of bytes transferred per second and the time of each transfer (both ways) for different sizes of the buffer transferred.

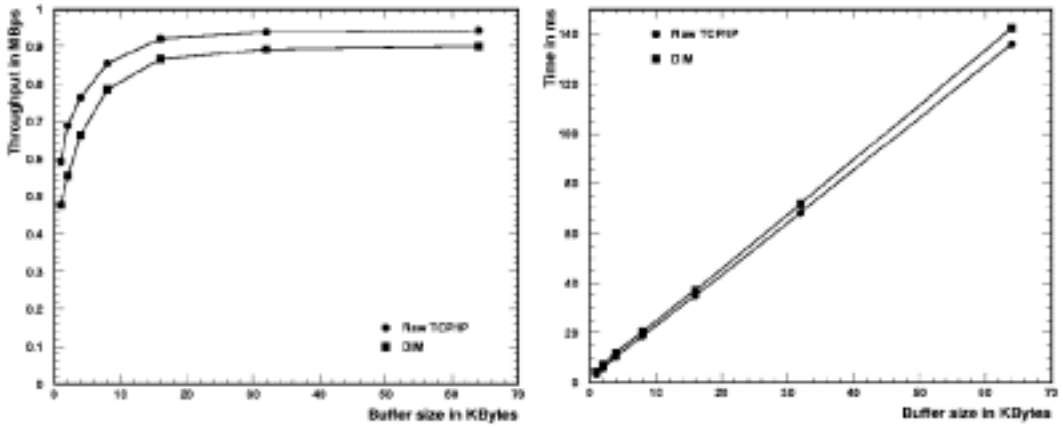


**Figure 5:** Performance Results over Ethernet

Ethernet's maximum data rate is 10Mbps. As we see from the graphic in Figure 5, both DIM and raw TCP/IP curves saturate due to the Ethernet bandwidth. DIM is about 1.05 times slower than raw TCP/IP and can be used to transfer data up to 900Kbytes/second on a "free" Ethernet segment.

# 8 DIM Usage

## 8.1 In Delphi

### 8.1.1 Control

DELPHI uses the same approach for the control of all areas of the experiment: Data Acquisition, Trigger, Detector Control (or classical Slow Controls), etc. obtaining what we called ”the Experiment Control System” [6]. Delphi uses throughout the control system a toolkit - SMI++ (State Management Interface)[5]. The SMI++ methodology is based on two concepts: Objects and Finite State Machines (FSM). This approach allows the description of the full experiment as a hierarchy of domains composed of objects, each object behaving as a finite state machine. The domains (and the objects) run in parallel distributed over several machines. Figure 6 shows the decomposition of DELPHI in terms of SMI++ domains.

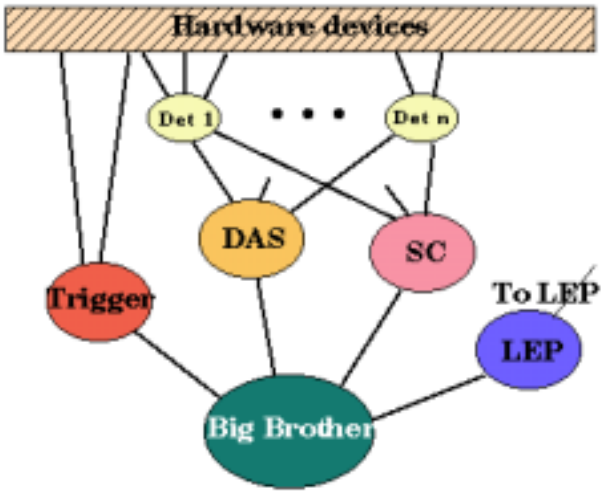


**Figure 6:** Main Delphi SMI++ Domains

Under normal running conditions a top-level domain -BIG BROTHER- pilots the system with minimal operator intervention. In other test and set-up periods the operator becomes the top-level object and using the user-interfaces he can send commands to any SMI domain.

The full system comprises about 1000 SMI objects in 50 different domains distributed over 40 machines.

DIM provides for the exchange of state information and actions between the different objects. The system is completely asynchronous, the reception of state changes and/or actions trigger FSM transactions that provoke state changes. Since DIM is a publish/subscribe mechanism the states of the different objects are made available to other interested processes like user-interfaces.

### 8.1.2 Data Acquisition

Delphi’s data acquisition[7] is partitioned at the level of sub-detectors (20). Sub-detectors can be read-out together (normal physics operation) or in stand-alone (for tests and calibration). In both cases the event or sub-event data are published as DIM Services by the OS9 processors (event builder and sub-event builders). This data is then available in the host machines to the event consumer processes: Data-loggers, Monitoring tasks, etc.

### 8.1.3 Monitoring

The Monitoring tasks that got the data (complete events and sub-detector spying events) via DIM analyze them and produce histograms that they again publish as DIM Services so that Histogram analyzers and presenters can get them.

### 8.1.4 User Interfacing

Delphi's user interface allows access to all parts of the experiment in an homogeneous and coherent way. From simple views of the full experiment down to detailed information about a particular subsystem all are available from the same tool - DUI (Delphi User Interface)[8]. Figure 7 shows a view of DUI configured for Run Control.

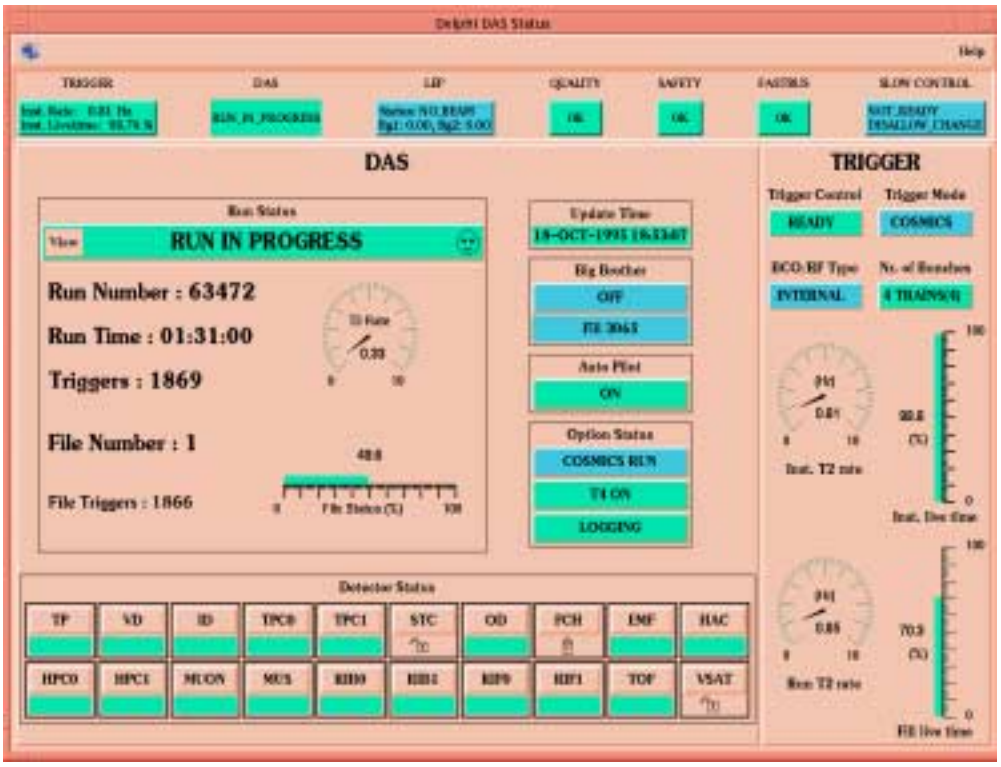


**Figure 7:** DUI view

All information displayed by DUI components comes via DIM (either from SMI++ Object states or directly from DIM Servers) and all user commands are sent via DIM to SMI++ objects.

## 8.2 In L3 and L3 Cosmics

In L3 DIM is user in order to exchange information between VMS, SUN Solaris (host machines) and OS9 (Embedded processors).

in L3 Cosmics DIM is used for inter-process communication between processes on different platforms, but also between several processes on a single machine. The use of different types of platforms (HP-UX, LynxOs, LINUX and even VMS) encountered in several parts of the L3 Cosmics online and monitoring system, proved to be transparent to the user. In particular, DIM is used:

- As a DataBase Server since the L3 database resides on VMS and L3 cosmics (on HP-UX and LynxOs) needs to access it.

- For job control: The mechanisms to check what processes are running and to start/restart/stop them are implemented using DIM.
- To read data out of LynxOS into HP-UX.
- To ship Online Data over to machines not normally belonging to the online system (eg at the home institutes) for monitoring purposes during data taking.

### 8.3 In NA50 and BaBar

DIM is mainly used as the underlying layer of SMI[10] (the previous version of SMI++ in NA50) and of SMI++ (in BaBar)[11].

BaBar's Run Control is composed of hundreds of SMI++ domains running on SUN Solaris machines.

## 9 DIM's Future

There are currently two tendencies for what concerns Operating Systems: Windows NT and UNIX. For several reasons it will be impossible in the near future to agree on using only one of them in the LHC experiments. For instance LHC experiments are planning to base their control systems on commercial tools (SCADA - Supervisory Control and Data Acquisition) [12] which run mainly on Windows NT while several LHC sub-detectors plan to build their control systems around VME processors running some form of UNIX (possibly Real Time). The problem of communications is still open. Since DIM runs on several platforms and it can easily be integrated with commercial packages like SCADA systems (it has been integrated with one) it can help solving this problem.

## 10 Conclusions

DIM provides efficient and reliable inter-process communications across different platforms. Its communication mechanism is based on the publish/subscribe method and allows for asynchronous communications and multiple destination updates.

DIM is responsible for basically all communications inside the DELPHI Online System, in this environment it makes available around 30000 services provided by 450 servers.

DIM is also being used by other experiments at CERN (L3, L3 Cosmics and NA50) and by BaBar at SLAC. Other projects (like the Atlas Data Acquisition) decided not to use it directly but to adopt a similar philosophy[13]. DIM has recently been adopted as an interim solution to be proposed to LHC sub-detectors (for their problems of communications within the control system) by the CERN IT Division's Controls Group.

## 11 Acknowledgments

We would like to thank our collegues Michel Jonker and Boda Franek for all the fruitfull discussions at the various phases of the development of the DIM package. We would also like to thank all DIM Users for their collaboration, support and for the numerous suggestions for improvement.

## References

1 C. Gaspar, M. Dönszelmann, "DIM - A Distributed Information Management System for the DELPHI Experiment at CERN", Proceedings of the 8th Conference on Real-Time Computer applications in Nuclear, Particle and Plasma Physics, Vancouver, Canada, June 1993.

2 A. D. Birrell, B. J. Nelson, "Implementing Remote Procedure Call", in ACM Trans. Comp. Syst., Vol. 2, No. 1, 1984.

3 Framingham, “CORBA: Common Object Request Broker Architecture and Specification”, Object Management Group Inc, 1995.
4 M. Dönszelmann, K. Rodden, “Gateways for the World-Wide Web in the ’Online’ Data Acquisition System of the DELPHI Experiment at CERN”, Proceedings of The Second International World Wide Web (WWW) Conference, Chicago, Illinois, USA, 1994.
5 B. Franek, C. Gaspar, “SMI++ - Object Oriented Framework for Designing Control Systems for HEP Experiments”, Computer Physics Communications, May 1998, Vol 110, Num 1-2, p. 87-90.
6 C. Gaspar, J.J. Schwarz, “The DELPHI Experiment Control System”, Proceedings of the first IEEE International Conference on Engineering of Complex Computer Control Systems, Florida, Nov. 1995.
7 Ph. Charpentier, et al., “Architecture and Performance of the DELPHI Data Acquisition and Control System”, Proceedings of the International Conference on Computing in High Energy Physics ’91, Tsukuba, Japan, March 1991.
8 M. Dönszelmann, C. Gaspar, “A Configurable MOTIF Interface for the Delphi Experiment at LEP”, Proceedings of the Second Annual International MOTIF Users Meeting, Washington DC, USA, 1992.
9 B. Petersen, T. Wijnen, “L3 Cosmics Data Acquisition”, CERN, L3 Cosmics Internal Note.
10 J. Barlow et al., “Run Control in MODEL: The State Manager”, IEEE trans. nucl. sci. 36, 1989.
11 B. Franek, “On-line Experiment Control System for the BaBar Detector at PEP-II at SLAC”, Proceedings of the International Conference on Computing in High Energy Physics, Chicago, 1998.
12 W. Salter, “Selecting and Evaluating SCADA Systems for the Slow Controls of the CERN LHC Detectors”, Proceedings of the 7th International Conference on Accelerators and Large Experimental Physics Control Systems, Trieste, 1999.
13 M. Caprini, P.-Y. Duval, R. Jones, S. Kolos, “Information System for the Atlas DAQ Prototype -1”, CERN, Atlas DAQ Technical Note 31.