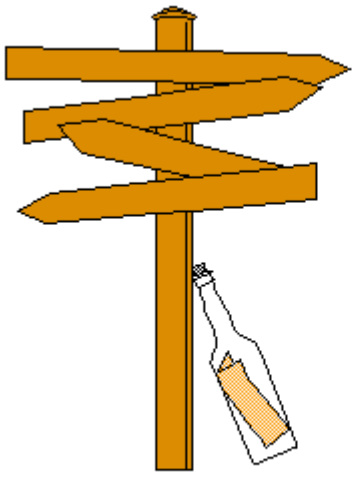


DIM

Distributed Information Management System

C.Gaspar, ECP Division, CERN

Abstract

DIM is a communication system for distributed / mixed environments, it provides a network transparent inter-process communication layer. This manual provides an overview of DIM's functionality and guidance on its usage.

Revision/Update Information: Version 10.0, April 2002

Contents (summary)

- **Introduction**
- **Availability and Tools**
- **User Manual (html) / User Manual (ps)**
- **Download DIM for: Windows NT or UNIX**

Other related documents

- **DIM - A Distributed Information Management System for the Delphi experiment at CERN (ps) Presented at: IEEE Eight Conference REAL TIME '93 on Computer Applications in Nuclear, Particle and Plasma Physics (Vancouver, June 8-11 1993)**
- **A Highly Distributed Control System for a Large Scale Experiment (ps) Presented at: 13th IFAC workshop on Distributed Computer Control Systems - DCCS'95 (Toulouse, Sep 27-29 1995)**

- **The Delphi experiment Control System (ps) Presented at: 1st IEEE Conference on the Engineering of Complex Computer Control Systems (Ft. Lauderdale, Florida, Nov 6-10 1995)**
- **Controlling a Large Physics Experiment; a Communication Issue (ps) Published in: IFAC Journal - Control Engineering Practice (Vol.4 Num. 2, Feb 1996)**
- **DIM, a Portable, Light Weight Package for Information Publishing, Data Transfer and Inter-process Communication (ps) Presented at: International Conference on Computing in High Energy and Nuclear Physics (Padova, Italy, 1-11 February 2000)**

DIM
Distributed Information Management System

Introduction

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.

Services are normally requested by the client only once (at startup) and they are subsequently automaticaly updated by the server either at regular time intervals or whenever the conditions change (according to the type of service requested by the client).

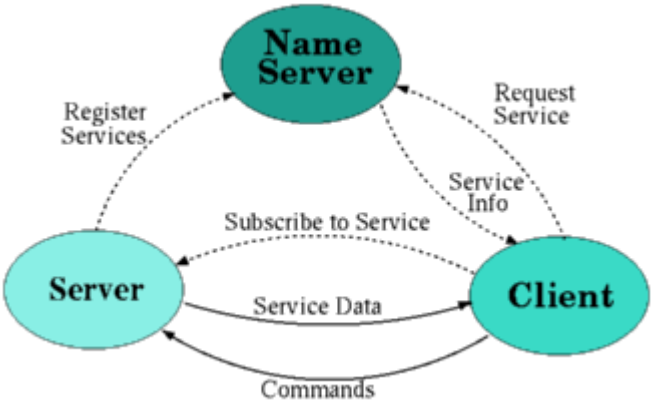
The client updating mechanism can be of two types, either by executing a callback routine or by updating a client buffer with the new set of data, or both. In fact this last type works as if the clients maintain a copy of the server's data in cache, the cache coherence being assured by the server.

In order to allow for 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 (normally once, at startup).

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 parameters.

The name server keeps an up-to-date directory of all the servers and services available in the system. The Figure shows how DIM components (Servers, Clients and the Name Server) interact.



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 an easy recovery, it also 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.

DIM
Distributed Information Management System

Availability and Tools

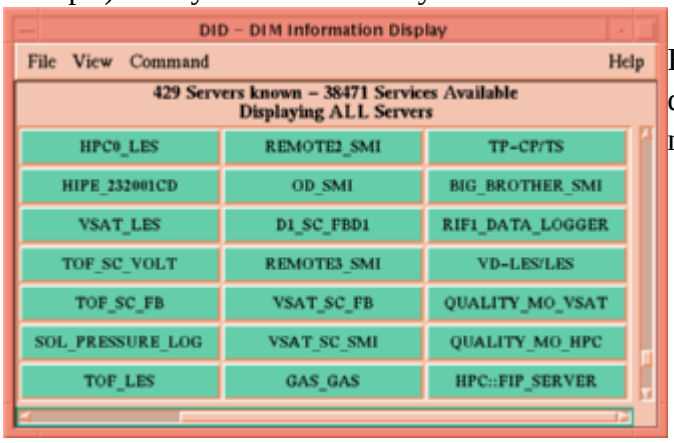
The DIM system is currently available for mixed platform environments comprising the operating systems : VMS, Unix, Linux, Windows NT and the real time OSs: OS9, LynxOs and VxWorks. It uses as network support TCP/IP.

The differences in data representation (e.g.: byte ordering, floating point format, data alignment and data type sizes) over different machines are automaticaly (transparently) negotiated between the server, the client and the name server.

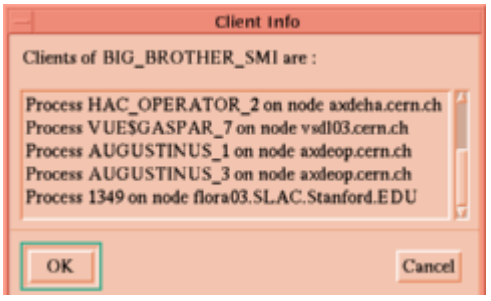
All DIM functionality is available as server and client libraries providing C++, C and Fortran callable interfaces.

The behaviour of complex distributed applications can be very difficult to understand without the help of a dedicated tool. The DIM System provides a tool - DID, the Distributed Information Display - that allows the visualization of the processes involved in the application as shown in the Figure.

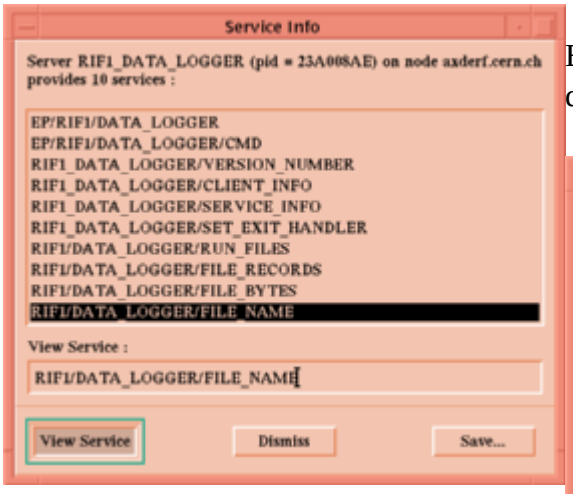
DID allows the visualisation of the Servers composing the application: they can all be desplayed (as in the example) or they can be selected by the machine where they are running or by the Services they provide.



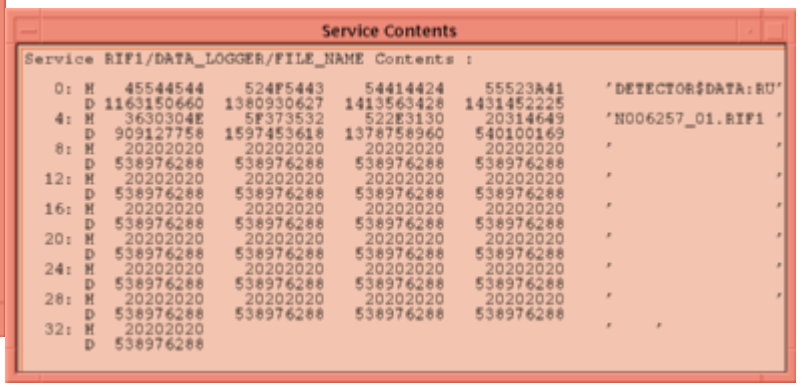
For each Server the list of current clients can be displayed providing information on their process names or PIDs and machine names:



And the list of Services provided can be visualised (together with information about the server itself like its PID and the machine where it is running):



By Selecting one of the services its contents can be displayed:



DIM
Distributed Information Management System

User Manual

This chapter describes the DIM interface , i.e. the functionality provided by the server and client libraries as well as some examples on how they can be used.

Two different interfaces are available:

- **The C++ Interface**
- **The C and Fortran Interfaces**

---

library : DIM

Authors : C. Gaspar and Ph. Charpentier

Version : 1.0

Please have a look at a few simple examples first.

Server Classes

- Class : DimServer
- Class : DimService
- Class : DimCommand
- Class : DimRpc

Client Classes

- Class : DimClient
- Class : DimInfo
- Class : DimCurrentInfo
- Class : DimRpcInfo
- Class : DimStampedInfo
- Class : DimUpdatedInfo

Utility Classes

- Class : DimBrowser
- Class : DimTimer

Last update : 02/11/99 15:53:02 by *MkHelp 1.1.0*

A Few Simple Examples

The Server publishes one service (integer value)

```
#include <dis.hxx>

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

The Client subscribes to the service, requesting it to be updated every 5 seconds. If the service is not available the value "-1" should be received instead.

```
#include <dic.hxx>

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

The Server publishes one service (integer value) and updates it from time to time (when it changes):

```
#include <dis.hxx>

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

The Client subscribes to one service and executes a method when the service gets updated (when the server executes updateService()):

```
#include <dic.hxx>

class RunNumber : public DimInfo
{
  void infoHandler()
  {
    cout << "Run Number " << getInt() << endl;
  }
  public :
    RunNumber() : DimInfo("DELPHI/RUN_NUMBER",-1) {};
};

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

The Server can receive commands (a string):

```
#include <dis.hxx>

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

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

The client sends a command:

```
#include <dic.hxx>
```

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

Class: DimServer

Library: DIM

Author: **Ph. Charpentier**

Version: **v1.0** **Update :** Thu Feb 11 15:52:46 1999

Description :

To be used by DIM Servers - Implements mainly static methods related to the Server.
DimServer::start(<server name>) should be called after DimServices and/or DimCommands have been created, in order to register them within the DIM Name Server.
DimServices created after "start" has been called will be automatically registered.
The class DimServer can be inherited by user classes wishing to handle multiple DimCommands
in the same class. Please refer to Usage for examples on the use of DimServer.

Constructors :

- *public* **DimServer** ( ) ;

Public Function :

- static void **start** (char * name) ; Starts the Server i.e. Publishes all declared services and starts serving Client Requests. Services declared afterwards may or may not be automatically registered within the Name Server depending on the **autoStart** flag.
- static void **autoStartOn** ( ) ; Instructs the server to immediately register any new services after the first **start(**name**)** to the name server without waiting for a new **start(**name**)**. This is the default.
- static void **autoStartOff** ( ) ; Instructs the server to wait for a new **start(**name**)** in order to register newly declared services.
- static int **getClientId** ( ) ; Get Current Client Identifier, returns a non-zero id only if the server is currently serving a Client, i.e inside the virtual methods : commandHandler and clientExitHandler.
- static char ***getClientName** ( ) ; Get Current Client Name, returns a non-zero name of the form "task@node" only if the server is currently serving a Client.
- static **addClientExitHandler** (DimServer *handler) ; Instruct the Server to execute the virtual method clientExitHandler whenever "special" clients die. A client can declare himself has a "special" client by executing setExitHandler or the Server can choose a client to be one of the "special" clients by using the two next methods.
- static **setClientExitHandler** (int clientId) ; Sets a client exit handler for the client which clientId was returned from getClient().
- static **clearClientExitHandler** (int clientId) ; Cancels the exit handler for clientId.
- virtual void **clientExitHandler** ( ) ; To be overloaded by the user. Gets called when "special" clients die.

Virtual methods for Command Handling

- virtual void **commandHandler** ( ) ; The Server can be a DimCommand handler when multiple commands are to be treated as in the example.
- DimCommand* **getCommand** ( ) ; Can be used inside "commandHandler" in order to return a pointer to the command beeing handled.

Usage :

The DimServer class implements Server functions - the most important is "start".
Example:

```
DimService runNumber("DELPHI/RUN_NUMBER",123);
DimServer::start("DelphiServer");
```

DimServer can be used as a base class when the user class wishes to handle DimCommands using
Handlers.
Example:

```
class Handler : public DimServer
 {
   DimCommand cmd;
   int val;
  void commandHandler( ) { val = cmd.getInt( ); };
public:
   Handler( ) :
      cmd("TEST/CMD", "I", this) {val = 0; };
 };
```

The DimServer can also be used to set up exit handlers to be executed when "special" clients die.
The installation of a exit handler for a client can be done in two ways:

- The client decides it is one of the "special" ones. Example: The Server should die when the client dies.

Server Part:

```
class ClientHandler : public DimServer
 {
public:
   ClientHandler() {
      DimServer::addClientExitHandler(this);
      DimServer::start("TheServer"); };
  void clientExitHandler( ) {
      cout << "Client " << getClientName() << " died" << endl;
       exit(0);
    };
 };
```

Client Part:

```
main()
{
    DimClient::setExitHandler("TheServer");
    ...
}
```

- The server decides a client is "special" - example: The Server wants to be "released" when the client that "allocated" it dies.

```
class Allocation : public Server
{
  DimCommand *allocate;
  int allocationState;
public:
  Allocation( ) {
    allocate = new DimCommand("SRV/ALLOCATE","I",this);
    DimServer::addClientExitHandler(this);
    DimServer::start("SRV"};
  void commandHandler( ); {
    if ( getInt( ) == 1) {              // Client Allocated
      allocateState = 1;
      DimServer::setClientExitHandler(DimServer::getClientId( )); // Set the exitHandler for this client
    }
    else {                              // Client Released
      allocateState = 0;
      DimServer::clearClientExitHandler(DimServer::getClientId( )); // Clear the exitHandler for this client
    }
  void clientExitHandler( ) {           // Client died  (while Allocated) - Release
    cout << "Client " << getClientName() << " died" << endl;
    allocateState = 0;
  };
};
```

Last update : 02/15/99 17:40:42 by *MkHelp 1.1.0*

Class: DimService

Library: DIM

Author: **Ph. Charpentier**

Version: **v1.0** **Update :** Thu Feb 11 15:52:46 1999

Description :

To be used by DIM Servers - Implements Service creation.
DimService constructors add a DIM Service to the list of known services.
The DimServices will only be registered (published) and served after "Starting" the DimServer.
Please refer to Usage for examples of DimServices.

Constructors :

First parameter is the Service Name. Since the clients might request the service at regular time intervals the value parameter should be the address of a variable that has a live time similar to the DimService itself.

- *public* **DimService** (char * name, int & value) ;
- *public* **DimService** (char * name, float & value) ;
- *public* **DimService** (char * name, double & value) ;
- *public* **DimService** (char * name, char * value) ;
- *public* **DimService** (char * name, char * format, void * value, int size) ; The format parameter specifies the contents of the structure in the form T:N[;T:N]*[;T] where T is the item type: (I)nteger, (C)arachter,(L)ong,(S)hort,(F)loat,(D)ouble and N is the number of such items. The type alone at the end means all following items are of the same type. Example: "I:3;F:2;C" means 3 Integers, 2 Floats and Characters until the end. The format parameter is used for communicating between different platforms.

Destructors :

- *public* ~ **DimService** ( ) ;

Public Functions :

Update Methods
All update methods return the number of clients updated.

- int **updateService** ( ) ; The current value of the same variable
- int **updateService** (int &value) ; The variable changed address (and size)
- int **updateService** (float &value) ;
- int **updateService** (double &value) ;
- int **updateService** (char * string) ;

- int **updateService** (void *structure, int size);

Selective Update Methods

Like the Update methods but do not update all clients.

- Update only the current client (if inside a Command callback) - clientIds = 0

- Update the list of clients specified by an array of client Ids (terminated by 0).

Client Ids can be obtained by DimServer::getClientId() inside a Command callback

- int **selectiveUpdateService** (int *clientIds) ;
- int **selectiveUpdateService** (int &value, int *clientIds) ;
- int **selectiveUpdateService** (float &value, int *clientIds) ;
- int **selectiveUpdateService** (double &value, int *clientIds) ;
- int **selectiveUpdateService** (char * string, int *clientIds) ;
- int **selectiveUpdateService** (void *structure, int size, int *clientIds);

Methods for Time Stamping and Quality flag (only meaningfull when the client requests it
by using DimStampedInfo). The Server time stamps the service and sets the quality flag to zero
when this feature is requested by the client but the following methods allow the user to override the default.
These settings will be sent to the client with the next update of service data.

- void **setQuality** (int quality) ; The quality flag - zero by default
- void **setTimestamp** (int secs, int millisecs) ; The time stamp - Set by default to the current time.

Usage :

DimService creation example:

- The Server Updates the service (i.e. sends it to the client) whenever the contents change

```
int run = 123;
DimService runNumber("DELPHI/RUN_NUMBER",run);
DimServer::start("DelphiServer");
 ...
run++;
runNumber.updateService();
```

- The Server Does not explicitly Update the service: it will get updated (i.e. sent to the client) with the current contents at the time interval specified by each client.

```
float trigger_rate;
DimService runNumber("DELPHI/TRIGGER_RATE",trigger_rate);
DimServer::start("DelphiServer");
```

```
 ...
trigger_rate = calculate_trigger_rate();
```

Last update : 02/11/99 15:53:02 by *MkHelp 1.1.0*

Class: DimCommand

Library: DIM

Author: **Ph. Charpentier**

Version: **v1.0** **Update :**Thu Feb 11 15:52:46 1999

Description :

To be used by DIM Servers - Implements Command creation.
DimCommand constructors add a DIM Command to the list of known services.
The DimCommand will only be registered (published) and served after "Starting" the DimServer.
DimCommands can be treated either by a command handler (asynchronously) or be queued and retrieved to the user when requested, please refer to Usage for examples.

Constructors :

- *public* **DimCommand** (char * name, char * format) ; Create a DimCommand. Format parameter specifies the contents of the expected data in the form T:N[;T:N]*[;T] where T is the item type: (I)nteger, (C)arachter,(L)ong,(S)hort,(F)loat,(D)ouble and N is the number of such items. The type alone at the end means all following items are of the same type. Example: "I:3;F:2;C" means 3 Integers, 2 Floats and Characters until the end. The format parameter is used for communicating between different platforms.
- *public* **DimCommand** (char * name, char * format, DimCommandHandler * handler); Format parameter as above. A handler can be specified if a different class is handling the command (specially useful if a class is handling multiple commands, as in the example).

Destructors :

- *public* ~ **DimCommand** ( ) ;

Public Functions :

Generic

- char* **getName** ( ) ; Get the command name.

Without a commandHandler

- int **getNext** ( ) ; Get Next queued command. returns 1 if there is a command queued, 0 otherwise. After a getNext is issued and until the next one the methods in "Access to the Command Data" can be used in order to retrieve the command data.

With a Handler

- virtual void **commandHandler** ( ) ; To be overloaded if the user specified a handler. The methods in "Access to the Command Data" can be used inside the Handler in order to retrieve the command data:

Access to the Command Data

- int **getInt** ( ) ; For an integer command.
- float **getFloat** ( ) ; A float.
- double **getDouble** ( ) ; A double.
- char* **getString** ( ) ; A character string.
- void* **getData** ( ) ; A structure or a vector.
- int **getSize** ( ) ; The size of the command data.

To find out which command it is (if same handler for multiple commands)

- DimCommand * **getCommand** ( ) ; The command received (See example)

Usage :

DimCommands can be created in three different ways
Examples:

- Without a handler:

```
DimCommand runCmnd("DELPHI/RUNCMND","C");
DimServer::start("DelphiServer");
 ...
while(1)
{
   sleep(5);
  while(runCmnd.getNext()) {
      //treat the command
      char *cmnd = runCmnd.getString();
       ...
}
```

- Using a commandHandler

```
class RunCmnd : public DimCommand // In order to inherit "commandHandler"
{
  // Overloaded method commandHandler called whenever commands arrive,
  void commandHandler()
  { //treat the command
     cout << "command " << getString() << " received" << endl;
   }
public:
  // The constructor initializes the DimCommand
  RunCmnd() : DimCommand("/DELPHI/RUNCMND", "C") {};
};
```

- Using a CommandHandler for multiple Commands

```
class RunCmnds : public DimServer // In order to inherit "commandHandler"
 {
   DimCommand *runNumber;
   DimCommand *runType;
   // Overloaded method commandHandler called whenever commands arrive,
   void comandHandler( )
    {
      if(getCommand( ) == runNumber)
       {
         int run = getInt( );
         // treat runNumber command
       }
       else
       {
         char *type = getString( );
         // treat runType command
       }
    }
public:
   // The constructor creates the Commands
   RunCmnds( )
    {
      runNumber = new DimCommand("/DELPHI/RUN_NUMBER/CMD", "I", this);
      runType = new DimCommand("/DELPHI/RUN_TYPE/CMD", "C", this);
    }
};
```

Last update : 02/19/99 12:49:03 by *MkHelp 1.1.0*

Class: DimRpc

Library: DIM

Author: **C. Gaspar**

Version: **v1.0** **Update :**Thu Feb 11 15:52:46 1999

Description :

To be used by DIM Servers - Implements RPC Service creation.
DimRpc constructors add a DIM Service of RPC type to the list of known services.
The DimRpcs as DimServices will only be registered (published) and served after "Starting" the DimServer.
Please refer to Usage for an example of DimRpc creation.

Constructors :

First parameter is the Service Name.

- *public* **DimRpc** (char * name, char * format_in, char *format_out) ; The format parameters specifies the contents of the data to be received (format_in) and to be sent in response (format_out) in the form T:N[;T:N]*[;T] where T is the item type: (I)nteger, (C)haracter, (L)ong, (S)hort, (F)loat, (D)ouble and N is the number of such items. The type alone at the end means all following items are of the same type. Example: "I:3;F:2;C" means 3 Integers, 2 Floats and Characters until the end. The format parameters are used for communicating between different platforms.

Destructors :

- *public* ~ **DimRpc** ( ) ;

Public Functions :

Handler: Gets Called when an RPC is requested by a Client (DimRpcInfo)

- virtual void **rpcHandler** ( ) ; // Has to be provided by the user.

Get **Methods: To be used inside rpcHandler in order to get the data received from the client**

- int **getInt** ( ) ; // Get an Integer
- float **getFloat** ( ) ;
- double **getDouble** ( );
- char ***getString** ( );
- int **getSize** ( ); //Get the size of the data (for complex types)
- void ***getData** ( ); //Get the data (for complex types)

Set **Methods: To be used inside rpcHandler in order to send the result back to the client**

- int **setData** (int &value) ; //Send back an Integer
- int **setData** (float &value) ;
- int **setData** (double &value) ;
- int **setData** (char * string) ;
- int **setData** (void *data, int size); //Send back complex data

Usage :

DimRpc example:

```
#include "dis.hxx"

class RpcInt : public DimRpc
 {

    void rpcHandler()
     {
        int val;
        val = getInt();
         val++;
        setData(val);
     }
public:
   RpcInt(char *name): DimRpc(name,"I","I") { };
 };

main()
 {
    RpcInt testRpcInt("TESTRPC/INT");

   DimServer::start("TESTRPC");
    while(1)
         pause();
 }
```

Last update : 02/11/99 15:53:02 by *MkHelp 1.1.0*

Class: DimClient

Library: DIM

Author: **C. Gaspar**

Version: **v1.0** **Update :** Thu Feb 11 15:52:46 1999

Description :

To be used by DIM clients - implements static methods relating to the Client
Mainly Sending Commands to Servers.
The methods sendCommand(...) will wait for the command to be actualy sent to the Server and return a completion code of :
1 - if it was successfully sent.
0 - if it couldn't be delivered.
The exception beeing: if the user calls this method inside a Handler (infoHandler, serviceHandler, commandHandler and clientExitHandler) the command will be sent but without waiting for reception and the return code is not reliable (in order to avoid deadlocks).

The methods sendCommandNB(...) are Non Blocking, i.e. they do not wait for the command to be sent.

They should be used inside callback routines to avoid deadlocks

Public Functions :

- static int **sendCommand** (char * name, int data) ; //Send Integer as Command
- static int **sendCommand** (char * name, float data) ; //Send Float as Command
- static int **sendCommand** (char * name, double data) ; //Send Double
- static int **sendCommand** (char * name, char * data) ; // Send a Character String
- static int **sendCommand** (char * name, void * data, int datasize) ; //Send a structure or a vector
- static int **sendCommandNB** (char * name, int data) ; //Send Integer as Command
- static int **sendCommandNB** (char * name, float data) ; //Send Float as Command
- static int **sendCommandNB** (char * name, double data) ; //Send Double
- static int **sendCommandNB** (char * name, char * data) ; // Send a Character String
- static int **sendCommandNB** (char * name, void * data, int datasize) ; //Send a structure or a vector
- static **setExitHandler** (char * serverName) ; //Inform the Server that this client would like the Server to execute an ExitHandler when it dies.

Methods for Service Info Handling

- virtual void **infoHandler** ( ) ; The method to be overloaded by the user, DimClient can be used as a base class when the user wishes to handle multiple DimInfo Services using the same handler. Example.
- DimInfo* **getInfo** ( ) ; Can be used inside "infoHandler" in order to return a pointer to the DimInfo service currently beeing handled.

Usage :

setExitHandler Example

sendCommand Example:

```
main()
{
  DimClient::sendComamnd("SRV/ALLOCATE", 1);
    ...
  DimClient::sendCommand("SRV/ALLOCATE", 0);
}
```

Last update : 02/19/99 12:49:03 by *MkHelp 1.1.0*

Class: DimInfo

Library: DIM

Author: **C. Gaspar**

Version: **v1.0** **Update :**Thu Feb 11 15:52:46 1999

Descrition :

To be used by DIM clients - implements DIM service subscription and reception
DimInfo constructors subscribe to DIM services.
DimInfo Services can be requested to be updated when the contents change and/or at regular time intervals.
The services are received at startup, when the server updates them or after a timeout limit if the parameter "time" is specified.
This gives the following possibilities:

- Receive the service at startup and then
- Receive the service only when the server updates it
- no "time" parameter
- Receive the service at regular time intervals (if the server doesn't update it) or
- Receive the service at regular intervals and also when the server updates it
- some "time" parameter
- Receive the service when the server updates it, but also after a timeout (to make sure the server is alive)
- longish "time" parameter

Note: The parameter "time" is sent to the server. It is the server that updates the services even on the time basis

Please refer to Usage for examples.

Constructors :

Constructors for Services updated only by the server

- *public* **DimInfo** (char * name, int nolink) ; Integer Service
- *public* **DimInfo** (char * name, float nolink) ; Float Service
- *public* **DimInfo** (char * name, double nolink) ; Double Service
- *public* **DimInfo** (char * name, char * nolink) ; Character String Service
- *public* **DimInfo** (char * name, void * nolink, int nolinksize) ; Structure (mix types) Service

Constructors for Services updated (also) on a time basis

- *public* **DimInfo** (char * name, int time, int nolink) ; Integer Service
- *public* **DimInfo** (char * name, int time, float nolink) ; Float Service
- *public* **DimInfo** (char * name, int time, double nolink) ; Double Service

- *public* **DimInfo** (char * name, int time, char * nolink) ; Character String Service
- *public* **DimInfo** (char * name, int time, void * nolink, int nolinksize) ; Structure (mix types) Service

Constructors for Services with a Handler for multiple Services (without time)

- *public* **DimInfo** (char * name, int nolink, DimInfoHandler * handler) ; Integer Service
- *public* **DimInfo** (char * name, float nolink, DimInfoHandler * handler) ; Float Service
- *public* **DimInfo** (char * name, double nolink, DimInfoHandler * handler) ; Double Service
- *public* **DimInfo** (char * name, char * nolink, DimInfoHandler * handler) ; Character String Service
- *public* **DimInfo** (char * name, void * nolink, int nolinksize, DimInfoHandler * handler) ; Structure (mix types) Service

Constructors for Services with a Handler for multiple Services (with time)

- *public* **DimInfo** (char * name, int time, int nolink, DimInfoHandler * handler) ; Integer Service
- *public* **DimInfo** (char * name, int time, float nolink, DimInfoHandler * handler) ; Float Service
- *public* **DimInfo** (char * name, int time, double nolink, DimInfoHandler * handler) ; Double Service
- *public* **DimInfo** (char * name, int time, char * nolink, DimInfoHandler * handler) ; Character String Service
- *public* **DimInfo** (char * name, int time, void * nolink, int nolinksize, DimInfoHandler * handler) ; Structure (mix types) Service

Destructors :

- *public* ~ **DimInfo** ( ) ; Dim Service Destructor

Public Functions :

- char* **getName** ( ) ; Get the Service Name
- void* **getData** ( ) ; Get the Service Contents
- int **getInt** ( ) ; Get Integer Service Contents
- float **getFloat** ( ) ; Get Float Service Contents
- double **getDouble** ( ) ; Get Double Service Contents
- char* **getString** ( ) ; Get String Service Contents
- int **getSize** ( ) ; Get the Service Size.
- virtual void **infoHandler** ( ) ; To be overloaded if Handler specified

Usage :

DIM Client Services (DimInfo) can be used in several ways:

- The user variable is of type DimInfo

Example : **The Service is received when the Server updates it**

```
main()
{
    DimInfo runNumber("DELPHI/RUN_NUMBER", -1);
    while(1)
    {
        ...
        cout << runNumber.getInt() << endl;
    }
}
```

Example : **The Service is received when the Server updates it and every 30 seconds otherwise**

```
main()
{
    DimInfo runNumber("DELPHI/RUN_NUMBER", 30, -1);
    while(1)
    {
        ...
        cout << runNumber.getInt() << endl;
    }
}
```

Example : **The Service is received at regular intervals of 5 seconds (the Server does not explicitly update it)**

```
main()
{
    DimInfo triggerRate("DELPHI/TRIGGER_RATE", 5, -1.0);
    while(1)
    {
        ...
        cout << triggerRate.getFloat() << endl;
    }
}
```

- Inheritance: The user class inherits from DimInfo

Example:

```
class RunNumber: public DimInfo
```

```
{
public:
     // subscribe to service with handler
    RunNumber(): DimInfo("DELPHI/RUN_NUMBER", -1) {}
    void infoHandler( ) {cout << getInt() << endl;} // update handler
};
```

- A class subscribes to many DimServices with only one handler
  Example:

```
class RumVars : public DimClient // inheritance necessary because a handler is to be used
{
    DimInfo runNumber;
    DimInfo runType;
public:
    RunVars:
        runNumber("DELPHI/RUN_NUMBER", -1, this), // subscribe with handler
        runType("DELPHI/RUN_TYPE", "not available", this), // subscribe with handler
         {}
    void infoHandler( ) {
        DimInfo *curr = getInfo() // get current DimInfo address
        if(curr == &runNumber) { int run = curr->getInt( ) };
        else { char *type = curr->getString( ) };
         //etc.
     }
};
```

Last update : 02/11/99 15:53:02 by *MkHelp 1.1.0*

Class: DimCurrentInfo

Library: DIM

Author: **C. Gaspar**

Version: **v1.0** **Update :**Thu Feb 11 15:52:46 1999

Description :

To be used by DIM clients - implements blocking DIM service reception
DimCurrentInfo constructors subscribe to DIM services.
Required Parameters: the Service Name and the value to receive when service not available
The methods get() wait for the Service to arrive if it has not arrived yet. Once the service contents
are received the Service is discarded (The client disconnects from the server).

Constructors :

- *public* **DimCurrentInfo** (char * name, int nolink) ; Integer Service
- *public* **DimCurrentInfo** (char * name, float nolink) ; Float Service
- *public* **DimCurrentInfo** (char * name, double nolink) ; Double Service
- *public* **DimCurrentInfo** (char * name, char * nolink) ; String Service
- *public* **DimCurrentInfo** (char * name, void * nolink, int nolinksize) ; Structure Service

Destructors :

- *public* ~ **DimCurrentInfo** ( ) ;

Public Functions :

- int **getInt** ( ) ; Get Integer Service contents
- float **getFloat** ( ) ; Get Float Service Contents
- double **getDouble** ( ) ; get Double Service Contents
- char* **getString** ( ) ; Get String Service contents
- void* **getData** ( ) ; Get the Service contents
- int **getSize** ( ) ; Get Service contents Size

Usage :

Example:

```
DimCurrentInfo runNumber("DELPHI/RUN_NUMBER",-1);
cout << runNumber.getInt() << endl;
```

Last update : 02/11/99 17:42:12 by *MkHelp 1.1.0*

Class: DimRpcInfo

Library: DIM

Author: **C. Gaspar**

Version: **v1.0** **Update :** Thu Feb 11 15:52:46 1999

Description :

To be used by DIM clients - implements RPC subscription (blocking or non-blocking)
Required Parameters: the Service Name and the value (nolink) to receive when RPC service not available
The methods set() and get() are used to send and receive data to and from the server.
Please refer to Usage for examples on how to use DimRpcInfo.

Constructors :

- *public* **DimRpcInfo** (char * name, int nolink) ; Integer reception expected
- *public* **DimRpcInfo** (char * name, float nolink) ;
- *public* **DimRpcInfo** (char * name, double nolink) ;
- *public* **DimRpcInfo** (char * name, char * nolink) ;
- *public* **DimRpcInfo** (char * name, void * nolink, int nolinksize) ; Complex data

Destructors :

- *public* ~ **DimRpcInfo** ( ) ;

Public Functions :

Set **Methods: To be used to send the RPC request (data) to the Server**

- int **setData** (int &value) ; //Send an Integer
- int **setData** (float &value) ;
- int **setData** (double &value) ;
- int **setData** (char * string) ;

  int **setData** (void *data, int size); //Send complex data

Handler: If the user implements it, it gets Called when an RPC answer is received (DimRpc)

- virtual void **rpcInfoHandler** ( ) ; // Can be provided by the user for non-blocking reception of RPCs

Get **Methods: To get the data received from the server: Can be used for blocking reception of RPCs or**
inside rpcInfoHandler in order to get the data received from the server

- int **getInt** ( ) ; // Get an Integer
- float **getFloat** ( ) ;
- double **getDouble** ( );
- char ***getString** ( );
- int **getSize** ( ); //Get the size of the data (for complex types)
- void ***getData** ( ); //Get the data (for complex types)

Usage :

DimRpcInfo non-blocking example:

```
#include "dic.hxx"

class Rpc : public DimRpcInfo
{
   void rpcInfoHandler() {
      int valin;
      valin = getInt();
      cout << "Callback RPC Received : " << valin << endl;
}
public:
   Rpc(char *name) : DimRpcInfo(name, -1) { };
};

main()
{
    int rpcCBValue = 0;
    Rpc rpcCB("TESTRPC/INT");

    while(1)
    {
        rpcCB.setData(rpcCBValue);
        sleep(5);
    }
}
```

DimRpcInfo blocking example:

```
#include "dic.hxx"

main()
{
    int rpcValue = 0;
    DimRpcInfo rpc("TESTRPC/INT",-1);

    while(1)
    {
        rpc.setData(rpcValue);
        rpcValue = rpc.getInt();
        sleep(5);
```

```
        }
    }
```

Last update : 02/11/99 15:53:02 by *MkHelp 1.1.0*

Class: DimStampedInfo

Library: DIM

Author: **C. Gaspar**

Version: **v1.0** **Update :** Wed Apr 05 15:52:46 2000

Description :

To be used by DIM clients - implements DIM service subscription and reception for time stamped (and quality flaged) services

DimStampedInfo requests the server to send together with the service data a quality flag and a timestamp

DimStampedInfo inherits its behaviour from DimInfo, please refer to it for detailed description.

Please refer to Usage for examples.

Constructors (as for DimInfo):

Constructors for Services updated only by the server

- *public* **DimStampedInfo** (char * name, int nolink) ; Integer Service
- *public* **DimStampedInfo** (char * name, float nolink) ; Float Service
- *public* **DimStampedInfo** (char * name, double nolink) ; Double Service
- *public* **DimStampedInfo** (char * name, char * nolink) ; Character String Service
- *public* **DimStampedInfo** (char * name, void * nolink, int nolinksize) ; Structure (mix types) Service

Constructors for Services updated (also) on a time basis

- *public* **DimStampedInfo** (char * name, int time, int nolink) ; Integer Service
- *public* **DimStampedInfo** (char * name, int time, float nolink) ; Float Service
- *public* **DimStampedInfo** (char * name, int time, double nolink) ; Double Service
- *public* **DimStampedInfo** (char * name, int time, char * nolink) ; Character String Service
- *public* **DimStampedInfo** (char * name, int time, void * nolink, int nolinksize) ; Structure (mix types) Service

Constructors for Services with a Handler for multiple Services (without time)

- *public* **DimStampedInfo** (char * name, int nolink, DimInfoHandler * handler) ; Integer Service
- *public* **DimStampedInfo** (char * name, float nolink, DimInfoHandler * handler) ; Float Service
- *public* **DimStampedInfo** (char * name, double nolink, DimInfoHandler * handler) ; Double Service
- *public* **DimStampedInfo** (char * name, char * nolink, DimInfoHandler * handler) ; Character String Service

- *public* **DimStampedInfo** (char * name, void * nolink, int nolinksize, DimInfoHandler * handler) ; Structure (mix types) Service

Constructors for Services with a Handler for multiple Services (with time)

- *public* **DimStampedInfo** (char * name, int time, int nolink, DimInfoHandler * handler) ; Integer Service
- *public* **DimStampedInfo** (char * name, int time, float nolink, DimInfoHandler * handler) ; Float Service
- *public* **DimStampedInfo** (char * name, int time, double nolink, DimInfoHandler * handler) ; Double Service
- *public* **DimStampedInfo** (char * name, int time, char * nolink, DimInfoHandler * handler) ; Character String Service
- *public* **DimStampedInfo** (char * name, int time, void * nolink, int nolinksize, DimInfoHandler * handler) ; Structure (mix types) Service

Destructors :

- *public* ~ **DimStampedInfo** ( ) ; Dim Service Destructor

Public Functions :

Inherited from DimInfo:

- void* **getData** ( ) ; Get the Service Contents
- int **getInt** ( ) ; Get Integer Service Contents
- float **getFloat** ( ) ; Get Float Service Contents
- double **getDouble** ( ) ; Get Double Service Contents
- char* **getString** ( ) ; Get String Service Contents
- int **getSize** ( ) ; Get the Service Size.
- virtual void **infoHandler** ( ) ; To be overloaded if Handler specified

New Functionality (The information retrieved by the following methods is set by the sever by using the class DimService)**:**

- int **getQuality** ( ) ; Get the Quality flag received with the service (set by the server)
- int **getTimestamp** ( ) ; Get the time stamp (in seconds since midnight, Jan 1st 1970)
- int **getTimestampMillisecs** ( ) ; Get the milliseconds

Usage :

DimStampedInfo can be used like DimInfo:

Example :

```
class RunNumber: public DimStampedInfo
{
```

```
public:
        // subscribe to service with handler
       RunNumber(): DimStampedInfo("DELPHI/RUN_NUMBER", -1) {}
        // update handler
       void infoHandler( )
       {
             time_t time;
             time = getTimestamp();
             cout << " Received: " << getInt() << " Time Stamped: " << ctime(&time) <<
                   "Quality: "<< getQuality() << endl;
       }
};
```

Last update : 02/11/99 15:53:02 by *MkHelp 1.1.0*

Class: DimUpdatedInfo

Library: DIM

Author: **C. Gaspar**

Version: **v1.0** **Update :** Wed Apr 05 15:52:46 2000

Descrition :

To be used by DIM clients - implements DIM service subscription and reception
DimUpdatedInfo constructors subscribe to DIM services.
DimUpdatedInfo Services can be requested to be updated when the contents change and/or at regular time intervals.

The difference between DimUpdatedInfo and DimInfo is that services are not received at Startup, i.e. the first time a client subscribes to a service.
The services are received when the server updates them or after a timeout limit if the parameter "time" is specified.

DimUpdatedInfo inherits its behaviour from DimInfo, please refer to it for detailed description.

Please refer to Usage for examples.

Constructors (as for DimInfo):

Constructors for Services updated only by the server

- *public* **DimUpdatedInfo** (char * name, int nolink) ; Integer Service
- *public* **DimUpdatedInfo** (char * name, float nolink) ; Float Service
- *public* **DimUpdatedInfo** (char * name, double nolink) ; Double Service
- *public* **DimUpdatedInfo** (char * name, char * nolink) ; Character String Service
- *public* **DimUpdatedInfo** (char * name, void * nolink, int nolinksize) ; Structure (mix types) Service

Constructors for Services updated (also) on a time basis

- *public* **DimUpdatedInfo** (char * name, int time, int nolink) ; Integer Service
- *public* **DimUpdatedInfo** (char * name, int time, float nolink) ; Float Service
- *public* **DimUpdatedInfo** (char * name, int time, double nolink) ; Double Service
- *public* **DimUpdatedInfo** (char * name, int time, char * nolink) ; Character String Service
- *public* **DimUpdatedInfo** (char * name, int time, void * nolink, int nolinksize) ; Structure (mix types) Service

Constructors for Services with a Handler for multiple Services (without time)

- *public* **DimUpdatedInfo** (char * name, int nolink, DimInfoHandler * handler) ; Integer Service
- *public* **DimUpdatedInfo** (char * name, float nolink, DimInfoHandler * handler) ; Float Service
- *public* **DimUpdatedInfo** (char * name, double nolink, DimInfoHandler * handler) ; Double Service
- *public* **DimUpdatedInfo** (char * name, char * nolink, DimInfoHandler * handler) ; Character String Service
- *public* **DimUpdatedInfo** (char * name, void * nolink, int nolinksize, DimInfoHandler * handler) ; Structure (mix types) Service

Constructors for Services with a Handler for multiple Services (with time)

- *public* **DimUpdatedInfo** (char * name, int time, int nolink, DimInfoHandler * handler) ; Integer Service
- *public* **DimUpdatedInfo** (char * name, int time, float nolink, DimInfoHandler * handler) ; Float Service
- *public* **DimUpdatedInfo** (char * name, int time, double nolink, DimInfoHandler * handler) ; Double Service
- *public* **DimUpdatedInfo** (char * name, int time, char * nolink, DimInfoHandler * handler) ; Character String Service
- *public* **DimUpdatedInfo** (char * name, int time, void * nolink, int nolinksize, DimInfoHandler * handler) ; Structure (mix types) Service

Destructors :

- *public* ~ **DimUpdatedInfo** ( ) ; Dim Service Destructor

Public Functions :

Inherited from DimInfo:

- void* **getData** ( ) ; Get the Service Contents
- int **getInt** ( ) ; Get Integer Service Contents
- float **getFloat** ( ) ; Get Float Service Contents
- double **getDouble** ( ) ; Get Double Service Contents
- char* **getString** ( ) ; Get String Service Contents
- int **getSize** ( ) ; Get the Service Size.
- virtual void **infoHandler** ( ) ; To be overloaded if Handler specified

Usage :

DimUpdatedInfo can be used exactly like DimInfo:

Class: DimBrowser

Library: DIM

Author: **C. Gaspar**

Version: **v1.0** **Update :** Thu Feb 11 15:52:46 1999

Description :

To be used by DIM processes - implements DIM environment browsing.
Allows Getting Information on Services, Servers and Clients available.
Please refer to Usage for examples

Public Functions :

- char ***getServices** (char * wildcardServiceName); // Check if a Service or Services (wildcards allowed) are available and if so what is their format. Returns an empty string if none available otherwise a string: <service name>|<service_format>[|CMD][|RPC]'\n'<service... (where CMD means the service is a command, RPC means it can be accessed by DimRpcInfo). The service_format is the one given to DimService, DimCommand or DimRpc
- char ***getServers** ( ); // Get the list of all servers available in the system. Format: <server_name>@<node_name>|<server...
- char ***getServerServices** (char * serverName); // Get the list of all services provided by a server. Format: <service_name>|<service_format>[|CMD][|RPC]'\n'<service... (where CMD means the service is a command, RPC means it can be accessed by DimRpcInfo). The service_format is the one given to DimService, DimCommand or DimRpc
- char ***getServerClients** (char * serverName); // Get the list of clients of a server. Format: <process>@<node_name>|<process...

Methods for Decoding the information of the above calls

- int **getNextService** (char * serviceName, char *format); // can be called after a getServices( ) call. It will return the type of the Service (DimSERVICE, DimCOMMAND or DimRPC) or 0 if no more services. If successfull it will return the service name and its format. The format is the one given to DimService, DimCommand or DimRpc
- int **getNextServer** (char *server, char *node); // can be called after a getServers( ) call. It will return 1 while there are servers in the list, 0 otherwise. If successfull it will return the server name and the node where it runs.
- int **getNextServerService** (char *serviceName, char *format); // can be called after a getServerServices( ) call. It will return the type of the Service (DimSERVICE, DimCOMMAND or DimRPC) or 0 if no more services. If successfull it will return the service name and its format. The format is the one given to DimService, DimCommand or DimRpc
- int **getNextServerClient** (char *clientName, char *node); // can be called after a getServerClients( ) call. It will return 1 while there are clients in the list, 0 otherwise. If successfull it will return the client name and the node where it runs.

Usage :

```
#include <dic.hxx>

main()
{
   DimBrowser dbr;
   char *server; *node, *service, *format;
   int type;

   dbr.getServers( );
   while(dbr.getNextServer(server, node))
   {
      cout << server << " @ " << node << endl;
      dbr.getServerServices(server);
      while(type = dbr.getNextServerService(service, format))
          ...
   }
}
```

Or :

```
#include <dic.hxx>

main()
{
   DimBrowser dbr;
   char *service; *format;
   int type;

   dbr.getServices("*DELPHI/MUON*" );
   while(type = dbr.getNextService(service, format))
   {
      cout << service << " -  " << format << endl;
          ...
   }
}
```

Last update : 02/19/99 12:49:03 by *MkHelp 1.1.0*

Class: DimTimer

Library: DIM

Author: **C. Gaspar**

Version: **v1.0** **Update :** Thu Feb 11 15:52:46 1999

Description :

Utility Class - Implements Asynchronous TimeOut Handling

Constructors :

DimTimer (int time) ; Start the Timer - parameter time is number of seconds
DimTimer ( ) ;

Public Functions :

- void **start** (int time) ; Start the Timer - parameter time is number of seconds
- void **stop** ( ) ; Stop the timer
- virtual void **timerHandler** ( ) = 0;

Usage :

Example:

```
class Tim : public DimTimer
{
public:
   Tim(int time)
   {
      start(time);
   };
   void timerHandler()
   {
      cout << "Timer Expired" << endl;
   }
};
```

Or:

```
class Tim : public DimTimer
{
public:
   Tim(int time) : DimTimer(time) {};
   void timerHandler()
   {
      cout << "Timer Expired" << endl;
```

```
    }
};
```

Last update : 02/19/99 12:49:03 by *MkHelp 1.1.0*

DIM
Distributed Information Management System

C (and Fortran) Interface

This chapter describes the routines provided by the server and client libraries as well as some examples on how they can be used.

The Server library implements the DIM server functionality. In order to become a DIM server a process can use the following routines:

- **DIS_ADD_SERVICE** - Declare each Service it can provide.
- **DIS_ADD_CMND** - Declare each Command Services (if any) it is willing to execute.
- **DIS_START_SERVING** - Send the list of provided Services and Commands to the Name Server and start serving client requests. The library takes care of handling all client requests and updating the Service data according to the mechanism specified by the client.
- **DIS_UPDATE_SERVICE** - A server can force an update of the Service when it knows the information contained in the service has changed.

The Client library implements the DIM client functionality. In order to become a DIM client a process can use the following routines:

- **DIC_INFO_SERVICE** - Request a service providing the update type (at regular time intervals or when the information changes) and the update mechanism (either a callback routine or a buffer update). The library will take care of contacting the Name Server, find out which server provides the service, connect to the server, request the service and update the service according to the specified mechanism whenever the service data arrives.
- **DIC_CMND_SERVICE** - Request the execution of a command. The library will take care of contacting the Name Server, find out which server provides the command, connect to the server and tell it to execute the command.
- **DIC_CMND_CALLBACK** - Same as above but a callback will be executed to report success or failure.
- **DIC_RELEASE_SERVICE** - Release a service if it becames unnecessary.

- Server Library (DIS)
- Server examples
- Client Library (DIC)
- Client examples
- Timer Utility Library (DTQ)

DIM
Distributed Information Management System

4.1 Server Library (DIS)

Detailed description of the routines contained in the Server library :

dis_add_service

Add an information service to the list of provided services.

Format

unsigned int dis_add_service (name, description, address, size, user_routine, tag)

Arguments

char *name;

Service name, this name should be used by the client when requesting the service.

char *description;

Service description string, the contents of the service can be described in the form "T:N;T:N;T" where T is the data type : C(char), L(ong), S(hort), D(ouble), F(loat) or X(tra long) and N is the number of items of that type. A data type alone at the end of string means all following items are of type T. The description string is not necessary and can be replaced by 0 when running in a non mixed environment.

int *address;

Service address, if the data provided by this service is stored in memory (global section, array, etc) and is to be sent as it is to the client, it's address should be given here.

int size;

Service size, the size in bytes of the data to be passed to the client if the previous parameter has been specified.

void *user_routine;

The address of a routine to be executed when it's time to send the data to a client. This routine will provide the data, the parameters of this routine will be specified later in this document.

int tag;

A parameter to be sent to the user_routine in order to identify the service.

Returns

unsigned int service_id;

The service identifier, to be used when (if) updating or removing the service (by calling dis_update_service or dis_remove_service).

Description

This routine has to be called once for every service provided by the server. There are two ways of providing the service : specifying the address and size of the data to be sent to the client or specifying the address of a routine that will provide the data.

---

dis_add_cmnd

Add a command service to the list of provided services.

Format

unsigned int dis_add_cmnd (name, description, cmnd_user_routine, tag)

Arguments

char *name;

Service name, same name used by client when requesting the execution of a command.

char *description;

Service description string, the contents of the service can be described in the form "T:N;T:N;T" where T is the data type : C(char), L(ong), S(hort), D(ouble), F(loat) or X(tra long) and N is the number of items of that type. A data type alone at the end of string means all following items are of type T. The description string is not necessary and can be replaced by 0 when running in a non mixed environment.

void *cmnd_user_routine;

The address of the routine to be executed when a command request is received from a client, the parameters of this routine will be specified later in this document. This parameter is optional, if it is specified as zero, the request will be queued and can be recovered later by calling the routine dis_get_next_cmnd.

int tag;

A parameter to be passed to the cmnd_user_routine or to dis_get_next_cmnd in order to identify the command service.

Returns

unsigned int service_id;

The service identifier, to be used when (if) removing the service (by calling dis_remove_service).

Description

This routine has to be called once for every command service provided by the server.

---

dis_start_serving

Start handling client requests.

Format

int dis_start_serving (task_name)

Arguments

char *task_name;

Task name, a string identifying the server, used by the Name server to recognize the server as well as for monitoring and debugging purposes.

Returns

int return_code

Returns 1 if the list of services and commands were succesfully sent to the Name Server, 0 otherwise.

Description

This routine will register within the name_server all the services declared with dis_add_service and dis_add_cmnd.

It will set up the server so that it will handle client requests.

It will also set up all the mechanisms so that when a service has to be sent to a client the user_routine that provides that service will be called or the data will be taken from the address and size provided in dis_add_service and sent to the client.

---

dis_update_service

Report the change of service contents to interested clients.

Format

int dis_update_service (unsigned int service_id)

Arguments

unsigned int service_id;

The service_id returned by dis_add_service.

Returns

int n_clients;

Returns the number of updated clients.

Description

This routine should be called by the server program when there is a change for a given service (when possible).

This routine will check whether there are clients interested in this service and if they have requested to receive the data uppon change (monitored service) the data will be sent, either by calling the data provider routine or by sending the specified data buffer.

---

dis_remove_service

Remove a Service from the list of provided services.

Format

int dis_remove_service (unsigned int service_id)

Arguments

unsigned int service_id

The service_id returned by dis_add_service.

Returns

int return_code

Returns 1 if the Service was succefully removed.

Description

This routine can be called by the server program when a service stops beeing provided.

This routine will inform the Name server and all the clients using this service that it is no longer provided. The service can from then on be provided by another server.

---

dis_stop_serving

Stop Serving DIM Services.

Format

void dis_stop_serving ()

Description

This routine can be called by the server program in order to stop beeing a DIM Server.

---

dis_get_next_cmnd

Get a command from the list of waiting command requests.

Format

int dis_get_next_cmnd (tag, buffer, size)

Arguments

int *tag;

This parameter returns the tag given to dis_add_cmnd, it allows the identification of the command.

int *buffer;

A buffer address where the command request is to be copied to.

int *size;

On entry this parameter contains the size of the buffer, on exit it contains the real size of the command request.

Returns

int return_code

Returns 0 if no command request is waiting for execution, -1 if the command request doesn't fit on the specified buffer (truncated) and 1 if the command as been copied successfully into the user buffer.

Description

This routine has to be called by the user if no cmnd_user_routine address has been specified in dis_add_cmnd. It allows the user to get and execute the commands requested by a client.

---

user_routine

Routine written by the user in order to provide a service.

Format

void user_routine (tag, address, size)

Arguments

int *tag;

The parameter that identifies the service, the tag given to dis_add_service. Passed by reference for FORTRAN compatibility.

int **address;

Should return the address of the data to be sent to the client.

int *size;

Should return the size in bytes of the data to be sent to the client

Description

This routine should be declared in dis_add_service if not using the address, size parameters, it will be called whenever the service has to be sent to the client.

This routine should provide the necessary gathering or computing of data and store it in a buffer in order to be sent as a service to a client.

---

cmnd_user_routine

Routine written by the user in order to execute a command when a command request is received from a client.

Format

void cmnd_user_routine (tag, address, size)

Arguments

int *tag;

The parameter that identifies the command, the tag given to dis_add_cmnd. Passed by reference for FORTRAN compatibility.

int *address;

The address of the buffer containing the command.

int *size;

The size in bytes of the command data. Passed by reference for FORTRAN compatibility.

Description

This routine should be declared in dis_add_cmnd for immediate execution on reception of a command request.

DIM
Distributed Information Management System

4.3 Client Library (DIC)

Detailed description of the routines contained in the Client library :

---

dic_info_service

Request an information service from a server

Format

unsigned int dic_info_service (name, type, timeout, address, size, user_routine, tag, fill_address, fill_size)

Arguments

char *name;

Service name, same name used by server when declaring the service.

int type;

Type of service, constants defined are: ONCE_ONLY, TIMED or MONITORED.

int timeout;

For a TIMED service "timeout" indicates the time interval the server should use to send new data, for ONCE_ONLY or MONITORED services it indicates the time after which the service is considered to have failed.

int *address;

Address of the buffer where to store the data when the service returns from the server. This parameter can be set to 0 and only the callback routine will be executed.

int size;

The size in bytes of the previous buffer (if specified).

void *user_routine;

The address of a routine to be executed when new data is returned from the server, the parameters of this routine will be specified later in this document. Can be set to 0 if no callback routine is necessary.

int tag;

A parameter to be sent to the user_routine in order to identify the service that has completed.

int *fill_address;

Address of a buffer containing data to be stored in the service buffer or passed to the callback routine in case the service doesn't succeed.

int fill_size;

The size in bytes of the previous buffer.

Returns

unsigned int service_id;

The service identifier, to be used when (if) releasing the service.

Description

This routine first contacts the name server in order to get the address of the server where the requested service is available, and then sends the request directly to the server. When the service arrives from the server the client buffer is filled with the data and/or the user_routine is executed.

After a timeout (timeout parameter for ONCE_ONLY and MONITORED services and 2*timeout parameter for TIMED services) the service is considered failed, the information in fill_address is copied into the client buffer and/or the user_routine is called.

If the server is not responding the client recontacts the name server and the name server will wake up the client as soon as the server is up.

---

dic_cmnd_service

Request the execution of a command by a server

Format

int dic_cmnd_service (name, address, size)

Arguments

char *name;

Service name, same name used by server when declaring the command service.

int *address;

Address of the buffer containing the command data.

int size;

The size in bytes of the previous buffer.

Returns

int return_code;

Returns1 if the command was successfully requested.

Description

This routine requests the execution of a command by a server, address and size contain the command parameters. This implies contacting the name server (if the command in not know already) and then contacting directly the server. This routine does not report back the completion of the command since these operations are only scheduled and may complete asynchronously.

If the server is not responding the command is discarded (in order to avoid having it sent later when it might not be desired anymore).

This routine allows clients (user interfaces for ex.) to send commands to be executed by the servers.

---

dic_cmnd_callback

Request the execution of a command by a server and registers a completion callback

---

Format

int dic_cmnd_service (name, address, size, cmd_callback, tag)

Arguments

char *name;

Service name, same name used by server when declaring the command service.

int *address;

Address of the buffer containing the command data.

int size;

The size in bytes of the previous buffer.

void *cmd_callback;

The address of a routine to be executed when the command completes, in order to report the success or failure of the operation. The parameters of this routine will be specified later in this document.

int tag;

A parameter to be sent to the cmd_callback in order to identify the command that has completed.

Returns

int return_code;

Returns1 if the command was successfully requested.

Description

This routine requests the execution of a command by a server, address and size contain the command parameters. This implies contacting the name server (if the command in not know already) and then contacting directly the server. This routine does not report back the completion of the command since these operations are only scheduled and may complete asynchronously. The cmd_callback routine will be executed on completion.

If the server is not responding the command is discarded (in order to avoid having it sent later when it might not be desired anymore).

This routine allows clients (user interfaces for ex.) to send commands to be executed by the servers

---

dic_release_service

Called by a client when a service is not needed anymore

Format

void dic_release_service (service_id)

Arguments

unsigned service_id;

The service_id returned by dic_info service.

Description

This routine tells the server not to update this service anymore and destroys all the references to it in the client.

---

user_routine

Routine written by the user, called when new data arrives from a server. The user routine is called with three parameters, please read carefully the parameter description.

Format

void user_routine (tag, buffer, size)

Arguments

int *tag;

A parameter in order to identify the service, the tag given to dic_info_service. Passed by reference for FORTRAN compatibility

int *buffer;

This parameter contains the address of the data received from the server. If the user has supplied "address" when calling dic_info_service buffer points to "address", otherwise it points to a temporarily allocated buffer where the data received has been stored (in this case this buffer address is valid only during the execution of this routine, if the data contained in this buffer has to be used later it's the responsibility of the user to copy it to another buffer).

int *size;

The size in bytes of the data actually sent by the server. Passed by reference

Description

This routine is called on reception of data from a server, it can be used for ex. by user_interfaces in order to update the screen whith the new set of data.

---

cmd_callback

Routine written by the user, called on command completion. This routine is called with two parameters, please read carefully the parameter description.

Format

void user_routine (tag, ret_code)

Arguments

int *tag;

A parameter in order to identify the command, the tag given to dic_cmd_callback. Passed by reference for FORTRAN compatibility

int *ret_code;

The return code: 1 if the command was successfully sent, i.e., the server was found and the command successfully written out; 0 otherwise. Passed by reference

Description

This routine is called on command completion to check or wait for successful delivery.

DIM
Distributed Information Management System

4.2 Server examples

Examples

#1

The following example implements a C server. This server provides two services which contain the same data, but the way of providing the data differs. The server can also execute one command and on its reception it updates one of the services.

```
#include <dis.h>

int buffer[] = { 0,1,2,3,4,5,6,7,8,9 };
int service_id;

void build_service(tag, address, size)
int *tag;
int **address;
int *size;
{
   *address = buffer;
   *size = sizeof(buffer);
}

void execute_cmnd(tag, cmnd_buffer, size)
int *tag;
char *cmnd_buffer;
int *size;
{
   if(*tag == 1)
   {
      printf("SERV_CMND: Command %s received\n",cmnd_buffer);
      dis_update_service(service_id);
   }
}

main()
{

   dis_add_service("SERV_BY_BUFFER", "L", buffer, 40, 0, 0);
```

```
    service_id = dis_add_service("SERV_BY_ROUTINE", "L", 0, 0,
       build_service, 0);
    dis_add_cmnd("SERV_CMND", 0, execute_cmnd, 1);
    dis_start_serving("DIS_TEST");
    while(1)
    {
       sleep(10);
    }
}
```

The following example implements a FORTRAN server. This server provides one information service and one command service. On command reception it updates the information service.

```
      program test_server
      implicit none

      common/test_ser/service_id
      integer*4 service_id

      integer*4 dis_add_cmnd, dis_add_service
      integer*4 dis_start_serving

      external do_cmnd
      character*80 str

      str = 'Server Answer'
      service_id = dis_add_service('TEST/INFO','C',%ref(str),80,%val(0), 0)
      call dis_add_cmnd('TEST/CMND','C',do_cmnd, 0)

      call dis_start_serving('DIS_TEST')
      call sys$hiber()
      end

      subroutine do_cmnd(tag, buf, size)
      implicit none
      integer*4 tag, size
      integer*4 buf
      character*80 str
      integer*4 dis_convert_str
      common/test_ser/service_id
      integer*4 service_id

C The routine dis_convert_str converts a C string into a Fortran string
      call dis_convert_str(buf, str)
      write(6,'(A,A)') ' Server : Received ',str
      call dis_update_service(service_id)

      end
```

DIM
Distributed Information Management System

4.4 Client examples

Examples

#1

The following example implements a C client. The client requests two services, one TIMED (every ten seconds) and one MONITORED. It also tells the server to execute a command.

```
#include <dic.h>

int buffer[10];
int no_link = -1;
int version;

buff_received(tag, bufferp, size)
int *tag, *size;
char *bufferp;
{
int i;

  if(bufferp[0] == -1)
     printf("Service SERV_BY_BUFFER not available\n");
  else
  {
     printf("received service SERV_BY_BUFFER\n\t");
     for(i=0;i<10;i++)
        printf("%d ",bufferp[i]);
     printf("\n");
  }
  printf("\n");
}

serv_received(tag, address, size)
int *tag, *address, *size;
{
int i;

  if(*address == -1)
     printf("Service SERV_BY_ROUTINE not available\n");
```

```
  else
  {
      printf("received service SERV_BY_ROUTINE\n\t");
      for(i=0;i<10;i++)
          printf("%d ",buffer[i]);
      printf("\n");
  }
  printf("\n");
}

main()
{

  dic_info_service("SERV_BY_BUFFER", TIMED, 10,
      buffer, 40, buff_received, 0, &no_link,4);
  dic_info_service("SERV_BY_ROUTINE", MONITORED, 0,
      0, 0, serv_received, 0, &no_link,4);
  while(1)
  {
      dic_cmnd_service("SERV_CMND", "UPDATE", 7);
      sleep(5);
  }
}
```

The following example implements a FORTRAN client. This client requests a MONITORED service and tell the server to execute a command.

```
      program test_client
      implicit none

      common/test_for/buff
      character*80 buff
      include 'delphi$online:[communications.dim]dic.inc'

      integer*4 dic_cmnd_service
      character*80 str
      external recv_rout
      external dic_info_service

      buff = 'empty'
      str = 'Command'
      call dic_info_service('TEST/INFO',MONITORED,0,%ref(buff),80,recv_rout,
     ,                      0,%val(0),0)
      do while(.TRUE.)
        call dic_cmnd_service('TEST/CMND',%ref(str),17)
        call lib$wait(10.)
      enddo
      end
```

```
subroutine recv_rout(tag)
implicit none
integer*4 tag
common/test_for/buff
character*80 buff
write(6,'(A,A)') ' Client : Received ',buff
end
```

DIM
Distributed Information Management System

4.5 Timer Library (DTQ)

Detailed description of the routines contained in the Timer Utility Library :

dtq_start_timer

Start a TimeOut request

Format

void dtq_start_timer (time, user_routine, tag)

Arguments

int time;

The time in seconds after which the user_routine should be called.

void *user_routine;

The address of a routine to be executed when the timer expires.

int tag;

A parameter to be sent to the user_routine in order to identify the Timeout request. Tag is also used as Request Identifier in order to stop the timer.

Description

This routine starts a timer. When the timer expires the user_routine will be called.

dtq_stop_timer

Stop (cancel) a TimeOut request

Format

void dtq_stop_timer (tag)

Arguments

int tag;

The parameter given to dtq_start_timer.

Description

This routine stops the timer. The user_routine will no longer be called.

---

dtq_sleep

Sleep for a number of seconds

Format

void dtq_sleep(time)

Arguments

int time;

The time in seconds the process should be suspended for.

Description

This routine suspends the process for the given number of seconds. In Unix the sleep function exits when the process receives a signal, since DIM uses signals, processes using the UNIX call directly will exit several times and it would be difficult to compute the actual time slept. dtq_sleep can be used instead it only exits when the sleep time elapses.

---

user_routine

Routine written by the user, called when the timer started by dtq_start_timer expires.

Format

void user_routine (tag)

Arguments

int tag;

The parameter that identifies the request. The tag given to dtq_start_timer.

Description

This routine is called when a timer started by dtq_start_timer expires.

Instructions for downloading, installing and running DIM on **Windows NT/2000**

Latest version Release Notes

1. Download

   - get the file dim.zip
   - unzip (extract) dim.zip onto a DIM folder

2. Installation

   - Insert .../DIM/bin in the "path" (Control Panel, System, Environment tab)
   - set the environment variable DIM_DNS_NODE to <node name>, <node name> is the complete name of the node where the DIM Name Server (Dns) will run, ex. hpplus003.cern.ch. (again Control Panel, System, Environment tab).

3. Running

   - If you want to run Dns in the PC: In the Command Prompt type "dns".
   - The DIM display tool for windows is DIM/bin/DID
   - From then on you can start DIM servers and clients, some example ones are available, you can start them by:
     - test_server <server name>
     - test_client <client name> <server name>
   - In order to make your own servers and/or clients you have to link them with dim.lib in DIM/bin (also dim.dll is there). The DIM/src/examples directory contains the source code of the examples and devstudio settings to compile and link them (dim.dsw) are available in DIM/Visual.

Instructions for downloading, installing and running DIM on **UNIX**

Latest version Release Notes

1. Download
    - get the file dim.zip
    - extract it onto a "dim" folder using: **unzip -a**
2. Installation
    - cd dim
    - (please use tcsh for the following to work)
    - setenv OS <unix flavour>, <unix flavour> can be: HP-UX, AIX, Solaris, SunOS, OSF1, Linux, LynxOS
    - source .setup
    - gmake [options] all
        - possible options:
            - CPP=yes or no (default = yes) : Create also the DIM C++ class library
            - THREADS=yes or no (default = yes) : Use the DIM multithreaded version
            - CC=cc, gcc, etc. (default = cc native compiler) : Define which C compiler to use
            - CXX=CC, g++, etc. (default = CC native compiler) : Define which C++ compiler to use
            - FLAGS=... : Extra user flags to pass to the compiler
            - EXTRALIBS=... : Extra user libraries to pass to the linker
3. Running
    - setenv DIM_DNS_NODE <node name>, <node name> is the complete name of the node where the DIM Name Server (Dns) will run ex. hpplus003.cern.ch
    - Dns & ! Starts the Name Server
    - Did & !Starts the DIM Display
    - From then on you can start DIM servers and clients (some example servers and clients source code is available in dim/src/examples) you can start them by:
        - Test_server <server name>

    - Test_client <client name> <server name>
- In order to make your own servers and/or clients you have to link them with …dim/<OS type>/libdim.a and with -lpthread (you can use makefile_examples in the top directory as an example).

DIM version 10.4 Release Notes

```
Notes 1 and 2 for Unix Users only
NOTE 1: In order to "make" DIM two environment variables should be set:
        OS = one of {HP-UX, AIX, OSF1, Solaris, SunOS, LynxOS, Linux}
        DIMDIR = the path name of DIM's top level directory
        The user should then go to DIM's top level directory and do:
        > source .setup
        > gmake all
        Or, if there is no support for C++ on the machine:
        > gmake CPP=no all

NOTE 2: The Name Server (Dns), DID, servers and clients (if running in
        background) should be started whith the output redirected to a
        logfile ex:
                Dns </dev/null >& dns.log &

NOTE 3: The Version Number service provided by servers is now set to 1004
        (version 10.04).

25/4/2002
Changes for version 10.0:
      - All source files are now common to Windows and Unix flavours
        (Linux included). Directories src/win and src/unix no longer
        necessary.
      - Fixed hopefully all compiler warnings (especially on Solaris 8).
      - In order to avoid potential deadlocks all tcpip writes (dna_write)
        are done by a separate thread (the timer thread, via a special
        "immediate" queue). Except service updates and sending commands
        (dna_write_nowait) since they are not blocking and to preserve
        backward behaviour compatibility.
      - Optimized servers, clients and the name servers for large number
        of services
      - Modified error messages to be more explicit

01/5/2002
Changes/Bug Fixes for Version 10.1:
      - Fixed the DimRpc class, it would hang sometimes.
      - Fixed a problem in the "immediate" timer handler (too slow)
      - Added "const" to service names in DimInfo and DimService methods
      - changed print_date_time to dim_print_date_time and made it
        available to users
      - Open_dns didn't always return the correct value (DID wouldn't
        reconnect to on Dns restart)
      - Did (on Linux) now shows services in alphabetical order

06/5/2002
Changes/Bug Fixes for Version 10.2:
      - Fixed dtq.c and tcpip.c for Linux, dim_wait() would not always
        return when required
      - The distribution kit now also contains the shareable version of
        the DIM library for Linux - libdim.so
        The makefiles use the shareable version for creating Dns, Did
        and the examples (.setup adds dim/linux to LD_LIBRARY_PATH)

27/5/2002
```

```
Changes/Bug Fixes for Version 10.3:
      - Changed dim include files not to include "windows.h" under
        windows. This was causing a conflict with Gaudi.
        (Had to change "DIM semaphores" from macros to subroutines)

18/7/2002
Changes/Bug Fixes for Version 10.4:
      - Two consecutive client requests for the same service would not
        implement the "stamped" flag properly (dic.c)
      - The DimBrowser class would not retreive service names containing
        the character "@". Fixed.
      - Re-fixed a bug that would make servers crash when clients exited
        while servers were updating a service (dis.c).
      - dtq_start_timer() did not always wait the requested amount of time.
      - Did (Linux version) now also prints timestamp and quality flag
        when Viewing service contents.

Please check the Manual for more information at:
    http://www.cern.ch/dim
```