MCS
My Customizable Server

Giorgio Calderone
IFC - INAF Palermo, Italy

Luciano Nicastro
IASF - INAF Bologna, Italy

June 11, 2011
Ver. 0.3.3-alpha3
http://ross.iasfbo.inaf.it/mcs

This page intentionally left blank.

Contents

1 Introduction

MCS is a collection of high level C++ classes and functions designed to easily implement the following kind of applications:

- multi-thread applications;
- network applications (through TCP);
- database (MySQL) applications;
- information servers;
- VOTable[1] and/or FITS access.

Aside from these tasks **MCS** provides several other features that can be used to solve common problems when developing applications in C++. The greatest advantage in using **MCS** is that it exploits the C++ capabilities in developing high-level tasks but does not require the developer to deal with networking, threading, database code, or in general low-level code. Furthermore some repetitive tasks like converting data between different types or creating and manipulating dynamically allocated data structures can be easily accomplished using the **MCS** classes.

MCS has been developed on the GNU/Linux platform and it is released under the GPL license. It can be freely downloaded from the site http://ross.iasfbo.inaf.it/mcs. This site contains all news, updates, documentation and downloadable software packages. The site is still under development, so check for updates.

The **MCS** project was developed at IFC-INAF (Palermo, Italy) by Giorgio Calderone and Luciano Nicastro (now at IASF-INAF, Bologna).

1.1 Motivations

Information services can be separated in two classes: those in which the information produced are addressed to humans, and those in which they are meant for the use by other software applications. In the former case there is a quite standardized way to develop such an information service, essentially based upon a web server, a database server, a scripting language and HTML pages. In the latter case instead there is no such standardization, and this is one of the reasons why **MCS** (My Customizable Server) was implemented. Thanks to its communication protocol indeed **MCS** can be used to easily develop information services that can be accessed from other software applications. Furthermore **MCS** provides several mechanisms to customize the server behaviour and integrate already existing applications. So **MCS** is an attempt to standardize the developing of these kind of applications.

[1]through the VOTPP package

By comparison **MCS** and its protocol are for software applications what a web server and HTTP are for the WWW: a simple way to access data.

Another reason that drove the development of **MCS** is the need in today's astronomical projects of computational systems capable to store and analyze large amounts of scientific data, to effectively share data with other research Institutes and to easily implement information services to present data for different purposes (scientific, maintenance, outreach, etc.). Due to the wide scenario of astronomical projects there isn't yet a standardized approach to implement the software needed to support all the requirements of a project. The new approach we propose here is the use of a unified model where all data are stored into the same database becoming available in different forms, to different users with different privileges.

Finally, **MCS** has been developed with the goal to simplify C++ developing through the use of high-level classes that relieve the developers from repetitive tasks. One of the most remarkable example of this higher level abstraction layer are the **Data**, **Record** and **RecordSet** classes (see Sect. 3.1).

MCS is the evolution of a software named SDAMS (SPOrt Data Archiving and Management System) that was implemented to support the SPOrt experiment, an Italian Space Agency (ASI) funded project which has been "frozen" because of the problems with the Columbus module on the International Space Station. At that time the approach used to build SDAMS seemed to be easily portable to other experiments so its features and usability have been generalized until it became the actual **MCS** . In fact it is now used to manage the data collected by the optical and infrared cameras ROSS/REMIR mounted on the robotic telescope REM at La Silla, Chile. Although **MCS** has been developed to build an information server to support an astronomical project, it is completely general purpose and can be used in the same manner in many other kind of projects.

1.2 MCS features

The **MCS** classes can be used to simplify application development in a variety of situations. Anyway we have to warn the reader about the fact that the **MCS** library is not supposed to be used as a system library replacement: if your application needs a low level control upon processes, threads, sockets, database connections, data conversions and memory allocation you simply have to use the system libraries. Only in this way indeed you can dispose of all the possibilities offered by the operative systems and the C++ language. However this kind of applications are just a minor part of all the applications, thus if your application needs to deal with this topics from a higher level then you can use the higher-level **MCS** classes instead of the low-level system routines.

The tasks that can be accomplished by the **MCS** classes can be subdivided in the following categories:

1.2.1 Multithreading applications

The **Thread** class can be used to create threads, that are separate path of execution running in the same process memory space. To create a separate thread you simply have to derive the **Thread** and implement the **run** method, which will became the body of execution of the separate thread. Synchronization between separate threads can be accomplished using the **Synchro** class. This class is also used as parent class for other classes such as **Record**, which acts as a shared resource between different threads.

1.2.2 Network applications

MCS provides several classes to develop network applications. The **HostInfo** class can be used to retrieve information about a network host, such as its IP address, while the **NetInterface** class can be used to retrieve information about the network interfaces installed on a computer. The **Socket** class is capable to open TCP sockets against remote hosts and send data through it. The **ServerSocket** can be used to open a server socket, that is a socket that waits on a specified port for incoming user connections. This class has been used to implement the **MCS** server, while **Socket** class has been used to implement the **Client** class which can connect to an **MCS** server.

1.2.3 The MCS data abstraction layer

The **Data**, **Record** and **RecordSet** classes can be used as a high-level abstraction layers upon data. Basically these classes offer a unique interface to access data from different sources (database, files, information sent by the **MCS** server, etc.). Furthermore they provide facilities to convert data between different formats, serialize data structures, implement thread-safe queues etc... These classes are intensively used to pass data between **MCS** classes. See Sect. 3.1.

1.2.4 Database access

The **DBConn** and **Query** classes provide the possibility to connect to a database server (actually only MySQL[2] is supported but other database server may be supported in the future) and execute queries on it. Data can be read from the database and written on it using the **MCS** data abstraction layer.

1.2.5 FITS file access

The **FITSReader** class can be used to read a FITS[3] file. Data will be read through the data abstraction layer.

[2]http://www.mysql.com
[3]http://heasarc.gsfc.nasa.gov/docs/heasarc/fits.html

1.2.6 Application server

An application server is an application that provides a service over the network. Typically a user connects to the service and issues a query that will be executed on the server, then the resulting data will be sent back to the user using a dedicated communication protocol. This process is very similar to the request of a web page. The **MCS** library provides an already working multithreaded TCP server like many others, but the **MCS** one has two interesting feature:

- the server behaviour is customizable (that's the reason for its name), that is the kind of queries that can be executed can be tuned on developer's needs.
- the communication protocol is perfectly integrated with the **MCS** data abstraction layer, so that data sent back from the server are in a well defined format (not like those of a web page) and thus it is very simple to implement an application that can access the service.

1.2.7 Other facilities

MCS has also a lot of minor features that help the developers to deal with common problems. Two of these facilities are:

- **CommandParser**: a class to parse a command line, with option and arguments support;
- **Conf**: a class to read and write configuration files, like the Windows `INI` files.

1.2.8 Facilities included in previous releases

A number of other facilities had been part of **MCS** in previous releases. Now these facilities had become separate projects because they don't depend upon **MCS** (except for **VOTPP**).

MyRO (My Record Oriented privilege system)
MyRO [4] is the name we use to refer to a technique used to implement a database privilege system on a per-record basis. Actually all database servers implement privilege systems based on tables or columns, that is if a user has grants to access a certain table (or a table column) he can access all records of that table (or that table column). **MyRO** lets you specify grants on a record level so a user can access only those records it is allowed to. A consequence of this is that different users reading the same table will see different records. The grants mechanism provided by **MyRO** is similar to that of a Unix file system, that is each record has an "owner" and a "group" to which the record belongs. Furthermore it has three sets of permissions (for the owner, the group, and all other users) that specify if

[4] http://ross.iasfbo.inaf.it/myro

that record can be read and/or written. The software components of **MyRO** are a Perl script used to perform administrative tasks, a C library and a set of MySQL functions. The process of protecting tables is completely transparent to the final user, that is once **MyRO** has been installed and configured by the database administrator users can access the database without even know that **MyRO** is working.

Astronomy related facilities
The most important Astronomy facility is the possibility to read VOTable[5] through the **VOTPP**[6] (VOTable C++ Parser) library. VOTable files can be read using the **Parser_Stream** or the **Parser_Tree** class, that is using the SAX model (read one node at a time) or the DOM model (read the entire file and build a browseable tree in memory). Note that **VOTPP** provides a distinct class for each possible VOTable node, so that the C++ code will be very close to the XML counterpart.

Another important Astronomy feature is **DIF** [7] (Dynamic Indexing Facility) which integrates the HTM[8] and HEALPix[9] indexing facilities inside database tables. This means that a query involving spherical coordinates (both in 2D or 3D environments) can take advantage from the database built-in indexing mechanisms to speed-up execution of a query.

1.3 Using MCS from other programming languages

MCS also provides an interface to use some of its classes from programming languages other than C++. The features exported through this interface are:

- the possibility to use the database related classes together with the data abstraction layer;
- the possibility to connect to an **MCS** server.

Due to the fact that not all languages support the object-oriented programming paradigm what is exported are just functions. An interesting feature of these functions is that they all have the same names in all languages.
Supported programming languages are:

- C;
- Fortran 77;
- PHP;

[5]http://www.ivoa.net/Documents/latest/VOT.html
[6]http://ross.iasfbo.inaf.it/votpp
[7]http://ross.iasfbo.inaf.it/dif
[8]http://www.sdss.jhu.edu/htm/
[9]http://healpix.jpl.nasa.gov

- Python;
- IDL.

Support for Java and Perl will be added soon.

1.4 License issues

MCS is distributed free of charge and according to the GNU General Public License. This means that you may copy this software and distribute it free of charge, as well as modify it provided that you retain this note and mention the authors name. The GNU General Public License comes with this package in a file named COPYING, for more information about this license visit `http://www.gnu.org`.

1.5 Some simple examples

The directory `share/examples` in the **MCS** package contains several examples that can be compiled and executed once **MCS** has been installed. To compile the examples simply execute `make` in that directory.

This section is aimed at giving a first sight at the code you should deal with if you decide to use **MCS** . Some aspects may appear unclear here but these are just examples. Check the following sections for further and more complete references.

In the first example (see List. 1) we'll show how to set and retrieve information from a **Data** object, which is the object upon which is built the entire data abstraction layer. Note in particular how the **Data** object takes care of all the necessary conversions between different data type.

In the second example (see List. 2) we'll execute a query on a database table, and print the result on standard output. We'll also show how to retrieve data through the data abstraction layer which, in this case, is implemented by the **Query** class (from line 15 to the end).

In the third example (see List. 3) we'll build the simplest **MCS** application server, without any customization but with all base features available (authentication support, base commands, database access, file download and upload, access to external programs).

Finally in our fourth example (see List. 4) we use the **Client** class to connect to a running **MCS** server (like the one in the previous example), issue some commands to upload and download a file and finally execute a query on the remote database and print the result on standard output. You should notice that lines from 24 to the end are exactly the same as those in listing 2, lines from 15 to the end. This is due to the fact that both the **Client** and **Query** classes implement the data abstraction layer, thus these classes provide the same interfaces to retrieve data, even if in one case data are being read from a database, and in the other data have been packed, sent through the network and then unpacked.

```
#include <mcs.hh>
using namespace mcs;

int main() {
  Data d(STRING, 20);

  //Assign an integer.
  d = 12;
  int i = d.ival();

  //Assigning a floating point number.
  d = 1.2;
  float f = d.fval();

  //Assign a date/time data.
  d = "2007-03-19 13:30";
  time_t t = d.tval();

  //Assign a string
  d = "My name is Giorgio";
  string s = d.sval();
}
```

Listing 1: Source of data1.cc

1.6 About this document

This document gives an overview of the **MCS** facilities and focuses on some of the basic topics of **MCS** . It is divided in several sections so that you can read only the information you need:

- **MCS installation and usage**: it provides information about the configuration of the package against all optional dependencies, the compilation and installation, furthermore it shows how to link against the **MCS** library;
- **Developer's manual**: it introduces the reader to some of the **MCS** basic concepts like the data abstraction layer, thread handling, database connections, etc., and focuses on various techniques to develop applications with **MCS** ;
- **The MCS server**: it shows how to implement, customize, execute and administrate an **MCS** server;
- **Client's manual**: it shows how to connect to an **MCS** server;
- **Using MCS with other programming languages**: describes all the available interfaces to connect to **MCS** through other languages;

```
#include <mcs.hh>
using namespace mcs;

int main() {
  //Connect to the database server
  DBConn db;
  db.connect("mcstest", "mcstest", "test","192.168.1.5");

  //Execute a query through the opened connection
  Query qry(&db);
  qry.prepare("SELECT * FROM mcstest");
  qry.execute();

  //Loop through the resulting record set
  while (! qry.eof()) {

    //Get a reference to current record
    Record& rec = qry.rec();

    //For each field print its value
    for (int i=0; i<rec.count(); i++)
      cout << rec[i].sval() << "\t";
    cout << endl;

    //Move to next record
    qry.setNext();
  }
}
```

Listing 2: Source of db1.cc

```
#include <mcs.hh>
using namespace mcs;

int main(int argc, char *argv[]) {
  //Start the server daemon
  mcsStart("simplest");
}
```

Listing 3: Source of server1.cc

```cpp
#include <mcs.hh>
using namespace mcs;

int main(int argc, char *argv[]) {
  //Connect to the MCS server.
  Client cli("./", "localhost", 6523);

  //Perform authentication.
  cli.login("mcstest", "mcstest", "test");

  //Upload a file
  cli.exec("PUT myfile");

  //Download a file
  cli.exec("GET myfile");

  //Execute a query on remote database
  cli.exec("QRY SELECT * FROM mcstest");

  //Retrieve the result set.
  cli.exec("QRES");

  //Loop through the resulting record set
  while (! cli.eof()) {

    //Get a reference to current record
    Record& rec = cli.rec();

    //For each field print its value
    for (int i=0; i<rec.count(); i++)
      cout << rec[i].sval() << "\t";
    cout << endl;

    //Move to next record
    cli.setNext();
  }
}
```

Listing 4: Source of client1.cc

- **libmcs reference**[10]: technical description of the **MCS** library implementation. It is produced using Doxygen[11].

The **MCS** documentation is *work in progress*, so you should always look for the last version. If you find any errors we'd appreciate if you could send us an email[12]. A users' mailing list is being prepared, so please contact us if you are a potential user. Thanks in advance.

[10]available only in HTML format at: http://ross.iasfbo.inaf.it/mcs

[11]http://www.stack.nl/∼dimitri/doxygen/

[12]Giorgio Calderone <gcalderone@ifc.inaf.it>, Luciano Nicastro <nicastro@iasfbo.inaf.it>

2 MCS installation and usage

The **MCS** software library is distributed in a `tar.gz` package. You can find the latest version at http://ross.iasfbo.inaf.it/mcs. To unpack the package simply issue the command:

```
tar xvzf mcs-x.y.z.tar.gz
```

where `x`, `y`, `z` are the version number (namely the first number is the major revision, the second number is the version, and third number is the subversion). A directory named `mcs-x.y.z` will be created containing all sources code as well as the documentation and the scripts needed to install **MCS** . Before installing **MCS** you should check that all mandatory dependencies are satisfied (see Sect. 2.1), then you must follow a three step procedure: Configure, Compile and Install.

2.1 Dependencies

The only mandatory packages required by **MCS** are:

- Perl (http://www.perl.com, version 5.8.5 or later);
- PCRE (http://www.pcre.org, version 6.4 or later);
- cURL (http://curl.haxx.se, version 7.12 or later).

Typically these packages are already installed in the system, if this is not the case you should install them before continuing. However there are a lot of **MCS** facilities which depend on other optional packages:

- MySQL (http://www.mysql.com/, version 5.1 or later): used to connect and manage a database, to handle client authentications and grants;
- Openssl (http://www.openssl.org/, version 0.9.7 or later): used to implement secure connections through sockets;
- CFITSIO (http://www.cfitsio.org/, version 0.9.7 or later): used to read data in FITS format;
- IDL (http://www.ittvis.com/idl/, version 5.6 or later): used to implement the IDL to MCS interface;
- PHP (http://www.php.net/, version 5.0.5 or later): used to implement the PHP to MCS interface;
- Python (http://www.python.org/, version 2.3 or later): used to implement the Python to MCS interface.

  You can enable or disable the compilation of each facility with the corresponding option of the configure script (see Sect. 2.2.1).

2.2 Installing MCS

2.2.1 Configure

Configuring **MCS** means checking your system for compatibilities, search for include files and libraries, and finally produce all necessary Makefiles needed to compile **MCS** . This is done automatically by the distributed `configure` script. Typically you can use this script without any option, as follows:

```
./configure
```

Anyway `configure` has a lot of options and switches (type `configure --help` for a list) to customize the compilation step. The options like `with-PACKAGE=PATH` can be used to specify the path where the `configure` script should search to find that package (default is `/usr/local`). Some of these options are specific to **MCS** :

- `--prefix=PATH`
  the directory under which all include files, libraries and other files relative to **MCS** will be installed, if this option is not used `usr/local` is assumed;
- `--enable-debug, --disable-debug`
  if enabled then the "-g" and "-O0" flags will be added to the compiler command line; these are needed to include debugging information in the library and to discard any optimization. By default this option is disabled;
- `--enable-all`
  by default all optional facilities are disabled, you can enable them with the relative options (discussed below), or you can enable all of them with this option. Note also that if you want to enable all option except, say, the interface to Python, you can simply use the following arguments: `configure --enable-all --disable-python`.
- `--with-pcre=PATH`
  if the PCRE package (which is mandatory) is not installed in a standard location (typically `/usr/local`) then you can provide the correct path with this option;
- `--with-curl=PATH`
  if the cURL package (which is mandatory) is not installed in a standard location (typically `/usr/local`) then you can provide the correct path with this option;
- `--enable-mysql, --disable-mysql, --with-mysql=PATH`
  enable or disable the compilation of MySQL facilities, by default this option is enabled;
- `--enable-openssl, --disable-openssl, --with-openssl=PATH`
  enable or disable the compilation of Openssl facilities, by default this option is disabled;

- `--enable-cfitsio, --disable-cfitsio, --with-cfitsio=PATH`
  enable or disable the compilation of CFITSIO facilities, by default this option is disabled;

- `--enable-idl, --disable-idl, --with-idl=PATH`
  enable or disable the compilation of IDL to **MCS** interface, by default this option is disabled;

- `--enable-php, --disable-php, --with-php=PATH`
  enable or disable the compilation of PHP to **MCS** interface, by default this option is disabled;

- `--enable-python, --disable-python, --with-python=PATH`
  enable or disable the compilation of Python to **MCS** interface, by default this option is disabled;

There are several other options and switches available, for further documentation see the `INSTALL` file.

2.2.2 Compile

To compile **MCS** , once the `configure` script has been correctly executed, simply issue the command:

```
make
```

If you got errors while compiling check Sect. 2.2.1 and the `INSTALL` file.

2.2.3 Install

If **MCS** has been correctly compiled you can install with the command:

```
make install
```

If your account doesn't have the permission to write in the path where it should be installed then you'll get an error. In this case you should login as "root" and retry.

2.3 Troubleshooting

In this section we show solution to the most common problems encountered while configuring and compiling **MCS** .

2.3.1 Problems with Python

To build the Python to **MCS** interface the Python include files are required. These are available in the Python source distribution or in the "Python devel" package of your distribution.

- The header `Python.h` has not been found: a symbolic link named `python` must point to the actual Python includes directory, and it must be located either in `/usr/include` or `/usr/local/include`.

2.4 Using the MCS library

Once you have installed the **MCS** library (see Sect. 2.2) you can use it in your C++ code as described here.

2.4.1 Include MCS header

All **MCS** classes and functions are defined inside the `mcs` namespace to avoid conflict with other libraries you may use. To use the **MCS** facilities you must include the **MCS** header in your C++ source code as follows:

```
#include <mcs.hh>
using namspace mcs;
```

To use the **MCS** library with other programming languages see Sect. 6.

2.4.2 Compile and link

The **MCS** package installs a script named `mcs-config` which can be used to retrieve several information regarding the installation itself, for example the location of the include files and of the libraries, as well as the list of the libraries to link to:

```
mcs-config --cflags
mcs-config --libs
```

For all other options type `mcs-config --help`. To compile and link a program (whose source file is, say, `myprog.cc`) against the **MCS** library you can issue the commands:

```
cpp ‘mcs-config --cflags‘ -c myprog.cc
cpp -o myprog myprog.o ‘mcs-config --libs‘
```

or with a single command:

```
cpp ‘mcs-config --cflags‘ myprog.cc -o myprog ‘mcs-config --libs‘
```

Note that the single quotes are "reversed single quotes" (ASCII decimal code 96).

3 Developer's manual

This section is aimed at showing to the reader some of the **MCS** basic concepts like the data abstraction layer, thread handling, database connections, etc., and focuses on various techniques to develop applications with **MCS** . We will often refer to classes, methods and function whose complete documentation can be found in the **MCS** library reference [13].

3.1 The data abstraction layer

The data abstraction layer is a set of classes aimed at providing a uniform access to data coming from different sources, and to easily manipulate and transmit those data. Namely the classes involved are **Data**, **Record** and **RecordSet**, with each one representing a different level of data hierarchy.

3.1.1 The `Data` class

The first level corresponds to the **Data** class, which can be used to store a single data, that is a number, a string, a date/time, a binary object, etc. The main feature of a **Data** object is its capability of performing conversions between different types of data, that is you can assign to a **Data** object different types of data, as in the following example:

```
//Create a Data object
Data mydata(STRING, 20);

//Assign an integer.
mydata = 12;

//Assign a floating point number.
mydata = 1.2;

//Assign a string
s = "My name is Giorgio";
```

As you can assign different type of data to a **Data** object, you can also assign a **Data** object value to different types of variable, as in the following example:

```
//Assign to an integer variable
int i = mydata.ival();

//Assign to a floating point variable
float f = mydata.fval();

//Assign to a string variable
string s = mydata.sval();
```

You should have noticed that in this case a simple assignment is not allowed, so you have to call the appropriate `*val()` method of the **Data** class, in which the first character is `i` for

[13] http://ross.iasfbo.inaf.it/mcs

integers, `f` for single precision floating point numbers, `s` for strings, etc. Note however that a **Data** object is not a "dynamic type" object (like those of other programming languages such as Perl, PHP, IDL), indeed it always has a "base type" that determines the size of its internal buffer and the behaviour of the different conversion routines (in the last example the "base type" is a string 20 characters long). A consequence of this fact is that an assignment can also lead to an error, just because the "base type" cannot change during the use of the object and some conversions do not make sense. In our previous example, assigning a string more than 20 characters long would have led to an error. Let's examine another case where an assignment could lead to an error:

```
 1 //Create a Data object with base type INT
 2 Data mydata(INT);
 3
 4 mydata = 12;  //Ok
 5 mydata = 1.2; //Ok, will be rounded to 1
 6 mydata = 1.5; //Ok, will be rounded to 2
 7
 8 mydata = "A string"; //Error!!!
 9 mydata = "123";      //That is correct.
10
11 int i = mydata.ival();     //Ok, i = 123
12 string s = mydata.sval();  //Ok, s = "123"
```

As you can see the assignment in line 8 doesn't make any sense, you can't assign a generic string to an integer base type. On the other hand the assignment in line 9 is perfectly allowed because that string can be converted to an integer. Finally, the last two assignment are allowed, even if in the first case the variable will be set to a value of `123` as an integer, whereas in the second the variable will be set to the string `"123"`, whose memory representation is quite different from the integer in the first case.

3.1.2 The `Record` class

On the second level of hierarchy there is the **Record** class which can be seen as a dynamically-sized array of **Data** objects. Here the word "record" is meant in the sense of a record in the Pascal language or the struct in C/C++; indeed a **Record** object can contain any number of **Data** objects of any base type. Creating and populating a **Record** is quite simple:

```
//Create a Record object
Record rec;

//Create some Data objects
Data d1(STRING, 20);
Data d2(INT);
Data d3(TIME);

//Give a name to the Data objects
d1.setName("A_String");
```

```
11 d2.setName("An_Integer");
12 d3.setName("A_Float");
13
14 //Add the Data objects to the Record objects
15 rec.addField(d1);
16 rec.addField(d2);
17 rec.addField(d3);
18
19 //Now we can refer to each Data object using the syntax
20 //we would use with any C/C++ array
21 rec[0] = "Hello";
22 rec[1] = 123;
23 rec[2] = 1.23;
24 Data& tmp = rec[0];
25
26 //We can also refer to a Data object using its name.
27 string s = rec["A_String"].sval();
28 int    i = rec["An_Integer"].ival();
29 float  f = rec["A_Float"].fval();
```

As can be seen from the example a **Record** object can be used like a usual C/C++ array, that is using square brackets(`[]`). Another interesting feature which links the **Data** and **Record** classes is the possibility to give a name to the **Data** object and use that name to search the object inside a **Record** object (see line 27 in previous example).

3.1.3 The `RecordSet` class

The third level of hierarchy is the **RecordSet** class, which is a dynamically-sized container of **Record** objects. It is not required that all contained objects have the same structure (that is the same sequence of base type for **Data** objects). Using a **RecordSet** object is quite similar to using a file handler, indeed there is a current **Record** object pointed by a cursor (like a pointer to a record in a file) that can be moved forward and backward in the file to search for the required data. The following example shows how to populate a **RecordSet** object, and how to visit all the contained **Record** objects:

```
//Create a RecordSet object
RecordSet rset();

//Insert the Record object created in the previous example
rset.insert(rec);

//Insert other Record objects as needed...
rset.insert(...);

//Visit all Record objects

//To visit all Record objects we have to move to the first
//Record object in the set
rset.setFirst();
```

```

//Loop through all Record objects in the set
while (! rset.eof()) {
  //Get a reference to the current Record object
  Record& r = rset.rec();

  //Use the record in some way...

  //Move to the next Record object in the set.
  rset.setNext();
}
```

The previous example, however, shows a quite rare usage of the **RecordSet** class, in fact it would be useless to populate a record and read it again inside the same programming unit. The most common way to populate a **RecordSet** is to derive the class and overload the `fetch()` virtual method. As an example see the code of the **FileReader** class.

It is worth to mention that, while the names "record" and "recordset" are often used in database terminology, the **Record** and **RecordSet** classes are not limited to database access, instead they are designed to be general purpose. For example the **RecordSet** is the parent class of the **Parser_Table** class, which is used to read VOTable files. Of course these classes are also used to perform database access, in particular the result of a query inside the **MCS** library is presented as a **RecordSet** object, a record of the result is presented as a **Record** object and a field of the record is presented as a **Data** object. Another important feature of these classes is that they implement the **Serializable** interface, so they can be easily sent through the network using the **Socket** class. Indeed all message passing between client and server during an **MCS** session is performed using these objects to incapsulate the information being sent.

3.2 Threads and synchronization

Threads are essentially identical copies of a code which run on different data, and let your program perform several "contemporary" tasks. **MCS** lets you implement code that runs in separate threads, and it provides a way to synchronize those threads using the **Thread** and **Synchro** classes.

3.2.1 The `Thread` class

To implement a code which runs in a separate thread you should derive the **Thread** class and overload its virtual `run()` method. This will become the body of execution of the separate thread. Suppose you want to implement an application which reads file names from standard input and "contemporary" perform some task on the given files. The derived class would be as in the following example:

```
//A ``global'' queue, while file names are read from stdin they
//are stored in this queue.
```

```
Record queue;

//Derive the Thread class
class MyThread : public Thread {

  void run() {
    for (;;) {
      //If there's at least one file name in the queue...
      if (queue.count()) {
        string fn = queue.pop().sval();
        //...perform some task on the file
      }
      sleep_ms(1000); //wait for one second.
    }
  }
};
```

As you can see this thread waits until there are file names in a queue, and once the data arrive it will take the file name and perform some task on the file. The main program should read the file names from stdin, as in the following example:

```
int main() {
  //Create the thread object and start it...
  MyThread t;
  t.start();

  string s;
  while (cin >> s) //Read from standand input
    queue.push(s); //Insert the string in the queue
}
```

This program will always be "alive" that is it will always react when you type a filename on stdin, even if it is working on a file in a separate thread.

The **Thread** class is used inside **MCS** to implement the multithreaded server. In the simplest case, when a user connects to the server it will create an instance of a class named **UserThread** (which derives **Thread**) which will listen for user commands and eventually will send back the requested data. Thus the main server program can continue to listen for new users connections. Also, if other users are already connected, they will have their own thread to work with. This way the server can provide services to different users at the same time.

3.2.2 The `Synchro` class

Once you have different threads running in your application you may need to synchronize them. A typical example occurs when the threads need to access a global variable. Suppose that in the previous example the computation inside the thread is very time consuming, and you want to elaborate two files in parallel. This can be easily done creating two (or even more) instances of the `MyThread` class in the main program:

```
//Create two instances of the MyThread class.
MyThread t1;
MyThread t2;

//Start separate threads.
t1.start();
t2.start();
```

That's all is needed to implement the parallel elaboration of two files. But there's is a hidden bug in this code, what would happen if both threads try to read from the global queue at the same time? Even if this is unlikely to occur, both threads could read the same data from the queue and start the elaboration of the same file, thus probably leading to an error. This bug can be removed using the **Synchro** class to implement "protected sections", that are sections of code that can be executed by only one thread at a time. The changes in the code are as follows:

```
class MyThread : public Thread {
  //Use the same Synchro object in all parallel threads.
  static Synchro syn;

  void run() {
    for (;;) {
      if (queue.count()) {
        syn.enter(); //Enter the protected section
        string fn = queue.pop().sval();
        syn.leave(); //Leave the protected section
        //...perform some task on the file
      }
      sleep_ms(1000); //wait for one second.
    }
  }
};
```

As you can see the global variable now is being used inside a protected section, thus the previous bug cannot occur anymore. Note also that the protected section is leaved before working on the file, that is because otherwise the other thread couldn't advance until the first has ended its job.

3.3 Database access

THe **MCS** library can be used to execute SQL queries on a MySQL database. The classes involved are `DBConn`, `Query` and `Table`.

3.3.1 The `DBConn` class

This class is used to open a database connection and perform authentication, as in the following example:

```
DBConn db;
```

```
db.connect("mcstest", "mcstest", "test");
```

If the connection or authentication fails an exception will be thrown.

3.3.2 The `Query` class

Once a connection to the database has been estblished it is possible to execute SQL queries on it through the `Query` class:

```
Query qry(&db);
qry.prepare("SELECT * FROM mcstest");
qry.execute();
```

In this case the query produce a result set which can be retrieved through the data abstraction layer (because `Query` derives from `RecordSet`):

```
//Loop through the resulting record set
while (! qry.eof()) {
  //Get a reference to current record
  Record& rec = qry.rec();

  //For each field print its value
  for (int i=0; i<rec.count(); i++)
    cout << rec[i].sval() << "\t";
  cout << endl;

  //Move to next record
  qry.setNext();
}
```

4 The MCS server

4.1 Architecture of an MCS based system

An **MCS** based system has four main components (see Fig. 1):

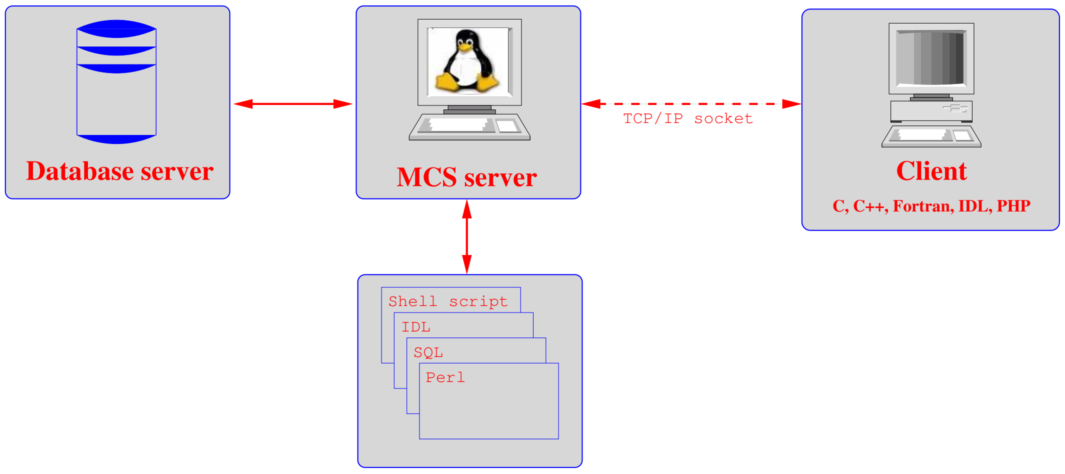

Figure 1: Main components of an **MCS** based system

4.1.1 Database server

The database server is used to handle clients authentication, to store all application specific data and anything else necessary to the application itself. This server isn't accessible directly from the clients, but it is visible only to the application server. At the moment the only supported database server is MySQL [14]. In the future other servers may become accessible through **MCS** .

4.1.2 Application server (MCS based)

The application server is the core of the information system. It implements the client/server model: a client opens a TCP socket towards the host running **MCS** and sends a request, then the server "computes" an answer, eventually querying the database and/or executing some external programs, and sends it back to the client. The behaviour of the **MCS** server can be customized deriving some classes.

4.1.3 External programs

External programs are software applications written in any language, which interact with the application server via command line and the standard output. Support to these pro-

[14]http://www.mysql.com

grams was added to easily integrate already existing applications within **MCS** .

4.1.4 Clients

Clients are programs which access the **MCS** service over the network. Such programs can be written in any language and run on any platform, provided that they implement the **MCS** protocol. Interfaces that implement the **MCS** protocol are provided by the **MCS** library for the following languages on the Linux platform: C++, C, Fortran, IDL, PHP, Python. Support for other languages (such as Java and Perl) and the Windows platform will be available soon (we hope).

4.2 The MCS server

The **MCS** server is an application server, that is an application that waits on a TCP port until a client gets connected. When a client is connected the server provides an information service, that is the possibility to request *information* to the server. Data are transmitted using the **MCS** protocol, which must be implemented by the clients. This protocol is flexible enough to transmit binary data and files, but also to let a client access the service offered by **MCS** from a simple telnet client . Due to the flexibility of the protocol, **MCS** is the natural solution to implement communication between different software tools running on different hosts, through the network. So **MCS** can also perform IPC (Inter Process Communication).

4.2.1 Comparison with a "shell"

Using interactively an **MCS** server resemble very closely the usage of a classic Unix shell, that is a command line interface with a prompt on which users can execute commands in their own environment and wait for the output before a new command can be issued. It is therefore possible to make a comparison between the "components" of a shell, and the ones from an **MCS** connection (see Tab. 1). The concepts listed herein will be described in details later.

Unix shell	**MCS server**
`stdin` e `stdout`	bidirectional TCP socket
system account	MySQL account
internal commands	base commands
programs, shell scripts	external programs (`EXEC` command)
home directory	work directory

Table 1: "shell" comparison

4.2.2 Temporal sequence of events during a connection

In this section we'll explain the temporal sequence of events during an **MCS** session, using Fig. 2. The arrow indicates the time flow; as we can see different blocks on client and

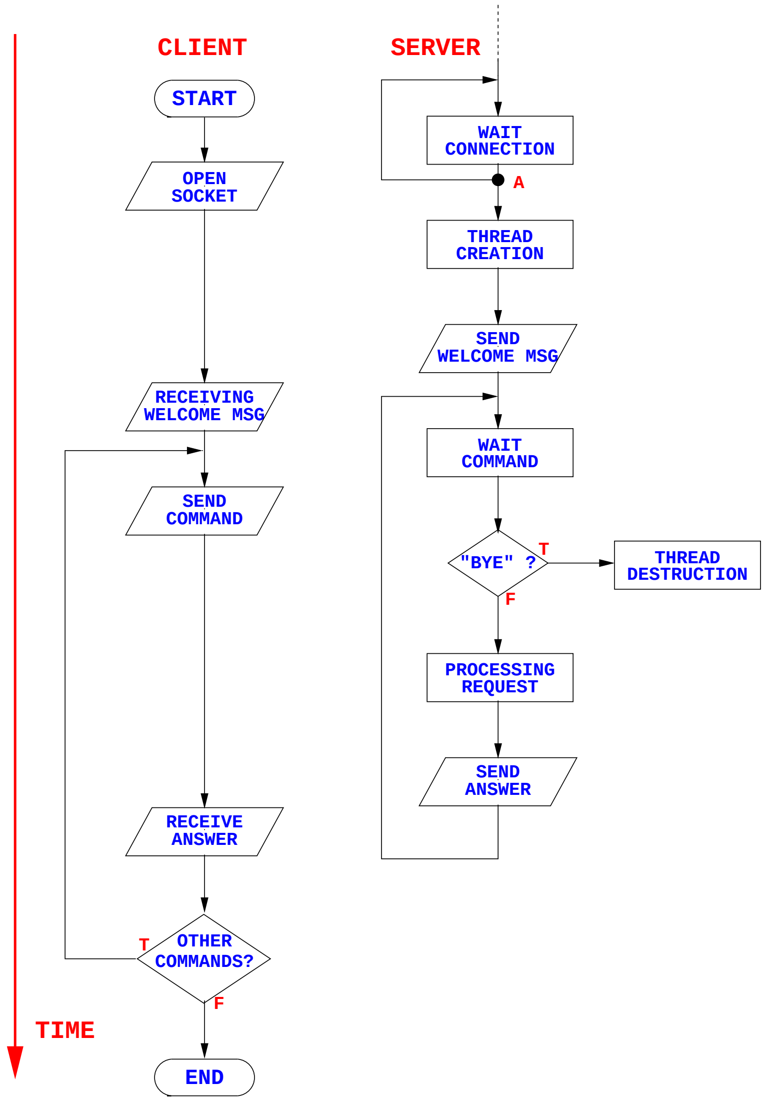

Figure 2: Flow diagram of a typical **MCS** session

server aren't contemporaries. Furthermore we see that while for the client the start and end points are well defined, for the server we don't have such points because we suppose it is executed indefinitely. Usually the server waits for a client to open a TCP socket, when such a socket is opened, the server creates a new thread of execution and assigns the connection to it. We can notice (point labeled "A" in Fig. 2), that at this point the execution path of the server is split in two, one of them returns to wait for another client, while the other starts processing client requests on another thread. Threads are basically identical programs working on different data. In this case the different data are the different sockets connected to different clients. That's how the server can process requests from several clients "simultaneously". As soon as the new thread is ready, it will send a welcome message through the socket. Receiving this message indicates that the server is ready to process client requests. From now on the server will wait for commands to arrive, while the client will enter a loop to send all necessary commands and receiving the related answers. When the server receives a command it will check if it is a `BYE` command, that is a command to close the connection. If this is the case then the thread will destroy itself. In all other cases the server will process the request and send the answer to the client until a `BYE` command is received.

4.2.3 Base commands

Users can request a service from an **MCS** based application server using **commands**. There are two types of commands: *a.* **base commands** are those implemented in **MCS** itself, *b.* **custom commands** are those implemented by the users (see Sect. "Developer's manual").

A command is a sequence of characters terminated by a newline character, very similar to the ones used in a typical UNIX shell. They are composed of a keyword (the command itself), zero or more options (with a "-" minus sign) and zero or more arguments, depending on the command. The command keyword and the options are case insensitive whereas arguments are not. Options and arguments are separated by one or more blanks, and can be enclosed in double quotes to be considered as a single argument (`''a single argument''`). The actual argument anyway won't contain the double quotes. If an argument must contain a double quote it can be escaped with a backslash (`\"`). The Tab. 2 shows a list of available **base commands**. Any base command provides the `-help` option, which will produce a brief explanation of the command usage. A command can be executed each time the **MCS** server sends a prompt, there can be three kinds of prompt:

- `#0--`: last command executed successfully;
- `#0W-`: last command report a warning;
- `#0E-`: last command report an error.

Options common to all commands There are a number of options that are common to all commands, so we report them here:

- `-help`: show a brief explanation about the command usage and doesn't execute the command;
- `-force`: continue execution of commands even if an error occurred;
- `-werr`: Turns all warning into errors, so that a warning can stop the execution.

Table 2: **MCS** Command codes

Command code	**Meaning**
`CID`	Retrieve the CID (Client Identifier)
`CLINFO`	Retrieve information about all connected clients
`NOP`	A "do-nothing" command
`USR`	Supply user name
`PWD`	Supply password
`DBN`	Supply application name
`CON`	Login
`BYE`	Close the connection
`GET`	Download a file from work directory
`PUT`	Upload a file to the work directory
`GETDATA`	Retrieve a `Data` object
`PUTDATA`	Send a `Data` object
`QRY`	Execute an SQL query
`QRES`	Send the query result as a file
`FETCH`	Retrieve the record at the specified position
`EXEC`	Execute an external command or script, with parameter

Command `USR <username>` Used to supply the user name to the **MCS** server during authentication. The user name identifies a list of grants. Example:

```
usr giorgio
```

Command `PWD <password>` Used to supply the password for a specific account. Example:

```
pwd my_password
```

Command `DBN <application_name>` Used to select the application to which the user wants to connect. This command can be used when a single **MCS** server implements different applications, otherwise it is not useful. Example:

```
dbn test
```

This way you will access the application named `test`.

Command `CON` This command doesn't need any parameter and it is used to finalize the authentication process. It must be used after the `USR`, `PWD` and optionally the `DBN` commands. If the user authenticates successfully, the following command will be automatically executed:

```
exec auto
```

The **MCS** server administrator may use the `auto` script to initialize the user environment.

Command `BYE` Logout and close the connection.

Command `CID` Every client has a **Client identifier**, that is a unique integer number that identifies the associated thread. This command can be used to retrieve such a number.

Command `QRY [-sqascii] [-sqfits] <query>` Execute SQL queries directly on the database DB server. The query doesn't need to be quoted, and you can use the quotes inside the query itself. If it is a selection it will return information about the record set, if it is not will return the number of affected records. If the option `-sqascii` or `-sqfits` are given then the result of the query will be written into a file in the work directory respectively in ASCII or FITS format. Example:

```
qry SELECT * FROM table
```

Notice: you don't need to supply the ';' at the end of the query, like you would do with the MySQL client.

Command `QRES` This command prepare an ASCII file containing all records from the last query, then it will send the file to the client.

Command `EXEC <name> [PARS]` This command executed an external program, an SQL script or a batch file. The name of external programs and/or script are specified on the server in the configuration file. If you're executing an external program, the parameters will be passed on the command line and its standard output and error will be written into the work directory in the `out` and `err` file respectively. If you're executing an SQL or batch script, the parameters will be substituted inside the script where a placeholder (like `$1`, `$2`, etc..) is found.

Command `GET [<file_name>]` Retrieve a file located into the work directory on the server. The parameter is the file name. If no parameter is passed then the file `out` will be retrieved.

Command `PUT <file_name> <size>` Store a file into the work directory on the server. Parameters are the file name and the file size in byte.

4.2.4 Grants

TO BE WRITTEN.

4.3 Running the MCS server

TO BE WRITTEN.

4.3.1 The configuration file

TO BE WRITTEN.

4.4 Customize the MCS server

One of the main feature of **MCS** (as its acronym suggests) is the possibility to be customized, adapting the server behaviour to specific needs/tasks. **MCS** can be customized in several ways:

- adding external programs, either real external applications or batch lists of **MCS** commands;
- adding SQL programs, to be executed on the database server;
- adding customized commands, deriving the `UserThread` class;
- modifying the behaviour of the local thread, deriving the `LocalThread` class;

We'll explain in detail how to customize the **MCS** server in the following sections.

4.4.1 Adding external programs

TO BE WRITTEN.

4.4.2 Adding SQL scripts

TO BE WRITTEN.

4.4.3 Adding BATCH scripts

TO BE WRITTEN.

4.4.4 Adding custom commands

TO BE WRITTEN.

5 Connecting to an MCS service

To connect to an **MCS** service you can use one of the available interfaces for the various languages: C++, C, Fortran, IDL, PHP or Python (Perl and Java will be added). They already implement all the features of the **MCS** protocol like handling connection and data transfer, either as a file or as record sets. One of the main features of these interfaces, as shown below, is that the syntax (except for the C++ interface) is almost identical for all languages.

5.1 The Client class

The `Client` class is the main interface to **MCS** ; all other interfaces are implemented through it. We recommend to read the reference documentation for the classes we'll mention.

The `Client` class constructor (line 9) accepts parameters about *a.* the `CLIPATH` directory (see Fig. 3), *b.* the host address of the host whose running the **MCS** service, *c.* the port on which the server is listening. Once you are connected to the server, you can use the `login()` method (line 11) to perform authentication and the `exec()` method (line 12) to issue commands. In this examples we issued the `CID` command to retrieve the client identifier. Each data exchanged between client and server passes through one of the public `Record` members of the `Class` client. In this case the client identifier is contained into a `Data` object in the `aux` member (line 14). As an example of another `Record` member we can require a brief help message about a command (line 16, 17, 18). Then we can perform a query on the database and print all records (line 20 to 33). Finally we close the connection and delete the `Client` object (line 35 and 36). Note that all the code is in a `try..catch` block to eventually catch exceptions that may be thrown (**MCS** code throws only exception based on the class `Event`).

5.2 User environment

Once a user has connected and logged in (see Sect. 5.3) to an **MCS** service, he/she has a dedicated environment consisting of:

- a work directory, where the user can upload or download files;
- a database connection, so that a user can perform database queries as well as create temporary tables visible only to the user itself.

In this document we will refer to the work directory on the server with `CLIPATH`. With the same name we'll refer to the directory on the client side from/to which files are uploaded or downloaded (see Fig. 3). Figure 3 shows a directory called `SRVPATH` above `CLIPATH`: this is the server main directory which contains all work directories.

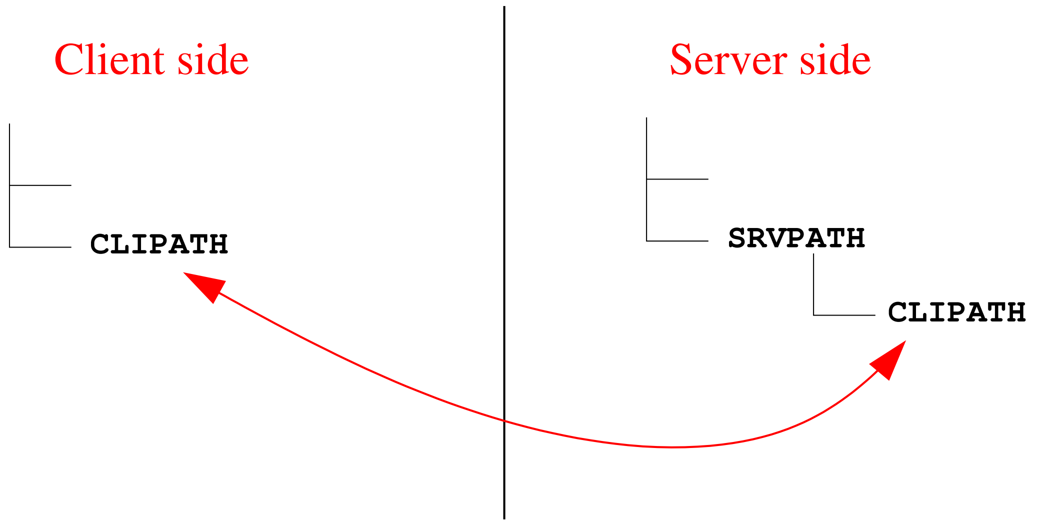

Figure 3: Directories in an **MCS** based system

5.3 Login

The authentication should be made just after the connection using the `USR`, `PWD`, `DBN` (optional) and `CON` commands. Some commands (`QRY`, `FETCH`, `EXEC`, etc...) require the user to be logged in, otherwise a permission error will occur.

5.4 Logout

To logout simply issue the command `BYE` then close the socket. It is important that the client closes the socket first, otherwise the resource on the server will remain locked.
When a user closes its connection the server will close the database connection, destroy the thread associated with the client and optionally delete all files in the user work directory. (**MCS** will use the **MyRO** package to handle grants).

6 Using MCS with other programming languages

An **MCS** session is quite similar to a telnet client, so user can connect to the server also with a simple telnet program. But to take advantage of all the feature of the **MCS** protocol you'll need to use one of the available interfaces.
MCS is written in C++, so most of its facilities are available as classes. In the following sections we'll assume that the reader has a minimum knowledge of the object oriented paradigm, as well as the C++ syntax. Furthermore we'll mention some of the **MCS** classes, whose reference documentation can be found at http://ross.iasfbo.inaf.it/mcs.
A user which connects to an **MCS** server needs to deal with the following classes:

- `Client`: the interface to connect to an **MCS** server;
- `Data`: the base class by which data are exchanged between client and server;
- `Record`: a collection of `Data` objects;
- `RecordSet`: a collection of `Record` objects.

Furthermore the following set of classes may be useful to a client program:

- `DBConn`: connect to a database server;
- `Query`: execute queries and retrieve results;
- `Table`: perform direct access on a database table;
- `Conf`: reads configuration files.

In Sect. 5.1 we'll explain how to write a program which connects to an **MCS** server using these classes and the C++ language. If a user wishes to use another language to connect to **MCS** he can use one of the available interfaces, which are simply wrappers to the classes mentioned above. For this reason the code to connect to **MCS** is quite similar in all languages, and we recommend to read Sect. 5.1 also if you don't plan to use C++. Successive sections will report information specific to each language. As already mentioned, the languages actually supported are: C, Fortran, IDL, PHP and Python. In the near future interfaces will be developed for Java and Perl.

6.1 Interface usage and naming convention

In this section we describe how to use the interfaces for languages others than C++, and the naming convention used. For a detailed description of all the classes involved and their methods check the reference manual as well as Sect. 5.1.

As mentioned above, the interfaces are simply wrappers around C++ classes, so when you're calling a function of the interface in your favourite language, you are actually calling a method of a C++ object that lives in the heap (dynamic) memory. That's the reason why there isn't a reference documentation for each interface, the main reference for the **MCS** classes contains all the information you need. Note that not all the class members are wrapped in the various interfaces (because only C++ supports overloading), check the classes documentation to see if a method is wrapped or not. To use the interface you should therefore create an object, call one or more of its methods and finally destroy him.

Before using an object you must call the appropriate wrapper to the constructor which returns the memory address where the object has been allocated. You should not modify this address, neither modify the type of the variable where the address has been stored (for those languages that let you do this), otherwise the object will become unreachable. We recommend to destroy objects when they are no longer needed. You can do this by calling the appropriate wrapper to the destructor and passing the address of the object to be destroyed. The memory address you got from the constructor must also be passed to all the wrappers to methods of that class.

Important note: What is returned by the wrapper to constructor routine is just a memory address, so the language you are using (even C!) doesn't know anything about the type of object that has been created. For this reason, if you use this address with a wrapper of another class, your compiler won't give you any compilation or syntax error but you will likely get a "segmentation fault" when the program is running.

There is only one exception to this rule: if an object derives from other classes then you can use the address of that object also with wrappers of parent classes.
In the following sections we'll explain the naming convention for a generic class named `CLASS`. Arguments in brackets (<>) are specific to a function or method. Arguments named `addr` are the memory address of an object.

6.2 Constructors

Constructors follow the name convention:

```
addr = new_CLASS(0, <PARAMETERS>)
```

where the parameters are the ones needed by the constructor. The first parameter is reserved for future use. Note that only one constructor can be wrapped so check the documentation to see which one is used. These functions return the memory address where the object has been allocated. You must use this address in the subsequent methods call. In all interfaces this memory address is stored in a numeric variable, you should avoid changing its value or the type of the variable.

6.3 Copying constructors

Copying constructors follows the name convention:

```
newaddr = copy_CLASS(addr)
```

These functions don't need any specific parameter but only the address of the object to be copied. Note that you should call the appropriate copy constructor (the one belonging to the same class with which the object was created), otherwise you will surely get a "segmentation fault". These functions return the memory address of the newly created object.

Note: actually only the `Data` class has a copy constructor implemented in the interfaces at the moment.

6.4 Destructors

Destructors follows the name convention:

```
del_CLASS(addr)
```

These functions don't need any specific parameter but only the address of the object to be destroyed. Note that you should call the appropriate destructor (the one belonging to the same class with which the object was created), otherwise you will surely get a "segmentation fault". These functions doesn't return any value.

6.5 Methods

Methods follow the name convention:

```
retval = CLASS_methodname(addr, <PARAMETERS>)
```

where the parameters are the ones needed by the method. Note that because overloading is not supported, only one method with a certain name can exist. The type and meaning of the returned values depend on the method called; check the class documentation.

6.6 Error handling

Many **MCS** classes use exceptions to handle errors, but this mechanism is not available in C nor in other languages for which we have an interface. Because we didn't want to use the "C-Style" error handling, that is checking the returned value of a function after each call to see if an error occurred, we implemented another mechanism: the **MCS** library maintains an internal "status" structure which tells if an error occurred or not. This "status" is checked each time an interface function is called, if an error occurred in a previous call the function will return immediately, otherwise the function will do its job. If an error occurs during the execution of the function the "status" will be updated with an error message. In this case all successive calls will return immediately. This way you can check if an error occurred only at the very end of a sequence of instructions, as in the following example:

```
Call MCS function 1
Call MCS function 2
Call MCS function 3
...

if (ifd_got_error()) {    //...handle error
  print ifd_last_error();
}

ifd_reset_error();
```

As you can see the check for the error is performed at the very end, not after each function call. Once you handled the error you may decide to continue execution, in this case you should reset the "status" information with a call to the `ifd_reset_error` function (as shown above).

6.7 The C interface

To use the C interface you should include the `mcs_c.h` file in your source code, it is located in the same directory as the main include file `mcs.hh` (see Sect. 2.2.1). To compile and link your program follow the same step as with any other **MCS** based program:

```
cc ‘mcs-config --cflags‘ -c myprog.c
cc -o myprog myprog.o ‘mcs-config --libs‘
```

As an example see Fig. 5 in which we implemented in C the program we developed in C++ (Fig. 4).

Some line needs a comment, first of all note lines 7, 9, 10 in which the number of arguments is different from the corresponding C++ code, that's because C cannot handle default argument values, so we have to specify them all. At lines 13, 19, 26, 2 you can see that the syntax is in reverse order in respect to the C++ code, let's examine the line 13 in detail:

```
cli->aux[0].ival()                                 C++

Data_ival( Record_field( Client_aux(cli), 0) )     C
```

In either cases we are calling the `Data::ival` method, of the object returned by the `Record::operator[]` at position 0, of the `aux` member of a `Client` object. But the `operator[]` doesn't have any equivalent in the C language, so it has been substituted by the `Record_field` function (see reference documentation). Furthermore the order in which members are called in C++ is reversed in the C language. At line 38 there is a check to see if an error occurred, and eventually the message is printed and the "status" is restored. Finally note at lines 24 and 29 that we passed the address of the `Client` object to a wrapper for the `RecordSet` class; this is allowed only because `Client` derives from `RecordSet`.

```c
#include <stdio.h>
#include <mcs_c.h>

int main(int argc, char* argv[])
{
  int i;
  IFD_OBJP cli = new_Client(0, "./", "localhost", 6523, 0, 10000);

  Client_login(cli, "mcstest", "mcstest", "");
  Client_exec(cli, "CID", 0);

  printf("Client identifier is: %d\n",
         Data_ival( Record_field( Client_aux(cli), 0) ));

  Client_exec(cli, "CID -help", 0);

  for (i=0; i<Record_count( Client_out(cli) ); i++)
    printf ("%s\n",
            Data_sval( Record_field( Client_out(cli), i ) ) );

  Client_exec(cli, "qry SELECT * FROM mcstest", 0);
  printf("Number of rows: %d\n", RecordSet_nRows(cli));

  for (i=0; i<RecordSet_nFields(cli); i++)
    printf ("%s\t",
            Data_name( Record_field( RecordSet_metarec(cli), i ) ) );
  printf("\n");

  while (! RecordSet_eof(cli)) {
    for (i=0; i<RecordSet_nFields(cli); i++)
      printf ("%s\t",
              Data_sval( Record_field( RecordSet_rec(cli), i ) ) );
    printf("\n");

    RecordSet_setNext(cli);
  }

  if (ifd_got_error()) {
    printf("ERROR: %s\n", ifd_last_error());
    ifd_reset_error();
  }

  Client_exec(cli, "BYE", 0);
  del_Client(cli);

  return 0;
}
```

Listing 5: Source of client_c.c

6.8 The Fortran interface

To use the Fortran interface you should include two files:

- `mcs_fortran.inc`: wrappers implementation;
- `mcs_facility.inc`: functions declaration.

These files are located in the same directory as the main include file `mcs.hh` (see Sect. 2.2.1). To compile and link your program follow these steps:

```
f77 `mcs-config --cflags` -Wno-unused-variable -c f_test.f
f77 -o f_test f_test.o `mcs-config --libs`
```

The `-Wno-unused-variable` is used here because in the `mcs_facility.inc` there is a declaration for each function of the **MCS** interface; without that option the compiler will annoy the user with a lot of warnings.
The comments relative to the C language apply here as well, furthermore notice that for some functions like `Client_exec`, `RecordSet_setNext` we used a dummy variable for the return value. This is necessary because these are really functions (not procedures like `del_Client`), even if the return value is of no interest.

6.9 The PHP interface

The PHP interface will be available only if the option was enabled when you executed the configuration script. Starting from MCS version 0.3.1 it is *not required* anymore to use the `php2mcs` script in order to be able to use the PHP interface. You still must include the file `php2mcs.php` in your code (`require("php2mcs.php");`).
The comments relative to the C language apply here as well, furthermore notice that instead of passing 0 to those parameters whose C++ counterparts require a “NULL” value, we used the `ifd_null` function.

6.10 The Python interface

The Python interface will be available only if the option was enabled when you executed the configuration script. Starting from MCS version 0.3.2 it is *not required* anymore to use the `python2mcs` script in order to be able to use the Python interface. You still must import the file `python2mcs.py` in your code (`from python2mcs import *`).
The comments relative to the C language apply here as well, furthermore notice that instead of passing 0 to those parameters whose C++ counterparts require a “NULL” value, we used the `ifd_null` function!

6.11 The IDL interface

The IDL interface will be available only if the option was enable when you executed the configuration script. To use the IDL interface you should execute the `idl2mcs` script in the directory where you'll store your source code, this will create symbolic links to the files needed. For efficiency reasons, all interface functions will be available as IDL Dynamically Loadable Modules (DLMs) rather than functions and procedures. No further action is required from the user to use these functions in the IDL code.
The comments relative to the C language apply here as well.