HELSINKI UNIVERSITY OF TECHNOLOGY
Laboratory of Telecommunications Technology

An Implementation of the Server Cache Synchronisation Protocol (SCSP)

Supervisor: Professor Francisco Mora
Instructor: Professor Raimo Kantola

Espoo, March ,1999

HELSINKI UNIVERSITY OF TECHNOLOGY **MASTER'S THESIS ABSTRACT**

Author: Jose Requena

Title: An Implementation of the Server Cache Synchronisation Protocol (SCSP)

Date: March , 1999
Pages: 69

Department: Laboratory of Telecommunications Technology

Supervisor: Professor Francisco Mora

An implementation of the Server Cache Synchronisation Protocol (SCSP) is presented in this document. Initially, the SCSP algorithm establishes the data synchronisation among a set of server entities of a particular protocol which are bound to a Server Group (SG). After that it maintains an actively mirroring state of any change in every cache information that form the SG.

The program is based on an object-oriented software environment. This leads to a modular implementation. Different processes take place between each "Local Server" (LS) and its "Direct Connected Server" (DCS).

The protocol code (which is written in C++) facilitates an easy translation to new communication architectures and services in a rapidly changing scenario. In that way the platform-dependence was minimised using one of the most extended programming languages namely C++, combined with the software functions provided by "Free Software Foundation, GNU". Probably, the GNU package compiles on more operating systems than the actual machine independent languages. UNIX, is the operating system chosen for the software development. UNIX provides enough speed to the protocol performance in those situations where the data replication is important for the system which is using the SCSP.

The basic structures used by the program were developed following the Management Information Base (MIB) specifications. The purpose is to permit later interactions with an SNMP agent, which checks the correct functioning of the protocol.

The most important application of this software package is focused in the IP Telephony and the new emerging directory services. For those services it is necessary to manage large amounts of information that is stored in distributed databases that need to be efficiently updated and replicated.

Having described the implementation of the software, the work ends with a discussion of future applications. In those applications the SCSP can be used to solve the problem of the data replication in distributed protocol entities. In the results chapter we show the tests done to check that the program works properly among different workstations in the Laboratory.
The main difficulty in the software development was handling all the different data and interfaces which were involved in the program. The problem was solved in modular structures and independent functions that were tested gradually in stand alone functioning. Thus the final purpose was reached, and the complexity could be managed.

Keywords: C++,SNMP, MIB, SG,UNIX

Preface

This master thesis has been written at the Laboratory of Telecommunications of the Helsinki University of Technology. I would like to thank all my collegues and friends who have helped me by giving advice every time I asked for it. I also thank my Instructor Raimo Kantola, for his guidance and valuable comments during the correction.
Finally I would like to thank my parents for their support during my studies and not falling into despair, specially my sister and brother for maintaining my enthusiasm alive and my witch “Queca” for her support and aid during my sketchs.

March 23, 199
Helsinki, Finland

Jose Miguel Costa Requena

Abbreviations

ASN.1	Abstract Syntax Notation One
CA	Cache Alignment
CAFSM	Cache Alignment Finite State Machine
CCITT	*Comite Consultatif Intenationale de Telegraphique et Telephonique*
CIP	Common Indexing Protocol
CMIT	Common Management Information Transaction
CPU	Central Processing Unit
CRL	Cache State Alignment Request List
CSA	Cache State Alignment
CSA	Cache State Advertisement
CSU	Cache State Update
CSUS	Cache State Update Summary
DARPA	Defence Advanced Research Projects Agency
DBA	Database administrator
DBMS	Database Management System
DCS	Direct Connected Server
DCSID	Direct Connected Server Identifier
DDL	Database definition language
DISP	Directory Information Shadowing Protocol
DMD	Directory Management Domain
DML	Database manipulation language
DSA	Directory Server Agent
FIFO	First In First Out
FSM	Finite State machine
GID	Group Identifier
HP 300	Hewlett Packard 300
HFSM	Hello Finite State Machine
IAB	Internet Architecture Board
IANA	Internet Assigned Numbers Authority
IDP	Internet Datagram Packets
IETF	Internet Engineering Task Force
IP	Internet Protocol
ISDN	Integrated Service Digital Network
ISO	International Standards Organization
ISP	Internet Service Provider
LAN	Local Area Network
LS	Local Server
MIB	Management Information Base
M/S	Master/Slave
OS	Operating System
OSI	Open Systems Interconnection
OSPF	Open Shortest Path First
PID	Protocol Identifier
RFC	Request for Comment
SCSP	Server Cache Synchronisation Protocol
SG	Server Group
SGID	Server Group Identifier

SMI	Structure Management Information
SNMP	Simple Network Management Protocol
SPI	Security Parameter Index
SPP	Service Provider Packet
TCP	Transport Control Protocol
TCP/IP	Transport Control Protocol/Internet Protocol
UDP	User Datagram Protocol
UTC	Universal Coordinated Time

Contents

List of Figures

Chapter 1

Introduction

1.1 About this Thesis

This thesis is about the implementation of a software tool, developed for database synchronisation and replication. In the implementation we have followed the protocol specifications from the Internet protocol for the Internet Community, titled “Server Cache Synchronisation Protocol ” (SCSP) [1].

The synchronisation is performed by a group of distributed protocol entities which compound a Server group (SG). The purpose of this software is to achieve a complete synchronisation of the data stored in any server of the SG. When the synchronisation is reached, the secondary objective is to update immediately every new information that has changed in the data stored in each server.

1.2 Structure of the thesis

The approach taken in this software project is to start with a basic introduction to the database systems. Thus in the second chapter, we describe the database systems and the different Internet models for their implementation. Afterwards, we show an introduction to some existing databases and the mechanism for information management. In those databases we can fit the SCSP as a solution for their synchronisation/replication. In this chapter we also review some exiting protocols with similar algorithms, to compare with the characteristics of the SCSP.

In the third chapter we explain the framework for the implementation. We describe the basis of the SCSP algorithm which is borrowed from the OSPF. Furthermore, in this chapter we clarify the mechanism used by the SCSP to solve the synchronisation problem. Afterwards we review the different steps of the protocol defined in the specifications RFC 2334 [1].

The fourth chapter shows the characteristics of the implementation and we argue the reasons for the choice of a specific language for its development. In this chapter we analyse the interactions between the program and the Operating System (OS). We introduce the facilities provided by the OS for the programming task. In addition, we describe the different steps followed in the program development.

In the fifth chapter we describe the basic implementation and the functional modules needed for the main structures. All the pieces that the SCSP requires for its development are

defined. In this chapter, we expose the internal specifications of the data structures which have to follow the SMI definitions.

In the sixth chapter, we present the interface module. In fact it is an independent program with the purpose of displaying the protocol state in every moment. This interface is a graphical module based on X-windows programming. To avoid interfering with the normal functions of the SCSP, this module acts without any direct interaction with the main program. The information that the interface requires is read through an information file which is filled by the main program.

The seventh chapter, shows the results of the final implementation. In this chapter we depict the global flow diagram and the results obtained.

Finally, in the eight chapter we describe the conclusions and future work. The characteristics of the SCSP are described as a useful facility for other protocols. Likewise, we show the features provided by the SCSP for future requirements in other protocols for their data replication.

1.3 Goals of the Thesis

The main goal of the thesis is to obtain a powerful tool for the database synchronisation and replication among different hosts, servers or distributed entities.

The second goal is a machine independent implementation that nevertheless is efficient. Thus there is a compromise between speed and platform-independence in obtaining an efficient and portable software program.

To achieve the first objective, we follow the SCSP RFC 2334 [1] for server cache synchronisation in distributed protocol entities. The specifications defines the algorithm to reach the data replication in a group of servers. That algorithm is borrowed from the well-known OSPF protocol. Behind the OSPF protocol are accumulated many years of experience and efficient work over IP networks. Hence we can assure that indeed the SCSP is based on a robust design. Thus by following the SCSP specifications, we will obtain an effective means for data replication in the SG.

The second purpose, to achieve a fast and platform-independent program is quite complicated. This implementation requires a considerable effort in writing functions based on separate modules. A design goal is that for the purpose of translating the program to another architecture we only have to change one of the basic modules. Hence, we can maintain intact the algorithm which is placed on an upper layer in the global skeleton. Therefore, we use an object-oriented programming environment which helps in modular implementation for each of the different parts in the whole protocol. The program is based on *class* definitions which also provide several features that make it suitable to maintain independent synchronisation processes among the LS and it associated DCS without wasting the system resources.

Following those two blueprints we target an efficient implementation of the SCSP protocol for the data synchronization. The goal is that different servers can use the SCSP features for data replication. Those servers can form already a Server Group or they can be part of independent Groups that desire to share the information in a common agreement. A real example could be the servers which are maintaining the databases of different ISPs that desire to share data in a common agreement, or the servers of an independent ISP which wants to synchronize its own information.

Chapter 2

Distributed databases

In this chapter, we review the database the different models and some existing examples already implemented. We will illustrate briefly the basic characteristics of each model and some details of the information management used by them. Finally, we will point out the specific problems of the data replication in each case and we will show how the SCSP can fit in the replication problem.

2.1 Introduction to Databases

The database technology is one of the most rapidly growing areas of computer information science. The total amount of data now committed to database can be measured, conservatively in terabytes.
What is a database system? Basically, it is nothing more than a computer-based record-keeping system [2]. Those records can store many types of data, from the personal information of one employer in a company, till the important data obtained from a routing algorithm for better performance in network management. The right behaviour of actual Intelligent networks are based on data stored in database records. Many new services in the networks require the management of large amounts of data.

Thus a database is a repository for stored data which is both *integrated* and *shared*.
"Integrated" means that the database may be thought of as a unification of several otherwise distinct data files, with any redundancy among those files partially or wholly eliminated. And by "shared" we mean that individual pieces of data in the database may be shared among several different users, in the sense that each of those users may have access to the same piece of data (and may use it for different purposes) [2].

Between the physical database itself and the users of the system is a layer of software, usually called the database management system or DBMS. All requests from users for access to the database are handled by the DBMS. One general function provided by the DBMS is thus the shielding of database users from hardware-level detail. Other elements in the database system are the data definition language (DDL), which provides the definition or description of the database objects, and a database manipulation language (DML), which supports the manipulation or processing of such objects.

Another important element is the database administrator DBA. This element has the freedom to change the storage structure or access strategy (or both). The DBA 's job is to decide exactly what information is to be held in the database. The DBA must also decide how the data is to be represented in the database, and must specify the representation by writing the storage structure definition using the DDL. The business of the DBA is also to

liaise with users, to ensure that the data they require is available, and to write the necessary external schemas, using the appropriate external DDL.

It is clear that the DBA will require a number of utility programs to help with these tasks. Such utilities would be an essential part of a practical database system. Thus the SCSP could be considered one of those complementary tools for the data management, which could be thought of as a special system-supplied application.

2.2 Database models

A database can be based on *centralised* or *distributed control*.

❑ In the former model, the DBA has the central responsibility for the operational data. The advantages that accrue from the *centralised* control of the data are:

> *Redundancy can be reduced*. In many systems each application has its own private files. This can often lead to considerable redundancy in stored data, with the resultant waste in storage space.
>
> *Inconsistency can be avoided*. Inconsistency can take place when two entries contain the same data and the system is not aware of this duplication. The problem appears when one of the entries change some information and the other has not been updated. Then the database is in an inconsistent state, and obviously is capable of supplying incorrect or conflicting information.
>
> *Standards can be enforced*. With the central control of the database, the DBA can ensure that all applicable standards are followed in the representation of the data. Standardising stored data formats is particularly desirable as an aid to data interchange or migration between systems.

❑ In the latter model, the Distributed database technology is a comparatively recent development within the overall database field. The database is not stored in its entirety at a single physical location, but rather is spread across a network of computers that are geographically dispersed and connected via communications links. The data is stored at the location at which it is most frequently used, but is still available, via the communications network, to users at other locations.

The advantages of such a distribution are:

> *Efficiency of local processing*. This refers to the fact of local operations without communications overhead. And all the advantages discussed above, in particular data sharing, provided by a centralised system.

But there are disadvantages too:

> *Communications overhead*. It could be high enough, and also there are significant technical difficulties in implementing such a system.

A key objective for a distributed system is that it should look like a centralised system to the user. The user should not normally need to know where any given piece of data is physically stored. Applications are independent of the manner in which the data is distributed, making it possible to change the distribution without affecting those applications. Thus the fact that the database is distributed should be relevant only at the internal level, not at the external level.

2.3 Database Examples

This thesis is focused on providing a new facility for the emerging services on IP networks. We have tried to avoid reinventing the wheel. Instead, we have put our best effort in the implementation of an efficient software package based on an existent algorithm well tested during many years, which is the OSPF protocol. Thus, with the implementation of the SCSP protocol, we try to introduce a new tool to manage the information that actually is available on the Internet.
It seems clear that with the vast amount of directory information potentially available on the Internet, it is simply not feasible to build a centralised directory to serve all the information. If we have to distribute the directory service, the easiest (although not necessarily the best) way of building the directory service is to build a hierarchy of directory information collection agents. In this architecture, a directory query is delivered to a certain agent in the tree, and then handed up or down, as appropriate, so that the query is delivered to the agent which holds the information which fills the query. This approach has been tried before, most notably in some implementations of the X.500 standard [3].

2.3.1 X.500 Directory

X.500 is the collective name given to a series of standards produced by the ISO/ITU-T defining the protocols and information model for a global directory service that is independent of the computing applications and the network platform. First released in 1988, the X.500 standards define a specification for a rich, global, distributed directory based on hierarchically named information objects that users can browse and search using arbitrary fields [4]. X.500 uses a model of a set of directory servers (DSAs), each holding a portion of a global directory information base. These co-operate to provide a directory service to users or user applications in a way that means these user applications need not be aware of the location of the information they are accessing i.e., the user applications can connect to any directory server and issue queries to access information anywhere in the global directory [5].

Elements of the X.500 are:

- **DAP** is the Directory Access Protocol defined by ISO/ITU-T as part of the X.500 directory standards. It provides a comprehensive open system protocol for accessing standardised directory servers. The ASN.1 is used in describing the data. DAP (and the other X.500 protocols) can run over Internet Protocols(e.g. TCP/IP).

- **DIT** Directory Information Tree defines the hierarchical naming model which is used to define a single global name space. The directory information is represented in entries (the standard describes the schema to define, for example, object classes and attributes), how entries are organised and named, how information within entries is protected from unauthorised access. The DIT is collectively managed, through splitting into management domains.

- **DSP** Directory Servers Protocol, is the protocol needed to chain user requests between directory servers. The DSP also manage the knowledge information needed to accomplish the connection between servers.

- **DUA** Directory User Agent, is the part of the standard which specifies the access to the directory from the user point of view. It passes the queries from the user

directly to the directory servers, and receives back their answers which will be forwarded to the user.

➢**DSA** Directory Server Agent, is the part of the specifications which defines the attributes of the application that will manage each node in the DIT. The DSA has to connect with the rest of DSAs to form a global directory accessible from everywhere. Transparent to users, X.500 directory servers can pass and resolve queries between themselves, in addition to passing referrals back to the user agent.

➢**DISP** Directory Information Shadowing Protocol provides a fully comprehensive replication model. DISP ensures consistency of knowledge and access control between replicated systems. *Thus DISP is the protocol needed to replicate selected information between servers* in a managed way such that for example access control is preserved and the information doesn't become stale. The specifications are written in X.525 [6].

An example of the application of the X.500 standards could be the implementation of a global directory service. This directory could be distributed world-wide and in each country a different DSA would manage its portion of the whole directory.
In this X.500 application, the country-level DSAs form the access path for the rest of the world to access directory entries associated with that country's organisations. Thus, the availability and performance of the country-level DSAs give an upper bound to the quality of service of the whole country's part of the Directory.

Country DIT information, including naming infrastructure information such as localities and states, should be replicated across the oceans - not only to serve when the transoceanic links go down, but also to handle name resolution operations for clients in other countries. There should be a complete copy of the US root in Europe and a copy of the Japanese root in Africa and North America, for instance. Generally, data should be replicated wherever it is heavily used, and where it will be needed in the event of a network partition.

The main objections raised against X.500 are:

It is too complex and over-engineered, maybe because it is an OSI standard and therefore too resource intensive. Another inconvenience is that the actual X.500 products are not mature and easy-to-use.

One of the Internet arguments against the OSI standardisation process is that it takes far too long to ratify required changes.

A recent version of X.500 - the 1997 edition - is about to be adopted as an international standard. This introduces yet more key features, such as a standard way of remotely managing directory system components, adding internationalisation into the directory model so that multiple languages and character sets can be supported, and further strengthening the use of X.500 as a repository for security-related information through defining encrypted attributes, signed attributes, and an encryption mechanism for protocol sessions.
This version may be a good option for a global directory implementation.

2.3.2 LDAP model

The Lightweight Directory Access Protocol (LDAP) [7] has become a new standard way to access directory services. LDAP is defined by the Internet community. The protocol provides a mechanism for passing text-based queries from an LDAP client to an LDAP server over the TCP/IP network protocol. The query language used, maps very closely to DAP. The aim is to let users quickly and easily create and query directories of people and information for example user names, email address, and telephone numbers. LDAP was proposed when a full OSI solution like DAP would not fit on the desktop technology available at the time. Nowadays, with the actual high technology available, the LDAP is far more widely used than DAP for access to directories. The main reason is that, the X.500 DAP protocol is seen as being complex, and hard to implement. LDAP offers a simpler model that fits more closely with the ideals of the Internet protocol suite.

The primary goal of LDAP is to minimise the complexity of the client so as to facilitate widespread deployment of applications capable of utilising directory services. The LDAP protocol is "designed to provide access to directories supporting the X.500 models, while not incurring the resource requirements of the X.500 Directory Access Protocol (DAP). This protocol is specifically targeted at management applications and browser applications that provide read/write interactive access to directories" [7].

LDAP is a protocol allowing users to access a directory. However, LDAP does not specify how the directory service itself operates. Hence the drawback is that key service aspects such as controlling access to the data, and facilities for replicating the data (for redundancy and performance) are not included in the LDAP specification. Any operational directory system needs to provide these components.

Actually, most of the vendors providing LDAP-based services have their own mechanisms for providing a directory model, access controls and replication. Many of these will be similar to X.500, but not compliant with the standard and may not be interoperable with products from other vendors, although it can be expected that interworking fora involving the major vendors will address many of these problems.

In X.500, these key services are fully defined, ensuring full co-operation between differing directory service implementations. And as we have seen in the above paragraph X.500 defines the DISP, which provides a full replication model.

As an example to solve that problem, the Michigan software, implements LDAP v2 [8], which includes a replication server (slurpd) [9]. This replication server is one specific implementation, it does not follow any standard and has not been widely-adopted.
These replication processes exchange data in a flat file format referred to as the "Lightweight Directory Interchange Format" (LDIF)¨ [10]. This has been put forward as a possible basis for an Internet standard exchange format, although it does only support full duplication of data and not selective, managed replication It is one of the main differences between X.500 and LDAP. DISP is using the ASN.1 notation [10], [11] for coding all the messages with the consequent overhead and complexity to form the packets. Instead of that the LDAP uses LDIF, for LDAP Data Interchange Format which is typically used to import and export directory information between LDAP-based directory servers, or to describe a set of changes which are to be applied to a directory. The LDIF is a simpler format which is perhaps easier to create and consists of a series of records using commands based on ASCII nomenclature and separated by line separators.

The process of seeking agreement on replication between LDAP-based directories through the IETF has only just begun. Several alternatives have been put forward. Currently no one mechanism is in the standardization process or widely adopted amongst multiple vendors.

Most commercial directory vendors have adopted some mechanism for performing replication between their own LDAP servers. Agreement on a common flexible replication mechanism for use by LDAP servers is still some way off. Most commercial implementations of X.500 available today support DISP. However, there are very few examples of multi-vendor inter-working using DISP. One of the reasons for this is that there are options in the standard which require any two implementations of X.500 to adopt the same configuration in order to be able to use DISP successfully. As a result from a user perspective, this is not an off-the-shelf process, and there is no guarantee that a replication agreement will reflect the original replication requirement.

2.3.3 WHOIS

The WHOIS is a service used as a very limited directory service. The current NIC WHOIS [12] system serves information about a small number of Internet users registered with the DDN NIC. The WHOIS NIC database consists of a series of individual records, each of which is identified by a single unique identifier ("handle"). Each record contains one or more lines of information. Over time the basic service has been expanded to provide additional information and similar services have also been set up on other hosts. Unfortunately, these additions and extensions have been done in an uncoordinated manner. Despite its utility, the current NIC WHOIS service cannot function as a general White Pages service for the entire Internet. Given the inability of a single server to offer guaranteed response or reliability, the huge volume of traffic that a full scaled directory service will generate and the potentially huge number of users of such a service, a trivial architecture is obviously unsuitable for the current needs for information services.

WHOIS++

Hence, in view of high utility of this service, a new architecture has been developed. WHOIS++ is a simple, distributed and extensible lookup service based upon small set of extensions to the original WHOIS information model. These extensions allow the new service to address the community's needs for a simple directory service, yet the extensible architecture is expected also to allow it to find application in a number of other information service areas.
These added features include an extension to the trivial WHOIS data model and query protocol and a distributed indexing service. The basic architecture of WHOIS++ allows distributed maintenance of the directory contents and the use of the WHOIS++ indexing service for locating additional WHOIS++ servers. In this way it is quite similar to the distributed directory model described in the X.500 standards.
The WHOIS++ server functions as a known front end, offering a simple data model and communicating through a well known port and query protocol. It is also proposed that individual database handles can be registered through the Internet Assigned Numbers Authority (IANA), ensuring their uniqueness. A WHOIS++ database may be seen as a single collection of typed records. And the WHOIS++ directory service has an architecture which is separated into two components; the *base level server* and the *indexing server*. In fact a single physical server can act as both a base level server and an indexing server. A base level server is one which contains only filled templates. An indexing server is one which contains forward knowledge and pointers to other indexing servers or base level servers.

Thus, the WHOIS++ directory service is intended to provide a simple, extensible directory service based on a template-based information model and a flexible query language. In this system, its general architecture has been designed for use indexes or resumes of the distributed database reducing the traffic and consequently the packets latency. The architecture can be applied to link together many of these WHOIS++ servers into a distributed, "searchable" wide area directory service. In this way WHOIS++ provides a distributed directory service which is also searchable. To have an efficient way to reach as quickly as possible the information allocated in the directory, it is essential to have an efficient *indexing server* . In this Index server whenever we issue a global query (at the root of the name-space), or a query at the top of a given sub-tree in the name-space, that query is replicated to "all" sub-trees of the starting point. However, every server to which the query has been replicated must process that query, even if it has no entries which match the specified criteria. This mechanism by which information servers can exchange indices of information from their database can make use also of the Common Indexing Protocol(CIP).

CIP Architecture

It is assumed that the structures defined in the CIP architecture can be used by X.500 DSAs, LDAP servers, WHOIS++ servers and many others.

CIP [13] proposes a mechanism for distributing searches across several instances of a single type of search engine with a view to creating a global directory. CIP provides a scaleable, flexible scheme to tie an individual database into a distributed warehouse that can scale gracefully with the growth of the Internet. CIP provides a mechanism for meeting these goals that is independent of the access method that is used to access the data that underlies the indices. Separate from CIP is the definition of the Index Object that is used to contain the information that is exchanged among Index Servers. For information servers that contain millions of records in their database, constant exchange of complete dredges of the database is bandwidth intensive. To avoid that problem, the tagged Index Object is specifically designed to support the exchange index of update information. In the case of WHOIS++, an index server will take a query in standard WHOIS++ format, search its collections of index and other forward information, determine which servers hold records which may fill that query, and then notify the user's client of the next servers to contact to submit the query. Thus the exchange of those index among servers allows hints to be given as to which information server actually contains the information. The tagged Index Object labels, the various pieces of information with identifiers that tie the individual object attributes back to an object as a whole. This "tagging" of information allows an index server to be more capable of directing a specific query to the appropriate information server. Again, this feature is added to the Tagged Index Object at the expense of an increase in the size of the index object.

CIP background

The background of the CIP specifications is based in the LDAP, and it defines a mechanism for accessing a collection of information arranged hierarchically in such a manner as to provide a globally distributed database which is normally a DIT. A distinguished characteristic of an LDAP server is that it is expected to respond to requests that pertain to portions of the DIT for which they have data, as well as for those portions for which they have no information in their database. Normally, the response given will be a referral to another LDAP server that is expected to have more knowledge about the appropriate sub-tree. Typically, an LDAP server is configured with the name of exactly one other LDAP server to which all LDAP clients are referred when their requests fall outside the sub-tree of

the DIT for which that LDAP server has knowledge. Based on the LDAP server mechanism for data management, the CIP specification go further and defines a new mechanism whereby LDAP server can exchange index information that will allow referrals to point towards a clearly accurate destination.
In addition, the X.500 series of recommendations defines the DISP which allows X.500 DSAs to exchange actual information in the DIT. Shadowing allows various information from various portions of the DIT to be replicated amongst participating DSAs. The design point of CIP and the Tagged Index Object is more appropriate for the exchange of index information than is DISP.

DISP is more targeted at DIT distribution and fault tolerance. DISP is thus more appropriate for the exchange of the actual data in order to spread the load amongst several information servers. DISP is tailored specifically to X.500 (and other hierarchical directory systems), while the Tagged Index Object and CIP can be used in a wide variety of information about large parts of the DIT, it would require a huge database to collect all of the replicas for a meaningful portion of the DIT. Furthermore, as X.525 states: Before shadowing can occur, an agreement, covering the conditions under which shadowing may occur is required. Although such agreements may be established in a variety of ways, such as policy statements covering all DSAs within a given Directory Management Domain DMD. This is due to the case that the actual data in the DIT is being exchanged amongst DSA rather than only the information required to maintain an Index. In many environments such an agreement is not appropriate, and in order to collect information for a meaningful portion of the DIT, a large number of agreements may need to be arranged.
All of these requests or data transactions can reasonably be translated into LDAP or WHOIS++, and other directory access protocol queries. They can also be serviced in a straightforward manner by the users home information server if it has the appropriate reference information into the database that contains the source data. In this situation, the first server would be able to “chain” the request on behalf of the user. Alternatively, a precise referral could be returned. If the home information server wants to service the request based on the index information that it has on hand, this servicing could be done by any number of means.
Finally, when a collection of Information Servers are operating against a large distributed directory, is more efficient to allow them to distributed index information amongst themselves. For that purpose, they can male use of the CIP features and in this way their own searches can be carried out with some degree of efficiency. But before a Tagged Index Object can be exchanged, the organisation which administers the object consumer must reach an agreement on how the servers will communicate.
In the CIP, there is also a mechanism to indicate that there is some information which must be updated. This system is achieved with data composition which has an attribute value that may only be empty in the case of an incremental update that contains a “Update Block” in which the index object indicates that certain attributes of objects are being removed. The intention of the Tagged Index Object is to supply a snapshot of the current index of the directory. The information Server administrators must decide what portions of their database are appropriate for inclusion in the Tagged Index Object.

2.4 Integrated Directory Services

The Integrated Directory Services (IDS) [14]Working Group is chartered to facilitate the integration and interoperability of current and future directories into a unified Internet directory service. This work will unite directories based on a heterogeneous set of directory

services protocols (X.500, LDAP, WHOIS++, etc.). The problem in achieving this purpose is that there are many solutions and different vendors are adopting their own solutions.
One of the lessons that can be learned from observing many directory environments is that there is a simple, common problem - too many directories - and many technical and political ramifications in resolving the ensuing mess.

Trying to clarify the different solutions to reach the objective of an IDS, we can depict the advantages and drawbacks of the different models that we have seen above, and the possible solutions:

❑ The X.500 information model uses the concept of a single virtual directory embracing all the countries of the world. This is one of the most powerful aspects of X.500, the ability to distribute data and the management of knowledge across thousands, possibly millions, of directory servers world-wide, thereby creating a logical database of billions of entries
X.500 provides a viable distributed solution for IDS on the Internet and the only one demonstrated so far. It is the only open set of standards that define protocols, replication, security and an information model. Hence, the X.500 standards provide a solid, proven blueprint for a global distributed directory.
The drawback of the X.500 solution seems to be that it is too complex to be adopted in an IP environment.

❑ The alternative could be a LDAP based solution. While X.500-based directories provide a mechanism for managing global interconnected directories, LDAP-based directories only assume the existence of a directory model. LDAP provides simple access and manipulation of a directory. Some of the lightness of LDAP is achieved by removing rarely used but useful features of X.500. Putting the useful features back into LDAP will mean relaxing this lightness requirement, but at least would make the LDAP more suitable for our purpose.

Nowadays, the vendors are trying to find out the best solution adopting the best characteristic of each model. Actually, there is an intermediary process - a protocol converter - called an LDAP server, that gateways LDAP into full DAP has been proposed. But Increasingly X.500 vendors also implement LDAP directly into their X.500 server, thus eliminating the intermediary process. The recent implementations of standalone LDAP servers (known as slapds) talk to a directory database directly. But otherwise slapds are not incorporated into a distributed directory service and are “islands of data”.
There are other vendor announcements about LDAP access to their own database servers. This approach suffers from the same problem: each of these LDAP-enabled database servers are in most cases cut off from the other vendors' LDAP directory servers. They become (or remain) database islands. This approach does not provide a (transparent) multi-vendor distributed directory solution although access to multiple LDAP directories can be achieved through manual configuration.

An alternative solution is the model called distributed LDAP. In this case LDAP servers return referrals to other LDAP-servers. However there still remain several open questions about this approach. For instance, where do the distributed LDAP-servers get this information from, and far more importantly how is it dynamically maintained?.
Many of the standalone servers will "borrow" ideas and concepts from X.500, but not adopt them in such a way as to provide compliance with the X.500 standards. Hence it appears at the moment that, over the next few years numerous heterogeneous directory systems will be available with a varying degree of conformity to X.500.

Definitely, much of the value of open standards is in providing interoperability. If LDAP-enabled directory servers become islands of data that cannot be easily consolidated and managed, the ideal of IDS can not be achieved.

❑ Another suggestion is that X.500 could be extended for use as a backbone in a distributed LDAP world, using its knowledge management model to identify the appropriate server whether this can be an X.500 or LDAP server. The role of X.500 in most corporations deploying X.500-based solutions is to provide the backbone infrastructure either as a standalone system or with multiple servers replicating with each other.

Definitely, LDAP servers solutions that do not also implement X.500 are unlikely to improve multi-vendor server-to-server connectivity or replication in the short term, and a viable co-operation between the two types of directory servers needs to be established.

❑ Other techniques suggest including the proposed Common Indexing Protocol, that was already mentioned in the WHOIS++ system, to build the meta-directory. However, this is still at a conceptual stage and has not yet been practically proven.

2.5 The SCSP in this context

In the previous section, we have reviewed some actual database solutions and different models for data storage. We have also described the main purpose that is the system to achieve a global directory in IP networks. In those models we have seen that the X.500 standard itself provides the data replication in the X.525 specification. X.525 defines the DISP protocol for the data updating between servers. The problem that we have found in the X.500 directories is that is too heavy to manage using the DAP for the access to the information stored in the DIT.
The solution seems to be the LDAP protocol but we have also shown that if it acts as a standalone protocol it forms “islands of data”. A solution for that problem could be to fit the SCSP as the mechanism for the LDAP servers synchronisation. Thus SCSP could be the simple and popular way of providing this service to standalone LDAP or other kind of servers. In this way, those servers can offer for the first time a common facility amongst heterogeneous directory servers, combining a simple access protocol, the LDAP, with a simpler backbone than the X.500.

Chapter 3

Protocol Standard Specifications

In the previous chapters we have described the problem of data synchronisation that is the actual weakness in many distributed database systems. Thus, with the SCSP implementation we are trying to solve that problem. In the second chapter, we have reviewed the actual solutions and the framework for future answers for this problem. In all the possible solutions for information replication there are two main dilemmas, the first one is following a standardised solution with the correspondent excess of complexity, and the other alternative would be to adopt a simple protocol that is well accepted among the users and vendors. Hence in this chapter we will describe the SCSP protocol as the solution for data synchronisation/replication among distributed entities such taht it has both properties of being standardised and without any complexity.
Thus in this chapter, firstly we will explain the objective we want to reach using SCSP that is the implementation of a reliable mechanism to provide a temporal-inconsistency system among replicated databases in a network.
Secondly we will look at the antecedents for the SCSP Algorithm, which is based in the algorithm for the synchronisation of the OSPF 's routing database.
And finally, we depict the algorithm specifications of the SCSP itself and the policies for managing the updates of the information on each server or distributed entity.

3.1 Data Consistency among replicas

An important feature in a distributed database is the consistency, meaning that when a user changes or modifies an entry in the database that must be reflected in all copies of the entry carried by all replicas of the database. The inconsistency happens when two users are accessing the same data simultaneously in different databases whether one of them change the data and that information is not been updated immediately in the other database. Consequently, the other user will access to incorrect information. To achieve a strong consistency it is necessary to implement an atomic operation when a user has to change some information in the database. During an atomic operation the changes in the attributes of any entry contained in the database are propagated to all the replicas as a single unit. With the strong consistency different users will access the updated data in different places without any incoherent information. The other alternative is to provide a weak consistency which allows a temporary inconsistency but later on after few transactions among the distributed entities it will be fixed and a complete replication will be achieved. In the weak consistency there is a transitory period when the databases could have different information for the same entry, but the databases will store always the newest data because the protocol implement many mechanisms to guarantee that the data is not corrupted or old fashioned.

The main goal of this thesis is to achieve an efficient architecture for a software system that provides a temporal inconsistency or weak consistency[15] among replicated databases in Internet. There have been many other solutions before this one, but they required a strong consistency and for that purpose they needed special channels for the communication. In this case, we are trying to achieve a software system well designed to allow database to be highly available and to operate reliably under difficult conditions such as unreliable communication, network partitions, and host failures. In this system, an important objective is that the software itself requires minimal support from the DBMS or the other protocol that will use the SCSP. It is important also that the system performs an easy tuning of the architecture's basic algorithms to other particular environments.

Database replication is an important technique for constructing viable distributed systems. The proliferation of networks and distributed applications has increased the demand for geographically distributed replicated database. The typical database replication systems, which were designated for local area networks, are not well suited for the new Internet networks applications. Internet introduces new problems because it often contains high - latency, low bandwidth links, node failures, and network partitions.

A replication system contains mechanisms for update propagation and inter-replica consistency. The update propagation mechanisms must ensure that updates are efficiency and reliably updated to all the replicas. But reliable propagation is more difficult to achieve in an Internet because of the limited capabilities of the links. The consistency mechanisms must maintain consistency among the database replicas. The traditional approach of strong consistency, as we have already mentioned at the beginning of this section, requires all available copies of a data item to have the same value, and is difficult to achieve across Internet. Executing atomic commitment protocols over slow, unreliable links can be very inefficient, causing transactions to suffer long delays and reduce throughput. Common techniques for maintaining strong consistency, such as "primary copy" [16] and "quorum consensus" [17], severely limit availability during a network partition because they allow a particular updated item to be updated in at most one partition and they allow reads of only the most recent values. Furthermore, the performance of strong consistency replication techniques does not scale well to large numbers of replicas.

In contrast to strong consistency, a database replication system that provides weak consistency permits greater availability by allowing temporary inconsistencies to develop among replicas. These temporary inconsistencies are the result of multiple users independently updating different copies of a database, as well as the time necessary for an update to propagate to all replicas. The SCSP algorithm guaranties to resolve these inconsistencies and return the replicas to mutual consistency. The SCSP mechanism perform a weak consistency that represent a trade-off between database consistency and availability.

The SCSP system is designed to be flexible and adaptable in a variety of ways. Its modular design permits its components to be configured for different networks and allows its algorithms, such as the consistency methods, to be selected and tuned for different environments and applications. In addition, the SCSP algorithm can be used with many Database Management Systems because it requires no DBMS modification and makes minimal assumptions about internal DBMS structure and mechanisms.

3.2 From OSPF to SCSP

The origin of the SCSP algorithm, is in the OSPF protocol [18]. Thus the basic structure in the SCSP protocol is borrowed from the OSPF protocol, where the *Hello* protocol is used to

discover and maintain neighbour relationship. The OSPF update mechanism is implemented by the “Link State Update” and “Link State Acknowledgement” packets. Each Link State Update packet carries a set of new link state advertisements one hop further away from their point of origination. A single Link State Update packet may contain the link state advertisements of several routes. Each advertisement is tagged with the ID of the originating router. In addition, the OSPF has mechanisms for initial synchronisation of the databases.

Hello Protocol

The Hello Protocol is responsible for establishing neighbour relationships. It also ensures that communications between neighbours are Bi-directional. The Hello packets are sent periodically out from all router interfaces. Bi-directional communications are indicated when router sees itself listed in the neighbour’s Hello Packet. The Hello Protocol elects “Designated Router” for the network, that Designated Router controls what adjacencies will be formed over the network. The Hello protocol has different behaviour on broadcast networks than on non-broadcast. On Broadcast networks each router advertises itself by periodically multicasting Hello Packets. This allows neighbours to be discovered dynamically. On non-broadcast networks some configuration information is necessary. Each router that may potentially become a “Designated Router” has a list of all other routers attached to the network. A router, having “Designated Router “ potential, sends Hello packets to all other potential “Designated Routers” when its interface to the non-broadcast networks first becomes operational.

Database Synchronisation

After a neighbour has been discovered, a bi-directional communication is ensured. The “Designated Router” is elected for the purpose of establishing adjacencies over point to point networks and virtual links. The first step in bringing up an adjacency to synchronise the neighbours topological database, because in a link-state routing algorithm, it is very important for all routers topological database to stay synchronised. OSPF simplifies this by requiring only adjacent routers to remain synchronised. The synchronisation process as soon as the routers attempt to bring up the adjacency.

Each router describes its database by sending a sequence of *Database Description* packets to its neighbour. Each Database Description packet describes a set of link state advertisements belonging to the router database. When the neighbour sees a link state advertisement that is more recent than its own database copy, it makes a note that this newer advertisement should be requested. This sending and receiving of Database Description packets is called *Database Exchange* process. During this process, the two routers form a Master/Slave relationship. Each Database Description packet has a sequence number. Database Description packets sent by the master are acknowledged by the slave through the echoing of the sequence number. Both packets sent from each party contain summaries of link state data. The master is the only one allowed to retransmit Database Description packets. It does so only at fixed intervals, the length of which, is the configured constant *RxmInterval*.

Database updating

During and after database exchange process, each router has a list of those link state advertisements for which the neighbour has more up to date instances. These instances are requested in *Link State Request* packets. The Link State Request packets that are not

satisfied, are retransmitted at fixed intervals of time *RxmInterval*. When the database description process has completed and all Link State request have been satisfied, the database are deemed synchronised and routers are marked fully adjacent. At this time the adjacency is fully functional and is advertised in the two router 's link state advertisements.

The adjacency is used by the flooding procedure as soon as the Database Exchange begins. This simplifies the database synchronisation, and guarantees that it finishes in a predictable period of time.

Conclusions

We can see in the later description of the Hello protocol in the OSPF, that the basis of this protocol is quite similar to the SCSP. In fact, as we should see, the principal algorithm used in SCSP have been borrowed from this protocol to allow an efficient synchronisation. Thus with this characteristics we perform an stand alone protocol which can be implemented with any other routing protocol that does not have to take care of its database synchronisation. This feature makes the SCSP protocol more portable to work with any other protocol in common symbiosis over different architectures. As a complementary protocol, also reduces the latency time of the main protocol on this issues.

3.3 The SCSP Protocol Specifications

In the above sections we have illustrated the main antecedents for the implementation of this Software package, the main goal of which is the database or server information synchronisation based in the SCSP algorithm. In the former step we have chosen the architecture to solve the data synchronisation problem. Our election has been a weak-consistency mechanism for database replication over non reliable networks. In the latter section we have searched for a well tailored algorithm and the election has been the well known mechanism used by OSPF routing protocol for its database synchronisation. This algorithm has been developed as stand alone protocol in SCSP[1] specifications, by J. Luciani. The algorithm efficiency has been rather proven over many years of well functioning performance of the OSPF on Internet networks. Thus to extend the feature of the SCSP protocol as a suitable system for many other kinds of database replications, we have used that specifications to implement a useful software packet for data replication. In the rest of the chapter we describe the framework of this software project development and the specifications of the algorithm.

The SCSP attempts to solve the cache replication for distributed protocol entities, which are bound to a SG. An SG is formed by the Local Server, LS, and a set of directly connected servers, DCS. By this way, when a server become aware of a change on its cache information it must immediately propagate the knowledge of this event to all the servers, which are actively mirroring that state information. All this process must be done without putting under difficulties the server resources. For this purpose the SCSP protocol places no topological requirements upon the SG. Thus it does not requires neither routing nor path calculations. This would impose additional memory requirements for these purposes. Beyond that, the protocol only requires a minimum memory space for the Finite State Machine (FSM) allocation corresponding to each Directly connected server or DCS. There is a ***H**ello **F**inite **S**tate **Ma**chine* (HFSM), and a ***C**ache **A**lignment **F**inite **S**tate **M**achine* (CAFSM) per each server that form the SG. The amount of memory is a function of the number of SG components. The amount of traffic will also increase depending on the

amount of servers in the SG because of the major number of packet transactions that will be needed for the synchronisation is a function of the number of members in an SG.

Thus, the basic elements in the protocol are:

LS: Local Server where will run the SCSP protocol entities under observation.

DCS: Direct Connected Server are the peer of the LS on each FSM, which are implemented to perform the SCSP individually between each LS in the whole SG.

SG: Server Group is composed of the LS and a set of DCS. It is identified by the SGID. This SGID will maintain independence between several SCSP processes running in the same physical network and on the same network nodes.

RS: Remote Server is an extra element, that are part of the SCSP nomenclature and indicate that it is an external element of the group.

All those elements are represented Figure 1.

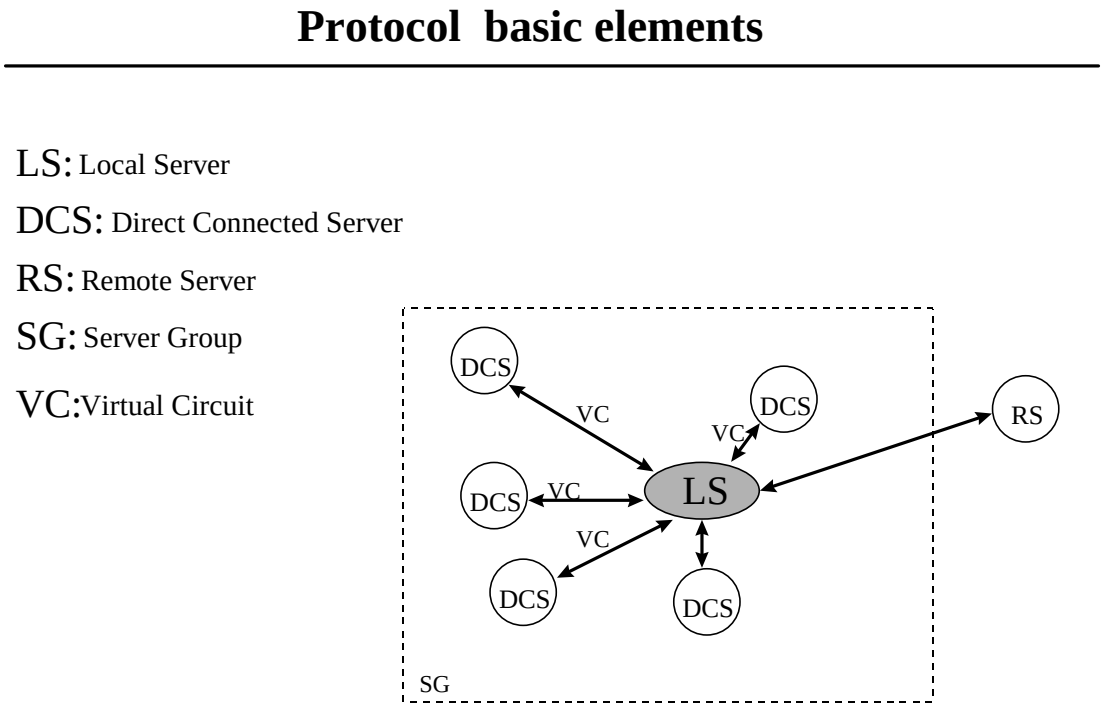
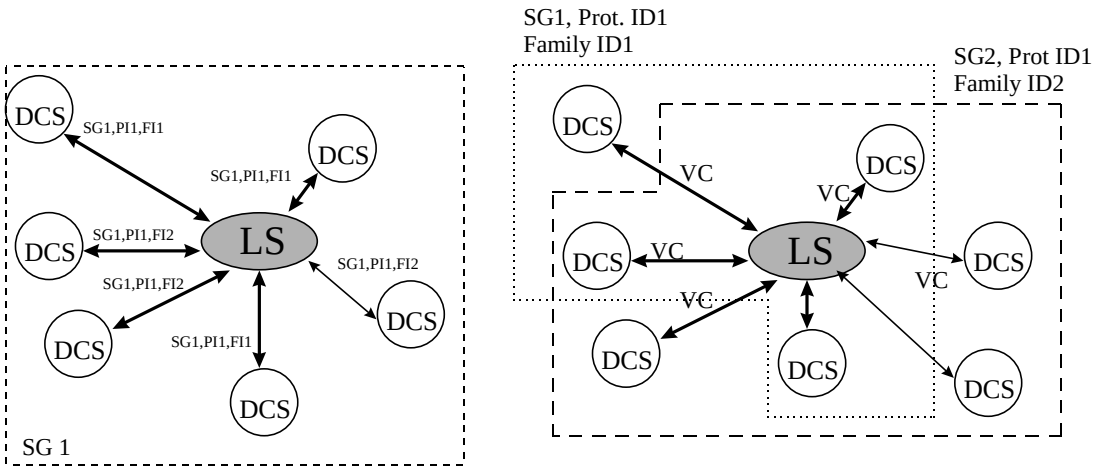


Fig 1. Basic Elements in the SCSP Algorithm

The SCSP protocol is implemented to allow running several instances of SCSP in the same group of nodes. This feature is depicted in Figure 2a. It is also possible that the same server or node belongs to different groups, then it is necessary to distinguish the instance of each group running in the same node. This characteristic is shown in Figure 2b. In this way, different groups can overlap and several instances of SCSP can run on the same set of network nodes. To achieve this feature we have implemented an initial configuration module where the protocol can read the correspondent data to fit the basic information for each FSM. This data is based on the MIB definitions [19] for protocol management that will be explained in the environment analysis chapter for the software project development.



a) Various instances of SCSP running in the same group

b) Instances of SCSP in the same server which belongs to different SG

Figure 2. Different instances of SCSP running in the same server

The SCSP algorithm provides the mechanisms necessary to allow the overlap of independent instances of the protocol running on the same node belonging to the same or different SG.

The protocol is identified by its Protocol Identifier (PID) and in the same way the group is identified by the Server Group Identifier (SGID). Thus with the PID/SGID pair plus an additional value called the Family Identifier (ID), we achieve our proposal to identify each instance of SCSP in the case that would be different instances of the same protocol running in the same group, or several instances of the SCSP if the same node belongs to different groups.

In several cases this mechanism is also appropriate to avoid situations where a single server could receive information from different servers which are implementing the same protocol. This situation happens quite often when a server is situated on the border of various groups domain or in the middle of different scenarios, then the server can receive wrong packets that it must drop out after checking the right PID/SGID or the Family ID. For that reason it is important to implement this auto-configuration module to read that information to check if they are receiving the packets from the servers that belong to the same SG. This configuration *class* will read the same text file in all the servers of that SG and this class also will read another text file with different PID/SGID, if it is going to run another SCSP process among the servers of another SG. An example of how the Family ID can be useful is the case when we have both relatively static and relatively dynamic information in the database. Different state of timeouts of the protocol can be tied to two different Family Ids in the configuration file for this database. The result is more efficient use of the network resources for database management. We can run one instance of the SCSP to synchronise the static part of the database. We can run another instance of the SCSP in the same LS that will read other configuration specifications to perform the replication for the dynamic data.

The program is capable of using an auto-configuration service, where it can read the information about the PID, SGID, Family ID and many other values that it will use in its normal operation. All that information is read from external files that can be managed directly by the network operator or any other program, which can write directly over that

text file. This module is implemented in the same way than the rest of the program using class structures for its development. Thus the configuration class used by the auto-configuration module is shown in the next Figure 3.

Configura class:

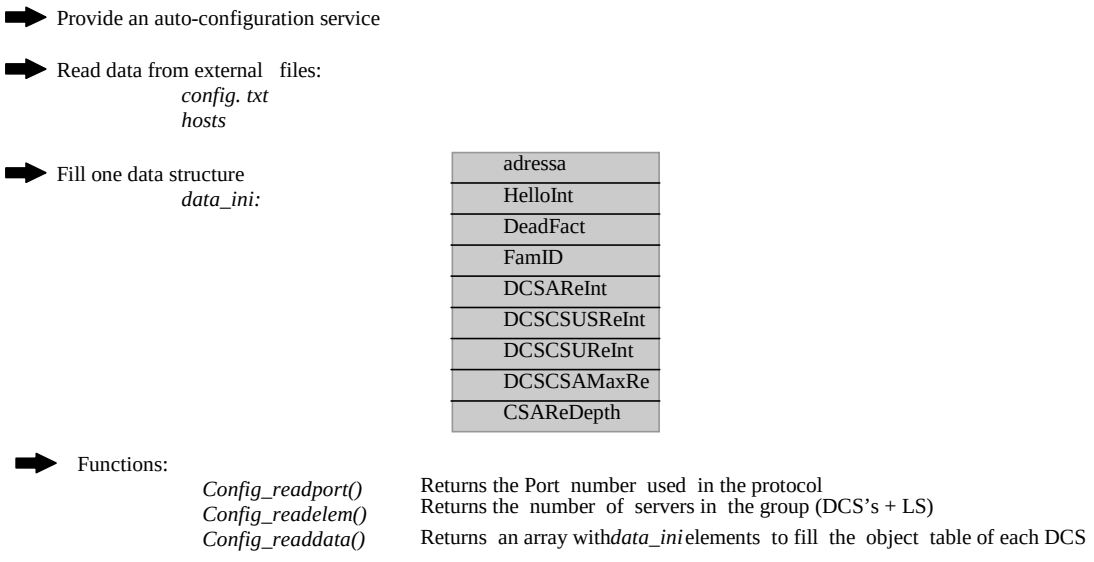


Fig. 3. Configuration class

All these information identifiers corresponding to each server and other information to fit on the machine tables are based on the MIB [19] standards. For this purpose we have implemented a series of basic *classes* to deploy on the background and to define the framework for the whole protocol implementation based on these definitions. Those definitions are based on the SMI specifications that will be detailed later in chapter 4.

After the definition of the basic elements of the protocol and the fundamental components, like the identifiers of the protocol, we are going to describe the SCSP protocol algorithm and the different sub-protocols that compound the whole specification.

Basically the SCSP is divided in three independent but interrelated protocols, which are:

- ➢ Hello Protocol
- ➢ Cache Alignment Protocol
- ➢ Cache State Update Protocol

3.3.1 Hello Protocol

This is the initial sub-protocol of the whole implementation. In this step two devices in the group determine if they can talk to each other.

An LS has associated a Finite State Machine (HFSM) with each of its DCSs. That machine monitors the state of the connectivity between the LS and the DCS. For this purpose we have implemented an independent object for each FSM which compose the Hello Protocol. When the connection between the LS and its associated DCS is available, the State Machine transitions to the Bi-directional state. In this moment the Cache Alignment protocol starts to request the update information on each database. All those steps are described in the flow diagram depicted in Figure 4.

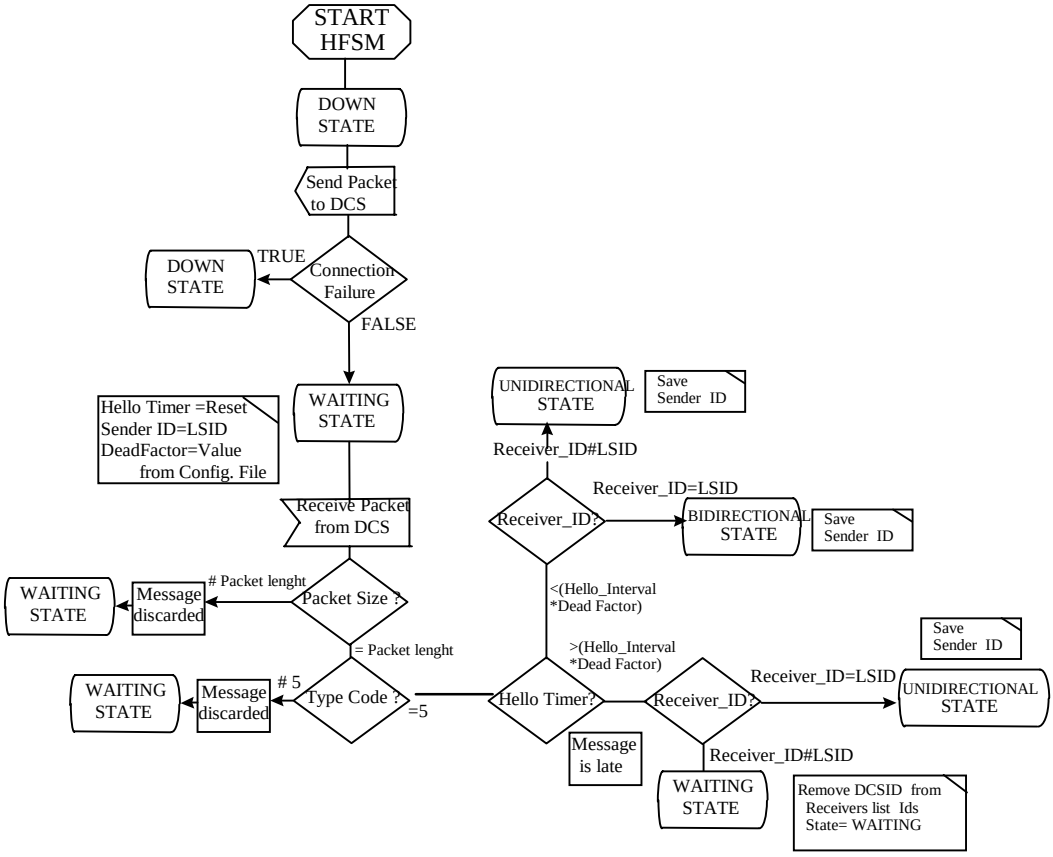


Fig. 4. Hello Finite State Machine, flow diagram

The Hello Protocol has four steps in the finite state machine.

- ❑ Down State
- ❑ Waiting State
- ❑ Unidirectional State
- ❑ Bi-directional State.

Hello Finite State Machine

This machine has been developed with the specifications as guidelines, so it has the following phases:

i) *Down State:* It is the preliminary state where the machine is just started.

ii) *Waiting State:* In this moment the Hello machine have just checked the physical connection between the LS and its DCS. When the Hello Interval expires the machine sends *Hello messages* packets. Those packets are formed with an additional list of addresses of other servers from which the LS has received other Hello messages during the past . Then FSM wait any incoming Hello message from any other server to check which kind of connection has the LS with that DCS. It depends of whether the message arrives during a period of time called *Hello Interval* or in another extended period composed by the multiplication of the Hello Interval by another factor called *Dead Factor*. Thus if the packet arrives during the latter period of time the packet is considered valid. In the other cases it is considered that the packet has arrived late. Then in this case the DCS could be stalled, or there will be an Unidirectional connection between both servers. Otherwise if the packet arrives in time the Hello machine transitions to Bi-directional state. It means that we have achieved a total connectivity between the two servers.

iii) *Unidirectional State:* When the LS receives a Hello message in time (it means during the Hello interval) and the LS address is not included on the additional list of odf addresses of other servers that the LS has received in one of the packet's field, then the LS transitions to the Unidirectional connection.

iii) *Bi-directional State:* This state is achieved when the Hello message received in time includes the LS address in the field of the packet which contains the list of addresses of other servers. It means that there is a bi-directional connection between both servers. Another possibility to reach this state, is when the LS receives a message after the Hello Interval expires but it is received within the interval comprehended by the Hello Interval multiplied by the Dead Factor and it also contains the LS's address in the packet. Each Hello machine has associated a Cache Alignment machine, which transitions from its Down state to the Mater/Slave Negotiation state.

3.3.2 Cache Alignment Protocol

The purpose of this part of the whole protocol, is the database alignment. In this process the neighbours exchange summaries information about the entries in their database. Summaries are used since the database itself is potentially quite large. Thus based on these summaries the neighbours can determine whether there is some information that each needs from the other, if that is the case, the pertinent information is requested and provided. At the end of this phase, the two neighbours will have the information in their database totally updated again. The implementation of this protocol is made in the same way as the Hello protocol. There is also one FSM associated with each LS-DCS pair. This machine has its own functions to manage the correspondent data that it will receive.

A description of all the algorithm of this part of the total protocol, is illustrated in Figures 5 and 6. Figure 5 shows the initial steps of the algorithm till the FM reaches the Cache Summarize State.

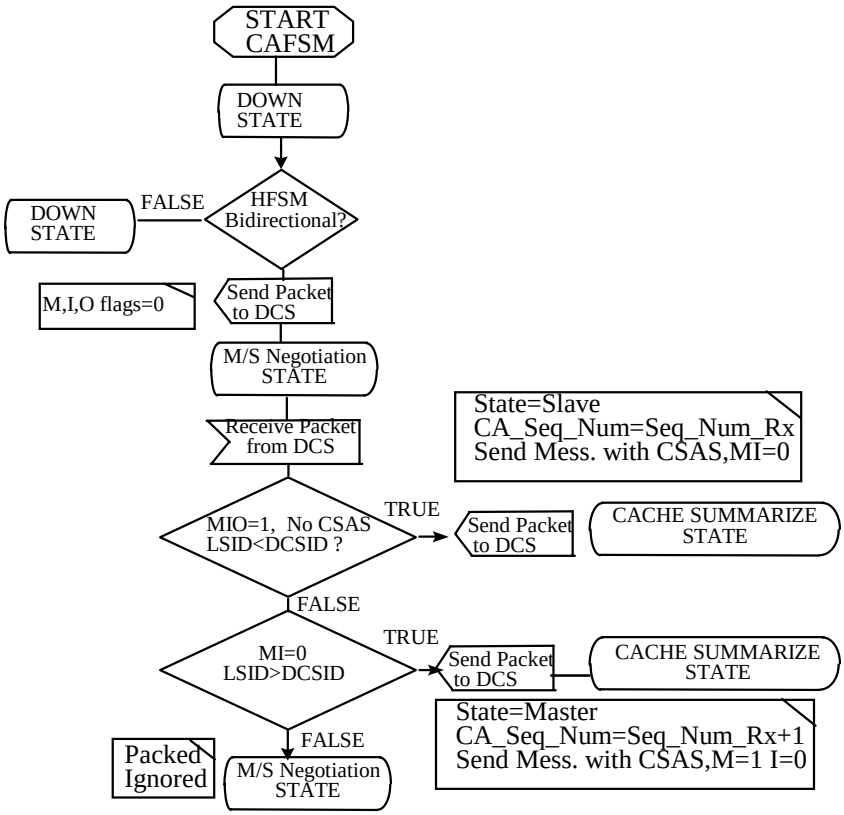


Fig. 5 Cache Alignment Finite State Machine (Master/Slave Negotiation)

Figure 6 shows the latest part of the algorithm, when the FM finally reaches the Aligned State.

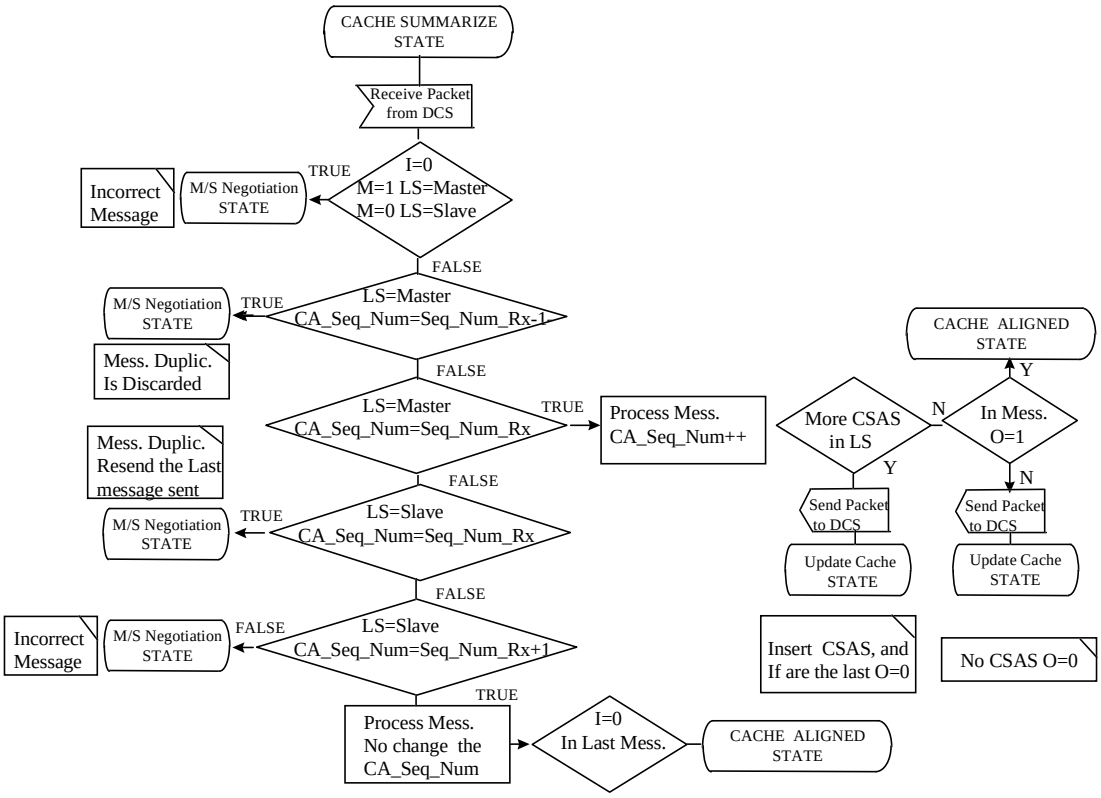


Fig. 6. Cache Alignment Finite State Machine
(Cache Alignment process)

Cache Alignment Finite State Machine

There is a CAFSM in the LS associated with each of its DCSs on a per PID/SGID basis. This machine monitors the state of the cache alignment between the servers in the same way than the HFSM were monitoring the state of the connection between the LS and the associated DCS. Quite similar to the HFSM, the CAFSM has different states that we have seen in figures 5 and 6. Those states are:

i) *Down*, Is the first state in the CAFSM. In this state the machine will keep inactive until the HFSM transitions to the Bi-directional state.

ii) *Master/Slave Negotiation*, In this state, when the CAFSM receives a packet, the machine can perform three different transitions:

> ➢ The first one, is when the CAFSM receives a packet with the flags field set and no other records in the rest of the packet's body. It means that the LS must take the role of Slave in the rest of cache information transactions. The LS informs to its DCS associated about the decision adopted sending a packet to its DCS with the Master's flag off. At the same time, if the LS's cache is not empty and there are CSAS records to send, it will fit all or part of them in the same packet and then the CAFSM transitions to the *Cache Summarise State*.
>
> ➢ The second mode, is when the LS receives a packet where the flag position, which indicates the Master flag is off. It means that the LS can take the role of

Master. If there are CSAS records in the rest of the packet the LS can process it. After that, the LS will send a new packet indicating its new status as Master and if there are also CSAS Records to send, all or part of them will be fitted in the same packet and the machine changes to the *Cache Summarise State* as in the first mode shown.

- Finally, when there is neither the first case nor the second the packet received will be discarded.

iii) *Cache Summarise State*, The connection and the role of each one have been already fixed for posterior transactions. Then only remains to start the cache alignment between the LS and its DCS associated in the respective HFSM and CAFSM. First of all, in this process all the packets which have something wrong will be discarded. We could find errors such as some discordance in different information fields of the same packet like the packet size field different from the real size, a mistake in the CA Sequence Number, or that the Master/Slave flags are wrong according at the real status of the LS. But otherwise, if the LS receives a packet with the CA Sequence Number and the M/S flags are correct then the message will be processed and if the LS's cache has some CSAS records to be sent, the LS will fit all or part of them in a new packet to the DCS (indicating that there are still some packets to send in the LS with one of the Flags field). In the case that there are no more CSAS Records to be sent the CAFSM transitions to the *Update Cache State*.

iv) *Update Cache State*, In the CAFSM there is a structure called **CRL** (**C**SA **R**equest **L**ist). It forms a buffer where the Cache State Advertisement (CSA) are stored. It contains those cache entries which we have to update in our LS's cache from the same data as the DCS has in its cache. Thus, if the CRL of this DCS associated at the CAFSM is not empty upon transition into the Update Cache State, then the LS solicits the DCS to send the corresponding CSA records. For that purpose the LS forms CSU Solicit (CSUS) messages from the CRL. This request will be sent to this DCS and it will respond by sending to the LS one or more CSU Request messages containing the information on its cache referent to the petition in the CSU Solicit that it has just received in the request. Then upon receiving the CSU Request, the LS updates its cache with the newer cached information that contains the CSU Request. At this point the LS will send one or more CSU Replies to the DCS. This process continues until all the CSA Records corresponding to the CSAS that were in the CRL have been received by the LS. At this moment the LS has a completely updated cache. Then the LS transitions the CAFSM associated with the DCS to the Aligned State.

v) *Aligned State*, The LS will behave according the *Cache State Update Protocol*.

3.3.3 Cache State Update Protocol

A flooding state is performed and any new learned information is sent to all the neighbours as soon as possible except to the one that has provided the information.

Figure 7 depicts the behaviour of the protocol during the Cache State Update sub-protocol

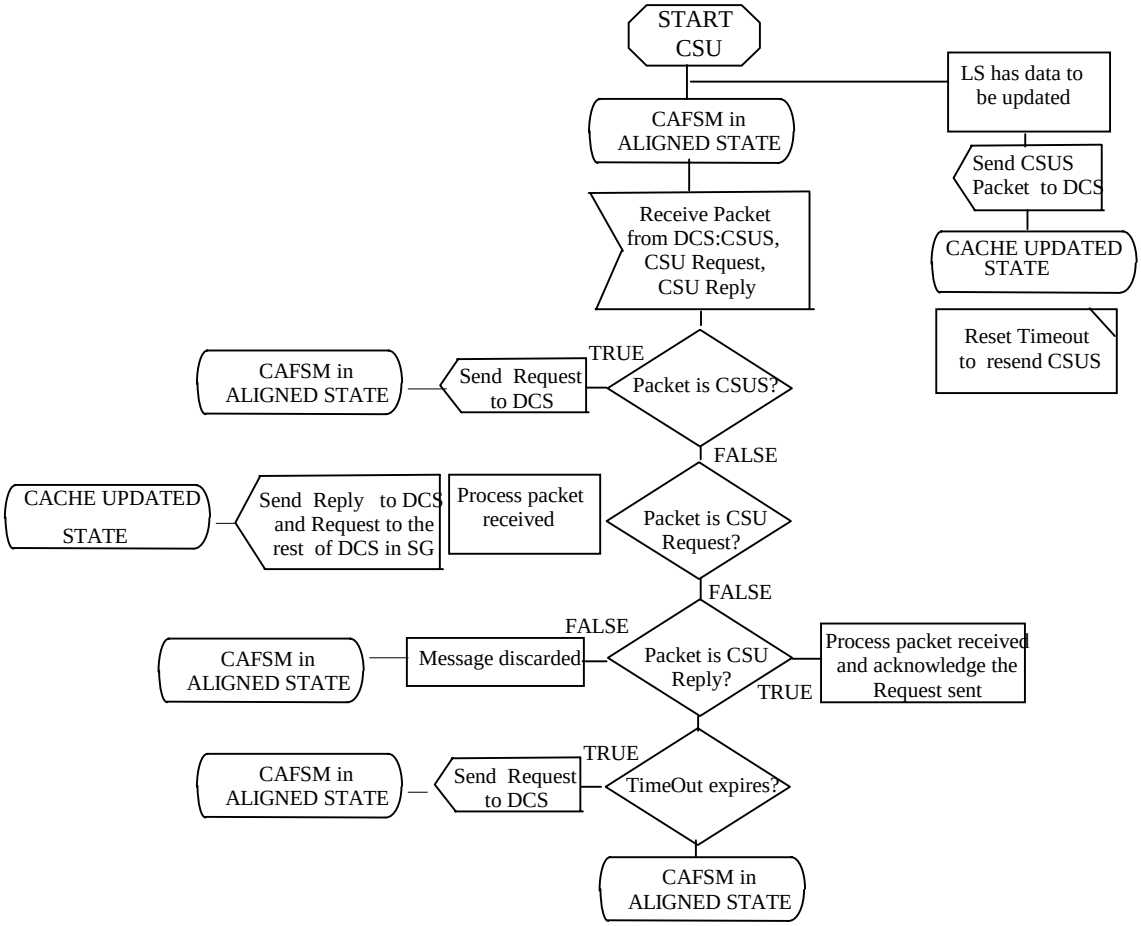


Fig. 7 Cache State update sub-protocol

This protocol uses the Cache State Update (CSU) messages to update dynamically the state of cache entries in servers when a new event has taken place (adding or removing an entry from one client or if the cache has been administratively modified,…). The CSU messages contain zero or more CSA records which have a piece of the state of a particular cache entry. This transaction is allowed only when the corresponding DCS's CAFSM is either in the Aligned State or the Update Cache State. In this protocol two kind of CSU messages are flowing: *CSU request, CSU Replies*.

❑ The former is sent from an LS to one or more DCSs for two reasons:

> ➢ When the LS has received a CSUS message and must respond to this DCS that has formulated the petition of updating information.
>
> ➢ The second case, is when the LS becomes aware of a change in a cache entry. This event could be possible in two ways, either through receiving a CSU Request from one of its DCS (in this case it has to send a CSU Request to each of its DCS s except to the DCS from which the newest information has been received. The other possibility is when the change is in its own cache. In the latter case it has to send a CSA Request to each of its DCSs and the new state is noted in a CSA record which is appended to the end of the CSU Request message mandatory part. In this way, the changes are propagated throughout the whole SG.

❑ The latter CSU message are used when a LS receives a CSU Request from one of its DCSs. Then the LS acknowledges the CSA records which were contained in the CSU

Request by sending a CSU Reply. This reply contains one or more CSAS records corresponding to those CSA records which are being acknowledged.

Thus in the above paragraphs we have described the SCSP specifications and the headlines for the implementations. Hence in the next chapters we focus our work in the software implementation which final behaviour must be the same that we have seen in the SCSP specifications.

Chapter 4

Protocol High Level Design

In this chapter we describe the basic environment and initial steps for any protocol development. We explain the purposes that we want to reach in the implementation and how we can design a portable software implementation of the SCSP specifications.
Afterwards, we explain briefly the Operating System (OS) facilities to allow that the SCSP implementation performs a data synchronisation efficiently. In this point we also describe the language chosen for the implementation. In this chapter, we present the basic data structures using the ASN.1 notation. Hence, we establish the object background for the implementation and achieve an open implementation framework.

4.1 Implementation Introduction

In the third chapter, was described the protocol specifications and the basic mechanism to achieve the servers synchronisation. In that chapter we explained how each of the three sub-protocols, that compound the whole SCSP protocol, is individually performed by a different FSM that will check the state between the LS and the DCS.
Thus before starting to describe in further details each part of the protocol implementation, in this chapter we show the global behaviour. Then in Figure 8, we can see the main modules of the implementation.

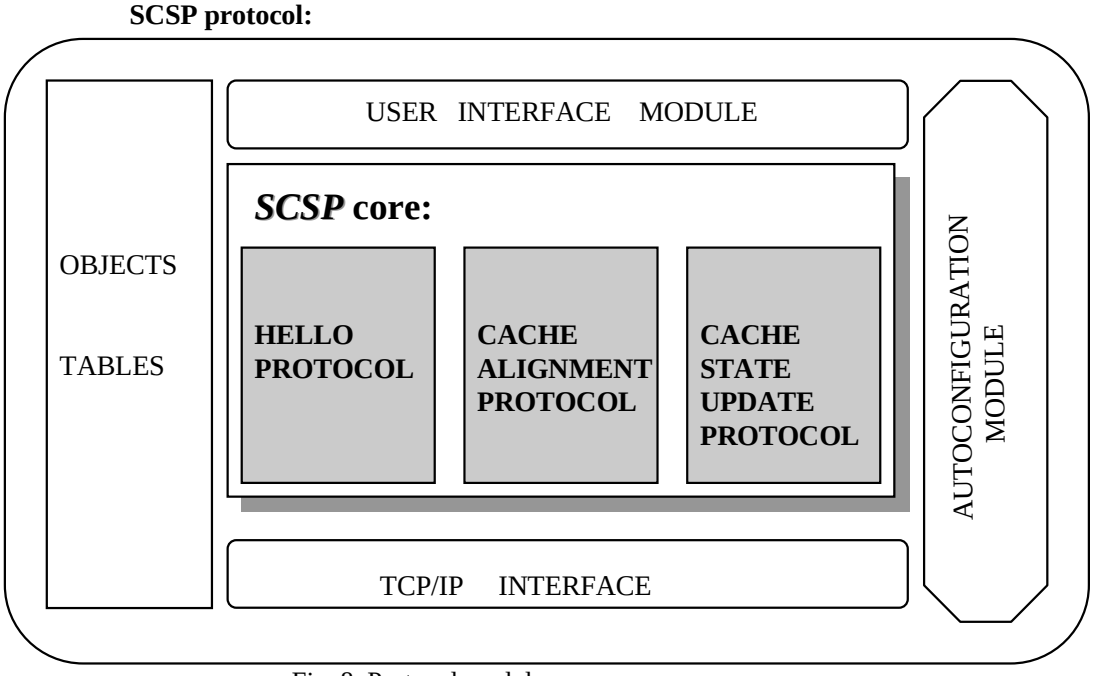


Fig. 8 Protocol modules

In the core of the software package is the implementation of the SCSP itself. In this core we have all the FMSs for each sub-protocol implemented in separate classes. Those classes perform the basic algorithm described for each sub-protocol in the specifications.

The main goal of this project is design software portability. For that purpose we have tried to implement the program in different layers. Thus we can develop in the upper layer the algorithm totally isolated from the functions that will access directly to the OS calls. Hence, we can achieve a modular package where in the case to change to another scenario we only have to change the pertinent functions that manage directly the calls over the specific platform where it is running. We can see the different layers of the implementation in Figure 9.

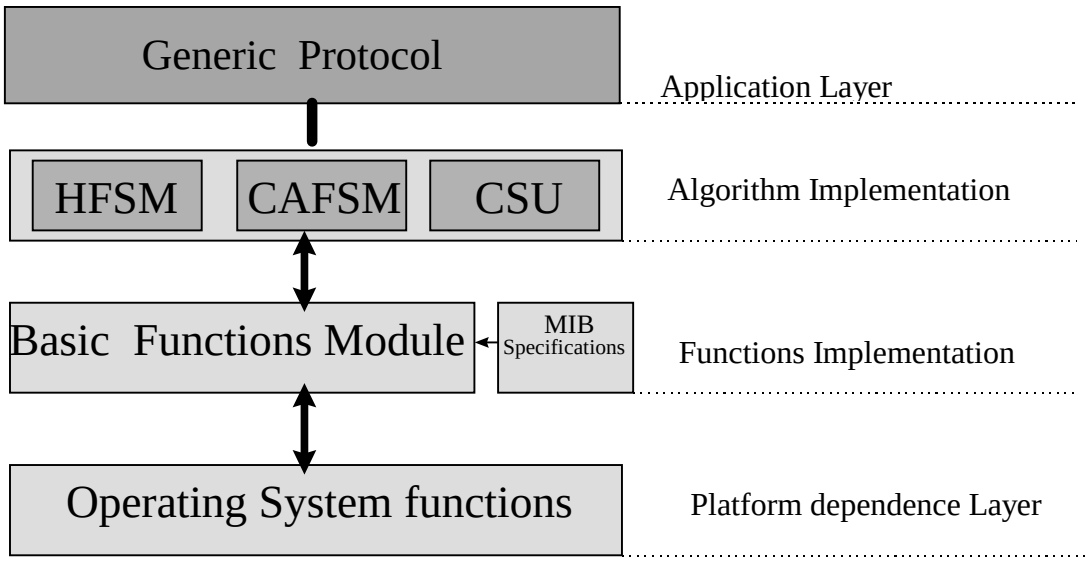


Fig. 9. Different Layers in the program implementation

Hence, we have already established the framework for the implementation. It means that we will try to follow the SCSP specifications but we also have to obtain a well tailored software to perform an efficient job over different scenarios. Thus in the above paragraph we have described the modularity we want to achieve in the implementation and the independence from the specific machine that the SCSP will run. Now, in the rest of the chapter we will try to go deeper in the OS details and the background for the definition of data structures using the different notations required for that.

4.2 Operating System Issues

The principal issues about any implementation are what language or compiler must we use to develop the network software. The first answer is quite often C, because it is the dominant language [20] for implementing systems software in general, and networks software are most certainly systems software. This is not accident, C gives the ability to manipulate data at the bit level (it is often called bit twiddling), which is necessary when you are implementing network software. The second question, is what operating system (OS) will the network software use. It depends on whatever OS runs the node in question. Perhaps the most extended OS in every node, host or server is the UNIX or LINUX systems, so it is necessary to understand the role played by the OS. Simply stated, network depends on many of the services provided by the OS, and in fact network software often runs as part of the OS. However we have tried to minimise this dependence using only few system calls

to achieve software portability. Anyway, network protocols have to call the network device driver to send and receive messages, the system's timer facility to schedule events, and the system's memory subsystem to allocate buffers for messages.

In addition, network software is generally a complex concurrent program. It depends on the operating system for process support and synchronisation. Hence, we will need to know few system calls for Timers and communications modules.

Thus, first of all we need to establish clearly all the modules we will need for the implementation. Afterwards, we describe the interaction among the different procedures or modules. For that reason it is essential to know the OS where the program have to run.

While each OS provides its own set of routines to read the clock, allocate memory, synchronise processes, and so on, we need to know these features in our OS very well to translate those calls into an objects abstraction which manage directly the system calls and will return the corresponding value to next level protocol calls, where we can concentrate on the algorithms that make up the SCSP specifications. From this point of view, it makes the protocol implementation more efficient than the case that we would have to make a direct system call from the normal level where the protocol is working. Thus, this will establish an initial groundwork needed for the upper level abstraction where the protocol will be running.

This background is simply a codification of the fundamental components where the protocol's algorithms can lie and where the definitions are taken from the SNMP and ASN.1 notation to give our implementation openness for posterior designers.

Thus, based on this guideline and also with the protocol specifications which we have already seen in the last chapter, we know that the implementation requires a series of finite state machines for each connection between LS-DCS. Thus, we need a language which provides an object-based framework for our implementation to obtain an abstraction of the objects hiding its (low level calls to the OS) performance to the rest of the implementation. Thus, those objects may be conveniently thought of data structure with a collection of operations that are *exported*. That is, we make those objects available for invocation by other objects or from the next step in the whole SCSP development.

Language Implementation

The fashionableness of object-based programming has been accompanied by a proliferation of object-oriented languages. One of them, probably the most well-known example, is the C++. That is the reason why we have chosen C++ for the implementation. All good computer scientist worship the good of modularity, since it brings many benefits, including the all-powerful benefit of not having to understand all parts of the problem and to deal with this we use the features of C++. The Alternative for C++ would be *Java*. The main benefits of JAVA concerned is that by writing as much of the software as is feasible in Java, it becomes platform independent. Java's virtual machine and compiler act as an isolating layer between the OS and the software. Otherwise, we can argue that it is possible to write OS independent code with C and C++. We have chosen to base our implementation on the characteristics of the software from the "Free Software Foundation" (the makers of GNU software) that actually can probably compile on more OS than Java has support for. Anyway, writing C and C++ code requires skills and knowledge that first have to be acquired. Also the compilers deal with OS and processors basics that vary a lot depending on the system. In those cases, we get into trouble with libraries, system calls, differing system header files and many things like that.

The drawback of Java is that it needs to do a lot of text processing and a fair amount of computing and operating system interaction is involved. Currently, the performance of Java make it less suitable for writing applications that need raw computing speed. Thus in this case we chose C++ because we felt that it is the best option for providing enough speed to the protocol. And with the object's definitions, which principal descriptions are based in the MIB specifications we provide our software the open characteristics for any protocol based on OSI guidelines. At the same time, we have an abstract implementation of modules that can invoke and return operations from other objects from the algorithm. Hence, the code has been implemented in the way that can interpret messages and maintain a state associated with the protocol situation at each moment.

OS issues and process types

Typically, any protocol implementation has to be concerned about a lot of OS issues. Most Operating Systems provide an abstraction called *process*, or alternatively *thread*. Each *process* runs largely independently of the others, and the OS is responsible of making sure that resources, such as address spaces and CPU cycles, are allocated to all the current ones. The process abstraction makes it fairly straightforward to have a lot of things executing concurrently on one machine that is the reason to bound the implementation to a well-known system. We need this processes mechanism for the implementation because the SCSP is a non stand-alone protocol, it serves like a complementary service provider for other protocols which will make use of the SCSP features for their database synchronisation. Hence, each application is carried on its own process and some additional goals inside the OS might be executed such as other processes. An inconvenience exist when the OS stops one process from executing on the CPU and starts up another one. This change, called *context switch*, means the corresponding waste of time for the reallocation of new resources, decreasing the global efficiency of the program.

When designing a protocol implementation the first question is to identify the processes and their behaviour. There are essentially two choices:

❑ The first, which is called ***process per protocol*** model, where each protocol is implemented by a separate process to achieve the aim of an efficient symbiosis between the *main* protocol and the SCSP in the same server, or host. In this way, a message can move up or down the protocol stack and also can be passed from one process/protocol to another. At the same time, in the SCSP we need to receive and update directly the data in the *main* protocol cache or database. How one process/protocol passes the message to the next process/protocol depends on the support that the OS provides for the inter-process communications, and in this case it is through the shared memory management [21]. There is a simple mechanism for enqueuing a message with a process, the important point, however, is that a *context switch* is required at each level of the protocol with the consequent time consuming operation. That is why we choose this kind of implementation only for the inter-process communication between SCSP and the *main* protocol, just to minimise those time expenses.

❑ The alternative, which is called the ***process per message*** model, treats each protocol as a static piece of code and associates the process with the messages. When a message arrives from the network, the OS dispatches a process which is responsible for the message and moves the message up to the relative high level protocol, according to the headers contained in the message. At each level, the procedure that implements that protocol is invoked. The eventual results obtained are sent to the next protocol level. For outbound messages, the application process invokes the necessary procedure calls until the message is delivered, and in the SCSP this procedure is quite efficiently succeeded with a I/O signal handler.

The *process per message model* is generally more efficient because a procedure call is an order of magnitude more efficient than a context switch. The former model requires the expense of a context switch at each level, while the latter model costs only a procedure call per level. For that reason, we choose the latter model instead of *process per protocol* for the different implementation levels inside the whole protocol, so we have this characteristic for the implementation of the different sub-protocols (Hello Protocol, Cache Alignment Protocol and Cache State Update protocol) that compound the whole SCSP protocol. With this model we manage the messages among the FSM directly without any *context switch*, thus a message from any *high level* part in the FSM does not reach down to receive a message from any *low level*, instead of that, the functions that perform the *low level* in the FSM deliver directly the message to the *high level* FSM protocol. This is because the receiving operation is being executed in the *high level* protocol, simultaneously with the sending function in the low level part of the protocol. It means that all the processes are in the same *context* and the *high level* FSM is waiting for new messages to arrive, which would result in a lower cost for that protocol switch context.

4.3 Protocol development states

Now, we can show the different states in the protocol development. The subsequent steps are generic guidelines to achieve any well suited protocol implementation. Mainly there are four states in any software project, the first is to clarify what kind of language we have to use for the implementation. The second one, is to know over what kind of OS the program has to work. Those points are already explained in this chapter. Thus, in the rest of the chapter we are going to describe the next two points involved in the development of the program itself.

Graph and Background

The first step in the protocol implementation, is trying to clarify the main functions and the algorithm that we are going to use. For that purpose, a graph is developed to see the right position of the protocol among the different Internet architectures and protocol definitions. Figure 10, is illustrates all the elements that we will need for the project development.

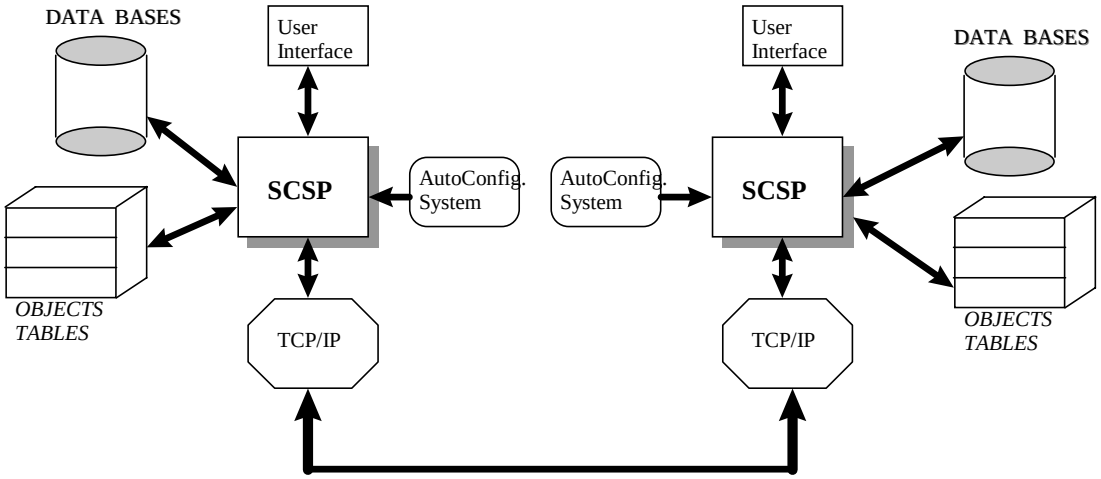


Fig. 10. Complementary modules for the SCSP implementation

This configuration graph reflects the operations and objects that are exported externally. We also have to know where the SCSP is allocated in the global structure of standardised bodies (ISO and the IETF). It is important to define the interfaces between those existing IETF blueprints, where the SCSP functions will lie. Thus, among the standardised set of protocols above the IP mechanism, like TCP or UDP, the characteristics of the SCSP specifications suggests that it is possible to use as “hard core” of the program implementation the UDP protocol. The reason is that the UDP protocol supports a more flexible mechanism which make it easier to plug our protocol in the IETF architecture. It is also quite simple to make use of the UDP features to obtain a rapid and efficient protocol, without taking care about strict and reliable connections which slow down the protocol functions.

Objects definitions

We have to implement the objects and routines to be accessible from the upper levels of the protocol guided by the protocol specifications. We also have to define the interface of each operation, and the performance which is expected from each module to export to the rest of modules in the program. All the data structures must follow the protocol directives. We can see an example of those data implementation in Figure 11.

Packets format:

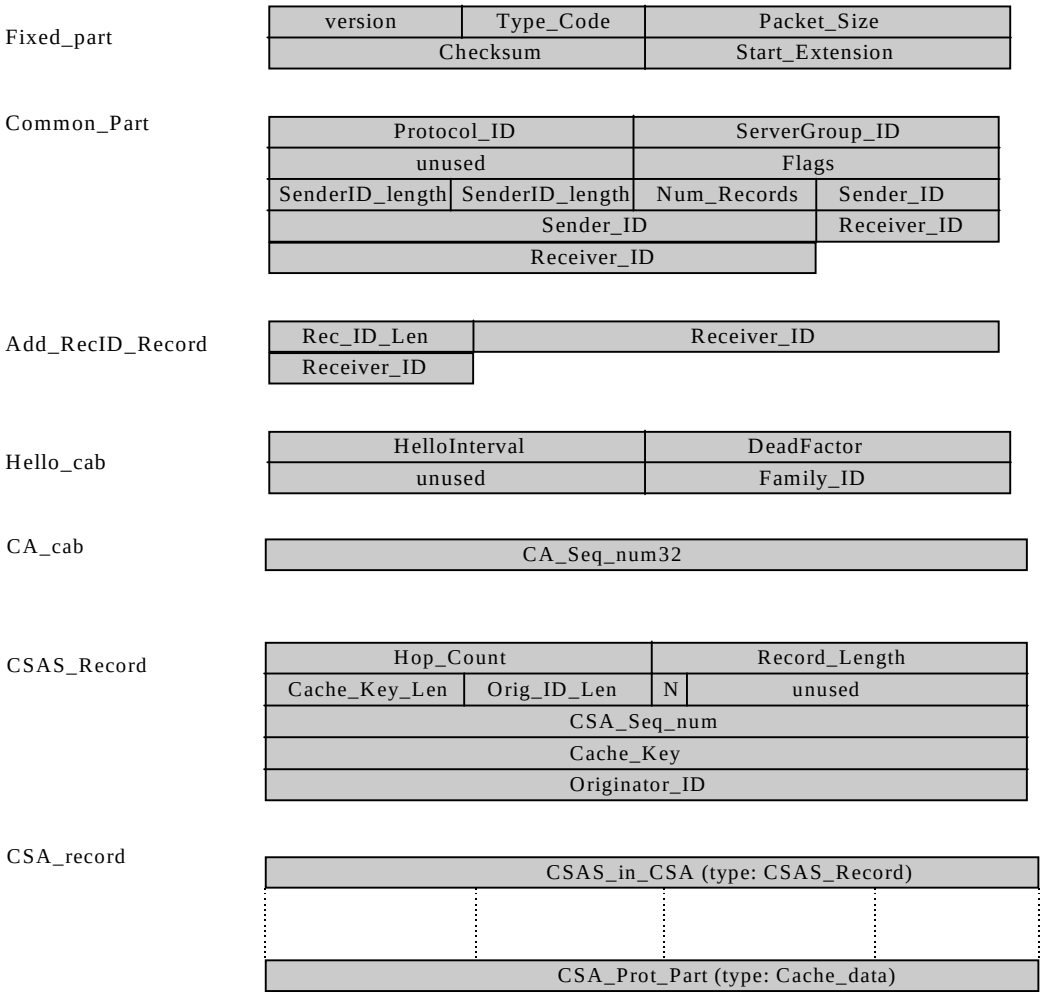


Fig. 11. Protocol specifications for data structures

Those definitions are based on the SMI objects with the ASN.1 nomenclature. An example of this type of notation is depicted in Figure 12. The decision to follow the SMI descriptions is to be able to create a framework for the later development of the SNMP management through a Proxy agent. Afterwards, the corresponding events will be defined to be forwarded between different modules or structures. The messages are implemented in fixed data structures which will be used to form the corresponding messages inside the Finite State Machines, which will be utilised in each protocol step.

Objects definition:

➡ **Objects description**

We need some previous data structures for the posterior implementation of the managed objects for hosts or routers that use SCSP.These structures follows the Structure of Management Information (SMI) for the Simple Network Management Protocol Version 2 (SNMPv2)

- *Integer32* , and *INTEGER*: represents integer information between -2^31 and 2^31-1 inclusive it is implemented by ***int*** value
- *INTEGER* (0...64): represents integer information between 0 and 64 inclusive it is implemented by ***integer_6*** class.
- *OCTET STRING*: represents text data, there is no SMI specified size for this struct in the Standard specifications, but it have a fixed limitation to 65 characters, it is implemented by ***Octet_String { char[65] }*** Structure
  (The Ip Address is a 32-bit address, and is represented as an OCTET STRING [4])
- *Unsigned32*: represents integer value between 0 and 2^31-1 inclusive, it is implemented by ***unsigned short*** value
- *Counter32*: represents a non-negative integer which monotonically increases until it reaches a maximum in 2^32-1 after that it starts again from zero, it is implemented by ***Counter32*** class.

Fig.12. Basic SMI definitions based on ASN.1 notation

After the definitions of the modules and basic structures, we also have the basic events and operations in the protocol that are:

Time out release which is sent to the correspondent FSM. There is an array of structures which achieve a systematic time counting which update its internal values through a system call. Those structures have been implemented without involving any tight system related to the OS that would be so much dependent from the platform. The goal of this structure is to maintain a stable and reliable time system to be checked at any moment with its own functions. Otherwise it can be managed independently from each FSM without the system requirements.

Another event is the **message reception** which is taken care of by the I/O signal handler. Its basic function is to store the address where the message is going to be allocated in a buffer. Then, the FSM is able to access directly to that message and process it. The headers are stripped from messages and then interpreted inward on each FSM. The details of that process are hidden to the rest of the protocol. All the functions to manage this stack are provided internally in the object definition like the *Timers* structures. Further details will be explained in the communications module and the sender module. Those will receive the message and address where it must be forwarded and they must try to achieve its function sending the message with a non-blocking system.

Application Interface

The application interface provides the mechanism used for data transaction between the application, which will use the SCSP for its data synchronisation and the SCSP itself. We will use the *socketpair()* function provided by the OS to send the data that the application wants to update through this pipe facility to the SCSP data strucutures.
The SCSP protocol keeps track of the cache information through the list_CSA structure, which is between both protocols. For that reason we have used an independent implementation for the cache to make future extensions easier and add features to the cache. A detailed description of the list_CSA is shown in section 5.4.4. Figure 13 shows a graphical example of how it works.

This is an important feature of the SCSP implementation. We use the list_CSA structure that forms an independent array of objects that we can fill with any kind of data. Each Directory provider will fit its data on the records of the array and the SCSP takes care of propagating this data and achieving the replication with other providers. The other party, the provider which receives the records will extract the pertinent data that he requires for updating his own DIT or database.
Thus, the SCSP provides the abstraction of the structures to allow the management of different kinds of information.

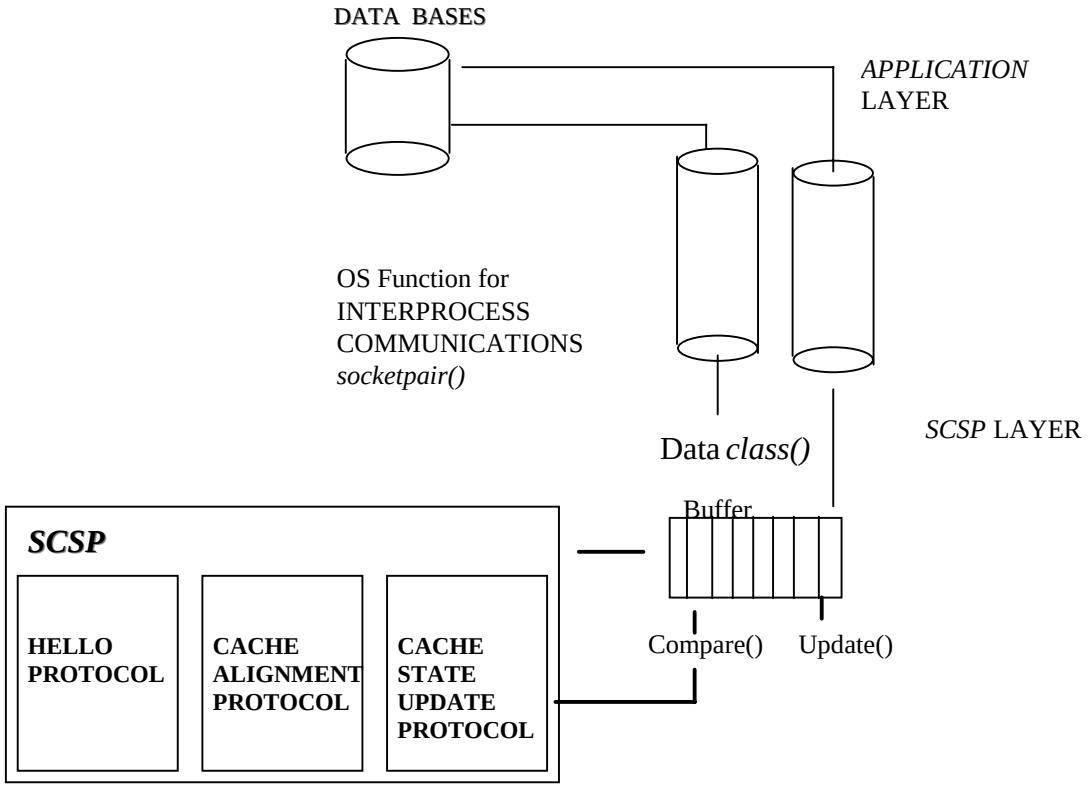


Fig. 13. Interprocess communications

Chapter 5

Protocol detailed design

In the previous chapters we have seen a description of the database synchronisation, which is the problem we try to solve in this Thesis. We chose the SCSP as the solution for that problem. In the next chapters were described the specifications and environments where the SCSP is allocated among other protocols for the information management. In this chapter we illustrate all the pieces that the SCSP requires for its implementation. Hence, below in the next sections are stripped all the details of the solution adopted for the system implementation. All the structures are exposed, and the system facilities required explained.

5.1 Timer Implementation

This module is quite important because it has to synchronise all the protocol functions. Each FSM needs its services to be aware of the interval between two consecutive requests. In case one of its requests had been lost, it has to formulate again the same request.

For this purpose, we have implemented an independent class called **Timer** that has its inner functions and a private counter. This maintains the time value and can be managed from the other objects through its own functions, those functions can stop, or reset the counter value, in function of the program requirements.

The *Timers* specifications are defined like global variables at the beginning of the program to be available from every part of the protocol. Its counter must be increased every new loop that the program reinitialises the array of classes for the process. Hence, we achieve our aim to minimise the dependence from the system's kernel. The alternative would be using an internal interruption with its handler module that we also need to catch the OS notification when the alarm event arrives.

5.1.1 OS facilities for Timer Implementation

We use a time function provided by the OS, which is translated to a C library in a function named *clock()*, which returns the amount of time in seconds since January 1, 1970, Universal Coordinated Time (UTC) also known as the Epoch. Most modern processors maintain a battery-backup time-of-day register. This system's clock continues to run even if the processor is turned off. In fact, when the system boots it consults the processor's time-of-day register to find out the current time. The system's time is maintained by the clock interrupts and at each interrupt the system increments its global time variable by an amount equal to the number of microseconds per tick. (For the HP300, running at 100 ticks per second, each tick represents 10.000 microseconds)[21]. The time is always exported from the system as microseconds, rather than as clock ticks, to provide a resolution-independent

format. Internally, the kernel is free to select whatever tick per rate best trades off clock-interrupt-handling overhead with timer resolution. As the tick rate per second increases, the resolution of the system timer improves but the time spent dealing with hardclock interrupts increases. As processors become faster, the tick rate can be increased to provide finer resolution without adversely affecting the applications.

All filesystem (and other) timestamps are maintained in UTC offsets from the Epoch. Conversion to local time, including adjustment for daylight-savings time, is handled externally to the system in the C library where we can find the function which will increase the counter value inside the Timer class.

5.1.2 Timer class structure

In the former definition we said that there is a Timers array defined at the beginning of the protocol. Each element of this array is for one of the different FSM running in the Hello Protocol and the Cache Alignment Protocol between the LS and each DCS that compose the SG. There is another array for the Cache State Update protocol, to keep track of the outstanding CSA records in the CSU request for each DCS to which the CSU was sent from the LS and for which an acknowledgment has not been received. Figure 14 depicts the basic functions defined for the Timer class implementation.

Timer class:

➡ Provides an homogeneous counter for all the protocol
There is a Timers array for each HFSM, CAFSM and CSU

*Timer * timerarrayHelloInt* ⟶ *timeHelloInt*
*Timer * timerarrayHelloRec* ⟶ *timeHelloRec*
*Timer * timerarrayCaInt* ⟶ *timeCaReXmInt*
*Timer * timerarrayCSUInt* ⟶ *timeCSUSReXmInt*
*Timer * timerarrayCSUInt* ⟶ *timeCSUReXmInt*

➡ Its value is updated every new program loop, with an internal function
The program is bound to the kernel function *clock()* To update each Timer in arrays element every time the protocol starts a new loop.

➡ Functions in the Timer class:

Timer_on() ..... Enable the Timer counting
Timer_off() ..... Disable the Timer
Timer_reset() ..... Reset the Timer value
Timer_value() ..... Returns the Timer value
Timer operator++() ..... Increase the Timer value
Timer_update() ..... Update the Timer counter with a system function *clock()*

Fig. 14. Internal functions in the Timer class

In all those transactions, if the related FSM receives the required answer or acknowledgment, the Timer associated is stopped and maybe reset for a new request. If the Timer has expired, then the FSM has to check the counter and to act according to its definitions. Maybe resending the packets or changing the Machine state, and restarting all the process afresh from the beginning. The Timer class is one of the important elements in the protocol implementation because on its functions depends the correct performance of the rest of the protocol.

In case there are no messages from other servers the protocol performs an idle state. Before going to sleep, we fix the interval where it must wake up to check the timers and make the necessary state transitions. The period of time that the program goes to sleep is fixed from the configuration file and depending of how accurate we want to make the behavior a tight schedule can be established.

5.2 Descriptors and I/O

The descriptors definition, are important in a protocol implementation, and we will use their features later in the communication module. For the user processes, all I/O are done through descriptors.

All the computers store and retrieve data through peripheral I/O devices. Storage devices such as disks are accessed through I/O controllers that manage the operation of their *slave* devices according to I/O requests from the CPU. The hardware peculiarities are hidden from the user by high-level kernel facilities, such filesystem and socket interfaces. Other details are hidden from the bulk of the kernel itself by the I/O system. The I/O system consist of buffer-caching systems, general device-driver code, and drivers for specific hardware devices that must finally address peculiarities of the specific devices.

5.2.1 Types of I/O Descriptors

There are four main kinds of I/O: filesystem, the *character-device* interface, the *block-device* interface, and the *socket* interface which is related to network devices. Regards our software, we will focus the attention in the network devices, which are accessible through only the socket interface.

System calls that refer to open files take a file descriptor as an argument to specify the file. The file descriptor is used by the kernel to index into the *descriptor table* for the current process to locate a *file entry*, or *file structure*. The file entry provides a file type and a pointer to an underlying object for the descriptor. The file entry may also reference a *socket*, instead of a file. Sockets have a different file type, and the file entry points to a system block that is used in doing inter-process communication. The virtual-memory system supports the mapping of files into the address space of the process.

The set of file entries is the focus of activity for file descriptors. They contain the information necessary to access the underlying objects to maintain common information.

The file entry is an object-oriented data structure. Each entry contains a type and an array of function pointers that translate the generic operations on file descriptors into the specific actions associated with their type. There are two descriptors types: **files** and **sockets**. The operations that must be implemented for each type are as follows:

- ❒ Read from the descriptor
- ❒ Write to the descriptor
- ❒ Select on the descriptor
- ❒ Do **ioctl** operations on the descriptor
- ❒ Close and possibly deallocate the object associated with the descriptor

Each file entry has a pointer to a data structure that contains information specific to the instance of the underlying object. The data structure is opaque to the routines that manipulate the file entries themselves.

Some semantics associated with all file descriptors are enforced at the descriptor level, before the underlying system call is invoked. These semantics are maintained in a set of flags associated with the descriptor. For example, the flags record whether the descriptor is open for reading only, an attempt to write it will be caught by the descriptor code. Thus, the functions defined for doing reading and writing do not need to check the validity of the request, we can implement those functions knowing that they will never receive an invalid request.

5.2.2 Descriptor Flags

Other flags that we will need for the file descriptor used for the communications include:

- ❒ The *no-delay* (NDELAY) flag: If a read or write would cause the process to block, the system call returns an error (EWOULDBLOCK) instead.

- ❒ The *asynchronous* (ASYNC) flag: The kernel watches for a change in the status of the descriptor, and arranges to send a signal (SIGIO) when a read or write becomes possible.

Other information that is specific to regular files also is maintained in the flags field, but this information is not relevant for our purpose:

- ❒ Information on whether the descriptor holds a shared or exclusive lock on the underlying file.
- ❒ The *append* flag, each time that a write is made the offset pointer is set to the end of the file.

Each entry has a *reference count.* A single process may have multiple references to the entry because of the *dup* or *fcntl* system calls. Also, file structures are inherited by the child process after a *fork*, so several different processes may reference the same file entry. This semantic allows two processes to read the same file or to interleave output to the same file.

5.2.3 Functions for the management of descriptors

For the descriptor's management the system provides the function **fcntl**, this system call manipulates the file structure that we will use in the communications module.
In this section we have a description of the management of file descriptors for the data reception. Figure 15 depicts the structure used to store the data received from the DCS and we will use the **fcntl** facilities to perform an efficient data management over the *socket* file descriptor. The steps to configure the reception function are:

- ➢ System interrupt is bound to the socket used for incoming messages

```
extern int sockfd;
sockfd = socket ( AF_INET, SOCK_DGRAM,0);
my_addr.sin_family = AF_INET;
my_addr.sin_port = htons(port);   /* we read the
           port value in the file config.txt */
bind(sockfd,(struct sockaddr *)&my_addr,
sizeof(struct sockaddr));
```

- ➢ Fix the socket to be aware of each I/O interrupt that take place in our port

```
fcntl(sockfd,F_SETOWN,getpid());
```

```
fcntl(sockfd,F_SETFL,FASYNC);
```

- Function implementation to handle the interrupt

```
void manejoin(int)
```

Finally, by this function we catch the just arrived data and fill the structure dato_bufer shown in Figure 14. After that this structure is inserted in the Template buffer called buferet, with its function pon().

dato_bufer

remite	DCS identifier, from which was received the message
mensaje	Message data, array of characters
len	Message length, number of bytes

Fig. 15 Data structure to store the message received

We can use the *fcntl* utilities to make the following changes to a descriptor:

a) Duplicating a descriptor as thought by the *dup* system call

b) Setting the close-on exec flag. When a process *forks* all the parent's descriptors are duplicated in the child. The child process then *execs* a new process. Any of the child's descriptors that were marked close-on-exec are closed. The remaining descriptors are available to the newly executed process.

c) Setting the descriptor into non-blocking mode. If any data are available for a read operation, or in any space is available for a write operation an immediate partial read or write is done. If no data are available for a read operation, or if a write operation would block, the system call returns an error showing that the operation would block, instead of putting the process to sleep.

d) Forcing all writes to append data to the end of the file, instead of at the descriptor's current location in the file.

e) **Sending a signal to the process when it is possible to do I/O.**

f) Sending a signal to a process when an exception conditions arises, such as when urgent data arrives on an interprocess-communication channel.

g) **Setting the process identifier or process-group identifier to which the two I/O-related signals in the previous steps should be sent**

h) Testing or changing the status of a lock on a range of bytes within an underlying file.

The fcntl function avoids to maintaining the program all the time pending of incoming data in our file descriptor. We want that our program must be able to read the data from the remotes DCSs connected to the receiver socket for incoming data. In the unfortunate case that in the moment the program decides to read the socket there is no data available it will be

normally blocked in the kernel until the data become available. That means losing time and system resources waiting for incoming information, so that blocking behaviour is unacceptable. Using the e) facility provided by the fcntl function we avoid the situation where the process makes a read request and blocks, then it will be unable to process the information that it has already on the buffers. So a deadlock would occur if the system spends all the time waiting, or otherwise we have to be constantly checking the socket to know if there is some new information to be read.

5.2.4 Facilities borrowed from OS for the management of Descriptors

For the reason that we have seen above, we are going to look at the subsequent features from the system:

- polling I/O, this facility is done with the *select* system call

- non-blocking I/O, the operations in the non-blocking descriptors complete immediately, partially complete an input or output operation and return a partial count, or return an error that shows that the operation could not be completed at all. (We have implemented this feature in the module for sending the messages)

- signal driven I/O, The descriptors which have the signalling enabled cause the associated process or process group to be notified when the I/O state of the descriptor changes. (We have already implemented this feature in the reception module)

There are four possible alternatives [21] to avoid the blocking problem when the descriptor is not ready to be read or written. Thus it is necessary to know all of them to decide which one is most suitable for our purpose.

- To set all the descriptors into non-blocking mode. The process can then try operations on each descriptor in turn, to find out which descriptors are ready to do I/O. The problem with this approach is that the process must run continuously to discover whether there is any I/O to be done. Thus this option has been discarded in the implementation, because it is waste of time checking every fixed interval of time whether there is any socket ready to obtain the data, and we also would need a stable mechanism to inspect the socket currently, so we will use a system signal to interrupt the program to check whether is data waiting or not, and return to the normal program course.

- To enable all descriptors of interest to signal when I/O can be done. The process can wait for a signal to discover when it is possible to do I/O. The drawback to this approach is that signals are expensive to catch. Hence, signal driven I/O is impractical for applications that do from moderate to large amounts of I/O. However, in the SCSP protocol we only use one socket to receive all the data, through the same port which is pre-configured at the beginning of the implementation. Thus we only have to mange a single socket, and afterwards the data that we receive is driven to the corresponding FSM which will manage the data according to the protocol state and premises.

- To have the system provided with a method for asking which descriptors are capable of doing I/O. If none of the requested descriptors are ready, the system can put the process to sleep until a descriptor becomes ready. This feature is implemented using a library function named *select*. The drawback is that the process must do two system calls per operation: one to poll for the descriptor that is ready to do I/O and another to do the

operation itself, so we also have discard this option, because we have to leave the process sleeping if there is no data ready. In this way we waste time and system resources, waiting for data available in the socket while the rest of the program could process the data allocated in a buffer.

➢ To have the process notified by the system of all the descriptors that the process is interested in reading, and then do a blocking read on that set of descriptors. When the read returns the process is notified on which descriptor the read completed. The benefit of this approach is that the process does a single system call to specify the set descriptor, then loop is doing reads[22].

5.2.5 Descriptor Implementation in SCSP development

The first approach is the non-blocking I/O. It typically is used for output descriptors, because the operation typically will not block. Rather than doing a **select** which nearly always succeeds, followed immediately by a write. It is more efficient to try the write and revert to using **select** only during periods when the **write** returns a blocking error. Thus, we use this first approach in the function **envia()** to send the data to the rest of DCS s in the SG. In the case that the call does not succeed it is not a big problem because the protocol specification take care of those inconveniences and after a period of time the same message will be sent again. So it is better than block the program when it has a new message to send and it fails trying to achieve this goal. This descriptor gives us the facility to avoid blocking the program while it is trying to send data. The descriptor of the **envia()** function is shown in Figure 16. In the figure we can see the behaviour of the program when the function **envia()** is called to send the data.

➡ *bool envia(int &,char *,char *,int &)*

This function is used to send the messages in every place where the protocol is running at this moment, It is a generic function to catch the packet and send it to the address which we give the function when it is called, use a non-blocking socket to avoid stopping the rest of the protocol on its normal functions.

fcntl(sockfd, F_SETL, O_NONBLOCK)

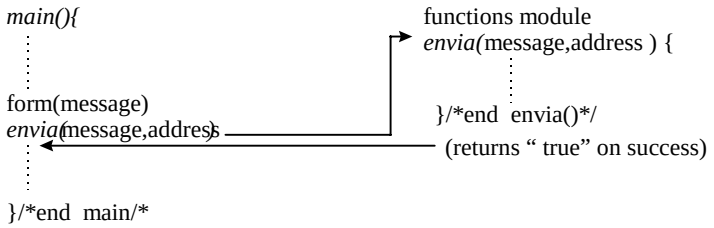


Fig. 16.Description of the function *envia()* and how it is embedded in the main program

Afterwards, if the **envia()** function does not succeed it returns without any delay, instead of trying to send the data again using the *slect* call. With the *select* alternative it means that the program has to stop and wait for this new attempt to send the data with *select* and it could be possible that in this moment we have data in the buffer to be processed.
The *select* interface takes three masks of descriptors to be monitored, corresponding to interest in reading, writing, and exceptional conditions. In addition, it takes a time-out value for returning from *select* if none of the requested descriptors becomes ready before a specified amount of time has elapsed. The *select* call returns the same three masks of descriptors after modifying them to show the descriptors that are able to do reading, to do

writing, or to provide an exceptional condition. If none of the descriptors has become ready in the time-out interval, *select* returns showing that no descriptors are ready for I/O.

The second approach is available as signal-driven I/O. It is typically used for rare events, such as for the arrival of out-of-band data on a socket. For such rare events, the cost of handling an occasional signal is lower than that of checking constantly with *select* to find out whether there is any pending data. Thus, this is the system feature that we choose to achieve the goal of an efficient and rapid data management when it arrives to the host or server. Thus, the goal we want to achieve is to store the data without spendding too much time with memory reservation and reallocation because when the data arrives the signal handler only have to use the system call **receivefrom()** to store the data coming in a character array, the address of which is allocated in a global buffer. Afterwards, it will be visible for the rest of the program when the handler returns from the signal call. This mechanism is reflected in Figure 17. The figure details the data management when data arrives from the other DCS to the I/O port of the LS.

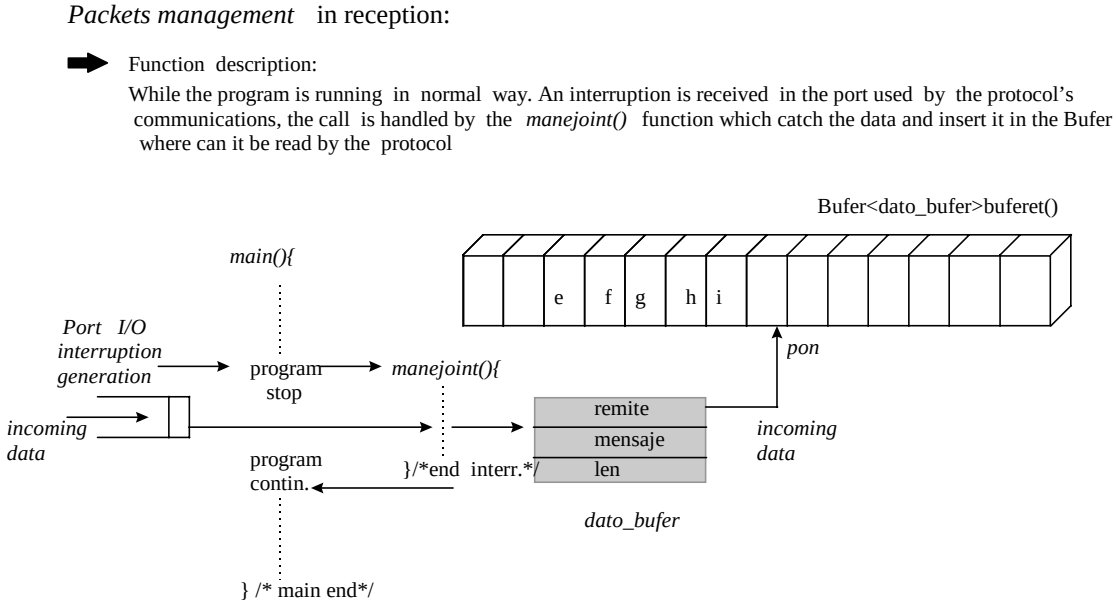


Fig. 17, Data management from the I/O port to the data buffer with the interruption handler

In this cyclic buffer which we will describe later in more detail we only manage the address of the characters array. In the buffer the data is stored, instead of making a new memory reservation and moving the whole block of data that has just arrived to that new amount of memory. Hence, we only put the first memory allocation address in the cyclic buffer and later when the program needs to process the data, it only has to read from the buffer the memory address and work over this data. Consequently, the data is not moved to any other place, avoiding the corresponding system resource reservation and reallocation, to store the identical data twice. This is the first rule in a network protocol implementation[20] we have to avoid copying data from one buffer into another. If a protocol adheres to this the implementation will probably be quite efficient. The reason is that copying data implies a loop that loads every word of one buffer into a CPU register and then stores it back to another memory location. In a sense, we should think in our computer's bus as the last physical link of the network and we have to manage this resource carefully. The result of this discussion is that it is critical that the message operations not touch the data in a

message, but rather only manipulate pointers. Thus the solution recommended and we have adopted in our reception module when we receive the message rather than allocating a new memory buffer and copy the bytes from the first message into that buffer we store the address of this first allocation in our cyclic buffer named **buferet**.

5.3 Communication module

This is the other half of the protocol skeleton, because all the transactions have to be managed by this module. It is fragmented in two parts; The function which take care of the sending task, and the part which have to receive the packets and allocate them to the Buffer.

The former part, is achieved by the definition of the **envia()** function. And the latter is the most important and is embedded in the protocol body itself.

The **envia()** function is implemented as an individual function that can be called from anywhere in the program. It acts like a stand alone function, it means that to send a packet it only need to receive the address of the data where is the packet sent to and the port it can use for its purpose. Afterwards, the function takes care of the socket reservation which is fixed like a non-blocking socket to avoid stopping the program. In the worst case that the socket is not available, it returns without send any message, and this message will be lost. Trying to minimise that problem we have **Timers** functions to resend a new packet in case the program did not receive any message in the corresponding time interval. Thus, we can solve this inconvenience without stopping the protocol or bind it in a imprecise cycle waiting for a free socket.

Subsequently, we are going to describe the internal details used to implement the **envia()** function. It includes the OS facilities provided by the system.

5.3.1 OS Characteristics

The basic building block for the communication is the *socket*. A socket is an endpoint of communication to which a name may be bound. Each socket in use has a *type* and one or more associated processes. Sockets exist within *communications domains*. A communication domain is an abstraction introduced to bundle common properties of processes communicating through sockets. TCP/IP socket programming can be seen as a way of writing programs that can separate the client from the server in a platform-independent manner. When we are working directly with socket based protocols we can send commands from the client to the server, thus implementing an application-specific protocol. The programmer needs to know the protocol and construct commands that are sent to the server through a socket connection. Upon command completion, the server generally replies using the same protocol.
The client server socket communications is not very difficult to implement, but it is tedious and requires quite a bit of code. When handling more complex tasks, the protocols generally get more complex and programming the protocol handling becomes more difficult.

Sockets normally exchange data only with sockets in the same domain (it may be possible to cross domain boundaries but only if some translation process is performed).
There are three separate communications domains:

UNIX domain, for on-system communication.

Internet domain, which is used by processes which communicate using the DARPA standard communication protocol.

NS domain, which is used by processes which communicate using Xerox standard communication protocols.

In the SCSP implementation we use the Internet Domain socket for our purpose.

5.3.2 Sockets for communications programming

Regards the *type* field, the sockets are typed according to the communication properties visible to a user. The processes are presumed to communicate only between sockets of the same type. Are currently available Four types of sockets:
1. Stream, 2. Datagram, 3. Raw, 4. Sequenced packet

1) A *stream* socket provides for bi-directional reliable, sequenced, and unduplicated flow of data without record boundaries.

2) A *datagram* socket supports bi-directional flow of data which is not promised to the sequenced, reliable or unduplicated. That is, a process receiving messages on a datagram socket may find messages duplicated, and possibly in an order different from the initial order in which it was sent. An important characteristic of a datagram socket is that record boundaries in data are preserved. Datagram sockets closely model the facilities found in many contemporary packet switched networks such as the Ethernet.

3) A *raw* socket provides users access to the underlying communication protocols which support socket abstractions. These sockets are normally datagram oriented, though their exact characteristics are dependent on the interface provided by the protocol. Raw sockets are not intended for the general user.

4) A *sequenced packet* socket is similar to a stream socket, with the exception that record boundaries are preserved. Sequenced packets allow the user to manipulate the SPP or IDP headers on a packet or a group of packets either by writing a prototype header along with whatever data is to be sent, or by specifying a default header to be used with all outgoing data, and allows the user to receive the headers on incoming packets.

5.3.3 SCSP communications framework

In the SCSP implementation we choose the *Datagram* socket for the communications because despite its unreliable packets delivery, it performs a faster communications method. Since our purpose is to achieve an efficient and quick synchronisation of the cache information it is the most suitable option for the communication block. A datagram socket provides a symmetric interface to data exchange where there is no requirement for connection establishment, instead each message includes the destination address.

To start operation we have to make a system call for a free socket which we can use for the message transactions. On this request the system creates a socket in the specified **domain** and of the specified **type**. A particular protocol may also be requested. If the protocol is left unspecified (a 0 value), the system will select an appropriate protocol from those protocols which comprise the communication domain and which may be used to support the requested

socket type. The user is returned a descriptor (a small integer number) which may be used in later system calls which operate on sockets.

The domain is specified as one of the constants defined in the file **<sys/socket.h>**. For the UNIX domain we have **AF_UNIX**, for Internet domain is **AF_INET** and for NS domain **AF_NS**. The socket types are also defined in this file and one of the **SOCK_STREAM, SOCK_DGRAM, SOCK_RAW** or **SOCK_SEQPACKET** must be specified.

Thus to create our particular socket we make this system call with the options of **AF_INET** domain and **SOCK_DGRAM** type, we leave the protocol option open and the system will select the best according its criteria. However, it is possible to specify a protocol other than the default.

int sockfd= socket(AF_INET, SOCK_DGRAM,0)

There are several reasons a socket can fail. Aside from the occurrence of lack of memory (ENOBUFS), a socket request may fail due to a request to an unknown protocol (EPROTONOSUPORT) or a request for a type of socket for whcich there is no supporting protocol (EPROTOTYPE).

A successful call will result in a datagram socket being created with the UDP protocol providing the underlying communication support.

5.3.4 SCSP over IP

Actually, many popular network applications have been built on top of TCP and UDP over the past decade. These have helped the Internet services and protocols to become widely-spread de facto standards. In the past few years the ISO and CCITT have defined a well-architected set of upper layer standards which include connection-oriented and connection-less session, presentation and application layer services and protocols. These OSI upper layer standards offer valuable services to application developers (e.g. dialogue control, transfer syntax, peer authentication, directory services, etc.) which are not currently offered by the TCP/IP standards. The SCSP is defined as an Draft Standard but the aim of this work is go beyond this standard and convert this Synchronisation Protocol into a valuable service for developers of other protocols. In that way the SCSP becomes in a connection-less session protocol, that can be used at the same level with other protocols or in upper level services.

The most important reason to choose the **UDP** protocol in our communications framework is the desirability to offer the OSI upper layer services directly in the Internet without disrupting existing facilities. This permits a more graceful convergence and transition strategy from IP-based networks to OSI-based networks in the future. Thus, in the SCSP we use the approach of RFC 768, that specifies how to offer OSI connection-less transport service using the User Datagram Protocol of the TCP/IP suite.

As a result, the SCSP packets can be delivered across the Internet. Thus the upper layers can operate fully without knowledge of the fact that they are running on top of UDP/IP.

It is observed that in the connection setup and tear-down protocol exchanges and complex connection-oriented processes take place. Those transactions create unnecessary overheads for a simple request/response exchange among the servers of the synchronisation group. Especially in reliable communications environment such as LAN and ISDN.

The OSI connection-less upper layers are thought to be highly effective and efficient both in time and space, for the distributed application classes.
Nowadays, the stability , maturity and wide availability of UDP/IP are ideal for providing solid connection-less transport services independent of actual implementations.

Thus among the servers we can implement a “request/response” application where the most prominent aims are quick transactions and reliable behaviour of the synchronisation service.

Using the UDP transport system the servers can communicate with each other using connection-less transport provided there is pre-arranged knowledge about each other (e.g. protocol version, formats, options, … etc.), since there is no negotiation before data transfer. During initialisation of the SCSP each server reads the basic data from a configuration file which contains the packet size and the port number. Before starting the synchronisation process that data must be agreed in common knowledge by the whole group. Afterwards, each server passes a message to its neighbours, that receive the message. Then, a sequence of interactions between servers described by the connection-less primitives will take place to achieve the final database synchronisation.

5.3.5 UDP for SCSP communications

This protocol provides a procedure for application programs to send messages to other programs with a minimum of protocol mechanism. The protocol is transaction oriented so delivery and duplicate protection are not guaranteed.

- ❏ General UDP architecture:

***Format*:**
In Figure 18 the structure of the UDP packet is depicted.

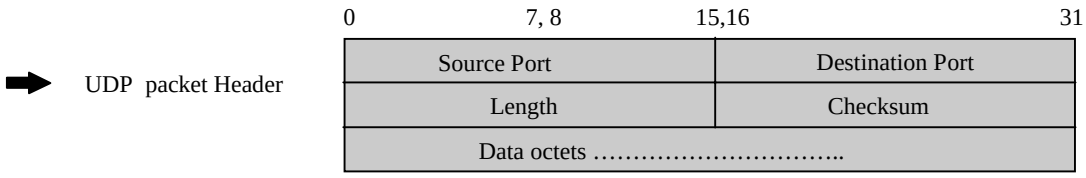


Fig. 18, UDP Packet header format

Fields:

Source Port is an optional field. When meaningful, it indicates the port of the sending process. It may be assumed to be the port to which a reply should be addressed in the absence of any other information. If not used, a value of zero is inserted.

Destination Port has a meaning within the context of a particular Internet destination address.

Length is the length in octets of this user datagram including this header and the data (this means the minimum value of the length is eight).

Checksum is the 16-bit one's complement of the one's complement sum of a pseudo header of information from the IP header, the UDP header and the data padded with zero octets at the end (if necessary) to make a multiple of two octets.

The pseudo header conceptually prefixed to the UDP header contains the source address, the destination address, the protocol and the UDP length. Those parts are described in Figure 19. This information gives protection against misrouted datagrams. This checksum procedure is the same as is used in TCP.

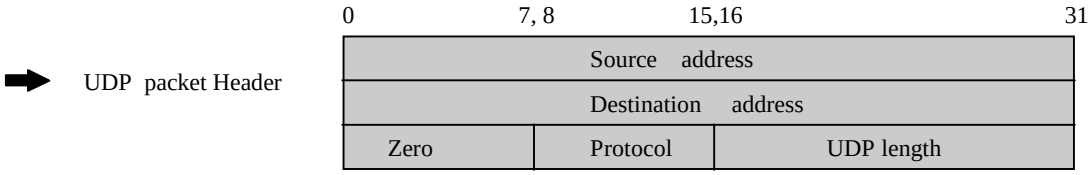


Fig. 19, UDP pseudo header structure

If the computed checksum is zero, it is transmitted as all ones (the equivalent in one's complement arithmetic). An all zero transmitted checksum value means that the transmitter generated no checksum (for debugging or for higher level protocols that do not care).

❏ UDP in SCSP implementation:

Another part of the UDP specification is the user Interface.
A user interface should allow the creation of new receive ports and receive operations on the receive ports that return the data octets. It also indicates the source port and the source address and an operation that allows a datagram to be sent. It specifies the data source and destination ports and destination address. In our case the configuration module takes care of the ports assignment and the interruption function bound to the reception port will check the data available on the port. This function will insert it in the buffer to be processed afterwards by the FSM. In the other side is the function **envia()** which manages the data and the address where the next packet will be sent.

The latter part is the IP interface.
The UDP module must be able to determine the source and destination Internet addresses and the protocol field from the Internet protocol header. One possible UDP/IP interface would return the whole Internet datagram.

Such an interface would also allow the UDP to pass a full Internet datagram complete with header to the IP for sending. The IP would verify certain fields for consistency and compute the Internet header checksum.

In the above description we have defined the socket through the system call **socket** and a connection-less communication is achieved. In the UDP user Interface specification defined above, we have to bind the socket obtained to some address and port.

Hence the second step is to bind the socket to a fixed port and name that we receive from the protocol specification RFC. Initially a socket is created without a name, so until a name is bound to a socket processes have no way to reference it and consequently no messages may be received on it. Communicating processes are bound by an *association.* In the Internet and NS domains an association is composed of local and foreign address and local and foreign ports. In most domains, associations must be unique. In the Internet domain there may never

be duplicate **<protocol, local address, local port, foreign address, foreign port>** tuples. Thus the **bind** system call allows a process to specify half of an association **<local address, local port>**

- ❏ OS calls for communications module

The system call for binding an address to a specific socket is as follows.

```
bind (socket, name, name_length)
```

And our particular system call is:

```
bind(socfd,(struct sockaddre *)&myaddre, sizeof(myaddr))
```

The bound name is a variable length byte string which is interpreted by the supporting protocol. Its interpretation may vary from communication domain to communication domain (this is one of the properties which comprise the domain). As mentioned, in the Internet domain names contain an Internet address and port number. To aid in the task to locate and construct network addresses in a distributed environment, there are a number of routines that have been added to the standard C run-time library that works over the OS. These routines are provided for mapping host names to network addresses, network names to network numbers and service names to port numbers. In this case the file **<netdb.h>** must be included when using any of these routines.

One of those structures is **struct sockaddr_in**. This structure holds socket address information for Internet. If we want to translate the program for another architecture, we can choose the **strucut sockaddr** that is available for many types of sockets.

```
int sockfd;
struct sockaddr_in myaddr;

myaddr.sin_family=AF_INET;
myaddr:sin_port=htons(port);
myaddr.sin_addr.s_addr=inet_addr(address);
bzero(&(myaddr.sin_zero),8);
```

This structure makes it easy to reference elements of the sockets address. The field *sin_zero* is included to pad the structure to the length of a *struct sockaddr* and it should be set to all zeros with the library function **bzero()**. The *sin_family* field is filled with the type of domain we have seen in the section 5.3.3. The *sin_port* holds the port number that we will use in the protocol and we read it from the configuration file at the beginning of the protocol functions. Finally, the *sin_addr* where we insert the IP address in the Network Byte Order, and also the port number must be in Network Byte Order. For that conversion we can use the function **htons()** and **inet_addr()** to convert the IP address in number and dots notation into an unsigned long number with Network Byte Order. It is necessary because due to the protocol specification the address must be stored in character format. Thus, in the Internet case we use the numbers and dots format, and in case of another system should change all this conversions according to the platform which we have to use). Thus finally we have filled the structure to bind the socket to the right address and port.

To make the **envia()** function non-blocking, once the socket has been created via the *socket* call, we use the function **fcntl()** as follows.

```
fcntl(sockfd, F_SETFL, O_NONBLOCK)
```
or
```
fcntl(sockfd, F_SETFL,FNDELAY)
```

With this function we can manipulate and perform miscellaneous operations on the socket. With the argument F_STFL we set the descriptor's flags to the value specified by the third argument. In that argument we have FNDELAY or O_NONBLOCK, in fact both of those values are bit masks which fix the socket flags to the non-blocking state in the file descriptor system.

In the reception module, following the same directive in the whole implementation, we try to avoid suspending the protocol process for waiting any incoming message. For that purpose we use several features from the system like signal management and the already described *fcntl()* function with different options.

The UNIX system defines a set of *signals* for software and hardware conditions that may be delivered to a process such that they can arise during the normal execution of a program. Signals are modelled after hardware interrupts, but in the sense that they are designed to be software equivalents of these hardware interrupts or traps. A process may specify a user-level subroutine to be the *handler* to which a signal should be delivered. Signals are *posted* to a process by the system when it detects a hardware event, such as an illegal instruction, or a software event such as a stop request from the terminal. When a signal is generated it is blocked from further occurrence while it is being *caught* by the handler.

Catching a signal involves saving the current process context and building a new one in which to run the handler. The signal is then delivered to the handler, which can either abort the process or return to the executing process. If the handler returns, the signal is unlocked and can be generated and caught again.

Alternatively, a process may specify that a signal is to be *ignored* or that a default action as determined by the kernel is to be taken. The default action on certain signals is to terminate the process. This termination may be accompanied by creation of a *core* file that contains the current image of the process for use in post-mortem debugging. This is very useful when we use the **gdb** debugger to fix problems during the program implementation.

Some signals cannot be caught or ignored. These signals include SIGKILL which kills runaway processes and the job-control signal SIGSTOP.

In POSIX system calls interrupted by a signal may cause a system call to be interrupted and terminated prematurely and an "interrupted system call" error to be returned. The *sigaction* system call can be passed a flag that requests that system calls interrupted by a signal be restarted automatically whenever possible and reasonable. Automatic restarting of system calls permits programs to service signals without having to check the return code from each system call to determine whether the call should be restarted. In our application we use the C-library routine *signal()* to set up the signal handler for the I/O interruption. The *signal()* routine calls *sigaction* with the flag that requests that system calls are restarted.

The SIGIO signal provided by the system allows a process to be notified via a signal when a socket (or more generally a file descriptor) has data waiting to read. The use of the SIGIO facility requires three steps:

➢The process must set up a SIGIO signal handler by issuing of the *signal* or *sigvec* calls. It is accomplished with the next function.

```
vvoid manejoin(int);
```

We allocate this definition at the beginning of the protocol implementation to be available from everywhere in the global process. The same attributes are established for the socket definition because it must be available for the interruption handler in every moment.

➢ Secondly, the process identifier which have to receive the notification must be set pending for that event. It is accomplished with the use of an *fcntl* function call.

```
if (fcntl(sockfd, F_SETOWN,getpid())<0){
```

➢ Finally, we must enable asynchronous notification of pending I/O over the socket with another *fcntl* call.

```
if (fcntl(sockfd, F_STFL,FASYNC)<0){
```

With this sample code we allow a given process to receive information on pending I/O requests as they occur at a socket *socfd*. With the SIGIO signal each socket has an associated process number. This value is initialised to zero, but may be redefined at a later time with the F_SETOWN option in the second argument of the *fcntl* function as we have done in our particular code for the messages reception. To set the process identifier for the socket positive arguments should be given to the *fcntl* call. A similar *fcntl* F_GETOWN is available for determining the current process number.

As a result, we have defined the major part of the whole code to receive any message without interfering the functionality and performance of the protocol. The signal handler has to take care of the incoming message in the socket when receiving the system call, which indicates that there is a message ready. Now, we show this handler.

```
Void manejoin(int);
```

Waiting for the SIGIO signal, with the function;

```
signal (SIGIO, manejoin);
```

And inside the handler body, we implement the code that will take the message from the socket, with the system function;

```
numbytes=recvfrom(sockfd,buf,MAXBUFLEN,0,struct
sockaddr*)&their_addr, &tam1));
```

The recvfrom() function will manage the data inserted in the characters array named **buf** by the latter function with an additional structure used for that purpose, to preserve the independence of the reception function from the rest of the program and maintain the maximum information of the message and sender available to be processed.

```
dato_bufer dato;
dato.mensaje=buf;
```

When we have data in the temporary structure **dato_bufer** we insert that information in the buffer **buferet**. Then the signal handler can return from its call to the last position in the program that it left when the signal iterruption was received.

```
buferet.pon(dato);
```

The **buferet** has been defined like a global entity to allow its management from the signal interruption handler and also from the rest of the protocol. Thus it is reachable when any protocol function decides to access to any new packet that have been allocated in its storage area to be processed. The basis of the **buferet** implementation is described below in the Buffer description.

5.4 Protocol data structure

In this section we describe the data structures used in the implementation. All those structures are based on the MIB definitions [19]. In this way we make it easier to manage the protocol through the SNMP requests. We will also analyse the basis of the MIB specifications and the ASN.1 notation used for those definitions.

5.4.1 MIB objects

In the late 1980's the Internet Architecture Board (IAB), felt that it should form a group to develop tools, protocols a common database for general network management. As a result of this, the Simple Network Management Protocol (SNMP) was born for TCP/IP with considerations based on the framework of the OSI model (RFC 1157). There are three basic concepts; The **manager** the **agent** and the **management information base (MIB)**

SNMP has its own monitor and control functions using transport devices to move information between the management station and the station being managed. As far as programming is concerned the manager is a client running on the management station and the agent is a server running on a network application program whose job is to collect data and interact with the manager. The manager and the agent use UDP services on the network to exchange messages. The database created on the network element is known as a view of Management Information Base (MIB), which contains objects in a tree structure (RFC 1156).

For that reason, first of all we have to adapt our work to the SNMP requirements for the management information. It allows its inspection or alteration of any network element by logically remote users. In Management objects for SCSP [23] the structure of the management information base for the objects needed in the SCSP is described. This document provides a simple workable architecture and system for managing TCP/IP based internets and in particular the Internet.

The Internet activities Board recommends that all IP and TCP implementations be network manageable. This implies the implementation of MIB, and at least one of the two recommended management protocols SNMP (RFC 1157) [24] or the CMIT (RFC 1095) [25].

5.4.2 SNMP framework

SNMP is a full Internet Standard and CMIT is a draft standard. For this reason in our work we have tried to adapt our implementation to the SNMP specifications. This architectural model is a collection of network management stations and network elements. The network management stations execute management applications which monitor and control network elements. The network elements are hosts, servers, gateways and terminals which have management agents. Those agents are responsible for performing the network management functions requested by the network management stations. The SNMP as we have already mentioned, is used to communicate management information between the network management stations and the agents. The SNMP explicitly minimises the number and complexity of management functions performed by the management agents. Those functions can be easily used by developers of network management tools. That has been an important motivation in following the management objects definition for the SCSP.

We have tried to implement the objects closely following the definitions in the paper [23] to obtain a functional code for monitoring and controlling every event which happens on the network. It is also extensible to accommodate additional possibly unanticipated aspects of the network operation and management. Another goal is that the protocol implementation should be, as much as possible, independent of the architecture and mechanisms of particular hosts or gateways.

To achieve this objective each FSM that implements a sub-protocol has access to these objects, and their values are constantly updated by the protocol on its normal task while the synchronization is taking place.

The scope of the management information communicated by the SNMP is exactly represented by instances of objects defined in Internet-standard MIB or defined elsewhere according to the conventions set forth in Internet-standard SMI. The support of aggregate object type in the MIB is neither required for conformance with the SMI nor by the SNMP. The management information communicated by operations of the SNMP is represented according to the subset of the ASN.1 notation, that is specified for the definition of non-aggregate types in SMI.
Figure 20 depicts LS and DCS data interactions with the SMI specifications. These definitions are used to implement the SG and the pertinent DCS tables to perform an Open System protocol. Hence the protocol follows the SNMP requirement to be managed through a Proxy agent.

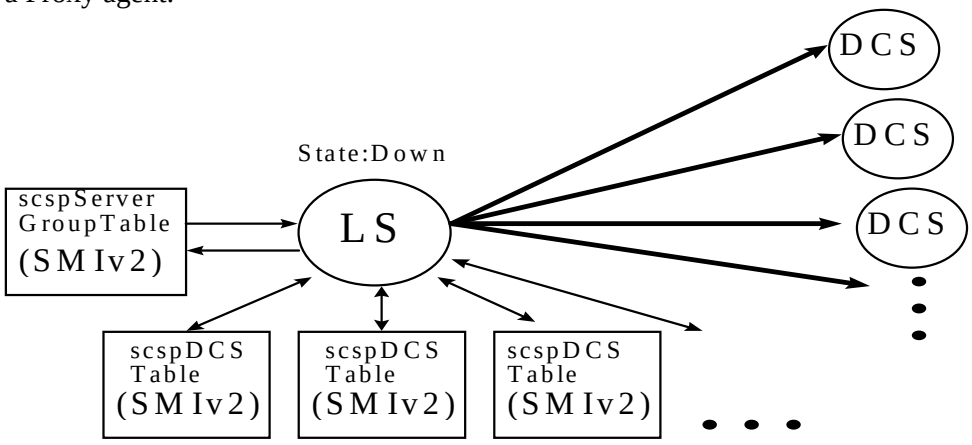


Fig. 20, LS and DCS data structures based on SMI definitions

The SNMP models all management agent functions as alterations or inspections of variables. Thus, a protocol entity on a logically remote host interacts with the management agent resident on the network element in order to retrieve (get) or alter (set) variables.

The SNMP embedded in the SCSP definitions

The idea of our *class* definitions arose from the purpose of a later development of tools that support the easy implementation of protocol handlers defined using ASN.1. Such tools read in an ASN.1 type definition and map it into an incore data structure(using the C++ language) which is capable of holding values of that type. Due to this way of implementing the SCSP protocol, we can develop an additional program that would be able to write to the incore data-structure to generate (in this local format) the value to be transmitted. Invoking a run-time routine provided by the tool would encode this value into the corresponding packet ready for transmission. The process is reversed on reception. This tool makes the mapping of the value of any ASN.1 type into an indefinitely large memory with dynamic memory allocation and with a known word-size (16 bits, 32 bits or 64 bits). To give an example, if the element is Boolean or an integer the word holds the element. If the element is a variable length character string, the word holds a pointer to a block of memory containing the string. The resulting structure is a strict tool that provides a more programming-language-friendly interface to ASN.1 style SNMP messages (Get request, Get Next Request, Set Request, Get Response, Trap) sent from the manager.
This kind of interaction between the program modules and the data structures that follow the SN.1 notation is shown in Figure 21.

- All the modules have been implemented in generic *classes* (C++)
- MIB objects based on ASN.1 to allow proxy agent management

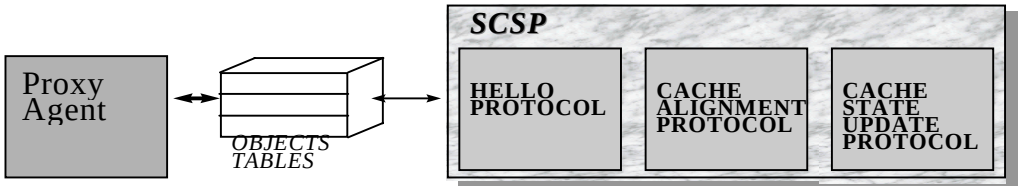
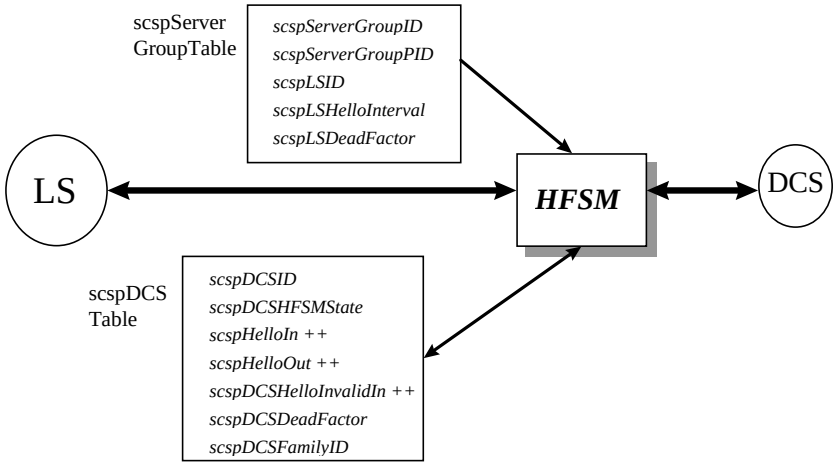


Fig. 21, Interaction between Proxy agent and SCSP through MIB objects

When the manager requires information from the MIB, it translates the information in our objects which are constantly updated by the protocol in ASN.1 messages format. The manager retrieves the information solicited through the corresponding port using UDP calls. All the request and response messages are transmitted or received via UDP calls using the port number 161. The Traps are unsolicited messages. It means that after a special event has occurred on the station being managed, the agent sends a Trap message to its manager to report the incident. Thus, when a trap is desired, a message is transmitted or received through port 162 with a UDP call.

An example of those data structures borrowed from the SMI specifications is shown in Figure 22. In that Figure we can see the objects used by the HFSM and how they are updated in the normal functions of the protocol. In this way the FSM structures are based on the ASN.1 guidelines to accomplish a portable protocol available through the SNMP mechanisms. The same characteristics are adopted by the rest of the FSMs in the SCSP development.

Hello Finite State Machine (*HFSM* class **)**



- Send Hello messages
- Start Hello Timers (*TimeHelloInt, timeHelloRec*)
- Check DCS State (*Down, Waiting, Uniconn, Biconn*)

Fig. 22, Data modules used in the HFSM

5.4.3 Packet structures

The message abstraction can be represented as a byte string of some length. For the purpose of our work, we use the message structure as an abstract string composed of different parts which depend on the protocol that is forming the message. The operations on the message object can be viewed as string manipulations. Hence, while processing an outgoing message, each of several protocols or in this case the FSMs, may add data to the message. Thus different strings are concatenated until the final message is ready. For incoming messages the process will be the other way around the message will be fragmented in various packets or data structures to be processed. Depending on which FSM has to manage the packet, it will strip the headers and the strings will be removed from the front of another string. Each FSM may save references to portions of a message for future use or to make the next decisions in the process.

Thus, any given byte may be attached to several different strings, removed from several different strings and referenced by several different HFSM states. For all those processes we manage the data obtained directly from the reception module which allocates the data in a character array. Its address is stored in a global buffer which is reachable from every part of the protocol.

From the programming viewpoint we choose basic structures for the data storage related to the program language characteristics. Thus we use **Bitfileds** for the packet structures

definitions [26] instead of **unions** technique because with this technique the code executes very fast and on some machines, the assembly code from this function may be smaller in size. The problem is that **unions** are not always portable due to the way that different processors store the bytes in memory. The same version of one program for instance runs correctly on "Intel" processors but displays bytes in reverse order on "SPARC" based workstations. For that reason, we would have to implement some preprocessors statements to identify the machine we are using and it may make the program complete and unreadable. Hence, we choose the **Bitfields** implementation.

5.4.4 Cache structures

For the cache implementation we have borrowed the structure of the OSPF cache format. Thus, we make it more reliable and similar to a well known routing table definition. The functions and data structures used in the cache are described in Figure 23. We can synchronise any type of data depending on the protocol that will use SCSP.

- The data structures are independent of the SCSP implementation
- We can use many different kind of data to be synchronized

Type of different database records :

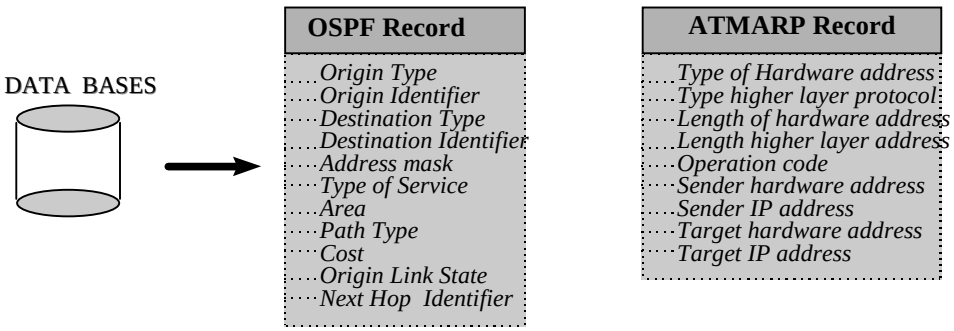


Fig. 23, Different data that can be stored in the Cache " class"

This is implemented in a separate class **class.cpp** to make it more independent from the rest of the SCSP protocol. This information must be used by the SCSP program and the protocol which is running with it and will make use of the SCSP facilities for its database synchronization. This implementation is needed because both of them will access client the same data. To avoid future interactions we make use of an additional data structure **list_CSA** which has the same information as the cache.

In Figure 24, we can see the most important structures used to manage the data stored in the cache. In the same figure there are the complementary functions to transmit the information from the cache to the protocol that will process the data.

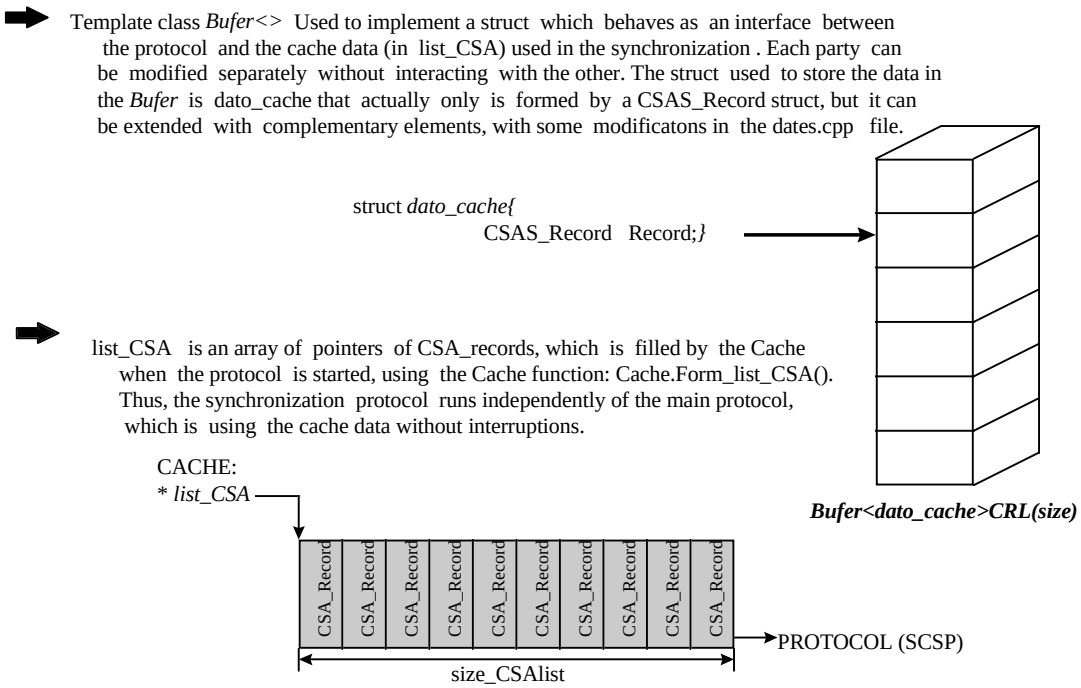


Fig. 24 Cache definitions and management process

5.4.4.1 DCS list on each machine

The first step in the protocol execution is to read the basic information from a configuration file. It must be allocated in the same directory where the protocol is running. Otherwise, we can read the file location from the standard input (from the keyboard buffer). It allows the user to manage the initial information. In this initial file there are the port number used for the message transmissions needed to fill the tables (based on the MIB objects) for the FSM. Those values, may be updated either by the results program or by the program which represents the SNMP agent. We have defined an individual class to take care of all these tasks. This class is represented in Figure 25. In this figure we can see that there are three functions which return the required values that the protocol needs to start its operation.

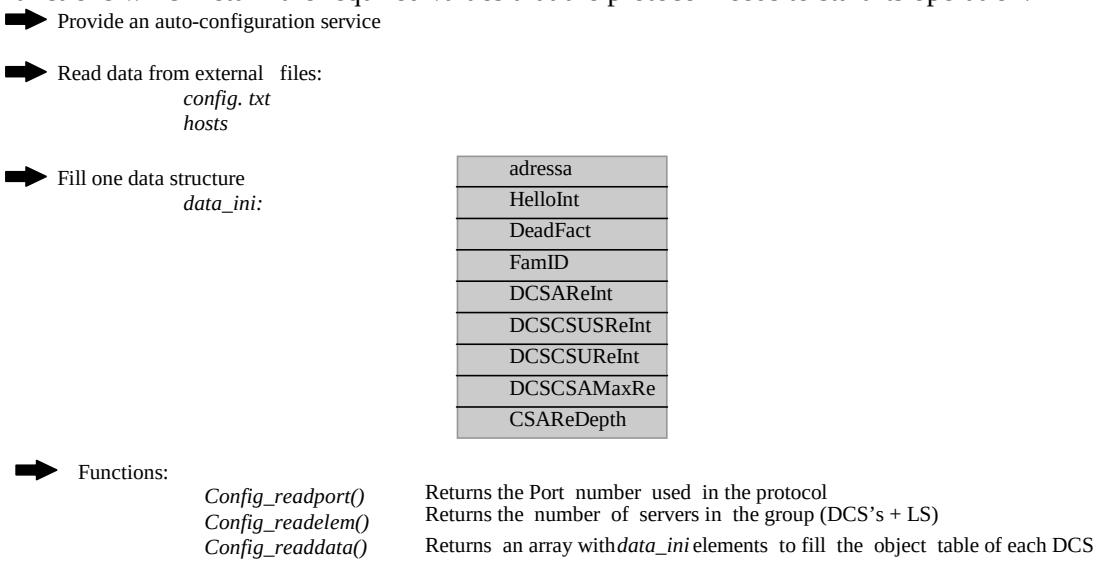
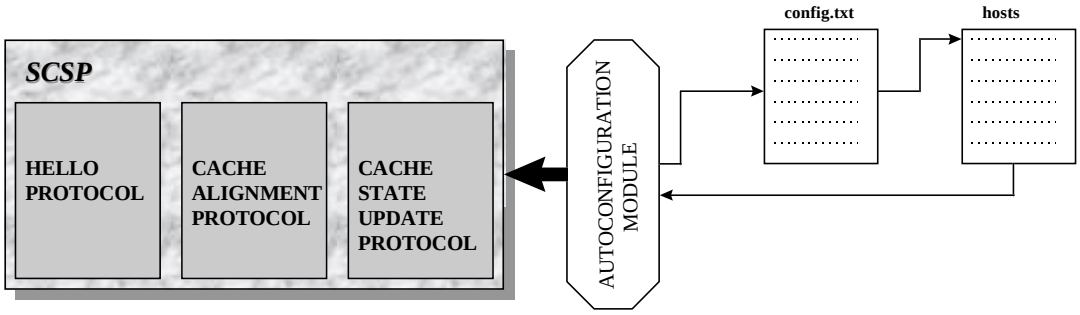


Fig. 25, Configuration class and internal functions

From these files we form the array with the addresses of all the servers which compose the SG. During the program implementation, we have tried to follow a modular way of working, and in Figure 26 we can see that in the configuration module this is also the case.



- Configuration information independent from Protocol Algorithm
- The configuration data is read from text files

Fig. 26, Configure module behavior

5.4.4.2 Buffer implementation

The buffer is an important structure in the program because it takes care of the data storage and memory allocation. An efficient buffer implementation helps with an efficient program behavior. For that reason, we have chosen a template definition to allow every type of data to be allocated in the same buffer. In this way, we have defined an independent *class* with its own functions for the data management from external requests. For the practical implementation we use a FIFO structure with a slight difference. The buffer acts like a cyclic structure. With this characteristic we obtain a useful module which can allocate any kind of data in the same space of memory without future buffer expansion or constrains. It avoids the reservation of a large amount of memory when new space is required. The characteristics of the buffer implementation are shown in Figure 27.

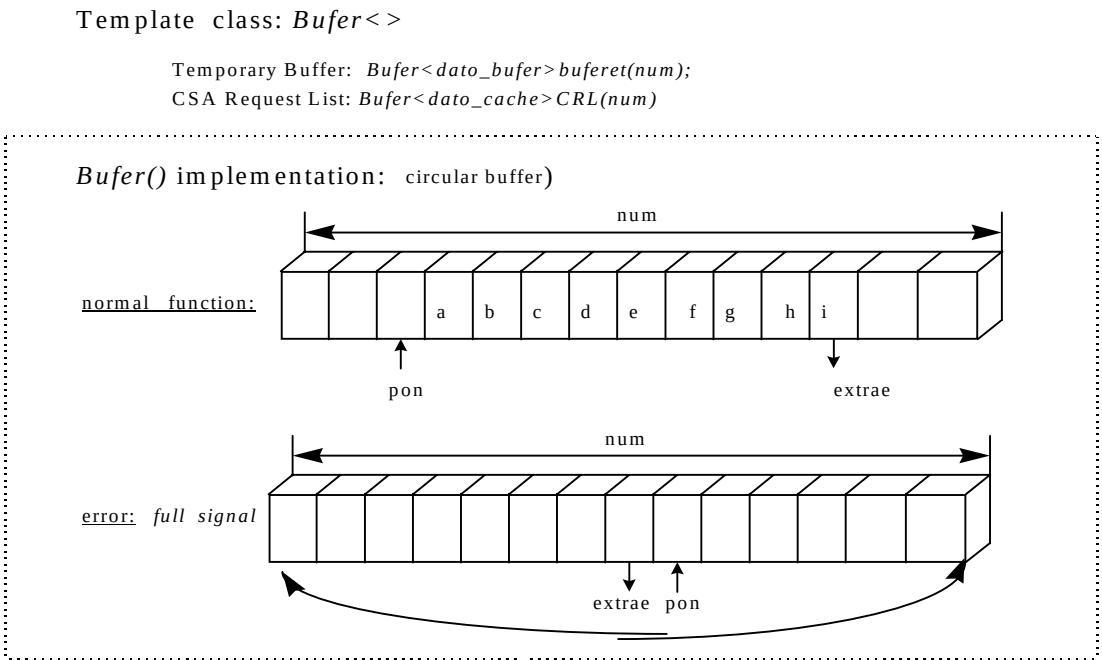


Fig. 27 Buffer Implementation

With the circular buffer we are reusing always the same space with the condition that the program is extracting the data from one side while the communications module is inserting the data that is being received in the other side. To avoid any collision between both components the extracting function have to check if the buffer is empty or not before picking up the data from the buffer. In the case that the protocol takes too much time for processing the message process and the communications module is continuously receiving packets which has to insert in the buffer an overflow in the buffer could be possible. This problem is solved by the inserting function that will check the space available on the buffer before inserting the new message. If there is no free space it will drop all the next messages that will arrive in the subsequent time interval. Thus, it could insert new data only after the program has extracted a message for processing and made some space available.

The same structure is used for other functions with the same characteristics like the CSA Request List (CRL).

5.4.4.3 CSA Request List (CRL)

The buffer definition of the previous section is also used for the CRL implementation. This structure contains a list of those cache entries which are more *up to date* in the DCS than the LS' s own cache. The CRL is filled from the data summaries which are composed by the protocol in the first step of its operation with the cache data. After the summary exchange step, we have in the CRL list the actual information that are in the cache or database when the protocol starts.
After that the SCSP have to check if its data is the same than in the rest of the DCS caches and update it in case that it is different. For that purpose, different message transactions take places according to the Hello Protocol or Cache Alignment Protocol. When the cache is aligned, the CRL will be filled with new CSAS Records from the data that have to be updated in the cache. Thus, the CRL acts like a buffer that stores the information summaries of the data that have to be updated. In the CRL, the protocol checks if there is any new information that needs to be refreshed in its own cache. The structure is depicted in
Figure 28 showing the process of inserting the data in the CRL and extracting the data that has to be sent with the **envia()** function.

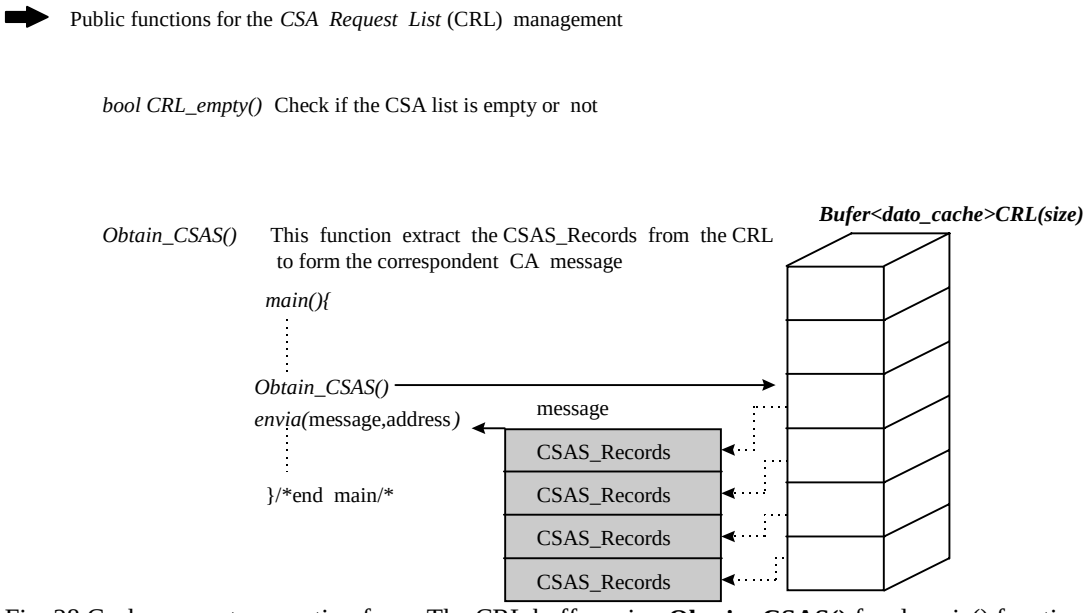
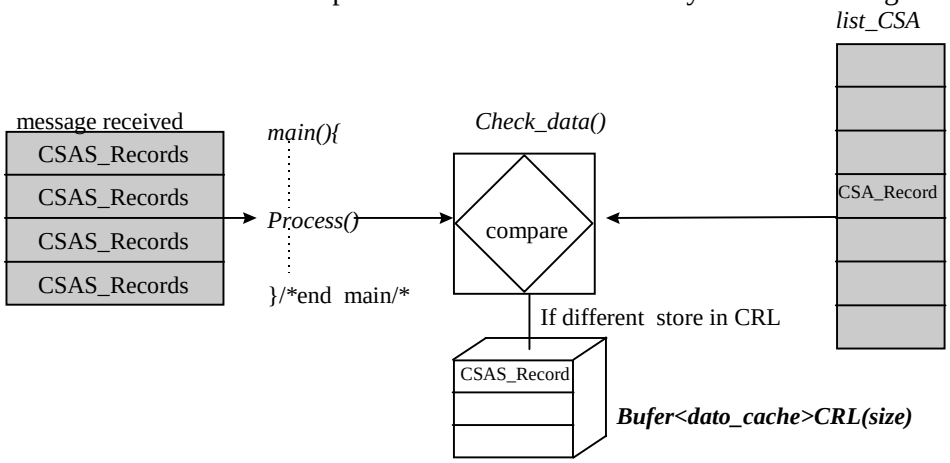


Fig. 28 Cache request generation form. The CRL buffer using **Obtain_CSAS()** f and envia() functions.

Other functions

There are many other functions that we need for the protocol implementation. Those functions are globally defined to be available from every part of the protocol. Where they will be needed. These modules provide a useful mechanism for managing the data among the different entities. A sample of those functions is briefly described in Figure 29.



➡ Public functions for the *CSA Request List* (CRL) management

bool CRL_empty() Check if the CSA list is empty or not

Obtain_CSAS() This function extract the CSAS_Records from the CRL to form the corresponding CA message

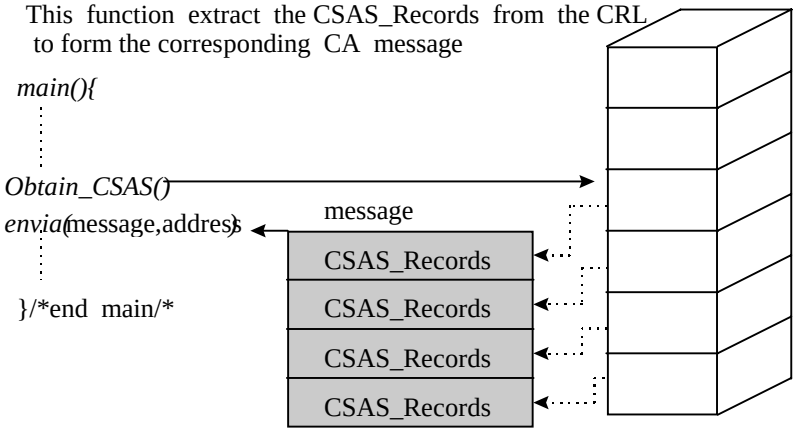


Fig. 29 Complementary functions for data management

The first function in Figure 29, shows the Check_data(). This function makes a quick comparation between a CSAS_Record and the data contained in the Cache, that is available in the list_CSA array in Figure 27, and it returns true if both are equal.

Another function shown in Figure 29, is Process(). This function compare the CSAS_Records received in a message corresponding data in the cache entry, but avoid interacting directly with the cache. We have a CSA_Records array filled by the Cache_data (is stored in list_CSA) just in case the information received is newer than the data contained in the list_CSA (we use the Check_data() function to check it) we have to update the Cache

data and for this purpose we need the CSA_record related to this CSAS_Record, so we have to insert this CSAS_Record in the CRL pendent to be requested in the next message.

The last function in Figure 29, is to extract the CSAS_Record from CRL and send packets requesting the newer data related to this CSAS_Record.

Chapter 6

Implementation interface of the monitoring

In this chapter, we describe the interface implementation. It is an interface for all the FSM of the protocol. The program is structured in three steps.
The first one, detail the facilities borrowed from the OS for the interface implementation.
In the second part, we explaine the data structures for the transactions between the SCSP program and the interface.
Finally, in the last part we illustrate the procedure used for the data management, which is borrowed from the SCSP basic data structures.

6.1 OS facilities

Following the directive of the whole work, we have tried to create a useful and adaptable interface. We have chosen the X-windows platform for its development. It provides a standard environment for the application software.
Applications which use X to operate a workstation display can easily be run on a variety of workstations from a variety of computer vendors, since all workstation vendors have by now accepted the X-Window System as a standard for their workstation hardware[27].

X-Window offers a rich and complex environment that we can use for tailoring our application according our needs without worry about the environment portability. The base window system interface is designed to work either within a single central processing unit (CPU) or between various CPUs. This system does not provide any special *backdoor* interfaces for privileged software. A C-language subroutine package known **Xlib** is provided with the X Window System for the purpose of letting applications interface to the network protocol and thence to the base window system.

The SCSP main program is written in C++, consequently the interface implementation follows the same characteristic and the X's base window system permit direct access to the **Xlib** functions with simple calls from C language programs. This feature allows quick transactions from our SCSP structures to the data formats that the interface will need to display showing the synchronization state in the SG. This feature let us not generate new structures to work with the same data in the interface program.
Then we only have to reuse the same definitions and classes predefined to work in the SCSP implementation. Therefore we can display immediately the protocol state and the different machine situation that is running in the protocol without any supplementary translations. A practical example of this translation is the architecture change from one system based on

Linux OS to Sun Solaris which was achieved with the addition of the extended X-window library XLIBEXT to the **makefile** that is used for the compilation process.

Any important feature of the interface is that the information it requires is read from a text file. That file is written by the SCPS main program when it is in an idle state, it means that in this moment when the process is not busy the file text is filled with the information of the protocol state. Afterwards, the interface can access directly to that file and display the information about the different machines that are running the protocol. This way of operation permits an efficient implementation of the interface program without slowing down the normal functions of the principal program.

6.2 Data structures

We have mentioned in the previous section that the structures needed are borrowed from the SCSP development. It means that we can reuse the information management facilities created before.

```
Configura  confi;
data_ini   *data;
Rsults_data   *results;
```

Thus, the Interface program reads an intermediate file that stores the state machine information and the values of the different variables which are updated by the protocol during its normal operation. This methodology avoids direct interaction between the interface and the protocol to obtain the recent information. Hence, the interface program has to read the last information from this file, refresh its data values and display the right information on the screen. The interface will make use of the same initial functions that are used by the SCSP for its configuration. These functions are available in the general **dates** file. Then, we only have to insert their definition at the beginning of the Interface implementation. The already known **configuration** function will provide the program all the necessary data for the Interface initialization.

```
tam = confi.Config_readelem();
data = confi.Config_readdata();
```

We make use of the graphical interface provided by X windows to show in a illustrative fashion the SG behavior during the synchronization process. A mesh of nodes is displayed on the screen where each of those nodes represent a different server of the group. If more information is required about a specific server we only have to place the mouse cursor over that server and click the mouse button. Immediately, the more detailed data about that node will appear in a window in a corner of the screen. At the same time the rest of the data will be updated. After that, if the mouse cursor drops this server area out, the window will disappear to allow displaying the information about other server if it is solicited.

6.3 Data transaction SCSP-Interface

The server data is updated with a local function defined in the program. That function reads the information from the file **results.txt**, which is filled by the protocol with the MIB objects information. Hence, the interface acts merely like a passive system with the unique purpose of reading the information that the SCSP has put in this file. Consequently, the

interface program could be considered like an agent in the SNMP. It has the limitation that it cannot modify any data nor send any messages to the protocol like a real proxy agent.

The function which takes care of picking up the data from the file and filling the data structure is:

```
Obtain_results(unsigned int &Last_clock, Results_data
*Results, int &len);
```

The first argument of this function is a reference time information needed to discard the information read during previous invocations. The second argument is the data structure which will be filled with the MIB objects information. It will be displayed in a window when the information required by the user.

As a result, the Interface program is suitable for displaying the Synchronization state in the group in a graphical format without interacting with the SCSP functions. It is more comfortable for the user who can check the server behavior every moment and display the most recent in MIB objects information from each server which have been updated by the SCSP protocol.

In conclusion, by using the facilities provided by the X windows, we obtain a simple and compact program that can be used in different vendors machines.

Chapter 7

Results

In this chapter we will briefly discuss the results of implementing the SCSP. We discuss a few issues that were encountered during the final tests in the equipment of the Laboratory.

7.1 SCSP implementation features

Figure 30 depicts the global distribution of all the states in the normal operation of the SCSP. It also illustrates the different steps and the data structures used to obtain a well tailored implementation of the protocol.

Protocol flow:

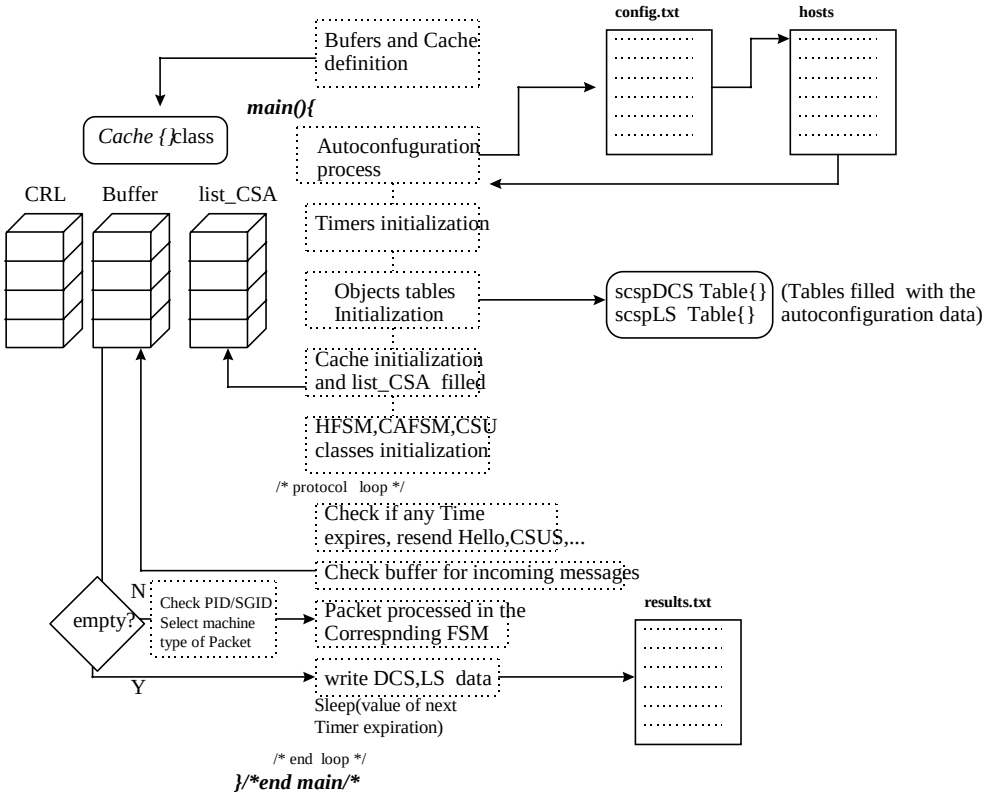


Fig. 30 SCSP general flow diagram

7.2 SCSP test

After the program implementation we have to search all the possible bugs that are hidden in the programming. It is a difficult task and for that reason we need some tools to follow the different steps during the whole performance of the program. In the project development we have used the software functions provided by “Free Software Foundation GNU”.

The GNU package also provides a useful debugger that we have used to run our program under its control and in case there was any breakpoint we can stop the program and analyze where was the problem.

In Figure 31, we can see all the modules of the implementation and despite that the protocol was defined in a modular way, we have to check each of those parts independently to assure their right behavior.

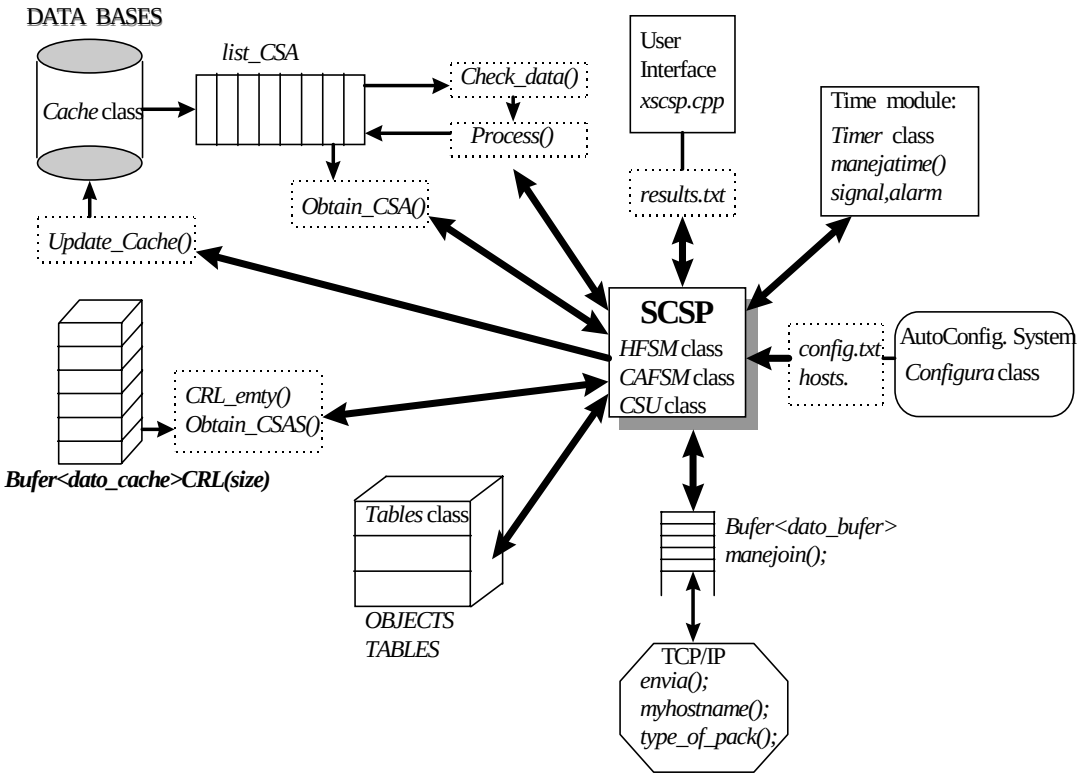


Fig. 31, Modules in the SCSP implementation

Another difficulty in the test was that the program have to be checked in different machines. It means that we have to run the program under independent debuggers on each machine where the SCSP is running. Fortunately, the GNU debugger provides a useful feature, namely the remote server debugger. Hence, we can run in our local machine a process with the debugger and its associated SCSP and at the same time run in a remote machine another process with the debugger and SCSP program, which is continuously reporting to our local machine the errors and breakpoints that happen in the remote platform.

Thus, this is the environment used for the program testing. The test was performed in different workstations in the Laboratory of Telecommunications and the final result is an

efficient program to update the information over the distributed entities in the Laboratory. The hardware environment was the Sun Solaris workstations distributed in the Laboratory.

Chapter 8

Conclusions and Future work

8.1 Conclusions

This chapter summarizes the software application that we have developed for data synchronization based on the SCSP specifications. The principal objective of this project was to provide a complementary program to be used by other systems for their data replication. For this reason, this software has been developed in a modular way to permit its integration with other mechanisms to achieve an efficient behaviour.

The main difficulty in this software development was handling all the amount of different data and interfaces which were involved in the program. The problem was solved in modular structures and independent functions that were tested gradually in stand alone functioning. Thus, the final purpose was reached, and the complexity could be managed. Finally, we obtained a standalone program that can be embedded in many applications to update their data following a standardized algorithm that can be globally adopted.

The aim of this work is to create a simple but efficient tool for data replication. The reason of the potential inefficiency is that in each server data may be moved and copied many times for the same analysis in different layers of the software system. The data may also need to be embedded inside complicated structures or primitives following strict specifications like in X.500 or others. This is a wasteful of processor time and resources. Thus, we have implemented the SCSP as an independent module that receives the data as a generic warehouse and takes care of its replication among the entities in the group.

8.2 Future work

Finally, we have a powerful program for database synchronization. This tool can be used as a stand alone protocol to work over various architectures with the same algorithm basis. We only need to make a few changes in some modules which actually are totally dependent from the OS where the protocol will be used. Those are the communications module (with its signals and event announcements) and the Timers module (which we have avoided to bound to the time signals in the OS with an additional internal function to update the time values continuously).

Actually, there are some Internet drafts and other work about this subject which make use of the SCSP facilities to provide synchronization and replication to their own database. It means that those protocols will use the SCSP like a complementary part of their whole implementation. An example of those protocols could be the Next Hop Routing Protocol

[28] in the case of database sync. We can see the MARS servers in a LIS where the LIS define the boundary of the SCSP SG [29] also in the ATMARP [30] protocol we can see how to fit the SCSP facilities and we could continue with many other examples.

From another point of view, this package could be a useful step in a modular implementation of the new incoming protocols. In that concept lies the idea of a modular protocol. In this way, each concrete function is achieved in several different blocks that take care of a specific task that must shape into an efficient and neat procedure forwarding the results to the next level that solicits its services. Hopefully, it would be the foremost solution in the next protocol generation development where the actual bottleneck is the packet processing on each node, server or host where the different protocols have to stripe the same packet many times for the routing analysis.

The philosophy we have chosen for the SCSP protocol development can be called a building_block design. Instead of relying on bulky protocol definitions tools or ad-hoc text encoding, the SCSP draws on existing well understood Internet technologies like UDP.

Hopefully this will lead to an easier later implementation and consensus building framework for other concrete protocols. It should also stand as an example of a simple way to leverage existing Internet technologies to easily implement new application-level services. The same philosophy can be adopted in other kind of architectures where we can readapt the SCSP program.

Hopefully SCSP implementation becomes widely used for providing the synchronization and replication of data for other protocols.

8.3 Applications of the SCSP

As we described in the second chapter the directories are becoming a key component of the service infrastructure in the emerging IP–based communications networks. The directories hold both static and dynamic information about the users of the communication services about the network and about the services themselves. Examples of controlled communication services using directories are the traditional e-mail, IP telephony and CuSeeMe. Due to IP Telephony, the service architectures of the traditional ISDN and Intelligent Networks and the new IP based communication networks will have to interoperate or converge.

We propose to use the Server Cache Syncronisation Protocol (SCSP) as a component of the emerging infrastructure for synchronising and replicating directories for services interoperable over IN and Internet communication networks. These directories could be accessed using a variety of protocols including the LDAP and even the Intelligent Network Application Protocol (INAP) in an IN. We hope that our implementation will contribute towards such experiments and development.

The best known example of directory is defined by the X.500 standards. In this case the directory information shadowing protocol (DISP) is used for data synchronisation and replication. The inconvenience of this protocol is its complexity and difficult interoperatibility among different architectures.

The alternative to the directory information access as we saw in the second chapter is the LDAP defined by the IETF. But the LDAP server data synchronisation and replication remain open issues. The Integrated Directory Services (IDS) Working Group has been chartered to facilitate the integration and interoperability of directory services in the Internet

and actually is working in the **LDUP** definition to provide the replication capabilty to the new LDAP v3.

GLP

Another application for the SCPS, is to achieve the roaming service over IP for every general IP service and in particular for IP voice[31]. There are many Regional Internet Servic Providers (ISP s) operating within a particular state or province looking to combine their efforts with those of other regional providers to offer services over a wider area. The national ISP s wishing to combine their operations with those of one or more ISP s in another nation to provide greater coverage in a group of countries or on a continent.

Businesses desire to offer their employees a comprehensive package of dialup services on a global system basis. The SCSP protocol could provide an architectural framework for the provisioning of roaming capabilities and it does directly support one of the most important requirements that must be needed by the roaming system namely the information updating about the elements of the architecture that compound any service.

To know how we can fit the SCSP in the basic structure of a new service, we will describe the elements required for the IP roaming.

Phone book. This a database or document containing data pertaining to dialup access. It includes phone numbers and any associated attributes.

Phone book server. This is a server that maintains the latest version of the phone book. The clients communicate with the phone book servers in order to keep their phone book updated.

The main functions involved in this architecture are:

Phone number exchange. Phone number exchange involves propagation of number of changes between providers in a roaming association. Current roaming implementations do not provide for complete automation of the phone number exchange process.

Phone book compilation. Once the ISP's phone book has received its updates it needs to compile a new phone book and propagate this phone book to all other phone book servers operated by that ISP.

Phone book update. Once the phone book is compiled, it needs to be propagated to the users. Standarisation of the phone book update process allows for providers to update user phone books independently of their client software or operating system.

Hence, the principal Phonebook requirement is the Phone book update protocol. For that purpose we can make use of the SCSP to create this portability service. The update protocol must allow for updating of clients on a range of different platforms and operating systems. The phone book provider only has to fit the data into the update records and provide a Key to check the information in the destination. The SCSP is like a lower layer that each party can use as a tool to synchronise the phone book. The algorithms and data management are hiddenfrom other providers. Contrary to the SCSP X.500 specifications are rather complicated. Each ISP needs to insert the data in a series of complex structures following strict directives and primitives. In case of SCSP, the update mechanism does not impose any operating system-specific requirements. The SCSP algorithm easily lends itself to portable implementation leaving a lot of room for vendor differentation in the cache implementation

itself. Future work could be focused in translating the SCSP package into the Roaming Service Protocol.

References

[1] J. Luciani, G. Armitage, J. Hlapern N. Doraswamy. *RFC 2334. Server Cache Synchronisation Protocol* . Network Working Group. Standards Track, April 1998

[2] C.J. DATE. *An Introduction to Database Systems*. Third Edition, Addison-Wesley System Programming, 1981

[3] ITU-T Recommendation X.500 (1993) . ISO/IEC-9594-1:1993,*Information Technology- Open Systems Interconnection- The Directory: Overview of Concepts, Models and Services.*.

[4] ITU-T Recommendation X.521 (1993) . ISO/IEC-9594-7:1993,*Information Technology- Open Systems Interconnection- The Directory: Selected Object classes*

[5] ITU-T Recommendation X.5181993) . ISO/IEC-9594-4:1993,*Information Technology- Open Systems Interconnection- The Directory: Procedures for Distributed Operation*

[6] ITU-T Recommendation X.525(1993) . ISO/IEC-9594-9:1993,*Information Technology- Open Systems Interconnection- The Directory:Replication*

[7] LDAP RFC 2251 M. Wahl, T. Howes, S. Kille *Lightweight Directory*

[8] LDAP v2 C. Sharon Boeyen, Tim Howes, Pat Richard. *Internet X.509 Public Key Infrastructure LDAPv2 Schema* <draft-ietf-pkix-LDAPv2-schema-02.txt>

[9] slurpd Implementation by the University of Michigan

[10] ITU-T Recommendation X.680 (1994) . ISO/IEC-8824-1:1993,*Information Technology - Open Systems Interconnection- Abstract Syntax Notation One (ASN.1):Specification of basic notation*

[11] ITU-T Recommendation X.681 (1994) . ISO/IEC-8824-2:1993,*Information Technology - Open Systems Interconnection- Abstract Syntax Notation One (ASN.1):Information objects specification.*

[12] Harrenstein K., Stahl M., E. Feinler. *NICNAME/WHOIS* RFC 954, SRI, October 1985

[13] Michael Mealling, J. Allen. *The Architecture of the Common Indexing Protocol (CIP)* 12/02/1997

[14] Integrated Directory Services (ids), http://www.ietf.cnri.reston.va.us/html.charters/ids-charter.html *Building an X.500 Directory Service in the US (RFC 1943)*

[15] Alan R. Downing, Ira B. Greenberg and Jon M. Peha. *OSCAR: An Architecture for weak-consistency replication*. DATABASES: Theory, Design and Applications. Edited by N. Rishe, S. Navathe and D. Tal , 1991

[16] M. Stonebraker, *Concurrency Control and Consistency of Multiple Copies in Distributed INGESS.* IEEE Transactions on Software Engineering, Vol. SE-5 3,May 1979

[17] M. P. Herlihy, *Genaral Quorum Consensus: A Replication Method for Abstract Data Types.* Technical Report CMU-CS-84-164. Department of Computer Science, carnegie Mellon University. December 1984
[18] OSPF RFC 1583, OSPF v2 RFC 2328 J. Moy April 1998 Obsoletes RFC 2178. (Status: STANDARD STD0054)

[19] K. McCloghrie, M. Rose. *Management Information Base for Network Management of TCP/IP based internets. MIB-II.* RFC 1213. March 1991

[20] Larry L. Peterson & Bruce S. Davie. *Computer Networks: A system Aproach.* Ed. Morgan Haufmann Publishers. Pp65, 1996

[21] M. Kirk McKusik. K. Bostic, M. Karels, J. Quarterman. *The Design and Implementation of the 4.4 BSD Operating System* Ed. Addison-wesley Publishing C. pp 212, 1996

[22] M. Accetta, R. Bolosky, D.Golub, R. Rashid, A. Tevanian. & M. Young. *Mach: A New Kernel Foundation for UNIX Development* USENIX Association Conference Proceedings, pp93-113, June 1996

[23] Internet Draft. *Definitions of Managed Objects for SCSP using SMIv2*

[24] Case, J., Fedor, M., Schoffstall, M., and J. Davin, *The Simple Network Management Protocol*, RFC 1157, University of Tennessee at Knoxville, May 1990.

[25] U. Warrier, L. Besaw *The Common Management Information Services and Protocol over TCP/IP (CMIT)* RFC 1095, April 1989

[24] Bjarne Stroustrup AT&T Laboratories. *The C++ Programming language*. Ed. Addison-Wesley pp. 71, 1993

[26] Oliver Jones. *Introduction to the Window X System* Ed. Prentice Hall pp 4, 1989

[27] J. Luciani.*RFC 2335 A distributed NHRP Service Using SCSP*. April 1998 (Status: PROPOSED STANDARD)

[28] J. Luciani, A. Gallo. *A distributed MARS service using SCSP.* 27 January 1998

[29] Gordon Good, *The LDAP Data Interchange Format (LDIF) - Technical Specification Filename: draft-good-ldap-ldif-03.txt* 22. February 1999

[30] Luciani, J. *A Distributed ATMARP Service Using SCSP* Work in progress.

[31] Matt Squire *A Gateway Location Protocol* Internet Draft, February 16-*1999. Draft-ietf-iptel-glp-00.txt*