Abstract

Author:	Julio Ramírez Yébenes
Title of the Thesis:	A Scalability Analysis of the Server Cache Synchronization Protocol (SCSP)
Date:	May 2001 Number of pages: 84
Faculty:	Networking Laboratory Helsinki University of Technology (HUT)
Supervisor:	Professor Raimo Kantola
Instructor:	MSc. Jose M. Costa Requena

In the today's new communications society, different fields of technology experience a strong boost in their development. The use of the IP network plays the main role in this framework.

The growth of voice and data services over IP produces significant scalability problems. Replication and caching technologies provide solutions to these problems. The Server Cache Synchronization Protocol (SCSP) uses a weak-consistency approach to provide a flexible replication methodology. The protocol works on limited scopes by creating replication groups composed of several servers. The use of smaller replication groups increases the consistency of the replicas within the servers involved.

Continuing from a former implementation of the SCSP, we study the functioning of the protocol and develop a scalable version. The new program relies on the use of threading techniques to generate different processing flows. Adjoined to this, the program commits with requirements as portability within several platforms and transparency of use.

In the last phase, the final product is put to the test. The program is augmented to self-generate performance reports and statistics. An interfacing prototype is developed to replicate via this implementation of SCSP. The tests evaluate the scalability and performance of the protocol in different stages of replication and different configurations of server groups. The data collected during the tests is analyzed and commented upon. The aim is to give a feasibility report of the current implementation and look for further improvements.

Keywords: Replication, cache, scalability, threading

Preface

This master's thesis has been written at the Networking Laboratory of the Helsinki University of Technology.

I am very grateful to my supervisor Raimo Kantola for the opportunity of working in this laboratory. I also thank Kimmo Pitkäniemi, Nicklas Beijar, Sampo Kaikkonen and Ignacio González for their help and company during my time here. I want to give special mentioning to my instructor Jose Costa Requena for his friendship, work, and support.

I would like to thank all my friends in Helsinki, specially the "winter people", for their constant support, they know I will never be grateful enough. I must say that I could not have done this thesis without their friendship. I also would like to thank my parents and brothers for their love and hours of phone calls, and my girlfriend Ana, for loving and bearing with me so long. To all of them for every second they gave me.

April 1, 2001
Helsinki, Finland

Julio Ramírez Yébenes

Index of contents

List of abbreviations and acronyms

CA	Cache Alignment
CAFSM	Cache Alignment Finite State Machine
CGMP	Cache Group Management Protocol
CK	Cache Key
CR	Critical Region
CRL	Cache State Alignment Request List
CRP	Content Routing Protocol
CSA	Cache State Alignment
CSAS	Cache State Alignment Summary
CSASL	Cache State Alignment Summary List
CSU	Cache State Update
CSU	Cache State Update
CSUS	Cache State Update Solicit
DB	Database
DBMS	Database Management System
DCS	Direct Connected Servers
DD	Data Dictionary
DFD	Data Flow Diagram
FSM	Finite State Machine
HFSM	Hello Finite State Machine
I/O	Input and Output
ICP	Internet Cache Protocol
IOD	Origin Identifier
IS	Information System
LDAP	Lightweight Directory Access Protocol
LDUP	Lightweight Directory Update Protocol
LS	Local Server
MQ	Message queue
MUT	Mean Update Time
NTP	Network Time Protocol
OM	Objects Model
PID	Protocol Identifier
SCSP	Server Cache Synchronization Protocol
SG	Sever Group
SGID	Server Group Identifier
TSAE	Timestamped Anti-Entropy
UTP	Update Transfer Protocol

List of figures

List of tables

1. Introduction

1.1. What is caching?

Traditionally, a cache is a computer memory with very short access time used for storage of frequently used instructions or data [1]. With the introduction of caching techniques in networking the definition is extended to: "*A small fast memory holding recently accessed data, designed to speed up subsequent access to the same data. Most often applied to processor-memory access but also used for a local copy of data accessible over a network*" [2].

Caches alleviate the origin data source from the most frequent requests (relieving bandwidth consumption) and reduce the access time needed to obtain them. Adjoined to this, caches can be located close to clients in order to minimize the costs of transporting the information.

A software cache consists of a local store of data and a subsystem which manages the contents. There are multiple policies to administer the collected information. The most important issue for caching performance is the "hit rate". Hit rate is defined as the fraction of all memory accesses that are satisfied from the cache [2]. The caching policy used in a cache must keep this rate as high as possible.

1.2. Replication and synchronization

To replicate is to create and maintain a duplicate copy of a database or file system on a different computer, typically a server [2]. Replication is a broadly used method for distributing the load of a server, e.g. NNTP [4] or FTP [5] mirroring. The aim of mirroring the cache contents is to place them in a closer or more accessible point for the clients. By replicating caches, the information is quickly accessible from different locations and the load of the cached server is reduced.

The drawback of replication is the need of synchronization of copies. A copy synchronizes with the original when their contents coincide [1]. Whenever an entry is changed, it must also be updated at the mirror cache. The concept of consistency [10] is bound to synchronization. The degree of consistency in a group represents the similarity of the replicas, i.e. a *full consistency* state means that all elements in the group are synchronized.

1.3. Research problem

The challenge now, in replication, is to develop scalable replication schemes that use efficiently the network topology and are able to keep replicas consistent. This master thesis undertakes the task of the scalability evaluation of one of those schemes, i.e. the **Server Cache Synchronization Protocol** (SCSP). It was originally suggested by J. Luciani [20] and implemented at the Networking Laboratory at the Helsinki University of Technology, Finland [23].

1.4. Tasks and steps

The goals of this master thesis are divided into two related topics. The first topic consists of providing the protocol with scalability and performance, i.e. eliminating bottlenecks and upgrading the code with parallel execution facilities. Adjoined to this, it is important to obtain a machine independent code, so the protocol can be portable to several platforms and tests can be extended to disparate environments

The second topic is the study of the protocol in terms of scalability[*]. The scalability analysis shows how the protocol functions when it works with an increasing number of inputs. The approach is to execute comparative analysis of the replication protocol to yield a feasibility report. A database environment is installed and used to recreate data sets for the trials. The interfacing module is developed to be the testing tool. Thus, a testing environment is built up, which can be further used later as application interface and monitoring tool.

In the first step, we analyzed the former implementation. The aim was to obtain a solution capable of executing the protocol steps for a high load context. This is required to begin with the second step. Therefore, we upgraded and debugged the program. We implemented also other improvements to enhance portability, transparency and performance of the protocol.

The second step is a set of scalability tests and analyses. The tests consist of different cases of server loads and group configurations. The aim is to obtain measurements for the resources consumption and the synchronization delay between replicas. There are two main parameters for the results. First, the tests and analysis is undertaken in terms of the replication load, i.e.

[*] *Ability to easily change in size or configuration to suit changing conditions. For example, a company that plans to set up a client/server network may want to have a system that not only works with the number of people who will immediately use the system, but the number who may be using it in one year, five years, or ten years* [17].

different database sizes and update rates. Second, the behavior under different group topologies is studied.

1.5. Structure of the thesis

The second chapter, *Theoretical analysis*, defines the scalability problem in the Internet. Two cooperative solutions emerge to solve it. They are caching and replication. The former section analyses several schemes of caching. It depicts the most used models and the state of the art of collaborative caching systems. The latter section shows how the replication of caches enhances the availability of resources in the net. The most common replication systems are also introduced.

The third chapter, *Development*, focuses on SCSP. It is subdivided into three sections. The first section describes the SCSP specification and analyses its replication process. The second section breaks down the protocol and describes the implementation of the different pieces. Adjoined to this, some improvements and changes in the implementation are explained. The third section explains the use of the program and its interaction with the interface module.

The fourth chapter, *Scalability test and analysis*, defines the test elements, the test plan and its results. The trials are undertaken for different server group configurations and update rates. The data obtained represent the resources consumption in terms of packet exchange, memory use, and CPU time.

The fifth chapter, *Conclusions and future work*, suggests different applications and improvements for this software.

Caching the web is not always a straightforward task. Different architectures fit the requirements better for different services. Added to it, cache replication is a technique used to spread caching services at a low cost. The following chapter depicts the state of the art of both practices in the Internet.

2. Theoretical analysis

This chapter relates the state of the art of the caching and replication techniques. Adjoined to this, it defines the position of the SCSP with this framework. The chapter is divided into two sections. First, we describe different caching methodologies. Second, we analyze two replication approaches and we depict the most interesting methods applied nowadays.

2.1. Different models of web caching

As the network grows, the most accessed services experience a degradation of the response times. This problem is typical in the Internet where individual servers receive huge service demands. One solution is to add more network capacity and use better performing servers. However, this idea tends to be insufficient in the long term. An exponential growth without due attention to scalability will eventually result in high network load and unacceptable response times. The most effective solution to create a scalable system rests on different caching and replication schemes [2]. Two interesting fields of application are the mobile networks and the Internet services. Figure 1 and Figure 2 depict the growth of these fields during the last years.

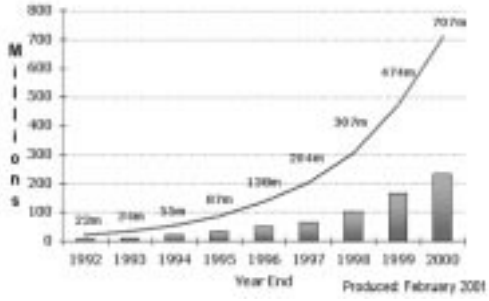


Figure 2. Growth of cellular subscribers up to Dec 2000 [8]

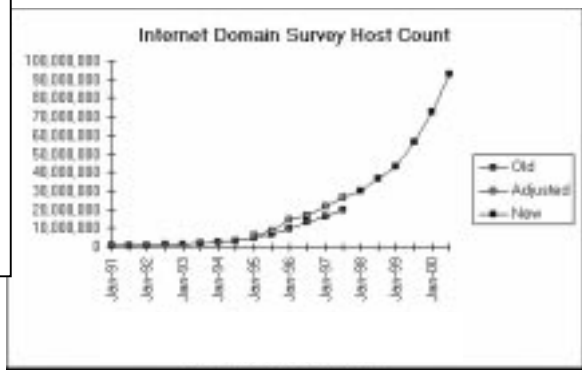


Figure 1. Growth of Hosts in the Internet [7]

2.1.1. Caching architectures

Caching can improve the perception of network performance in two ways [9]. First, when serving clients locally, caches hide wide area network latencies. On local cache miss, the original content provider will serve client requests. Second, if the network is temporarily down, the cache can provide, meanwhile, the most solicited information.

Different topological and functional situations use different caching strategies. Hereafter, as an overview, the main approaches are depicted.

❒ **Client Caching** is a local method. This cache is stored on the client's disk. It provides individualization of the cache service. The major drawback is that it cannot guarantee up to date entries as long as it does not send any query to the origin server.

❒ **Local Network Caching** has been shown to be more effective than client caching [13]. This approach is implemented mainly in web proxy servers. Proxy servers work as firewall access machines for subnets. Therefore, the traffic is managed through this service point. It makes the proxy the best candidate to undertake caching tasks. Nowadays, the terms proxy and web cache are used indistinctly.

❒ **Push Caching** [15] tries to keep data close to those clients requesting information. Thus, data is dynamically mirrored to caches that are more convenient. The major point here is to find a methodology that allows maintenance of the consistency of replicas.

❒ **Adaptive Web Caching** [14] is trying to optimize the global data dissemination. The approach consists of multiple, distributed caches. The caches join and leave cache groups dynamically according to policies based on demand, i.e. self-organization of caches is a response to scenarios where the access to elements evolves in time. Cache meshes are organized into overlapping multicast groups. Two protocols are used to implement the cache group management and communication. The Cache Group Management Protocol (CGMP) controls the formation of groups. All caches and web servers automatically organize themselves into groups according to the CGMP. The Content Routing Protocol (CRP) is in charge of the administration of the cache contents for the mesh.

It is important to note here that SCSP can be studied as a useful tool to fulfill the requirements of intra/intergroups replication for the Adaptive Network Caching model.

2.1.2. Cache design

Caching systems are designed to obtain high performance, scalability and reliability [9]. Different techniques enhance these features. Two topics regard networking at this point. They are hierarchical caching and intercache communication.

❒**Hierarchical Caching** is a topological enhancement to the caching methods. It is based in the concept of "*hierarchical* routing" [2]. Hierarchical routing simplifies the network by breaking it into a hierarchy of smaller subnetworks. Each level in the hierarchy is responsible for its own routing. In caching, each level holds an independent cache. The Internet has three basic levels: the backbones, the regional networks, and the stub networks. The backbones know how to route between the regional networks. The regional networks know how to route between the sites (or stub networks). Each site (being an autonomous system) knows how to route internally.

The idea here is to distribute the load of requests and minimize the access latency. The "Harvest

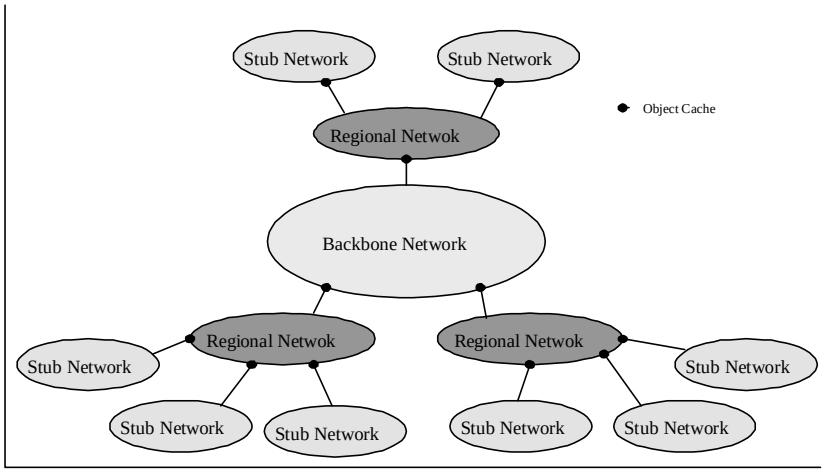


Figure 3. Example of a Hierarchical Cache Arrangement

Cache" is the main example [12]. Caches are accessed hierarchically in a net to subnet sort. Each cache in the hierarchy decides independently whether to fetch an entry from its parent or from its sibling caches according to a resolution protocol [12].

❒ **Intercache Communication** policies are a set of protocols for maintaining contents among cooperative caches [9]. Caching systems are composed of numerous distributed caches (to improve scalability and availability of the contents). The most used communication protocols are the Internet Cache Protocol (ICP), Cache Digest, CRP, CARP (Cisco) and WCCP

(Microsoft). The study of their features is out of the scope of this thesis. However, it is important to notice some important similarities in the communication process used in SCSP and these protocols. Thus, it can be put forward that SCSP uses a request-reply paradigm over UDP. The group-based replication and the creation of overlapping caching meshes to forward objects among groups is similar to the one used in CRP. Another important technique adopted for cache-to-cache communication is the notion of "cache digest" (as in SQUID and Summary Cache). Digests are used to reduce intercommunication load by exchanging summarized information instead of the whole object.

2.2. Cache replication and scalability

Replicated systems are used to improve the availability of contents. As sources are closer to clients two benefits raise: communication locality and load sharing. These increase scalability, alleviate bottlenecks and reduce net traffic and communication latency [16].

Replication protocols consists of two procedures working in parallel: a group membership scheme and a group multicast protocol. The first one is in charge of the management of elements and connections of the group. The second floods the required entries between replicas. The main differences between replication protocols come from the message delivery policies. Implementations are classified into two replication trends. They are *Strong Consistency Protocols* and *Weak Consistency Protocols*.

2.2.1. Strong consistency approach

In the strong consistency approach, the system provides a multicast service which ensures that every message is received in a controlled order. Two elements can never differ at any time by more than a limited number of messages (e.g. *Lazy Replication* or *Psync*).

A relevant approach is called "two-phase-commit" [22]. Each time a transaction occurs in one server, all copies are updated at the same time. The update is not considered stable until every entity involved has committed with the change (or rejected it). Mainly, real-time applications make use of these techniques. Despite the good results it can provide, this solution does not scale well and, in terms of network performance, produce costly waste of resources. Delays in transactions produce high communication costs and overhead.

2.2.2. Weak consistency approach

In this approach, each member of the group keeps a copy of the state of the other elements. A group communication policy is used to coordinate changes within the elements. The degree of consistency of replicas depends on the speed and reliability of the multicast protocol to exchange messages.

Weak consistency implementations allow any two members of the group to have different copies at any instant. The idea is that all elements in the group will receive the same set of update messages. The group eventually converges to a synchronous state. Timestamped Anti-Entropy (TSAE) protocols are main examples of the weak consistency approach [16].

2.2.3. Strong vs. weak consistency

Compared to strong consistency, weak consistency can have important benefits. The main advantages appear in wide-area networks and mobile systems[*]. In these scopes, strong consistency has two major drawbacks. First, it requires expensive protocols and does not perform properly when communication is unreliable (i.e. problems to deliver messages orderly). Second, if the network is partitioned by a group element (i.e. mobile systems), the efficiency of the system is degraded.

Moreover, weak consistency protocols use fewer network packets. This reduces consumption of communication resources. Adjoined to this, weak replication allows caching and delayed operations. These features provide "*fault tolerance*". Fault tolerance is the ability of a system to stand ruptures of some connections without affecting the overall functioning. For mobile systems, it means the ability to reconnect and synchronize after long periods off-line.

2.2.4. Replication models

❒ **Timestamped Anti-Entropy** (TSAE) protocol is a weak consistency method [16]. It performs delayed communication establishing "*anti-entropy sessions*". Instead of sending updates as they are received, TSAE places them in a queue. Pairs of elements periodically contact each other to exchange these queues. That is defined as an "*anti-entropy session*".

[*] Mobile system is a dynamic group of cooperating elements. Any of them can disconnect or reconneced to the group at any moment.

The session begins allocating a timestamp on the queue. The timestamp identifies the newness of the updates. Next, the pairs exchange summaries of their update queues. Summaries consist of the minimal information needed to uniquely identify an update in the pool of messages. The timestamps together with the summaries are used to evaluate which updates must be sent to the pair element of this anti-entropy session. In the last step, updates which are not in both servers are exchanged and acknowledged using a reliable protocol. If at any moment the procedure fails, the anti-entropy session is aborted and changes are discarded.

❒ **Flood-d** [10] is the replication method used in the Harvest System [12]. It was proposed as an extension to TSAE with added features. Flood-d is designed to scale for thousands of weak consistent replicas, i.e. replication systems which might not achieve full consistency.

The replication elements are organized into groups. There are two main enhancements proposed in this system. The first is the use of a hierarchical inter-group topology. It improves system scalability by limiting the amount of consistency information that each replica needs to keep [6]. The second regards the internal topology of the group. Elements of the group change dynamically. A group management algorithm is executed to obtain a resilient topology and optimize bandwidth use.

As in TSAE, Flood-d also uses timestamps to identify new messages and determine duplicates. The timestamp is a sequence number. New updates are flooded into the group following the established logical topology. New topologies generate different timestamp sequences. Thus, changes in the topology are communicated without any need for new procedures.

❒ **Lightweight Directory Update Protocol** (LDUP) [18] is the replication approach implemented in the *Lightweight Directory Access Protocol* (LDAP) [19]. It is a widely developed weak-consistency replication protocol.

In this protocol, replicas are grouped hierarchically. As with TSAE, it uses timestamps for the messages. However, there are three big differences with the above protocols. First, replication occurs in a supplier to consumer base. Second, no multicasting protocols are used. Third, concurrent replications are not allowed in a consumer server. These two last points are a comparative drawback to the previous protocols in terms of performance, but provide consistency to replicated data. Adjoined to this, this protocol supports several replication procedures instead of a fixed one.

The replication procedure is previously negotiated between pairs of connected servers. Two processes rule the replication session, *a* "*Replication Schedule*" and an "*Update Transfer Protocol*" (UTP).

❒ The *Replication Schedule* manages the times of the replication sessions. Two mechanisms can be selected. They are "*Schedule Driven*" and "*Change Event Driven*". The former defines fixed periods to initiate the replication sessions. In the latter, sessions are initiated by the supplier of changes as soon as new updates are received.

❒ The *UTP* selects among two protocols: "*Full Update*" and "*Incremental Update*". These are the basic protocols although others can be implemented. The former transfers the whole contents from one replica to another. The latter sends only those updates required to achieve synchronization.

The replication session can be managed by any combination of these two processes (E.g. Schedule Driven with Full Update, Schedule Driven with Incremental Update, etc…). Also both replication schedules can be used in a configuration, e.g. the server can use Change Event by default and execute Schedule Driven if no changes appear within thirty minutes.

The changes that were received by the server are acknowledged to the origin and forwarded to other replicas. Acknowledgements are used to track the exchange process.

❒ **Server Cache Synchronization Protocol** (SCSP) [20] is another weak-consistency protocol for cache replication. It inherits its replication scheme from OSPF [21].

SCSP focuses on replication of cache information. Caches are bound to one or more groups. Replication is done on a group basis. Changes of entries in a cache are replicated to other caches in the group. SCSP has two basic features. First, it allows concurrent replications in a server. Second, it optionally supports multicast services.

In terms of the replication procedures described for LDUP, SCSP is categorized as a Change Event Driven protocol. However, SCSP shares with LDUP the same UTPs, i.e. Full and Incremental Update. The former is executed during the synchronization of caches (i.e. the Cache Alignment phase). The latter is used to keep copies synchronized (i.e. the Cache State Update phase). SCSP is the protocol we are concerned with in this thesis and it is further described in the next chapter.

3. Development

This chapter relates the development of a new version of the SCSP implementation. In the first part, we describe the protocol and its environment. At this point, we study the protocol to find out its possible weaknesses. We identify the different parts as different elements of execution. In the second part, we describe the implementation of the elements of the protocol.

3.1. SCSP description

SCSP is intended to perform synchronization and replication tasks among distributed caches. The replication model is analogous to the one used in OSPF. OSPF uses *Hello, Database Synchronization* and *Flooding*. Likewise, SCSP names for its phases are *Hello*, *Cache Alignment* and *Cache State Update Protocol*. The main difference of SCSP against other protocols is that no algorithm for routing or group administration is implemented, i.e. SCSP is a pure replication protocol with no added load.

3.1.1. Elements

SCSP uses two basic communication items:

❒ A "*Cache State Alignment*" (CSA) record stores an update of a cache entry. The CSA is composed of a CSAS and a field containing the update information.

❒ A "*CSA Summary*" (CSAS) record identifies CSAs. Each CSA has one CSAS containing three major fields. First, a unique identifier for the CSA record or "*Cache Key"* (CK). Second, a sequence timestamp. Third, an "*Origin Identifier"* (OID) pointing the server that generated the CSA.

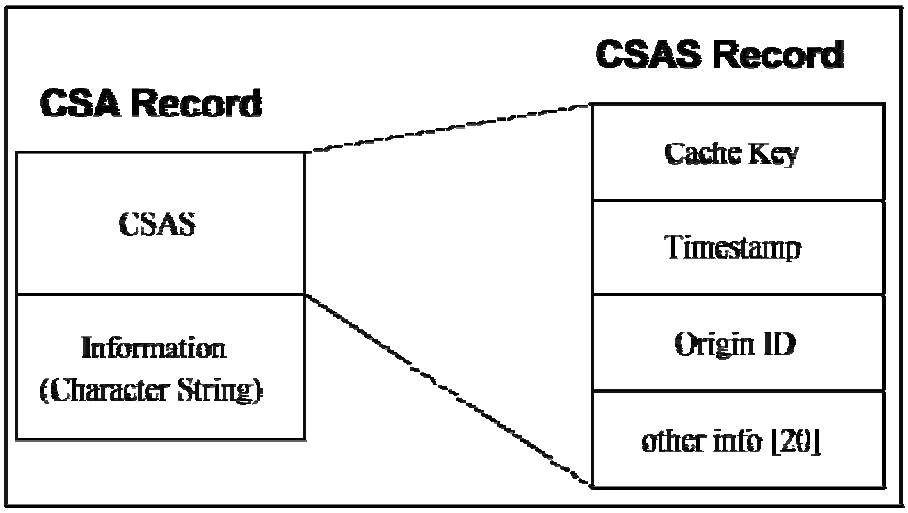


Figure 4. CSA and CSAS

3.1.2.Topology of replication

The replicas are statically organized into Server Groups (SG). Each instance of SCSP mirrors its information only with other replicas in the same SG. An SCSP instance running in a server defines two concepts, a "*Local Server*" (LS) and a group of "*Direct Connected Servers*" (DCS). LS refers to the server executing this SCSP instance. DCS refers to every server connected to the LS in a SG.

A running SCSP instance has two identifiers: a "*Protocol Identifier*" (PID) and a "*Server Group Identifier*" (SGID). PID refers to the application protocol that uses SCSP for replication. Examples of such application protocol are NHRP [30] and TRIP [43]. The SGID defines the replication scope in terms of servers involved. Thus, each running SCSP instance is identified with a pair of identifiers PID/SGID. This feature allows concurrent replications in an LS.

For a single PID/SGID, any server can be connected to more than one DCS, and can synchronize simultaneously with all of them. The SG works as a distributed cache. Thus, every new CSA received in an LS is flooded to other DCSs. Control of duplicated messages is done by tracking the CSAS of the CSAs received. Timestamps are needed in the case of collision of versions for a CK. The protocol identifies the replica with higher timestamp as newer.

SCSP allows communication between SGs. Any LS can be configured to replicate within several SGs. These servers work as connection nodes. Although replication is done inside each

group, when one update is received in a node, it loops sending the CSA to every DCS in any of its SGs.

Several instances of SCSP can provide replication services to different applications on an SG. It is achieved by using different PIDs, e.g., several instances can execute synchronization procedures at the same time on a single server for a MIB [31] and NHRP [30] without interfering with each other.

3.1.3. Functioning

Three steps are the basis for the SCSP working model. The "*Hello Protocol*" manages and tracks the communication of the LS with its DCSs. The "*Cache Alignment Protocol*" is in charge of the initial synchronization of replicas. The "*Cache State Update Protocol*" keeps flooding the changes between synchronized servers to maintain the group consistency. For a detailed description of the FSMs refer to [23]. The protocol works as follows [20]:

At the beginning, the **Hello Algorithm** is executed. One independent state machine runs for each DCS. It is called the "*Hello Finite State Machine*" (HFSM) [23]. The state machine tracks the state of the DCS connection. It can be bidirectional, unidirectional or nonfunctional. The LS tries to contact the DCSs and determines the state of the communication. This is done by sending "*hello*" messages periodically. These messages must be acknowledged by the destination to confirm that the communication is active. When bidirectional connection is achieved, the SCSP starts the *Cache Alignment Protocol*.

The sending of hello messages is maintained all along the replication process. It works as a "*keep alive*" procedure. This way, the LS tracks the state of its connections with every DCS.

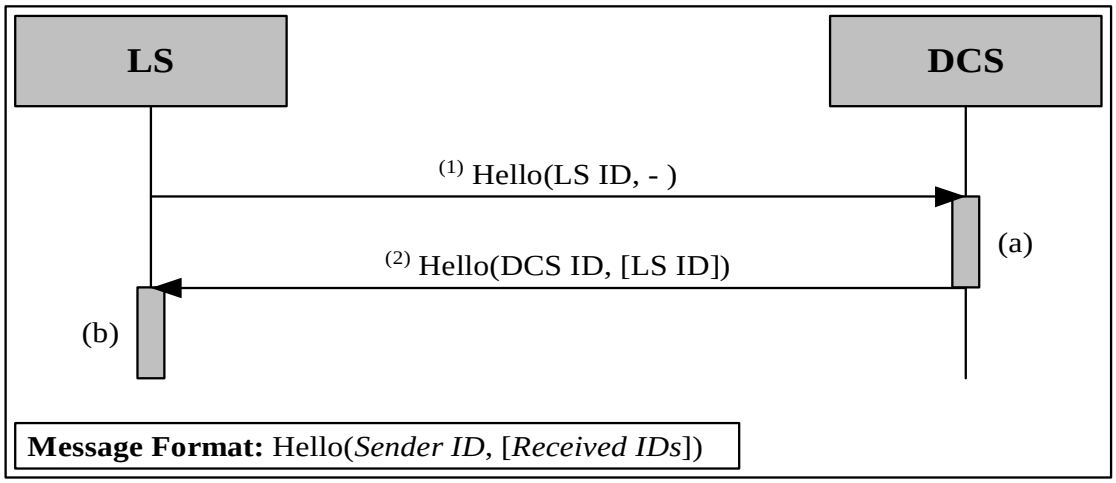


Figure **5** depicts the message interchange required to achieve a bidirectional connection in a replication session. In Figure 5, only the two major parameters are represented for simplicity. The "*Sender ID*" contains the identifier of the server sending the message. The "*Receiver IDs*"* contains a list of the DCSs from which the DCS has received a Hello message before. For further detail on the packets refer to [20].

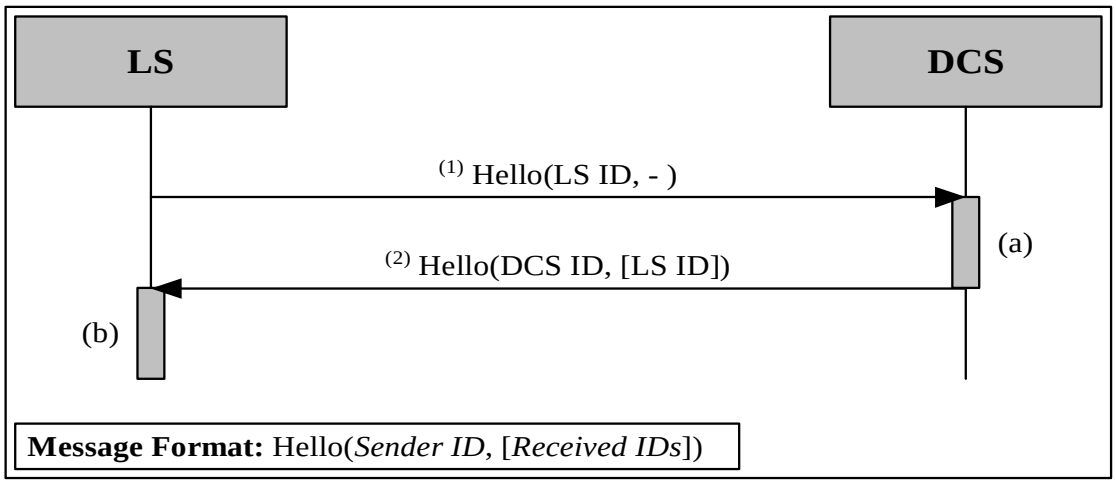


Figure 5. Trace Events Diagram of the Hello Protocol in SCSP

Procedures description

(a) Add the *Sender ID* to the *Received IDs* list.

(b) If the LS ID is included in the *Received IDs* list, the LS transitions to the bidirectional Connection state. In other case, the LS transitions to Unidirectional Connection state.

Messages description

(1) The LS begins the replication session sending a Hello message which includes its server ID, i.e. LS ID.

(2) The DCS sends a Hello message with its server ID (I.e. DCS ID) and its local "R*eceived IDs*" list.

The **Cache Alignment Protocol** deals with the initial synchronization of information, i.e. the servers exchange their current caches. As in the previous stage, one independent FSM runs for each DCS. It is called the "*Cache Alignment Finite State Machine*" (CAFSM) [23]. Figure 6 describes the process of alignment. In this figure, only the major fields of the CA messages are depicted. For further detail on the packets refer to [20].

* We use the name "Received IDs" for clarity, although it is referred as "Receiver IDs" in [20].

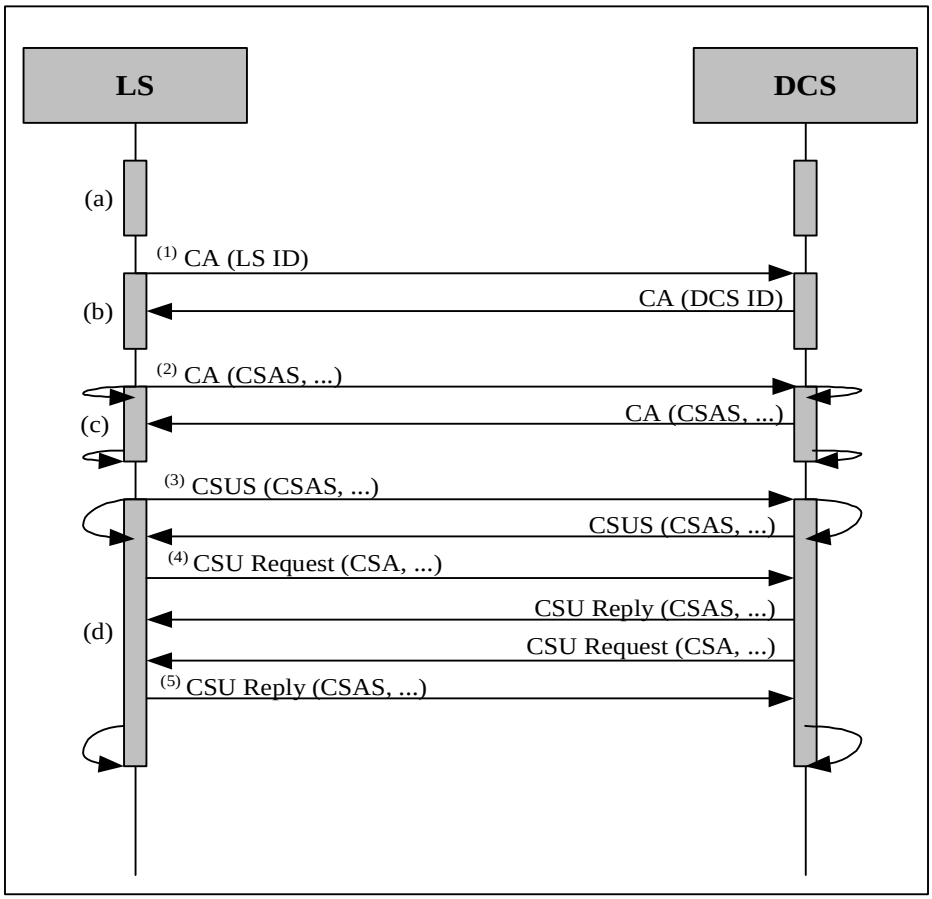


Figure 6. Trace Events Diagram for Cache Alignment Protocol in SCSP

Procedures Description

(a) Bidirectional Connection state achieved.

(b) Master/slave negotiation. Replication roles are established.

(c) The LS exchanges summaries of its content with each DCS. CSASs received from the DCS are compared against the local cache records. The LS creates a list of records in the DCS which do not appear in its cache. This list is called "*CSA Request List*" (CRL).

(d) The LS requests to the DCS the CSAs corresponding to the CSASs in its CRL. The LS receives the CSA records requested. When the CRL is emptied, the LS transitions to the "*Aligned*" state. This means that the synchronization has been achieved. At the alignment point, the CAFSM finishes and gives rise to the *Cache State Update Protocol*.

Messages Description

(1) CA messages (for Master/Slave Negotiation) are labeled with the *Sender ID* to negotiate replication roles[*]. No CSASs can be included into them.

(2) CA messages (for Cache Summarize) transport the CSASs of the entries in the local cache. Should not all the CSASs fit in one message, several messages are sent.

(3) Cache State Update Solicit messages (CSUS) transport the CSASs in the CRL of the LS to the DCS. These messages are used to request CSAs from the DCS. Should not all the CSASs fit in one message, several messages are sent.

(4) Cache State Update (CSU) Request messages transport CSAs from the DCS to the LS. Should not all the CSAs fit in one message, several messages are sent.

(5) CSU Reply messages transport acknowledgements from the LS to the DCS. Acknowledgements are CSASs of the received CSAs. These messages are sent when a CSU Request has been received in the LS. Should not all the CSASs fit in one message, several messages are sent.

The **Cache State Update Protocol** is the last stage of the SCSP. At this moment, the caches involved in the replication process are identical. Now, only new updates must be synchronized.

Whenever an LS detects changes in its cache, it sends a notification of the changes to the local instance of SCSP. The LS creates a CSA record for each update and floods it to all the DCSs in a CSU Request. When the DCS receives a CSA record, it checks if it is new in its local cache. The server receiving the CSU Request sends a CSU Reply message acknowledging the change.

It is possible that the DCS has a newer record for that entry in its cache than the one received. In this case, the acknowledgment message sent is filled with the summary information of the local record and a flag set to one (For detailed explanation about the flags in the SCSP messages refer to [20]).

The message flow is the same than the one used in Figure 6-procedure (d), excepting the CSUS messages exchange. SCSP keeps the *Aligned* state until the HFSM transitions from Biconnected to any other state.

[*] During the replication session, the master is in charge of tracking the messages sequencing.

3.2. SCSP environment

The replication system is divided into three modules. The SCSP module is the main one. Secondly, the caching tasks are under control of a Database Management System (DBMS). Third, an interface is in charge of communicating between all the modules, i.e. SCSP, DB and applications using the replication service. This part describes a working environment case for the SCSP, i.e. interface and DB. Figure 7 depicts the interrelationships between the different modules in the system, and the role played by the interface.

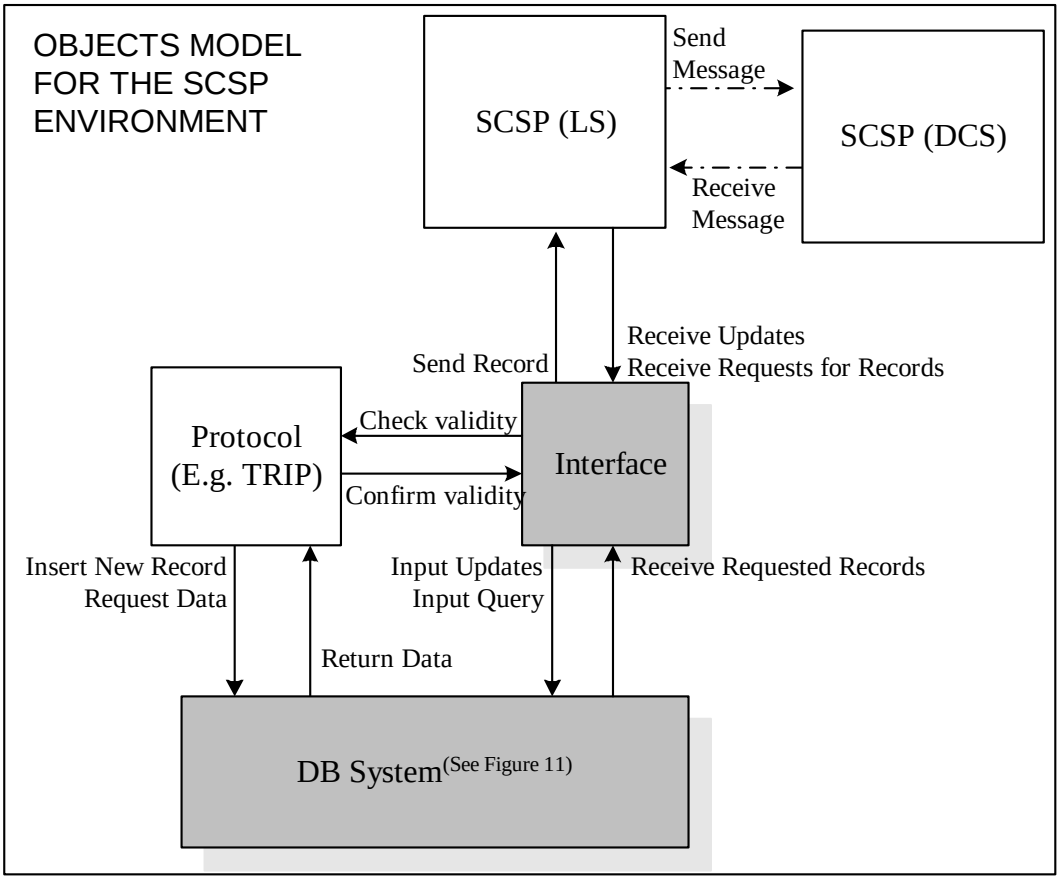


Figure 7. SCSP environment

The following figures help to understand the flow of events executed for the three basic operations of replication. Figure 8 describes the sequence of events executed when a local update is generated in the cache by the user application. Figure 9 relates the inverse process, i.e. when a remote update is generated in a DCS. Figure 10 indicates the messages exchange in the SCSP environment when a DCS is reinitialized. In this case, the LS needs to retrieve the information stored in its cache. The records are kept in the DB System. Therefore, the SCSP in the LS must request the records to the DB System through the interface.

The events "*Request Data*" and "*Return Data*" in Figure 7 are not directly involved in the SCSP fuctioning. They represent the basic intercommunication between the user application and the DB System (I.e. data storing and reading).

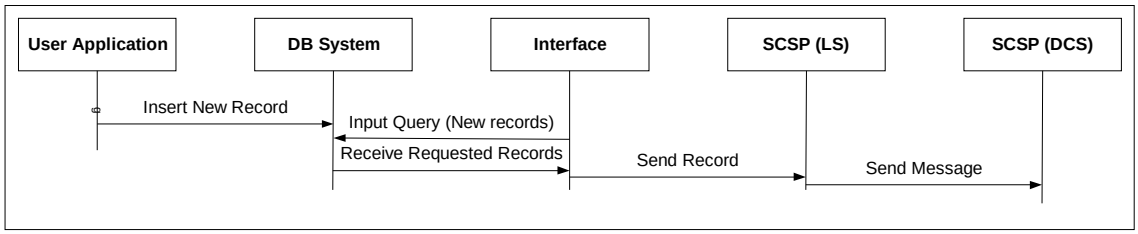


Figure 8. Flow of a local update in the SCSP environment

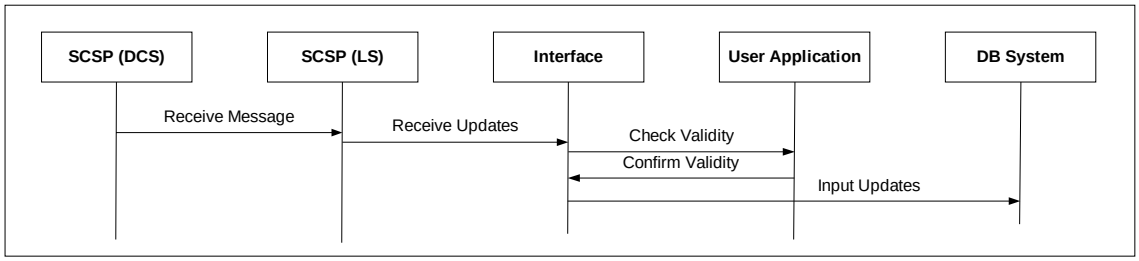


Figure 9. Flow of a remote update in the SCSP environment

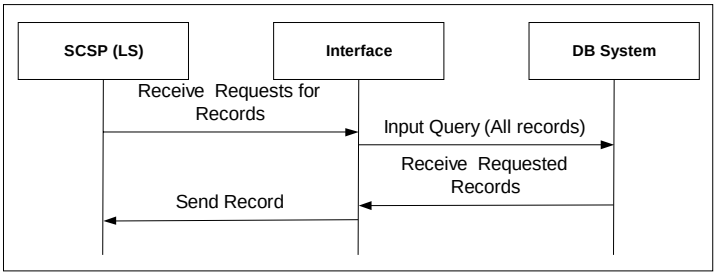


Figure 10. Flows in the SCSP evironment for a DCS reinitialization

Figure 11 shows a model of internal functioning of the DB system. This chart shows which parts of the DB system deal with the different messages. The following paragraphs give an explanation of the figure.

The messages of the application are entered directly into the information system holding the data, i.e. "*Insert new Record*" and "*Request Data*". The results are returned to the application by the "*Return* Data" flow. The DBMS is in charge of processing the requests and retrievals of information. Indeed, a cached DB system consists of two components of storage, the "*Information System*" (IS) and the "Cache". The former component holds all the information

managed. The latter holds a relatively small copy of the information in the IS. This copy is intended for fast-retrieval data. The selection of records to store in the cache is made according to disparate policies. The "*Caching Procedure*", internal to the DBMS, is in charge of selecting the records that must be cached. The flows managing the data cached by the "*Caching Procedure*" are "*Get Records*" and "*Update Cache*".

In order to use the performance of the system, the queries are passed first to the cache ("*Query*"). If the data is not found in the cache, it is looked into the DB itself ("*Query Not Cached Data*").

When the SCSP is added to the DBMS, it can modify cached data directly with no intervention of the "*Caching Procedure*". Therefore, the "*Caching Procedure*" must be augmented with procedures which detect changes in the cache ("*Retrieve External Updates*") and update the new records in the IS if required ("*Update External Updates*").

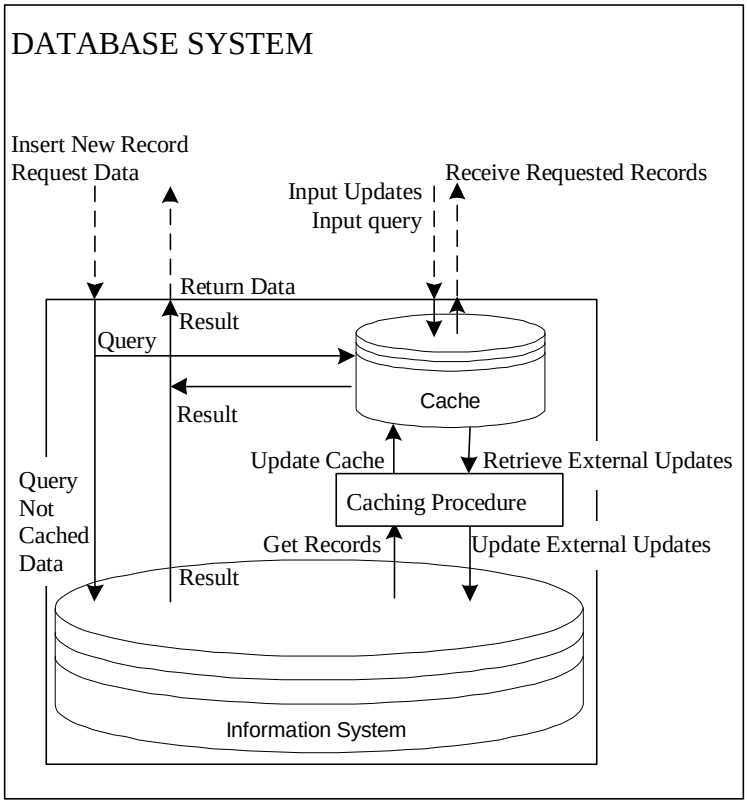


Figure 11. The Database system in the replication environment

3.2.1. Interface

The interface module is in charge of the internal communications between the elements of the system. It must receive the messages from the SCSP, pass them to the DB and return the results to the SCSP. This is organized into four mandatory tasks. First, it serves to SCSP the required data for initialization, i.e. SCSP in Cache Alignment. Second, it asks the user application to commit with updates received from remote replicas (if needed). Third, it updates remote changes received to the local cache. Fourth, it transmits local cache updates to the SCSP.

Data for cache alignment (First task)

In the CA phase, the SCSP requires an access to all the cached data. This task can be executed either at initialization of the local SCSP instance or re-initialization of any DCS into the replication group. Thus, the interface must be able to provide the contents of every cached record to the SCSP if required.

Remote updates (Second and third tasks)

As remote DCSs replicate their data, the LS receives new updates. The interface must be aware of the new records received from DCSs and update them in the local cache.

When a record arrives to the interface, a validation check by the user application might be needed. Once the record is confirmed, it can be updated in the local cache and DB.

An example of a previous implementation of an interface for SCSP can be seen in [25]. In this case, the interface works with ATMARP information cached in the kernel of the system. The same interface checks the validity of the remote updates received. This model is different than the one proposed in Figure 7. The aim of the model designed for our implementation is to leave those decisions to the protocol using SCSP. Thus, the interface proposed here is more independent of the data contents.

Local updates (Fourth task)

The user application managing the information system enters new updates into the DBMS. The interface must read the local changes in the cache and serve them to the SCSP.

3.2.2. Database

A DBMS is the facility used to implement the software caching for each replication server. SCSP replicates cached information independently of the information itself. The replication items can be selected from two basic models. The first model is based on change records; i.e. the item is a description of a change executed in the DB. These descriptions are called "*deltas*". The second model uses the record itself as replication item. Thus, whenever a cache record is modified in the DB, the content of the new record is replicated.

Replication of deltas is implemented forwarding the query which executed the update in the DB (cache) to the other replication entities. The drawback of this approach is that all deltas must be executed in the same time sequence in every remote server. This requirement is against the principle of weak consistency replication. Thus, this method must be discarded for application within SCSP.

Moreover, cache records should be small amounts of information. Their main attribute is the likelihood of change; i.e. they may be highly dynamic data. They produce flows of small changes in the DB. Thus, the replication of the whole content of the record adds consistency to the system with minor additional consumption of resources.

MiniSQL

Systems for DB management provide with complex selection and storage schemes that are strong in extraction of data. They can be very useful for knowledge extraction systems or classification procedures. However, they cause more consumption of resources than the necessary for caching tasks.

MiniSQL [24] engine is specially designed to work with small amounts of data which need to be rapidly accessed. The approach it takes consists of a sophisticated use of indices and trees. Once the indexing fields have been established, the system can now create lookup keys in real time. These features provide a performance more dependent on the quantity of data returned than in the total amount stored in the DB. It makes this DBMS to be well suitable for the storage and caching tasks.

3.3. Implementation of the SCSP

3.3.1. Introduction

This part of the thesis is the result of the analysis and upgrade of a previous SCSP implementation in this laboratory [23]. The main topics to analyze focus on four attributes of quality for any software product: execution performance, portability, scalability and transparency. The recommendations in the development of the new code are taken from [32] for the main aspects (i.e. server performance and scalability). The threads system is the major issue.

To improve **performance** the original code was migrated from C++ to C. To obtain a scalable implementation I followed the guidelines at [32].

C language contributes to increased software **portability**. Although communication and threading facilities are of relevance to achieve performance, they are not enough. I tested and studied several libraries in order to find out the most feasible solution to the portability problems. I concluded that the code must rest on standard libraries complying with POSIX specifications. It provides standardized function calls, simplifies coding and increases readability. Thus, I adapted the implementation to commit with POSIX standards.

In this thesis project, I have improved the **scalability** of the program in several ways. First, we obtained an optimised code by eliminating memory leeks. Second, I used a revision of the previous code provided with dynamic buffers and lists. The new changes were adapted to these dynamic structures. Due to this, memory is reserved when needed and freed after use. Program execution causes minimal memory consumption. Third, I implemented a threaded code. It reduces consumption of kernel resources and allows parallel processing of information. The level of threading must be taken to an acceptable degree in order to avoid overuse of threads. The main challenge at this point is to control the concurrency and synchronisation of threads.

To obtain a **transparent** program I use the message queues facilities for the input and output tasks. It allows any process to connect to the SCSP program by a simple procedure. The SCSP runs independently from any program using its functionality. Adjoined to this, no program in the system needs to look to the SCSP implementation to replicate its information.

The description of the implementation is structured in two sections. First, the dataflows section describes the use of information inside and outside SCSP in the replication process. Second, the algorithms section describes the methodologies and algorithms of the implementation.

3.3.2. Data Flows

This section explains briefly the distribution of information in SCSP. The aim is to describe the information exchange between the different tasks of the protocol. The notation used follows the guidelines given for Data Flow Diagrams (DFD) in [27].

External flows

This part describes the external flow of messages between the SCSP and the interfacing process (See Figure 12). It is divided into two kinds of flows: input and output.

Input Streams

The interface uses two streams to input data to the SCSP.

❒ *Get Initial Data*[1] is used in the "*Cache Alignment*" stage of the protocol. It is in charge of collecting the summaries of the cache data that are used in the initial (alignment) steps.

❒ *Get New Entries*[2] stores every entry that arrived after the protocol started the *Alignment State*. The records that arrived in *New Entries* are considered as up to date data. Thus, they are not checked against other copies for newness. SCSP sends them in the next flood session to every DCS. The procedure followed for flooding is described in the *Cache State Update Protocol*.

In the implementation, SCSP receives all the inputs in a single point. A priority is attached to each entry depending on its type. If the entry belongs to the initialisation group it is identified with higher priority.

Output Streams

The program uses two streams to return the outputs to the interface (See Figure 12).

❒ *Update*[3] returns the records that must be updated in the local cache. These records are the result of the replication and synchronisation process.

❒ *Erase*[4] returns the records that must be deleted in the local database. The *Erase* flow is caused by the reception of a CSAS indicating that a record is out of date (i.e. it is old, or it has been deleted in any DCS).

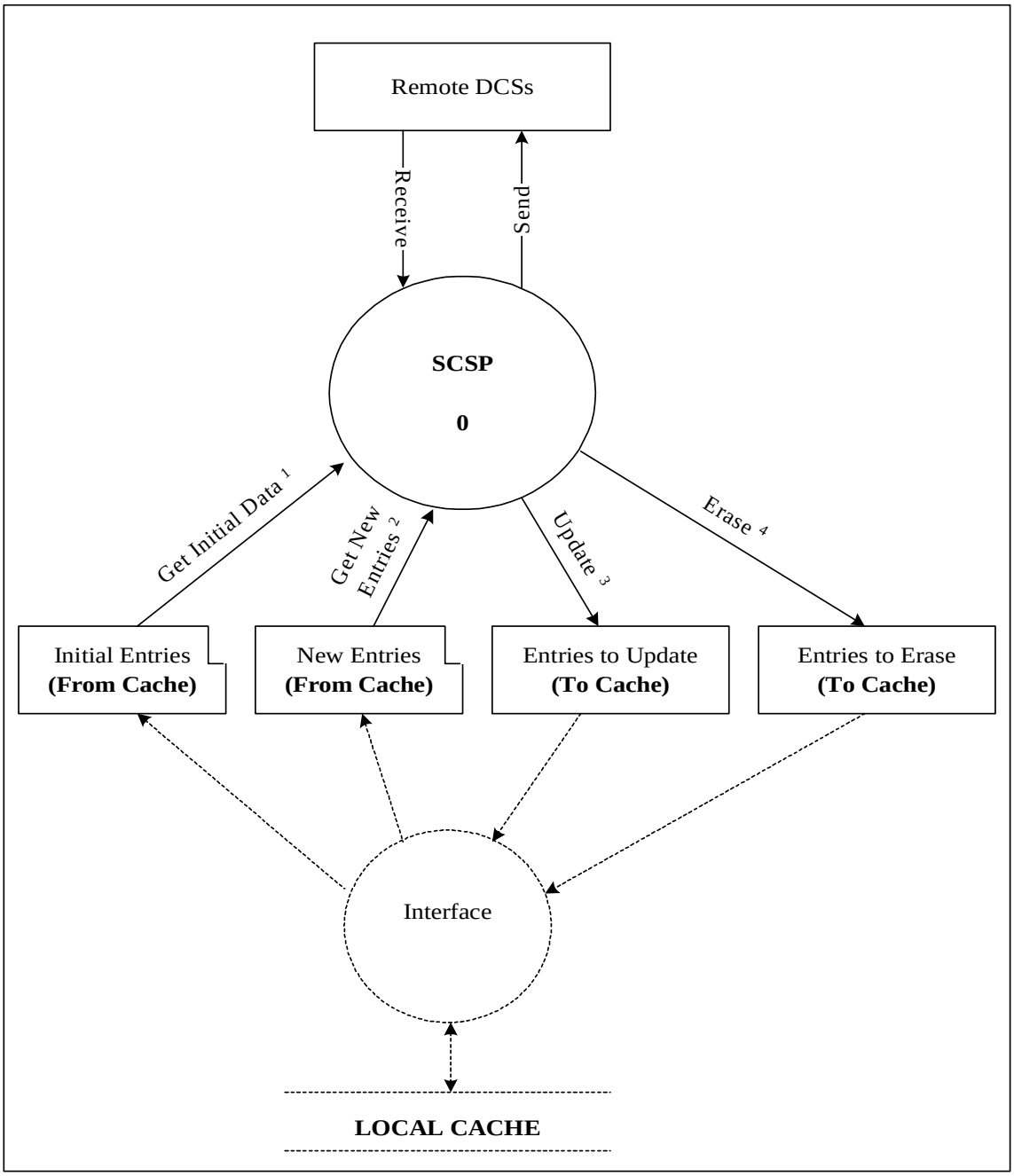


Figure 12. Level zero DFD for SCSP

The input and output files (*Initial Entries*, *New Entries*, *Entries to Update*, *Entries to Erase*) hold the SCSP messages for communicating with external processes, i.e. the interface. The implementation of these connection points is done by message queues. Section 3.3.4 deals with the use of message queues.

Internal flows

Inside SCSP, the tasks are divided into three: *Input*, *Process* and *Output*. Each of them is processed in a different thread to reduce delays (eliminate I/O waiting periods) and improve performance.

This part describes how the protocol manages the information received. Different buffers collaborate with the exchange of CSA and CSAS between the three threads running in SCSP. These buffers are also explained here.

This part is divided into two subsections. The first relates the internal processing of inputs. The second explains the processing of outputs.

In Figure 13 and Figure 14, the process *SCSP Finite State Machine* (FSM) represents a generic FSM. The protocol can run either *Hello*, *Cache Alignment* or *Cache Update* protocols at one point. Thus, the SCSP FSM represents the algorithm regarding the state of the LS at that moment.

Input streams

The *Read Input Process* retrieves the inputs and classifies them depending on their priority. There are four buffers where it deposits the different entries. Three of them are registers of CSASs; i.e. they hold control information. The last one stores the CSAs of the updates waiting to be sent or acknowledged.

❒ *Cache State Alignment Summary List*[3] (CSASL) holds the summaries of the initialization records (i.e. *Cache Alignment*). The CSASL is formed by the entries labeled with higher priority and is used in the alignment steps.

❒ *NewInLocal*[4] is in charge of storing the record (CSAS) of the entries that must be flooded to the DCSs after alignment (i.e. *Cache Update*). CSAs with lower priority are placed here.

❒ *List_Buffer*[1] acts as a second level cache. It holds the CSAS of all the CSAs stored in the cache. The protocol must store all updates so that it can track missed updates and check for consistency of its replicas [33]. Keeping summaries helps to save memory instead of using the complete CSA records. Summaries hold all the information about the CSAs. Thus, if there is a

need to check the newness or existence of any entry, the summary information is rapidly obtained from *List_Buffer*. Otherwise, the protocol would be forced to extract the information querying to the cache, with the corresponding delay.

❒*NewCSA* is a consistency buffer. It keeps all CSAs that have not been replicated in every DCS. Whenever a CSA has been flooded and acknowledged by all DCSs, it is removed from this buffer. Note that an empty *NewCSA* means that the replica has achieved a *full consistency* state.

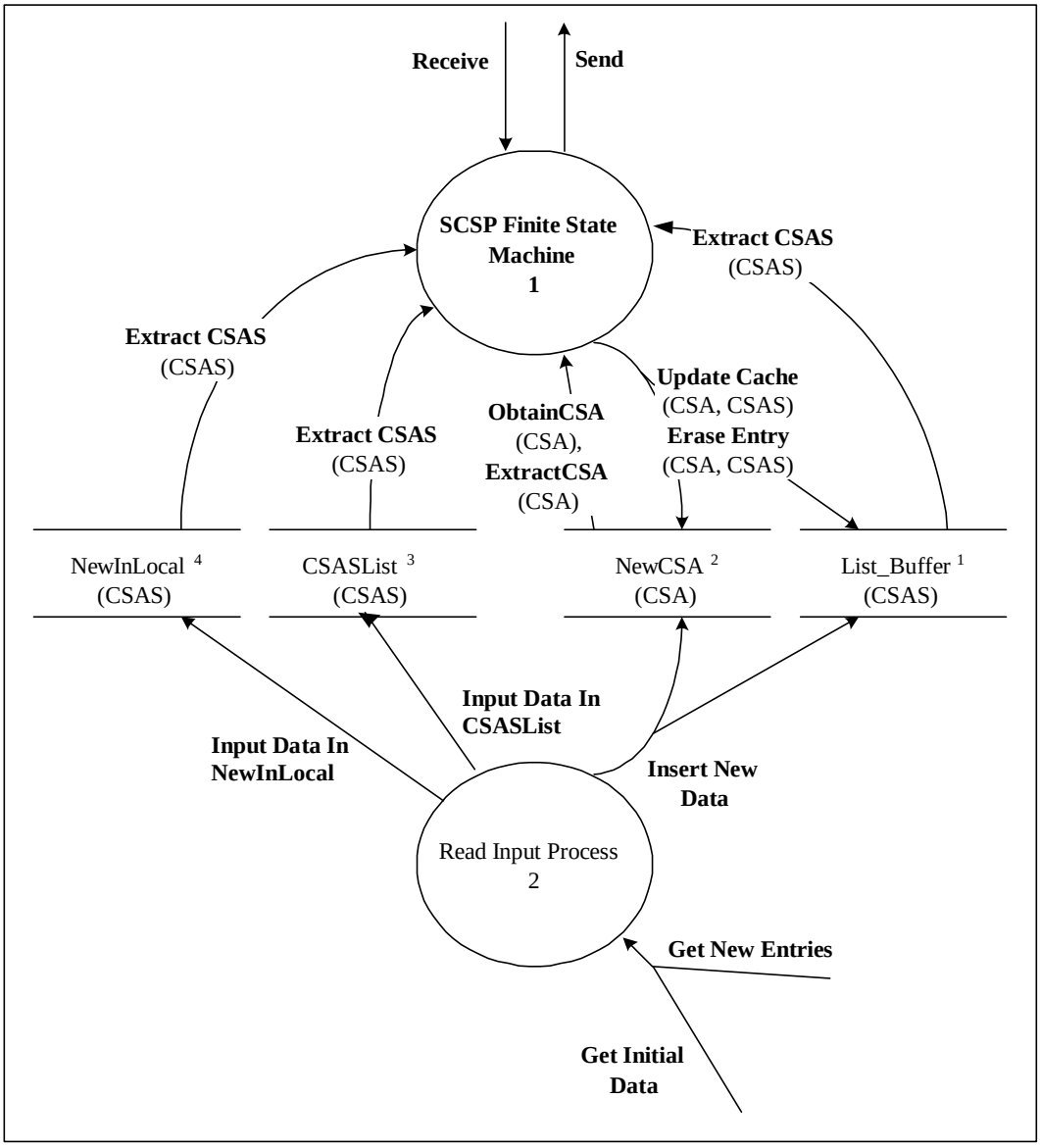


Figure 13. Level one DFD for SCSP: Input flow

The entries coming from the local input or from a DCS are stored in the same buffer, *NewCSA*[2]. However, they are tracked in different manners. If the new entry is a local server's input it will be flooded to every DCS. Hence, its CSAS has to be registered in *NewInLocal*. Otherwise (if the entry is an update from a remote server), it is forwarded to every cache in the group but the one that originated the update. The forwarding procedure is done automatically. Thus, the CSAS does not need to be registered in the *NewInLocal* buffer. Note, however, that *List_Buffer* registers every entry received.

The SCSP checks *CSASL* or *NewInLocal* when it has to execute either alignment or flooding sessions. When a CSA is needed, the protocol extracts it from *NewCSA*[2].

Note that the flows *Update Cache* and *Erase Cache* in Figure 13 are also involved in output tasks (See Figure 14). These flows are in charge of passing the remote updates from DCSs to the SCSP and the local cache. The updating towards the local cache is described as output flow in Figure 14. The updating inside the SCSP is described as input flow in Figure 13. The internal updating consists of the synchronization of the buffers *NewCSA* and *List_Buffer* with the same records that are sent as output to the local cache.

Output streams

The flow of output runs in two different threads, one for the updates and another for the erases (See Figure 14). The output is controlled by the functions *Update Cache* and *Erase Entry*. These functions output the update records received from the DCSs to the local cache.

The output buffers are called *NewCSAtoUP*[1] and *OldCSA*[2]. The former stores the copies of the updated entries that must be inserted in the local cache. The latter is a list of CSAs which have to be removed from the local cache.

The procedures *Out Updates* and *Out Erases* are in charge of unloading the regarding buffers. They infinitely loop collecting updates received from DCSs and send them as outputs towards the DB, i.e. to the interface.

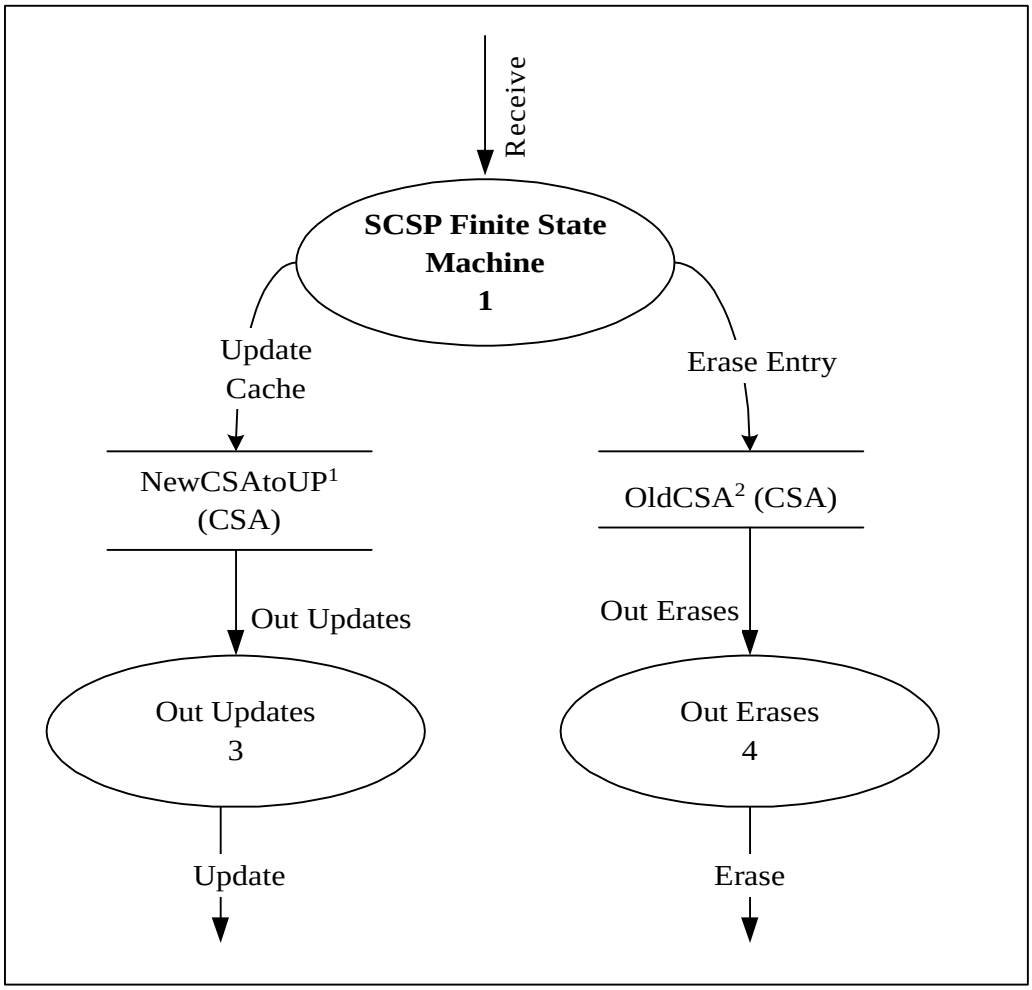


Figure 14. Level one DFD for SCSP: Output flow

3.3.3. Threading

Introduction

The aim of this work is to improve the concurrency features of our previous implementation of the SCSP. The initial implementation [23] worked as a batch process. Batch processing is not efficient and should be avoided when possible.

In the former implementation, SCSP worked in a batch sequence. First, a data set is input in the SCSP. Second, SCSP reads a part of these inputs and builds a packet to be sent to the DCSs. Third, SCSP sends the messages and returns to the first step. Figure 15 depicts the process.

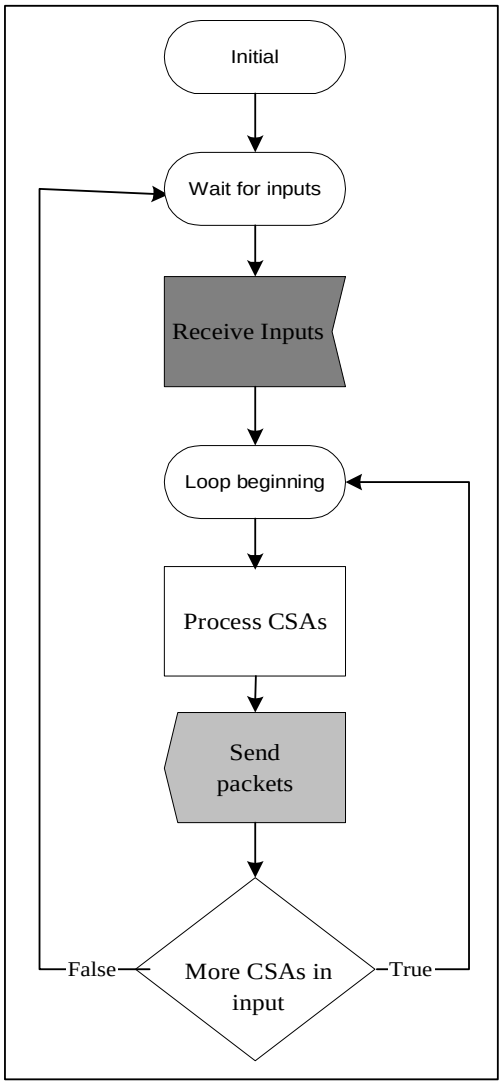


Figure 15. Sketch of batch processing in SCSP

The problems of this implementation are diverse. The three major drawbacks are pointed out here:

- Client processes are not served until the input buffers are emptied. Thus, they have to wait before setting their inputs into SCSP to be replicated.
- Communications are stopped if SCSP is serving the clients (receiving inputs).
- Delays in any point of the execution sequence are transmitted to all the parts in the process.

In the new implementation, the use of threads and message queues avoids any problem due to input and output tasks.

As seen before, data flows of SCSP run in two different scopes: internal and external flows. To implement those flows in SCSP, two different approaches are used.

The internal flows are implemented using threads. Advantages of threads are multiple: parallelism, execution, increased throughput, efficient use of system resources, alleviation of bottlenecks... Analyzing the advantages of threads in a system is out of the scope of this thesis. For further detail, refer to [34]. The most relevant issue in SCSP is the I/O throughput congestion [32]. In this implementation, one thread runs to alleviate congestion on each critical access point, i.e. local input, local output, and communication with DCSs.

The external flows carry the communications between the SCSP instance and the external interface and the cache. Therefore, interprocess cooperation is needed. UNIX systems provide several mechanisms for accomplishing interprocess communication tasks [28]. When several processes want to communicate via IPC, they are using kernel system calls. The computational cost of this is high [35]. Thus, the use of IPC facilities must be minimized. The selected tool is the *Messages Queue* (MQ) library.

The rest of this section is divided in six parts. The first part briefly explains the required policies for inter-thread communication. The second part explains the algorithms used in the threads (I.e. *Fill_CSAList*, *OutErase* and *OutUpdate*). The third part compiles the functioning of all pieces together. The forth part describes the communications with other processes in the system through message queues. The fifth part relates the timestamping approach implementented. The Sixth part concludes annotating the scalability features provided in the new implementation.

Inter-thread communication

Buffers are intermediate elements between threads. Parallel flows might perform operations simultaneously in shared buffers. The system must avoid global locks and serialization of threads [32]. Determining the shared memory regions and setting local locks on these regions is the solution to this problem. Executions in shared regions must be serialized to avoid errors in the operations.

The problem of access to shared objects is defined as the *Critical Region* (CR) *Problem*. In terms of concurrency, a CR is represented with a *monitor*. Figure 16 represents the design for the CRs in SCSP. The model selected for the monitor "*Buffer*" is the *producer-consumer model*, with the added feature of a *test*$^{(1)}$ procedure (This procedure accesses as consumer but does not modify it). For a description of the syntax used in this chart refer to [37].

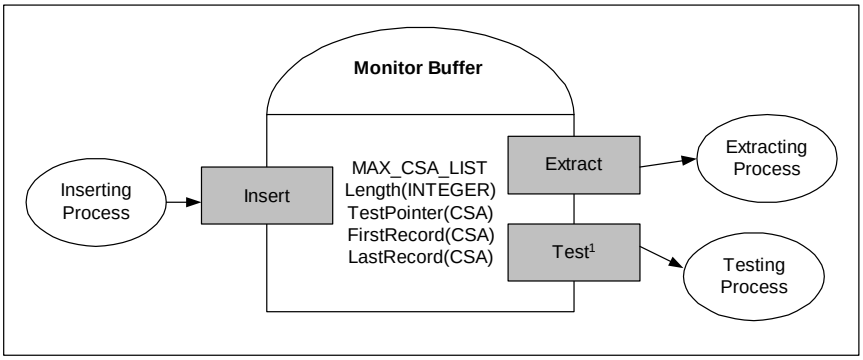


Figure 16. Monitor of access to shared buffers

Functions, constants and variables inside the monitor can not be used by external processes. *MAX_CSA_LIST* is a constant which sets a maximum limit to the size of the buffer. *Length* holds the number of items stored in the buffer. *TestPointer* points to the last item accessed by *Test*[1]. *FirstRecord* points to the first item stored in the buffer. It is used by the *Extract* and *Test* functions. *LastRecord* points to the last item in the buffer. It is used by the *Insert* function. The buffer is implemented as a FIFO linked list.

Algorithms

For paralleling the execution and to divide tasks among threads, different software models are established [38]. The model used in this program is a version of the *pipeline* model. It is intended to work as an IO pipeline. Four threads are in charge of the three IO tasks. One for data *Fill_CSAList*, two for data output, and another managing a list of output messages to send to the DCSs.

Input thread: "*Fill_CSAList*"

This thread is in charge of retrieving the inputs from the MQ system and posting them into the adequate buffers. The thread stops either trying to store new entries when the buffer is full, or waits for new inputs to come in. The SDL in Figure 17 depicts the algorithm for this thread.

The thread running the *SCSP FSM* does not have to deal with the local inputs. It is only in charge of the replication and must receive the CSAs in the corresponding buffers (See Figure

13). The Fill_CSAList thread frees the FSMs of delays caused retrieving inputs. While external processes are providing information fast enough, input can be considered a continuous stream*.

Note that, in Figure 17, the input "*Receive (message)*" refers to the composition of the "*Get new Entries*" and "*Get Initial Data*" flows from Figure 13.

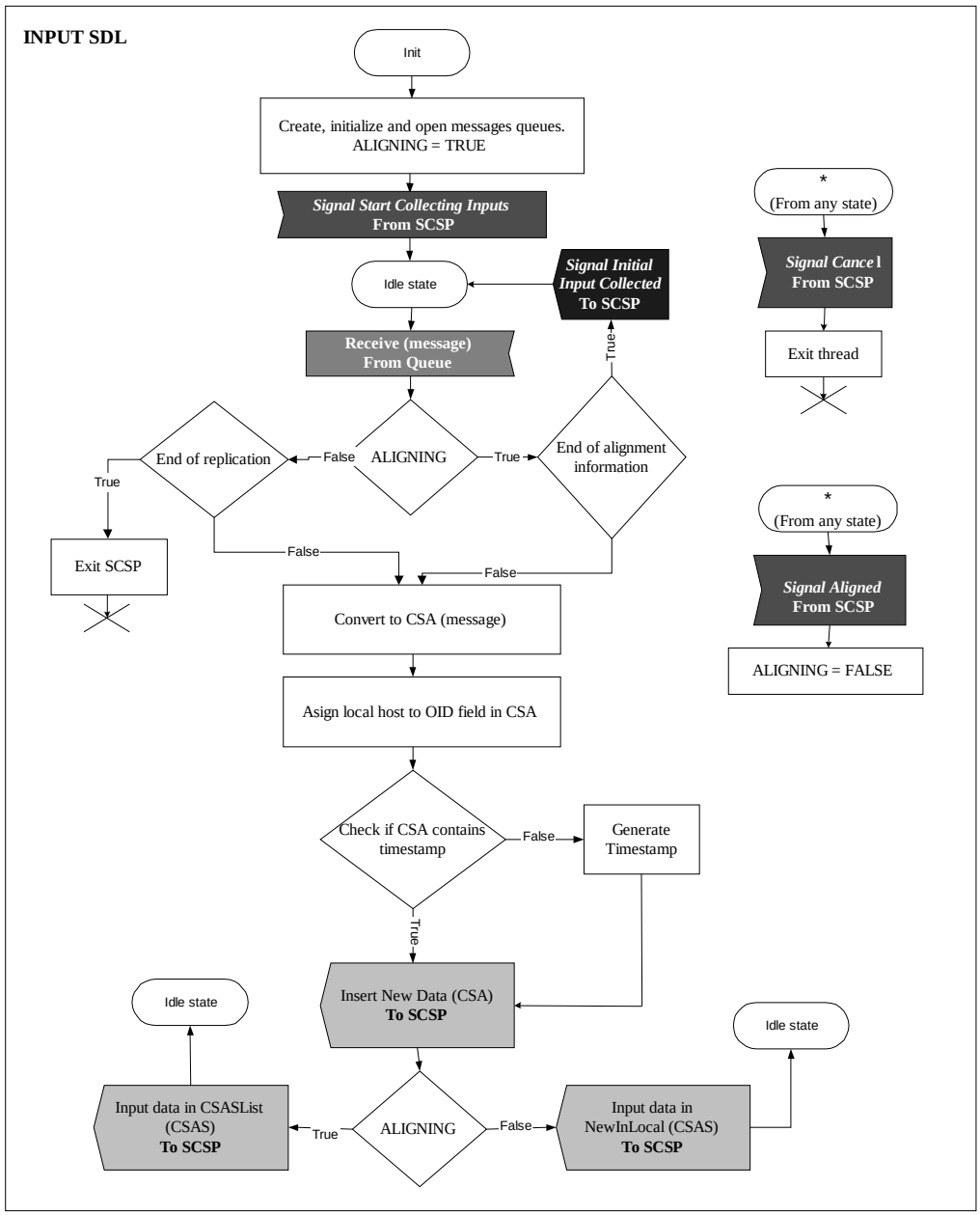


Figure 17. SDL for Input thread

* Note that the flow of inputs will still depend on the inter-thread switching policy running if the process is running in monoprocessor machines (not parallel).

Output threads: *"OutUpdate"* and *"OutErase"*

As pointed out before, there are two more threads in charge of the output collection. Therefore, the *SCSP FSM thread* is released of the output tasks. Indeed, the benefits are the same as for data entry. Outputs are deposited in either *NewCSAtoUp* (for updates) or *OldCSA* (for erases) buffers. Each of these threads retrieves the information from each of the output buffers. This avoids the complete filling of these units (It would cause the *SCSP FSM thread* to block until there is some space to deposit its data). The retrieved entries are sent as output to the system of Messages Queues. The SDL in Figure 18 depicts the algorithm for this thread.

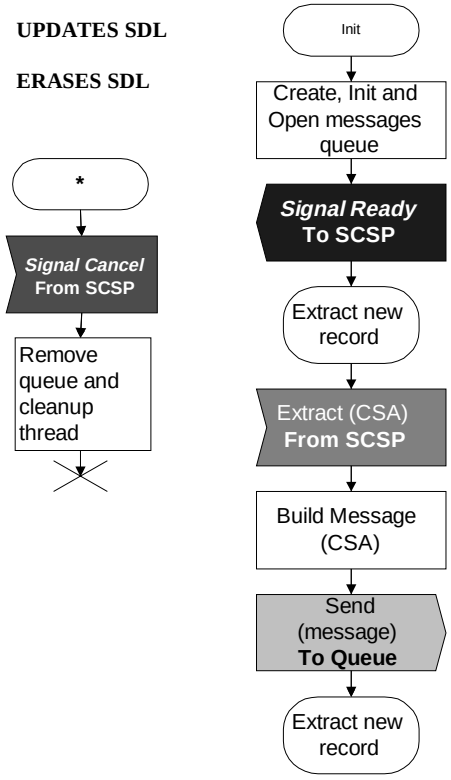


Figure 18. SDL for the update and erase streams

Thread for communications: *"ReceiveUpd"*

Another bottleneck identified is the management of communications. A new thread is created to perform these tasks. It processes the communications with DCSs. Communication packets (CA, CSUS, … [20]) are generated in the DCSs and received by the LS. The *SCSP FSM* thread can not afford to keep listening in the socket waiting for messages to arrive. To perform this task the *ReceiveUdp* thread is created. An intermediate buffer deals with the message exchange between the *SCSP FSM* and the *ReceiveUdp* threads. The *ReceiveUdp* loops listening in the

communications socket and writes the received messages in the buffer. The messages are extracted following a FIFO order and processed by the *SCSP FSM* thread. If there is no messages in the buffer, *SCSP FSM* thread follows its normal execution. If there are no messages in the socket, *ReceiveUdp* waits for more messages.

Functioning inside SCSP

To begin replication, the server executes an instance of SCSP. SCSP must run as system daemon [29]. The interface links with the associated MQ and starts deploying information packets into queues. The execution sequence of the threads in SCSP is depicted in Figure 19*.

The *ReceiveUdp* thread is omitted in the chart. It is created at the beginning of the process and never stopped. No signals are exchanged and it runs completely independent from the rest of the system.

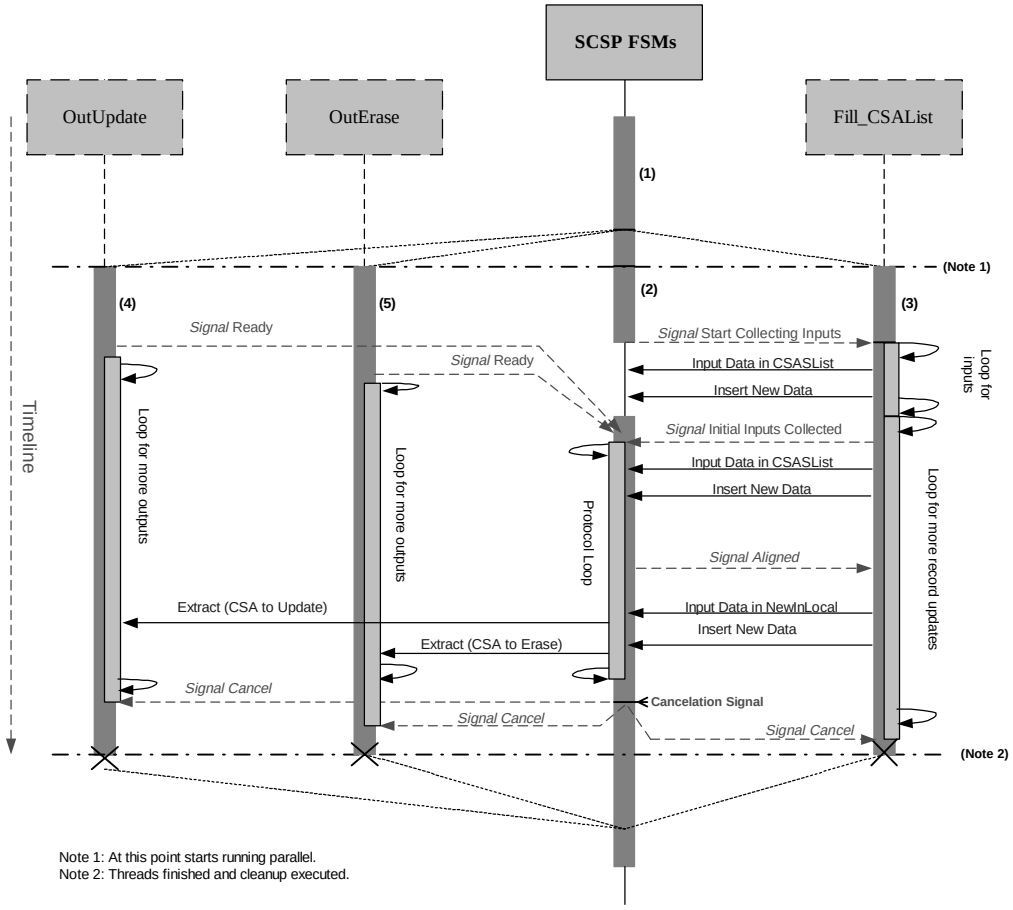


Figure 19. Inter-thread signals and messages

* Note: Notation used follows the recommendations at [39]

The process starts by setting the synchronization semaphores and the I/O queues (1). The main flow branches three threads and reads the configuration file (2) ("*hosts*").

The three new threads are in charge of the I/O tasks and configure their respective output (4, 5) and input (3) buffers (See Message Queues below). When SCSP receives the "*Initial Inputs Collected*" signal, the SCSP processing begins. At this point, all the tasks are executed simultaneously*.

For further reference, **Figure 17** and **Figure 18** describe the internal functioning of each thread. The following data dictionary defines the functionality of the events in Figure 19.

Event	Description
Extract (CSA to erase)	Reads from the *OldCSA* buffer which holds the records to erase and sends a record to the *OutUpdate* thread. Blocks if the buffer is empty (See Figure 14).
Extract (CSA to update)	Reads from the *NewCSAtoUP* buffer which holds the records to update and sends a record to the *OutErase* thread. Blocks if the buffer is empty (See Figure 14).
Input Data in CSASList	Provides CSASs to the SCSP during the Cache Alignment phase. CSASList holds the summary information of the CSAs in the local cache which must be synchronized with the DCSs.
Input Data in NewInLocal	Provides CSASs to the SCSP during Cache State Update phase. NewInLocal holds the summary information of the CSAs which must by sent in the next flood session.
Signal aligned	Signals the Fill_CSAList thread that the Cache Alignment phase has ended. After this, the thread uses "Input Data in NewInLocal" instead of "Input Data in CSASList".
Signal cancel	Termination signal. The thread receiving this signal must clean up the resources that it is using and exit.
Signal initial inputs collected	Signals the SCSP that the summaries of the records in the local cache have been read and entered in the buffers. The replication process can begin now.
Signal ready	Signals to the SCSP FSM that the thread has configured its buffers. The SCSP can begin execution.

* When the input thread receives the Aligned signal, the Cache Alignment procedure has finished. This signal is needed because now the *Fill_CSAList* thread deposits the new CSAs into a different buffer (See Figure 13)

Signal start collecting inputs	Signals the Fill_CSAList thread that the protocol has been correctly configured and replication needs the records in the local cache to begin the replication.
Insert New Data	Provides local updates to the SCSP during the Cache Alignment and Cache State Update phases (See Figure 13). It is called by the *Fill_CSAList* thread. Blocks if there is no new updates from the input.

Table 1. Data dictionary (DD) for Figure 19

3.3.4. The message queues system

The environment of SCSP was depicted in section 3.2. SCSP must execute as system daemon to be independent from any process or information being replicated [29]. The interface process must connect with SCSP to provide local updates and retrieve remote records. This connection must be asynchronous, i.e. both processes work independently from each other.

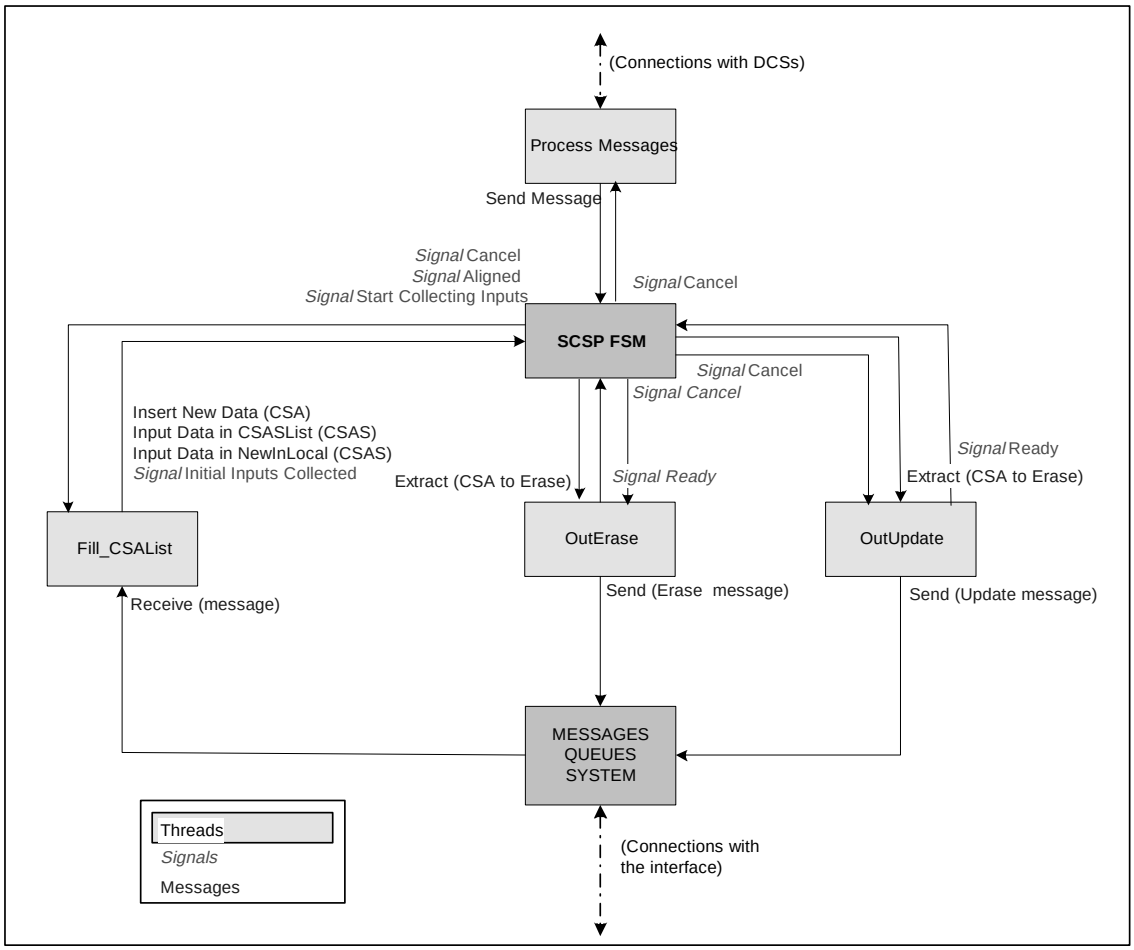


Figure 20. Objects model (OM) for elements in the SCSP system

MQs allow a process to store messages asynchronously. Messages are kept there until another process extracts them. MQs consist of a set of FIFO queues. The queues can be ordered according to priority labels [28]. This allows selection of messages to the process which receives them.

The interface replicates the data by linking with the MQs associated to an SCSP instance. Messages are entered into the MQ and SCSP retrieves them when needed.

To give an overview of the SCSP system, Figure 20 depicts the message exchange between the MQs system, IO threads, and the main thread (thread running SCSP).

The queues system is composed of two queues. The first one holds the data inputs from the LS. This queue is accessed by the *Fill_CSAList* thread. The second one holds the data outputs received in SCSP from remote DCSs. The OutErase and OutUpdate threads access to this queue. The output records in the MQ are retrieved by the interface, which is in charge of communicating with the cache.

The data dictionary in Table 1 describes most of the signals in Figure 20. Table 2 describes the events in this figure which are not in Figure 19.

Event	Description
Receive (message)	Retrieves elements from the queue according to the SCSP phase running at the moment.
Send (Erase message)	Sends a record to erase to the MQs System.
Send (Update message)	Sends a record to update to the MQs System.

Table 2. DD for the OM in Figure 20

3.3.5. Timestamping

The timestamping method in replication with SCSP provides a consistent algorithm to identify the newness of an update. Any cache record might be replicated by any of the replication servers in a SG. Two points are considered here. First, SCSP must maintain consistency of the updated records inside a SG. Second, once this consistency is achieved in the scope of the SG, some nodes can replicate their caches within several groups, i.e. several replication scopes.

Adjoined to this, servers can leave and join the SGs in different periods during a replication session. Thus, the SCSP requires a timestamping system able to keep a global timestamping sequence for every server which might be involved in the replication.

In our implementation, the field "CSA sequence number" [20] of every CSAS is filled with the value returned by a call to the function "time(0)" (See UNIX man pages). It returns the local time at the operating system in seconds.

The framework of the SCSP replication is a distributed system. Clocks in each machine are likely not synchronized. To solve this, servers must run a synchronization daemon which keeps local clocks synchronized. The "*Network Time Protocol*" (NTP) is recommended for this task [41].

Whenever a server joins a replication group, it must ensure that its local clock is synchronized with the other clocks in the system. Thus, if a server belongs to two SGs, the protocol must ensure that every server replicating the same cache shares a common clock sequence.

Another solution for this problems (not implemented here) would be the use of "*Lamport's algorithm*" [42]. This algorithm synchronizes a distributed system ensuring that events maintain the order they were executed. Clocks are not synchronized. However, the system can control the sequence of changes in the group. Although the study of this solution could be interesting for the SCSP timestamping problem, it is out of the scope of this thesis.

3.3.6. Conclusions

The implementation here tries to follow the recommendations of implementation for scalable programs for servers given in [32]. The following points are committed:

First, to provide caching of frequently used data, the buffer "*List_Buffer*" was used as a second lever cache internal to the process. Thus, as explained before, it stores the CSAS of every CSA recorded in the local cache. The system does not need to query the DB to check the information in the summary records. Those check are executed very often and would consume resources.

Second, to avoid the creation of many threads, the system has been implemented by task-associated threads; i.e. each thread performs one task (Input, output, communications and

processing). This solution commits with the recommendations listed in the article for this subject.

Third, to avoid the use of global locks, the critical regions have been allocated in the shared buffers. The locks can not be situated more locally than in these points.

Fourth, to avoid the delays due to blocking calls the I/O operations are executed in independent threads. This solution is the same used in the second point above.

Fifth, to express the functioning of the protocol in numbers, several files and statistics are produced during and after execution. These data are commented in the next chapter of this thesis: Scalability tests and analysis.

4. Scalability tests and analysis.

This chapter deals with the evaluation of the SCSP program. Tests are planned and documented to obtain a record of the analysis. The aim of the tests is to analyze how the protocol responds to growth of the replication load and consequently give a baseline of SCSP scalability evaluation in comparison to other similar protocols. Adjoined to this, the tests are used to find out the weakness of the current implementation.

4.1. Test goals

The objective of this study is the measurement of the protocol scalability in five cases of simulation. The scope of the tests is the SG. They are grouped in two categories, one for different SG load models and another for different SG topologies.

4.1.1. Server group load

The analysis of SCSP under different loads of data is the basic scalability analysis. Note that, for these analyses, all the trials use a fix topology consisting on two servers. The only parameter modified is the data set to replicate. The evaluation is done in respect to three parameters:

❒ The *DB size* is the main factor for the alignment time. Its evaluation shows the time and messages required initializing different sizes of caches. The aim of these tests is to measure the proportional increment of resources consumption associated to the increment in the size of the DB.

❒ The *Mean Update Time* (MUT) is the rate by which updates arrive to an LS. Lower MUTs generate higher update rates, i.e. higher number of CSAs to replicate per unit of time. The Update Rate is studied according to an exponential model, i.e. the number updates arriving to the LS increase exponentially in time. The aim of these tests is to show the response of the protocol to several update flows and find out the upper limit of the protocol in terms of update rate.

❒ The relation "*CSA size / maximum message size*" influences in the number of CSAs sent per message and the bandwidth consumption. This proportion can be tested for fixed MUTs and DB sizes to measure the influence that it has on the replication. Longer messages send more replication data in a packet, but are less resilient to errors. The influence of *maximum message size* in the replication depends on the communications bandwidth. The tests focus on the SCSP performance. Thus, they concentrate on the analysis for different sizes of CSA records, i.e. different sizes of information records.

4.1.2. Server group topology

SCSP works on a group basis. Different numbers of elements in the group, with disparate topologies, raise different load flows. These tests show the behavior of SCSP for two generic configurations: *star and inline*. Indeed, other configurations are not but composition of these two basic models.

Note that an important issue for this study is the consistency analysis. For this evaluation, the test model is taken from [33]. The aim of these tests is to analyze the convergence of the SG to a consistent state, i.e. to which extent the servers in the group are equal during a replication session.

❒ *Star* configurations show the distribution of load for caches serving as link points to more than two servers, i.e. they generate convergent replication flows that must be processed in the server and delivered to other DCSs.

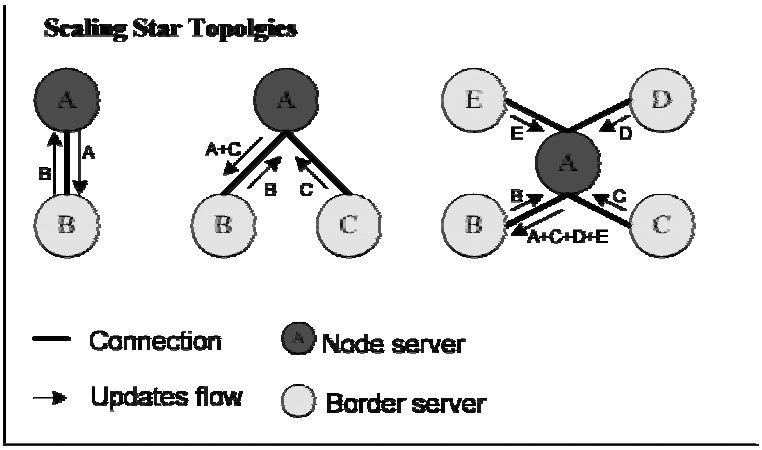


Figure 21. Progressive scaling for topological analysis

These tests analyze the behavior of the protocol under loads coming from increasing number of DCSs. The basic topology consists of two connected servers. In the analyses, the configuration is scaled by adding DCSs always to the same server. The aim of the evaluation is to determine the load in the server acting as the nucleus for other nodes. Adjoined to this, it raises data about the synchronization time required for different configurations.

❒ *Inline* configurations show the processing of cumulative replication load. Updates received in the LS from any DCS must be flooded to every DCS in the SG but the one that originated the update. *Inline* topologies generate longitudinal replication flows that hop several times before terminating.

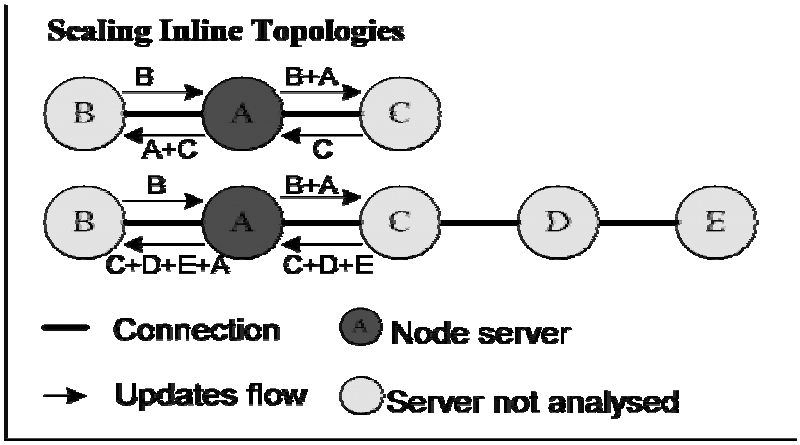


Figure 22. Progressive Scaling of Inline Topologies

The evaluation is always executed on an inner node and scaled to different number of group lengths. The aim is to analyze the required time to replicate data as a function of the number of intermediate servers that replicas traverse.

4.2. Test elements

This section explains the different types of data extracted from the tests and the kind of information that they represent. The information returned by the SCSP is sorted into seven files. One of them records the final statistics of the execution. The other six files store dynamic information.

❒ A statistics log is written into the "*stat*" file. This is the statistic record of the test. The information returned corresponds to the stated in [40]. For each DCS involved in the replication, the following data is collected:

	Hello	Cache Alignment	Cache State Update
Hello Messages(1)	♦		
Cache Alignment Messages(2)		♦	
Cache State Update Solicit Messages(3)		♦	
Cache State Update Request Messages(4)		♦	♦*
Cache State Update Reply Messages(5)		♦	♦*

Table 3. Data collected on each phase of the SCSP

The *Hello Messages*(1) are used to check connectivity of the DCSs with the LS.

The *Cache Alignment* (CA) *Messages* (2) allow synchronizing the cache of an LS with its DCSs. They are used in the Cache Alignment Protocol [20] to send the description (CSASs) of the local cache records.

The *Cache State Update Solicit* (CSUS) *Messages* (3) are used during the Cache Alignment Protocol to solicit the entirety of CSA records to a DCS. They are composed of the CSASs received in CA messages which do not exist or which are older in the LS.

The *Cache State Update* (CSU) *Request Messages* (4) are used to update cache entries in DCSs. They contain one or more CSAs with new updates or requested CSAs (by CSUS messages).

The *Cache State Update* (CSU) *Reply Messages* (5) are sent by the DCSs acknowledging CSU Request Messages.

In addition, the file records some extra data:

- The total amount of CSAs exchanged during the test.
- For each replication step (I.e. Cache Alignment and Cache State Update), the number of input records received from the local server.

* CSU and CSUS messages can be exchaged during the Alignment or Update phases. Thus, to easily identify the messages for each phase, tests must be executed separately.

- The time spent for each DCS on each replication step (CA and CSU).

❒ The file "*consistency*" periodically reports about the consistency state of the LS [33]. The information is delivered each second and classified into four columns as indicated in Table 4.

Time	Size of the consistency buffer	Updates received since the last check	Total updates received

Table 4. Information of consistency

The number of entries is the reference to the consistency state. As explained before, the *NewCSA* buffer holds the CSA entries which currently need to be replicated. A low quantity of elements in it means that the protocol is receiving fast enough the acknowledgments to the updates. Thus, the consistency of the system is high (regarding the LS).

The evaluation of these results is interesting in the scope of different MUT rates and topologies.

❒ The files "*communications_in*" and "*communications_out*" trace the CSAs in each message at any time. The measurement is done in the incoming and outgoing messages respectively. The aim is to track the relation CSAs per message in the test. Histograms and time charts of these data are interesting for the evaluation of the use of the bandwidth and the overhead of the protocol. This facilitates acquiring a global view of the communications within the server group from the analysis of only one of the servers.

Sometimes, the protocol suffers sending problems due to overload. This file also keeps counting the socket errors.

The files record a row for each message with the format:

Time of the message	Number of CSAs in the message	Error in message (0 or 1)

Table 5. Information of communications (Input and output)

❒ The file "*updates*" displays the log register of entries received in the LS from its DCSs. It records all the information about every update with the format:

Cache Key	Origin Identifier	Message length	Sending time	Arrival time	Transmission time	Number of servers traversed

Table 6. Information of remote updates arrived

The information in this file is used to calculate the delay in CSAs delivery which is caused by scaling the tests load.

❒ The file "*inputs*" is the log register of all updates locally entered in the SCSP. The CSAS data of each new record is sorted chronologically by time of arrival. An update is entered in the system when it is extracted from the message queue. Each row represents an update with the following format:

Arrival time	Cache Key	Origin Identifier	Record length

Table 7. Information of local updates arrived

The information here permits to register the time every entry was actually received by the SCSP.

❒ The file "*erases*" records every CSA received from a DCS which must be erased from the local cache. The format of this file is the same used in "*updates*".

4.3. Test environment

This part is the description of the network, servers and test programs used. It aims to serve as record of the scope of the tests.

4.3.1. The network

The schema in Figure 23 describes the network used in the tests and the position of each server used on it.

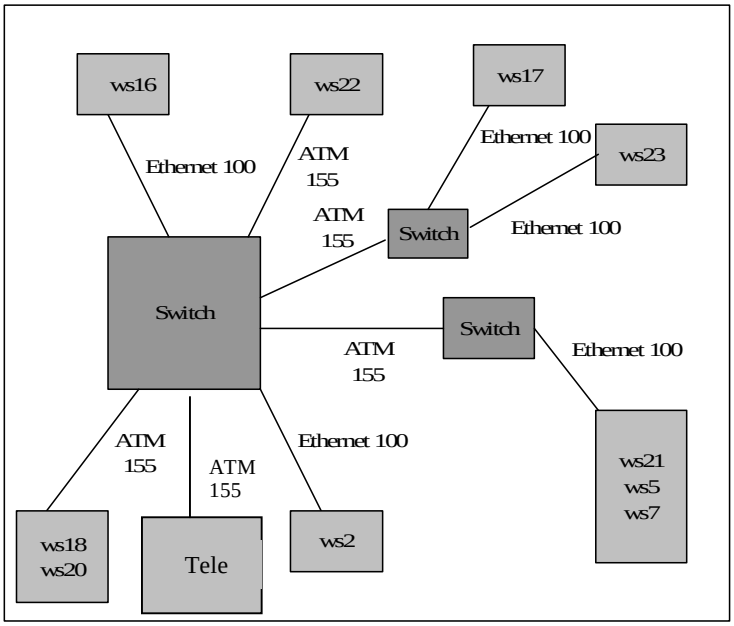


Figure 23. Schema of the connections between servers

The following workstations are used as node servers:

Node	System	Release	KernelID	Machine	CPU#
ws2	SunOS	5.7	Generic_106541-12	sun4u	1
ws5	SunOS	5.5.1	Generic_103640-31	sun4u	1
ws7	SunOS	5.5.1	Generic_103640-31	sun4u	1
ws16	SunOS	5.8	Generic_108528-06	sun4u	1
ws17	SunOS	5.6	Generic_105181-17	sun4u	1
ws18	SunOS	5.6	Generic_105181-23	sun4u	1
ws20	SunOS	5.8	Generic_108528-01	sun4u	1
ws21	SunOS	5.5.1	Generic_103640-34	sun4u	1
ws22	SunOS	5.8	Generic_108528-02	sun4u	1
ws23	SunOS	5.7	Generic_106541-08	sun4u	1
Tele	SunOS	5.6	Generic_105181-20	sun4u	4

Table 8. Description of the servers

This information is collected using the “uname –X” and “uname –a” functions.

4.3.2. The tools

Three tools are used to perform the tests. The “*scsp*” program, the “*test*” program, and the “*formatea*” program.

Program “scsp”

The execution of *SCSP* returns data about the process performance in different files. These files are explained in 4.2. For testing issues, the implementation is temporally modified to provide the results to the “*updates*” and “*erases*” files and not to the corresponding output message queues. This permits to analyze the resulting records more easily. The impact of the collection of data can be neglected in the scope of the performance of the protocol. They consist of print commands which generate the different record files. The files are opened at the beginning of the program and closed at the end to avoid calls to the file system during execution. The statistical files are generated in a different way. The values are kept in records and printed out at the end of the process in their corresponding formats. These values are integer values and counters that do not affect the performance of the protocol in measurable terms.

Program “test” (Emulating the interface)

The *test* program emulates a prototyped interface. The only function implemented is the input task. It links the message queues of the running *SCSP* being tested and provides the records to replicate.

Functioning

The program is executed with four parameters:

test “initial file” “updates file” “updates rate” “queue name”

❒ “initial file” is the file containing the cache records used in the *Cache Alignment* phase.
❒ “updates file” is the file containing the cache updates used in the *Cache State Update* phase.
❒ “updates rate” is the selected time gap (in microseconds) between two consecutive updates. This number is an unsigned integer between 2 and 999999.
❒ “queue name” is the name of the message queue associated with the SCSP instance which is used to replicate the records.

The formats for the “initial file” and “updates file” are the same. These formats are defined by the “formatea” process (See below in *Program “formatea”).*

The functioning steps are two:

First, the procedure opens the “initial file” and loops reading the inputs. During the loop, it builds messages and sends them to the inputs message queue specified in “queue name”. The priority label of these messages is two. The format of the messages is analogous to the format of a CSA (See Figure 4).

When the file is empty, it sends a “finalization message” to the MQ (also with priority two) and closes the “initial file”. The “finalization message” contains the string “END” in the cache Cache Key field and the total count of records sent in data length field. The data field is empty.

Second, the procedure opens the “updates file” and loops reading the updates. Messages are built in the same way and sent with priority one. The difference here is that messages are not sent in a continuous sequence. The time function (See Variants below) calculates the gap between two updates in each loop and sleeps the program during the resulting time before sending the next message to the SCSP. Different time functions implement different variations of the protocol.

When the file is empty, it sends a “finalization message” to the MQ (with priority one) and closes the “updates file”. The format of the “finalization message” for this phase is the same than the one used in the previous phase. The count of records is set to the amount of updates sent during this second phase.

Third, the procedure has finished the test session. It unlinks the message queues used and exits.

Variants

The code permits to implement different variants of the test program. These variants are based on the *time function* used. This function calculates the delay between two consecutive updates. Note that the *time function* selected does not affect the initialization steps of the SCSP. The idea is to provide different models of update rate to the *test* program; i.e. it can simulate statistical update rate models.

Three variants implement three update models:

❒ The "constant model" uses the initial "updates rate". The "updates rate" is entered in the program call to fix the update rate. Each record is delayed during "updates rate" microseconds before being sent*.

❒ The "exponential model" simulates exponential rates of update arrivals. Thus, it reduces the delay time between consecutive updates exponentially. Figure 24 shows the update function to simulate. The coordinate (*y*) axis represents the number of updates received.

The aim is to calculate the time gap between updates at any point in the function. The simulation is done supposing two consecutive updates (y coordinate) y_i and y_{i+1}. Their values for the exponential model are:

$$y_i = e^{x_i}$$
$$y_{i+1} = e^{x_{i+1}}$$

y_i and y_{i+1} are consecutive. Then:

$$y_{i+1} = y_i + 1 \Rightarrow e^{x_{i+1}} = e^{x_i} + 1$$

The resulting time sequence in which updates arrive is:

$$x_{i+1} = Ln\left(1 + e^{x_i}\right)$$

The values required are the sequence of consecutive gaps between updates in the update session. Therefore the final formula used in the *time function* is:

$delay = x_{i+1} - x_i$

In each loop, the x_{i+1} value is recalculated and subtracted from the value in the previous loop. The sequence begins with: $x_0 = 0$.

❒ The "quadratic model" simulates quadratic rates of update arrivals (See Figure 24). The functioning is the same than the exponential case. The sequence function for this case is:

$$x_{i+1} = \sqrt{1 + x_i^2}$$

* The time consumed for processing the records in the program is not included in the delay calculation.

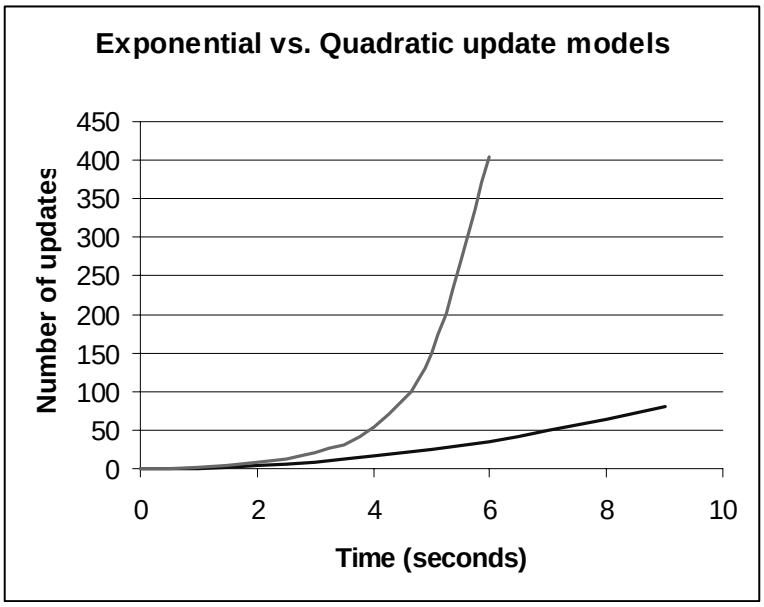


Figure 24. Standard Update Models

Outputs

The program accesses two files in the test environment. First, the "*stat*" file is modified to insert the update rate used in the replication session. Second, the tool generates an internal file. The name of this file is "*register_inputs*". It traces the time sequence of updates inserted in the message queues during the update phase.

The format of the "*register_inputs*" file consists of four fields:

Time	Cache Key	Gap	Counter

Table 9. Information of updates sent from the test tool

❒ "time" represents the time in which the message has been sent to the message queue.
❒ "cache key" is the Cache Key of the sent record.
❒ "gap" is the delay of this message with respect to the previous one. It permits to trace the behavior of the system regarding the time gap used for the message.
❒ "counter" is the number of the message in the message sequence.

Program "formatea"

This is an auxiliary tool to generate the files of records used in the *test* program. These files are composed of two columns of data. The first column corresponds to the Cache Key of each record. The second column corresponds to the data in the record. Each row in the file represents a record in the file (Cache Key and data).

This program helps to generate arbitrary cache records used by the *test* program with a selected size.

Functioning

The “formatea” tool reads from a raw ASCII file and generates a file in the format expected in “test” for the “inintial file” and “updates file”. The function call is:

formatea “input file” “output file” “slide length” “number of records”

❒ “input file” is the raw text file from which characters are taken.
❒ “output file” is the target file with the records.
❒ “slide length” is the size of the data field selected for every record in the output file. The Cache Key size is set to 16.
❒ “number of records” is the amount of records generated in the output file, i.e. the number of rows of the output file.

The program loops reading characters from “input file” in two steps. First, it reads sixteen characters. They conform a Cache Key. Second, it reads the next “slide size” characters. They are the data corresponding to this Cache Key. At this point, one record is built and written to the output file in the format: Cache Key, blank character, data, end-of-line. The counter of records is incremented and the read characters are discarded. When the end of the “input file” is reached or the specified “number of records” is generated, the program exits and closes all the files in use.

CPU use calculation

To obtain the percentage of CPU consumed by scsp during execution the UNIX “ps” tool is used. “ps” is executed in a shell script. The script loops executing “ps –opcpu | grep scsp” each second, which returns the CPU use of the program “scsp”. The output of the script is redirected to a file named “datafile” in the working directory.

4.4. Test and analysis

The following point relates the development of the tests. It presents and analyzes the most remarkable results. All the graphs in this section show the behavior of the protocol regarding different scaled parameters.

The data returned during the execution of SCSP is related in section 4.2. This information is received during execution time. It tracks the performance of the protocol. To obtain the charts below, the files containing those data are exported to Microsoft Excel format.

One generic test case is used as reference. The parameters for this case are shown in Table 10. In each test, one parameter is modified. The others maintain their value that was specified in the generic case.

Number of servers	2
Data Set Size in the LS (Either DB or Updates Set)	1,000
CSA data size (in characters)	100
Topology (Star or Inline)	Inline
Update Rate (microseconds)	10
Time Function (See section “Program “test”)	Constant

Table 10. Base configuration

Note that, if N is the Data Set Size for the test, the total update load in the SG is multiple of that value, i.e. for two servers, the size of the shared cache in the SG is *2*N*. E.g. the size of the generated DB after a replication session with 2Mbytes Data Set Size is 4Mbytes. Adjoined to this, the DBs used in every server are different, i.e. the replication always consists of the copy of all the records in every LS to every DCS.

The configuration of the servers is fixed in the “hosts” file. For a detailed description of it refer to [23]. Only the addresses of the DCSs are modified in the topology tests. Times of resending keep the same value for every test. Table 11 shows these values. This configuration aims to avoid the retransmission of lost messages. Thus, it allows easy tracking of leaks of messages.

Port	47123
Packet size	1000 bytes
Hosts	x.x.x.x (Depending on the LS and DCSs used)
Group Identifier	35 … (Same for every server)
Family Identifier	80 … (Same for every server)
Hello Interval	12000 … (Same for every server)
Dead Factor	40000 … (Same for every server)
CSA_Rtx_Depth	40000 … (Same for every server)
DCSA_Rtx_Interval	40000 … (Same for every server)
DCSCSA_MAX_Retx	90000 … (Same for every server)
DCSCSU_Rtx_Interval	40000 … (Same for every server)
DCSCSUS_Rtx_Interval	40000 … (Same for every server)

Table 11. SCSP configuration file ("hosts")

The working environment for the tests has on average a 90% of idle CPU use before the tests are executed. The consumption of CPU by the test program is never higher than 8% during CSU, i.e. for a Data Set Size of 20000 records. The CPU use of the test tool during the CA phase does not influence in the CPU use of the SCSP (In this case, the SCSP waits for the test tool to input the records before beginning the replication).

4.4.1. Scalability tests for different loads

Four kinds of tests characterize the behavior of the protocol under different loads of data. These tests must show the communications and processing responses when scaling the load of SCSP in a server. The input files and parameters described here are related to the "*test*" process and "scsp" process described in 4.3.2*.

Fixing clock gaps between servers

As explained in the section 3.3.5, the servers need a distributed time facility. It must provide consistency to the timestamps attached to the records. Therefore, during the execution of the tests, the servers must run the NTP daemons. Adjoined to this, the gaps between servers are calculated. In the cases in which some gap is found, the times obtained (see 4.2) are adjusted accordingly.

To obtain the gaps, a process is executed in pairs of servers. It runs a request of the local machine time in both servers at the same time. The difference of times results in the time gap between machines. To do this the process must perform a call to a remote server. The connection times can raise errors in the times returned. However, the response times in the net used here are fast enough to take advantage of it. Thus, the process is composed of two steps that must be executed in one of the servers involved in the calculation.

The first step executes a request of the local time in the remote machine. This call can take some delay, but the elapsed period between the extraction of the remote time and receiving it in the LS can be neglected[**]. The second step starts when the remote time is received. The process executes a request of the local time. Both times can be considered simultaneous in the framework in which these tests are running. However, the calculations are executed several times from both servers. The average delay (in seconds) is the value used to adjust the times.

Database size tests

These tests change the "Data Set Size" parameter. They are executed for 2.500 (≈270Kbytes), 5.000 (≈550Kbytes), 10.000 (≈1Mbyte), and 20.000 (≈2Mbytes) records. The records must be allocated in the "initial file". The "updates file" is empty. The "updates rate" does not influence in the CA phase and it has no consequences in the tests. The servers used are ws17 and ws18. Table 12 defines the parameters of the test. The results are analyzed from the data returned in ws18.

Number of servers	2
Data Set Size (Either DB or Updates Set)	**Variable**
CSA data size (in characters)	100
Topology (Star or Inline)	Inline
Update Rate (microseconds)	0 (not used)
Time Function (See section "Program "test")	Constant

Table 12. Tests for DB sizes

A special test is executed in this analysis to test the processing on inputs in a multiprocessor machine. The name of this machine is "Tele" (See Table 8). In this case, the configuration of the

[*] By default, the name of the messages queue in the scsp implementation used here is "tablabla"

test is described in Table 13. The machines used are Tele and ws18. Data is collected from both servers to obtain a comparative chart.

Number of servers	2
Data Set Size (Either DB or Updates Set)	**20000**
CSA data size (in characters)	100
Topology (Star or Inline)	Inline
Update Rate (microseconds)	0 (not used)
Time Function (See section "Program "test")	Constant

Table 13. Test for inputs in multiprocessor machines.

The charts depicted for these tests are focused on the exchange of messages. Two points are of most relevance. First, the messages required to achieve the *Aligned* state in SCSP. Second, the influence of the cache size in the time required to replicate it.

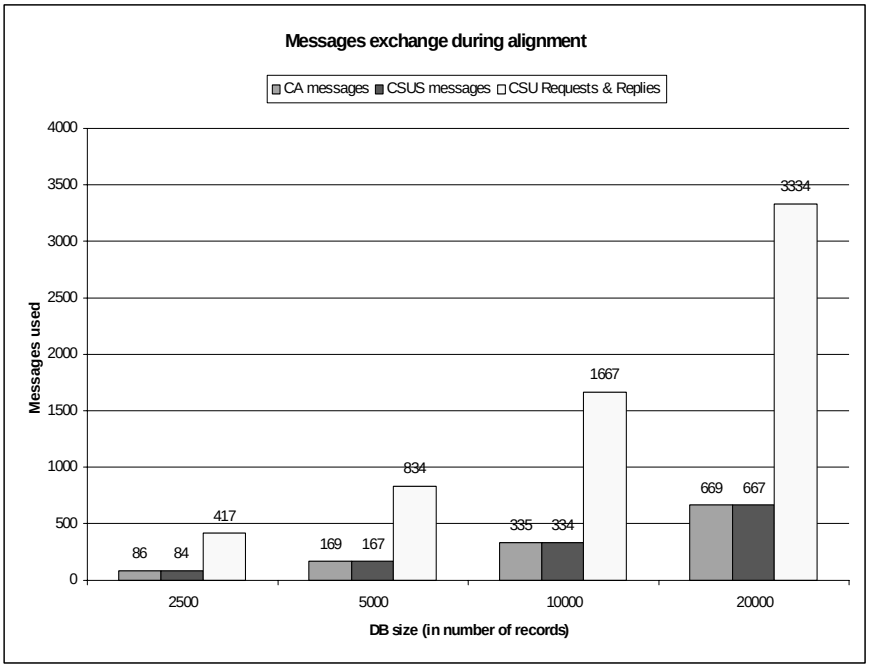


Figure 25. Alignment and Cache Summarize upgrowth times

Figure 25 shows the number of messages sent to synchronize different sizes of caches. It is expressed in three terms. First, the number of CA messages. These messages are used to exchange the summary information between caches and establish the roles of the servers (Master/Slave). Second, the number of CSUS messages. They are used to request summary information to the peer server. Third, the number of CSU (Request and Reply) messages. They

** Notice that the times recorded in the test files are in terms of seconds

carry the CSA information requested by CSUS messages (Requests) and the acknowledgement of this information (Replies).

The analysis of Figure 25 shows the breakdown of the information into SCSP messages. We can see that the maximum CSAs that fit into a CSU Request message is 6 (for a CSA data size of 100 characters). Adjoined to this, the CSAS records inside a CSUS or CA message cannot be more than 30.

The number of CSAS or CSAS in a message and the number of messages required can be forecasted. Hence, for a CK of 16 bytes the size of the CSAS is 32 bytes. The size of a CSA record includes the CSA data (E.g. 100 characters). Thus, the CSA size is 132* bytes. The records must be inserted with the added SCSP header (See [20]) in packets of 1000 bytes size (See Table 11). Thus, the space left by the header is free to be filled with records.

The amount of CA messages is a few elements higher than the number of CSUS messages depending on the number of messages required to perform the Master/Slave negotiation.

Figure 26 shows the messages growth for aligning different sizes of caches. We can see linear increments in both of them as we increase the number of records being replicated. We must notice here that the tests do not undertake the case of leaks of messages. Therefore, there is no resending of messages.

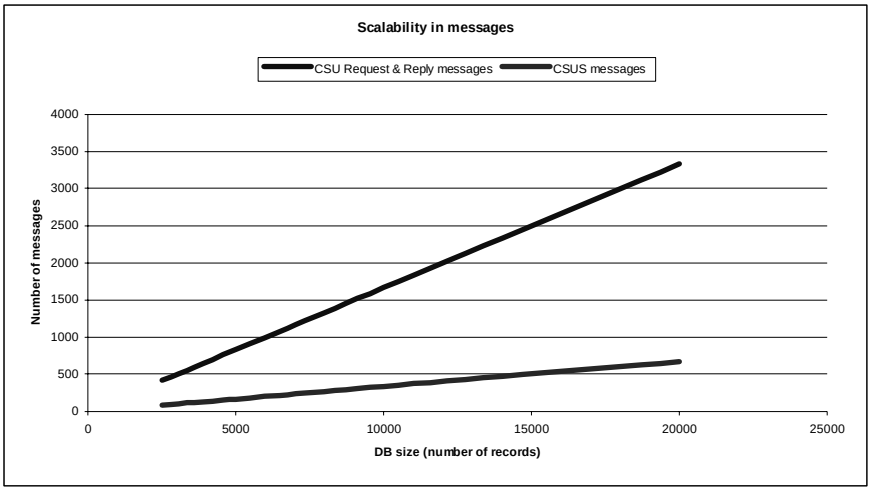


Figure 26. Scalability analysis of the number of messages consumed in CA

* In the implementation, the size of the CSA packets is 8 bytes longer than it should. This can be optimized by removing the field "CSA_Prot_Data_len", which contains redundant information (See CSAS format) and correcting the packing of messages. These flaws are the reason of some incorrect packing of CSA records, i.e. the case of tests for records of 100 characters.

Figure 27 shows the growth functions of the times required to achieve the aligned state and to finish the replication session. These times are different due to delays in the reception of CSU Reply messages. An LS is aligned when it has received all the CSAs previously requested. This is defined in Figure 27 as "Alignment time". However, the replication does not finish until all the CSU Reply messages from the DCS have been processed, i.e. the LS notices that the DCS is aligned also. The total time spent in the replication is defined as "Replication time" in Figure 27. The trend lines show the growth of time required for both phases. They can be used to forecast the time required to replicate other sizes of caches.

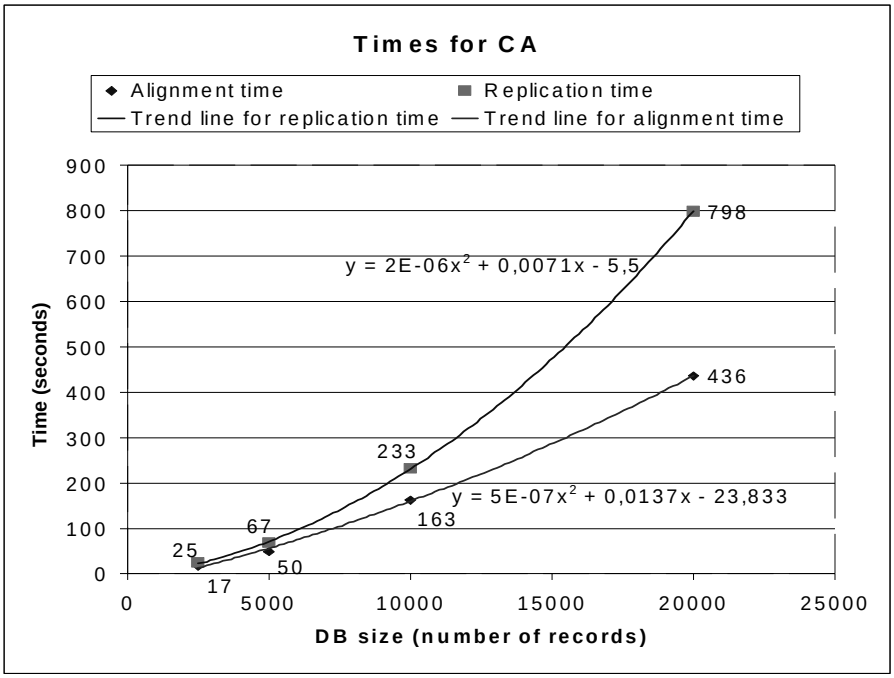


Figure 27. Times required for cache alignment

Figure 28 is a comparative analysis of the input of messages. The "Sending" line represents the input of messages to the MQ by the test tool. The "Reception" line represents the extraction of records by the SCSP. It only analyzes the case of 20000 records.

In the chart, we find the first problem in the implementation. The time consumed in the collection of 2 Mbytes of records is high. Two factors affect this misbehavior. First, the switching between threads in the SCSP produces oscillations in the retrieval of inputs. It is analyzed in Figure 26. Second, the processing of records (in program "test" or program "scsp") delays the generation and collection of inputs. The possibility of delays caused by the intermediate MQs system between the processes is analyzed in Figure 26.

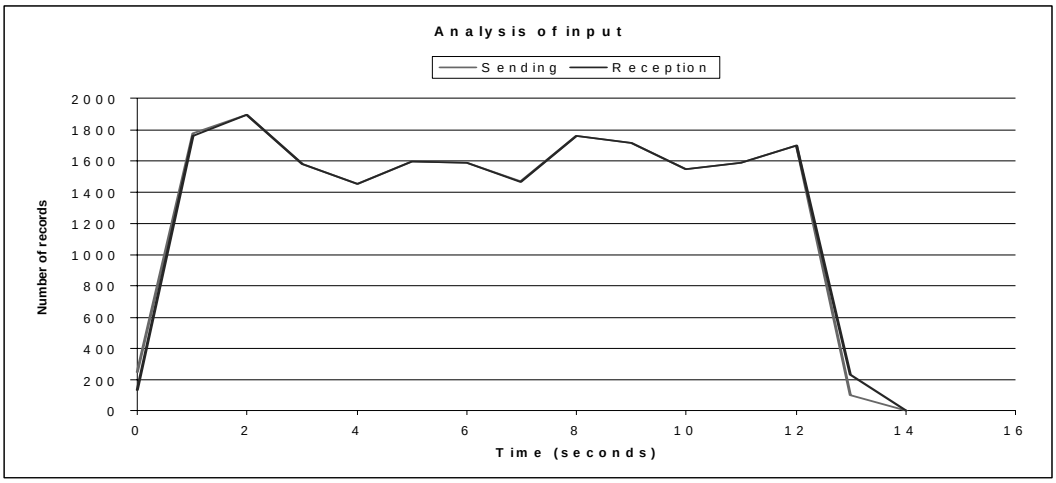


Figure 28.. Analysis of input during CA

Figure 29 shows a comparative analysis of the input in a machine with four processors against a monoprocessor machine. In this chart we can see the performance of the threading in a parallel environment. In the multiprocessor, the average input per second is increased from 1250* (monoprocessor) to 3333'333; i.e. 1 update each 0'3 milliseconds.

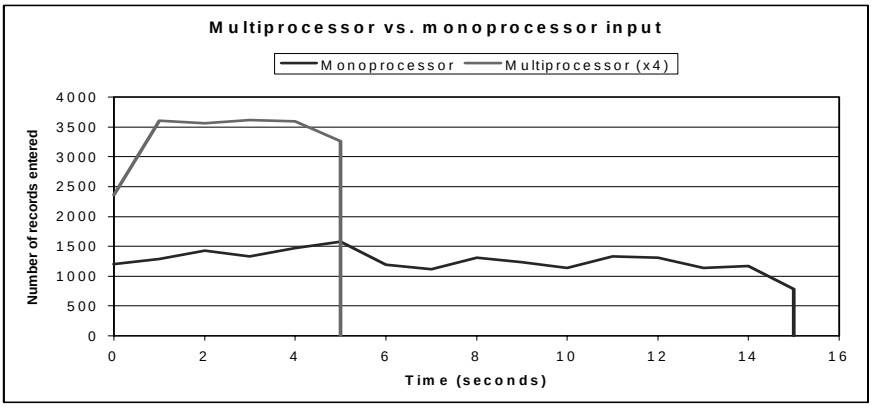


Figure 29. Comparative analysis of input in different machines

Figure 30 shows the trace of the input for a machine with four processors. If delays were produced by the use of MQs as communication facility between the interface (program "test") and "scsp" the analysis of this figure would show some gap between the lines. The only visible differences in the chart appear at the first and last seconds of the input process. The delay produced during the first second in the input for the SCSP could be caused by the MQ. However, this gap is reduced and canceled in the peaks of the functions (seconds 1 to 3). Thus, we can consider the MQ a suitable facility for communication between these processes.

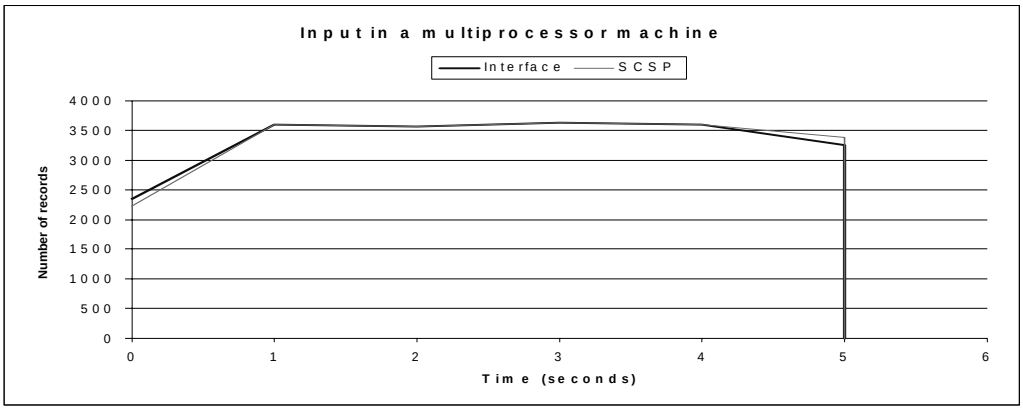


Figure 30. Input in a multiprocessor machine

Figure 31 shows the use of the CPU in a monoprocessor machine. The percentage of CPU used is maintained up to 90% during the most of the time. It means that the high times needed in replication are due to slow processing and not to delays in I/O operations (See 3.3.3).

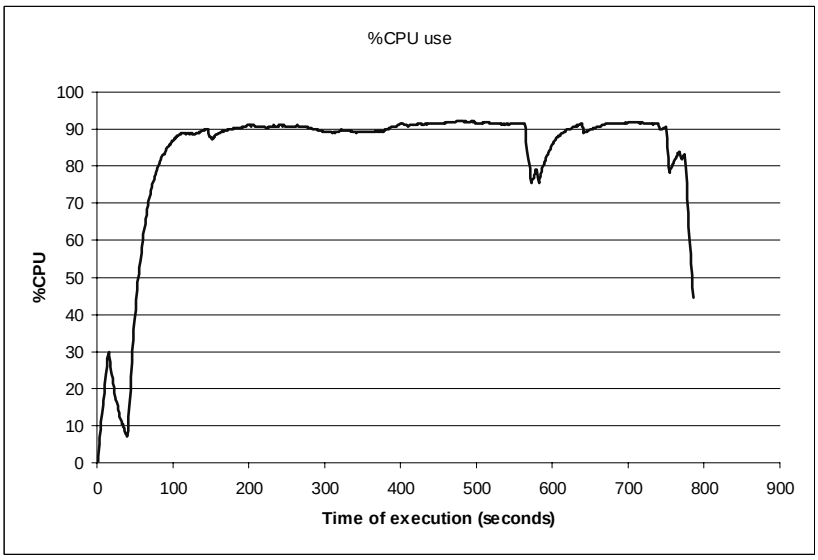


Figure 31. Percentage of CPU used by SCSP during CA

CSA data size tests

These tests change the size of the CSA data. The selected sizes are 100, 200, 300, and 400 characters. The tests are performed during the CA phase. The records are allocated in the

[*] 1 update each 0'8 miliseconds.

"updates file". The "initial file" is empty. The servers used are ws17 and ws18. The results are analyzed from the data returned in ws18. Table 14 defines the parameters of the test.

Number of servers	2
Data Set Size (Either DB or Updates Set)	1,000
CSA data size (in characters)	**Variable**
Topology (Star or Inline)	Inline
Update Rate (microseconds)	10
Time Function (See section "Program "test")	Constant

Table 14. Tests for different sizes of data

Figure 32 shows the average CSAs in the CSU Request messages for the different cases of CSA records. The calculation of the number of CSA records that fit in a message was pointed out in the previous tests for database sizes. In this chart, we also present a trend line of the average number of CSAs in the messages. The function of the trendline is included in the chart.

It is important to notice that the values of the average CSAs per message are not integer in all the cases. It is due to the functioning of the CA phase. In this phase, the CSU Requests are sent in response to CSUS messages[*]. E.g. if the CSUS messages contain 30 records each, the LS receives 30 CSAS records and replies with 30 CSA records. If the size of the cache records is 200 characters, the requests are sent a series of "4, 4, 4, 4, 4, 4, 4, and 2". Thus, the average CSAs per message is 3'75.

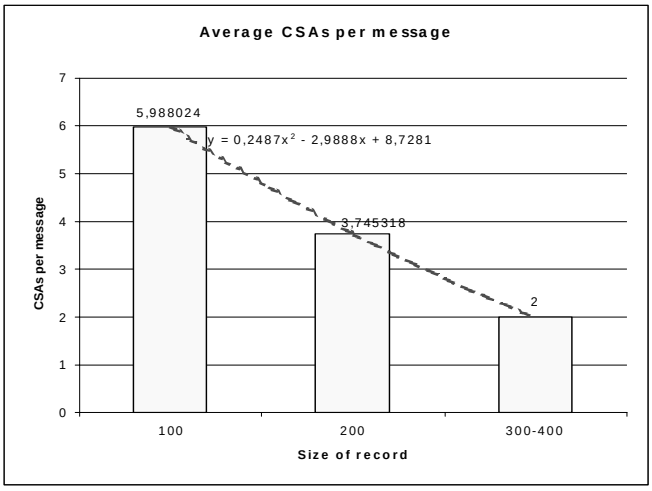


Figure 32. Growth of the average CSAs per message for different sizes of CSA record

[*] A second reason is the sending of the last CSA records left in a non-full packet.

Figure 33 shows the growth of the number of CSU Request messages delivered depending on the size of the CSA record. Figure 32 showed that when the size of the CSA increases, the number of records in a message decreases. The protocol reacts to this by sending more messages per second. The reason of this can be understood looking at the implementation. When the protocol processes the new CSAS arrived in the "NewInLocal" (See "Input thread: "Fill_CSAList" at point 3.3.3) it loops until sending all of them. The aim is to avoid delays in the replication of the records. If the records are longer, the protocol inserts fewer records in a message. Thus, it needs more loops to send every record arrived.

The polynomic trend line in the chart shows the growth function. It allows forecasting the results for other sizes of records.

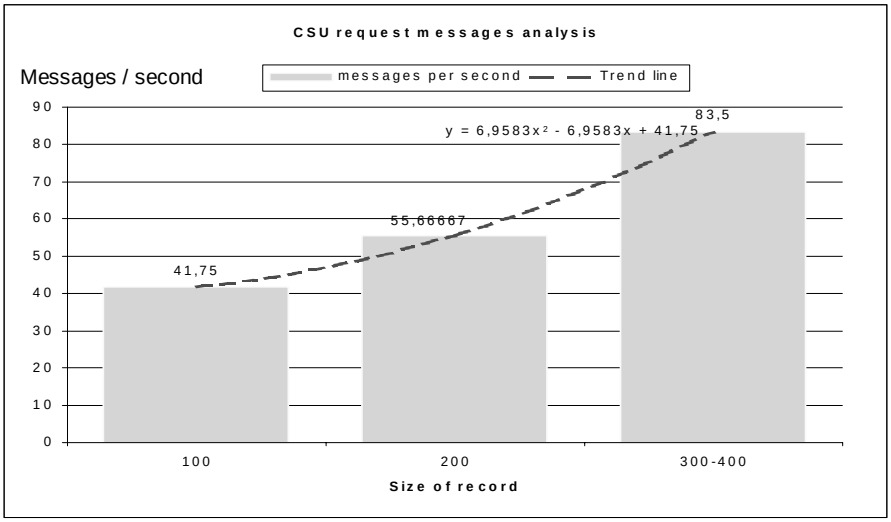


Figure 33. Growth of CSU Request messages for different sizes of records

A problem found at these tests is the leak of messages. When the rate of messages per second increases, the protocol sends more messages to the communication sockets. In the program "scsp", the buffers which are supporting the sockets are extended to the maximum. However, this is not a scalable solution. A feasible correction is to implement a windowing sending system that increases the resilience of the protocol.

Figure 34 shows the influence of the CSAs per message in the processing. Although the number of records in the replication session is the same for all the trials, we can see how it affects the performance of the SCSP. The factor that generates the delay is not the size of the record but the number of messages to process (See also Figure 35).

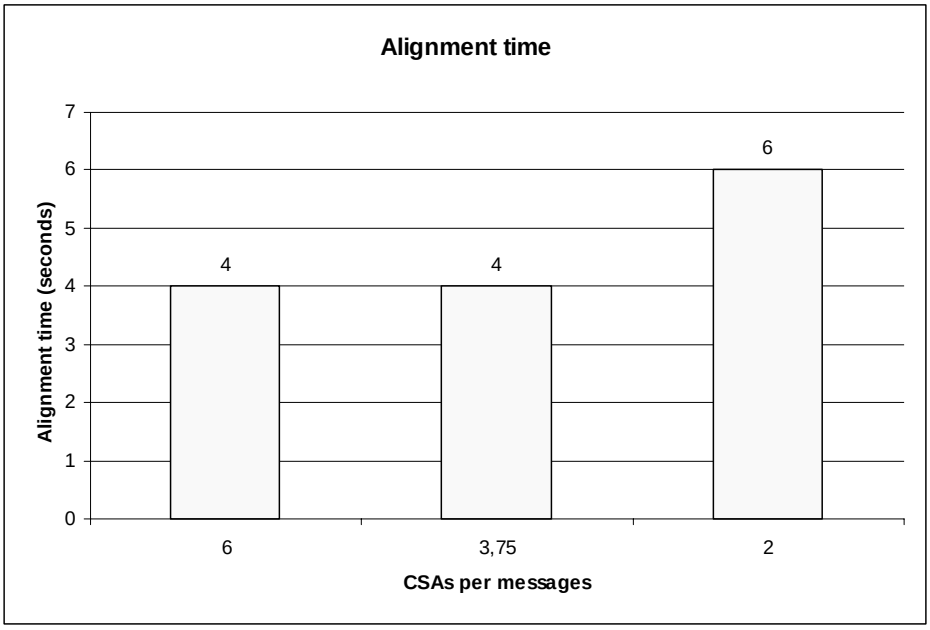


Figure 34. Influence of the size of the CSA record in the replication time

Figure 35 shows the growth of the alignment time in terms of the CSU Request messages. The coordinate axis represents the number of CSU Request messages used during the CA for different sizes of records. The ordinate axis represents the time consumed during CA in the same cases. The trendline guesses a linear increment of the relation.

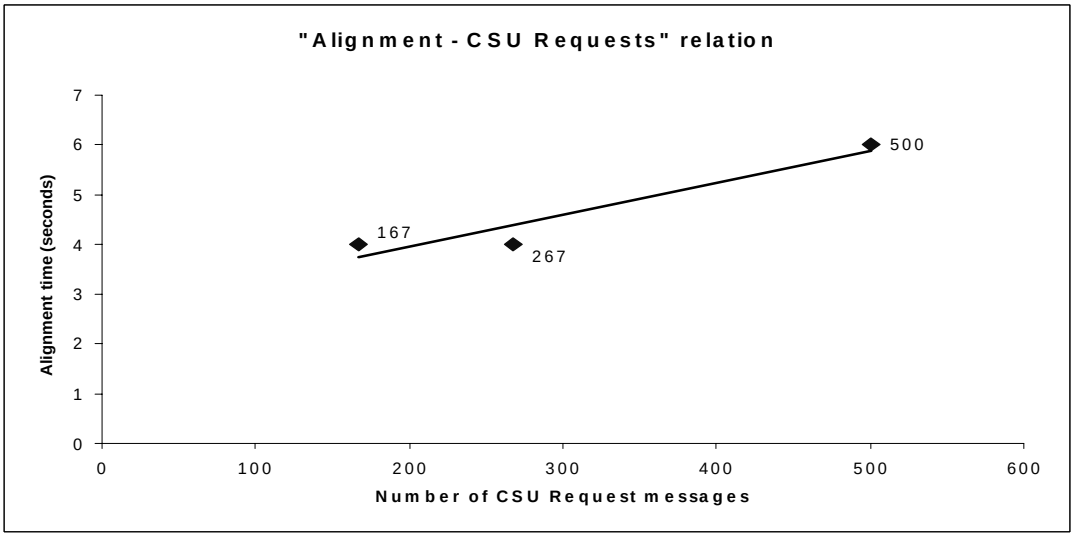


Figure 35. Alignment time against number of CSU Request messages processed

Mean update time tests

These tests use the exponential and quadratic variants of the *test* program. One exception raises here. The data set does not provide enough entries to obtain a long time evaluation of the protocol. Thus, this parameter is modified to obtain a feasible data set size. The selected size is 5,000 records.

The records are allocated in the "updates file". The "initial file" is empty. The "updates rate" does not have any effect in the test. The servers used are ws17 and ws18. Table 15 defines the parameters of the test.

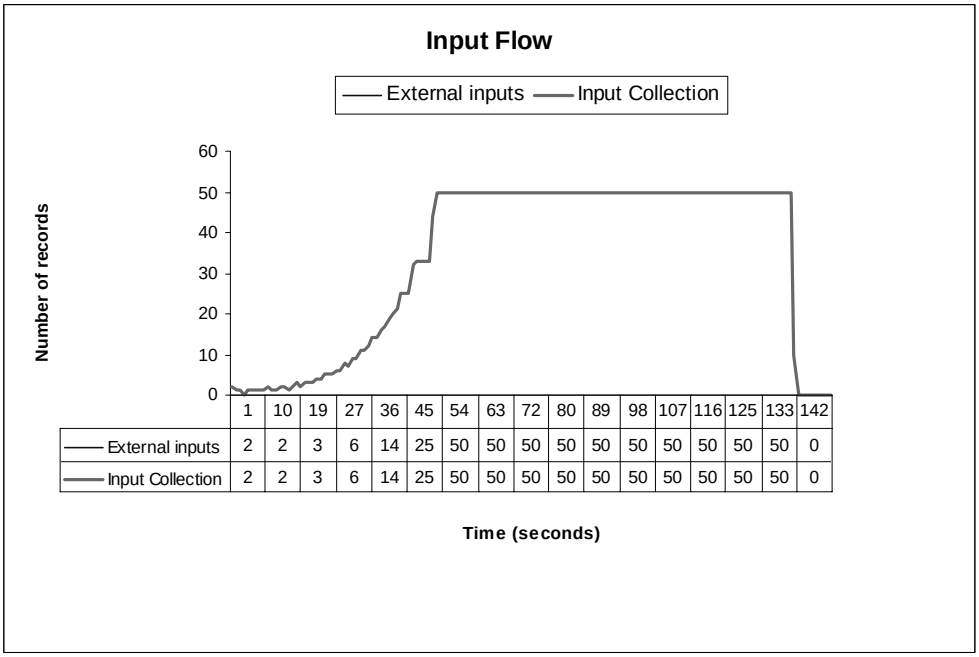
Number of servers	2
Data Set Size (Either DB or Updates Set)	**5,000**
CSA data size (in characters)	100
Topology (Star or Inline)	Inline
Update Rate (microseconds)	10 (unused)
Time Function (See section "Program "test")	**Variable**

Table 15. Tests for different update rate models

Exponential case

The analysis of the replication model for an exponential increment of updates is dealt from the input retrieval perspective. The protocol receives more updates each second. It allows exploring the reaction of the protocol to different update rates.

Figure 36 shows the response of the protocol to increasing update rates in the input thread. The analysis of the chart shows a limit in the update rate of 50 updates per second. This rate is very low and we saw in previous tests that it must be higher (See Figure 29). Therefore, the test tool is analyzed and checked for corrections. After several test we find out that the origin of the incorrect functioning is the use of the library function "usleep". This function is in charge of delaying the next update in the sequence a specified number of microseconds. The function call does not perform properly and the delay has its lower limit in 20 milliseconds. In order to finish with these tests the test tool must be provided with a correct function call able to execute this delay properly. However, Figure 37 describes the behavior of the protocol up to that update limit.



	1	10	19	27	36	45	54	63	72	80	89	98	107	116	125	133	142
External inputs	2	2	3	6	14	25	50	50	50	50	50	50	50	50	50	50	0
Input Collection	2	2	3	6	14	25	50	50	50	50	50	50	50	50	50	50	0

Figure 36. Response to the growth of external input

Figure 37 shows the relation input vs. consistency state in the exponential case. For small update rates, the consistency is kept at the same level than the input (0 to 20 seconds of execution). When the update rates are high enough, the two graphs separate each other progressively (20 to 50 seconds of execution). Fluctuations in the input rate and consistency state are sharper.

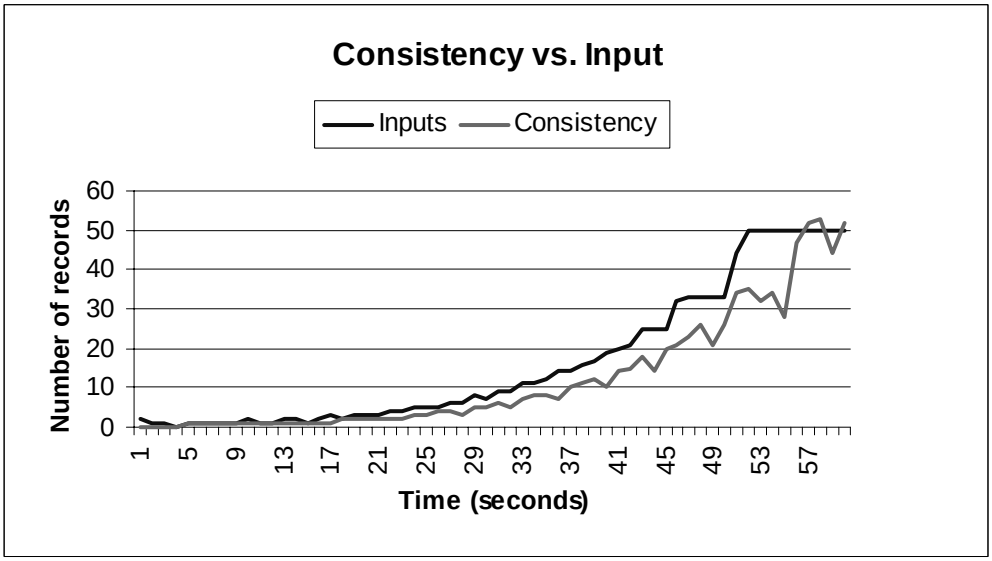


Figure 37. Influence of the input rate in the consistency state

4.4.2. Scalability tests for different SG configurations

Two kinds of tests characterize the behavior of the protocol working within different group topologies and sizes. These tests aim to show the data flow and consistency of the SG regarding its configuration. The results must show the time required updating the records and the influence that the group topology has on it. Data is collected during the Cache Update phase.

The servers are used as required according to the following list: ws17, ws18, ws16, ws20, ws22, ws23, ws7, ws5, ws21, and ws2. (E.g. the inline topology for six servers uses ws17, ws18, ws16, ws20, ws22, and ws23)

Inline topology tests

These tests show the influence of the number of intermediate servers in the replication. The generic case of two servers is extended longitudinally (See Figure 22). For each test, a number of servers are attached to the last node in the replication sequence. The selected group sizes are 3, 6, and 10.

The number of updates must be constant for the SG in every test [33]. To generate the same set of updates, the amount of inputs entered in each server must be divided by the size of the SG. The data set used in the generic case is 2000. E.g. for the test using five servers, each server must be loaded with 400 records. Table 16 describes the values used for testing. The intermediate node analyzed is ws17. The border server analyzed is ws16. The sequence of servers for the different SG sizes tested is always: ws16, ws17, ws18, …

Number of servers	**Variable**
Data Set Size (Either DB or Updates Set)* above	**Variable**
CSA data size (in characters)	100
Topology (Star or Inline)	Inline
Update Rate (microseconds)	50
Time Function (See section “Program “test”)	Constant

Table 16. Test for different sizes of SGs in inline topology

The results in Figure 38 show the functioning of the protocol for an increasing number of severs in the group. The *Average update delay* represents the average time required for an update to be replicated in an intermediate node. The *Update time* represents the elapsed time since the first update was generated in the group until the last update was received in this node. It is calculated by checking the earlier and latest timestamps in the updates received. The *Replication time* function represents the time elapsed between the Cache Alignment phase and the reception of the last CSU Reply message.

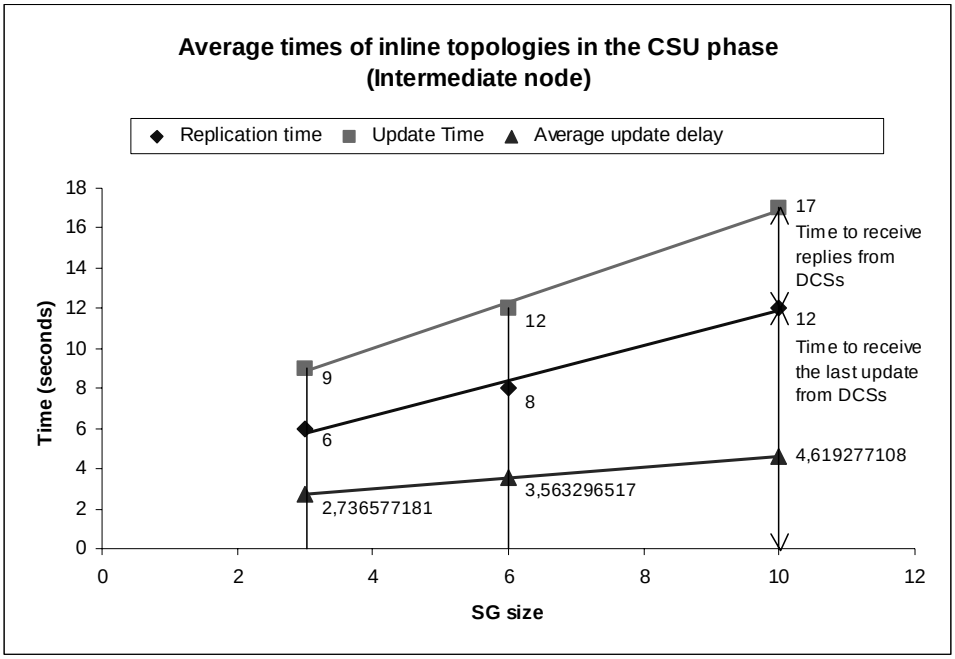


Figure 38. Times for the intermediate node in the inline topology (See Figure 22)

We can determine that the time to receive the CSU Replies experiences a small growth as the size of the SG increases. The main reason to explain this increment can be found in the subdivision of the group records into more packets. The update set of the SG is the same but the update set for each server is smaller. It generates more flow of messages and affects the performance of the protocol even though they replicate the same amount of records (See Figure 35 and its analysis).

Figure 38 describes also a linear growth of the delay between the updates for the scales used in the tests. These configurations are more dependent on the network delays than in the SCSP performance. The network used in the tests is fast enough to introduce only a negligible delay in the replication. The factor that influences the growth of the time to replicate the data is the distance (in hops) that the records must traverse. In each hop, the server introduces a delay,

which affects the replication in the SG. The global delay in the group is linearly proportional to its size.

Figure 39 shows the results of the same tests as in Figure 38 for one endpoint node. In the chart below, we can observe the same behavior explained before. The major difference with the previous chart to point out here is the growth of the times of replication in both lines of the chart (*Replication time* and *Average delay*). It is due to the longer distance that the updates must traverse to reach these final nodes in the group.

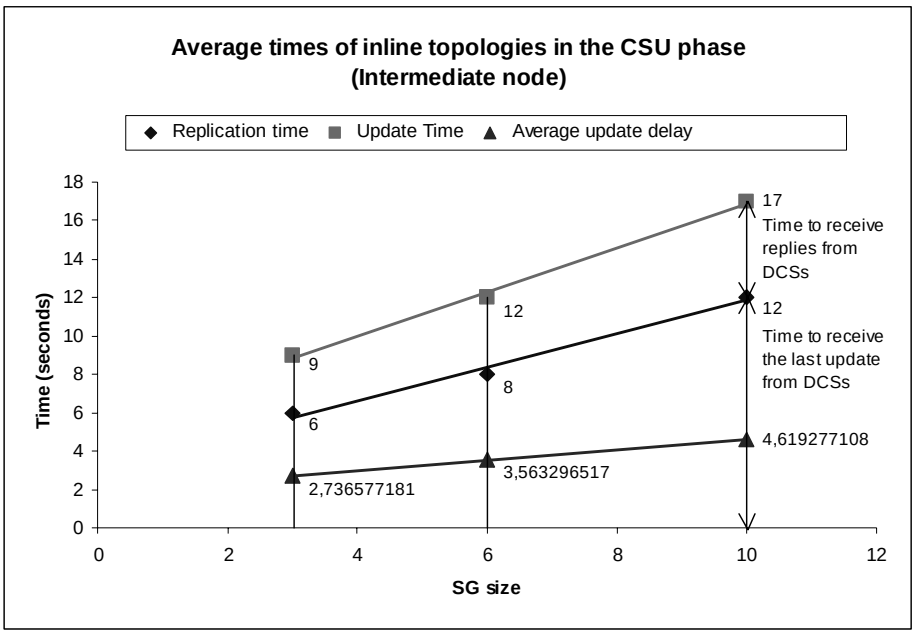


Figure 39. Times for endpoint nodes in the inline topology

Star topology tests

These tests show the impact of the number of servers converging to one point. The convergence point is a node that must communicate the updates received from each server to every other in the SG (See Figure 21). The tests increment the number of servers attached to the same node in every step. The values selected for testing are 6, and 10. The case with three servers for star topology is the same than for linear topology.

The description of the test parameters was the one depicted in the inline case (See Table 16). Only the configuration of the SG is changed to acquire star configuration. The intermediate node used is the ws17. The endpoint server analyzed is the ws16. Every other server is an endpoint server connected to the ws17.

Figure 40 describes the same test as in Figure 38 for star topologies. For explanation of the lines in this chart see Figure 38.

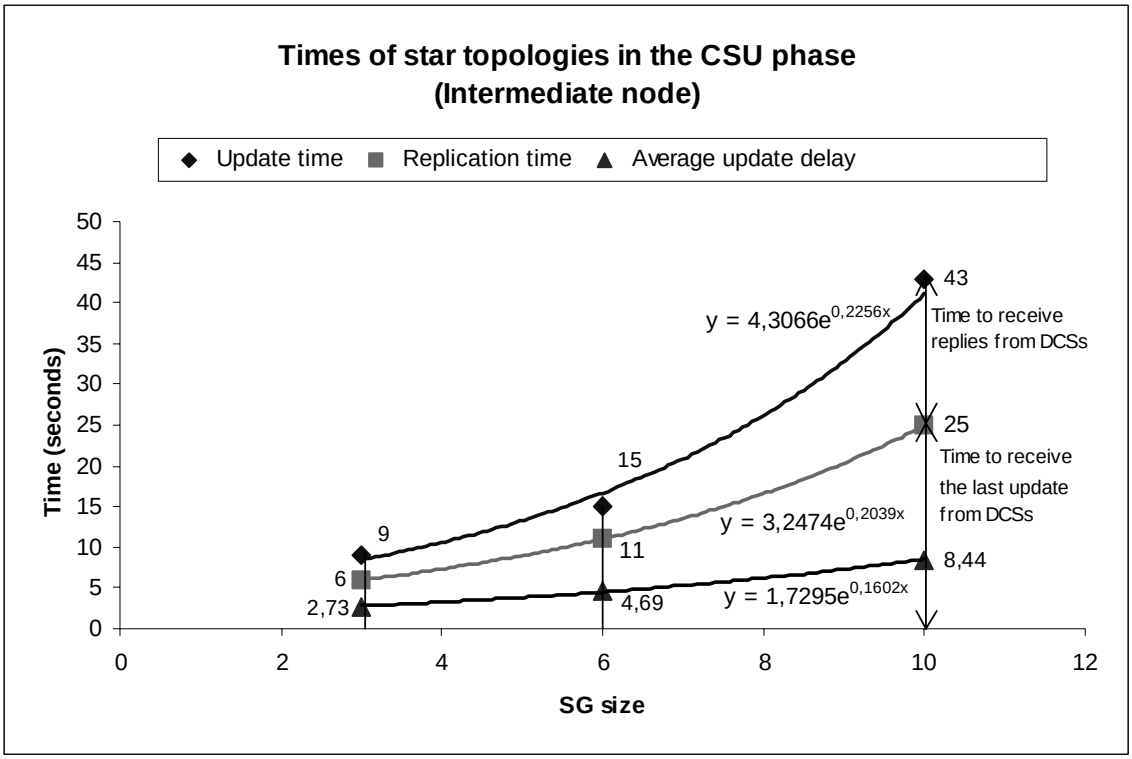


Figure 40. Times for intermediate nodes in the star topology

Figure 40 shows the effect of a concentration of update flows in a single node. The growth of the delay is sharper than in the previous cases. The intermediate node funnels the updates from every neighboring server. The node must process all this information, reply to it and transfer it to the other DCS. The trend lines for each growth function of the different time values are displayed in the chart above. These times are interpolated by exponential functions.

Figure 41 shows the effect of the size of the SG in the endpoint nodes. The growth of the delay functions is higher than in Figure 39. This test pushes harder the performance of the protocol. Although the elements in the SG are closer, the replication is slowed down. We can see how the concentration of flows in a node is an unwise solution and damages dramatically the performance of all the SG.

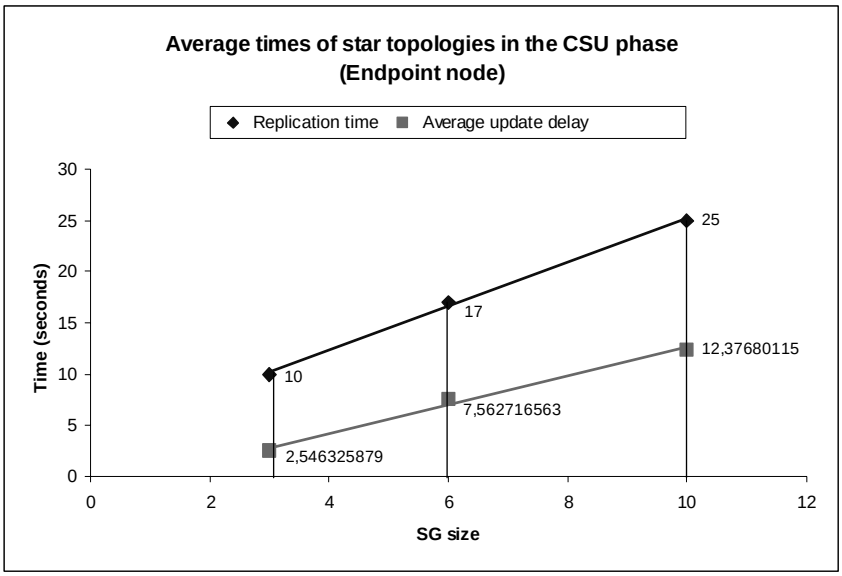


Figure 41. Times for endpoint nodes in the star topology

Star topologies are generated in inter-group nodes, i.e. border nodes connecting different SGs. A good configuration of the replication topology must take into account the effects of star topologies and their influence in the whole replication system. Figure 42 and
Figure 43 show comparative delay functions for the star and inline models.

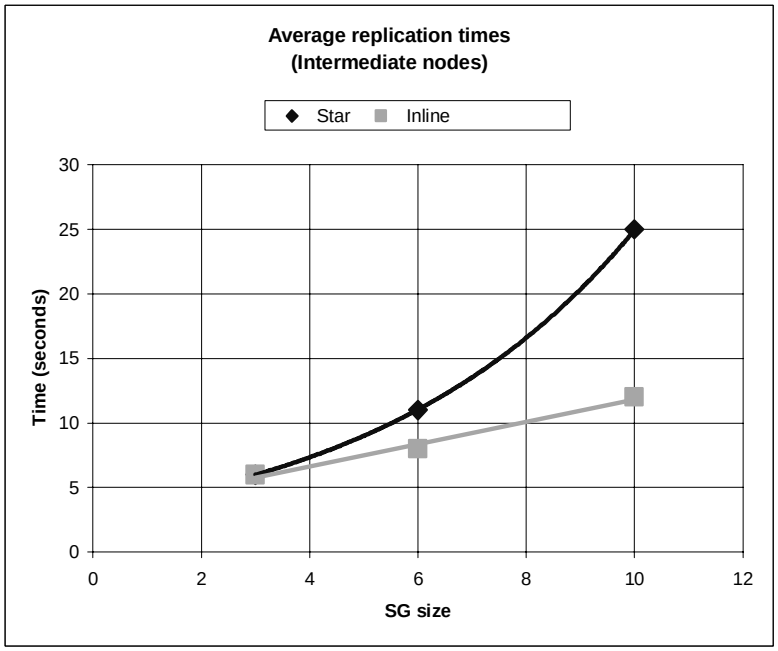


Figure 42. Comparative average replication time for intermediate nodes (Star vs. inline topology)

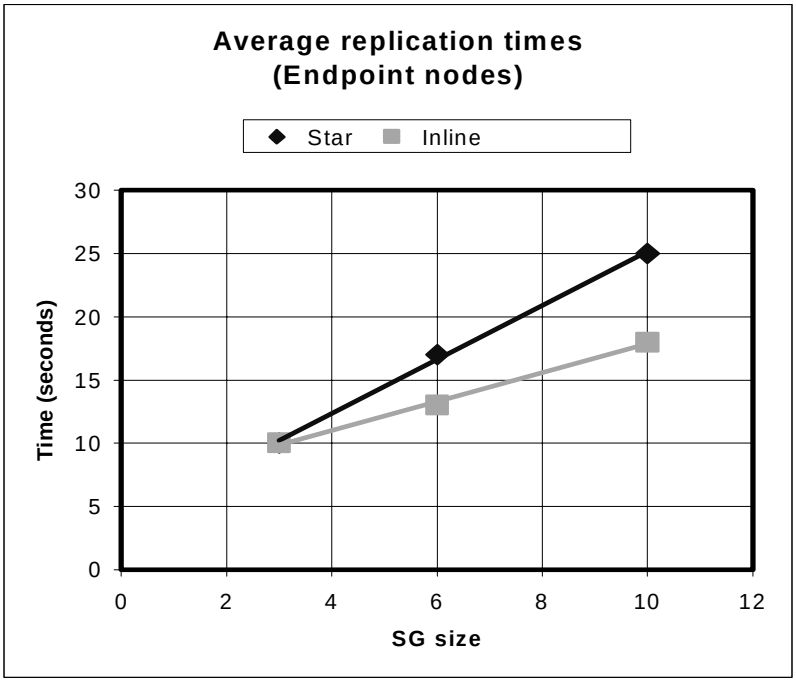


Figure 43. Comparative average replication time for endpoint nodes (Star vs. inline topology)

5. Conclusions and future work

In this thesis, we have upgraded a previously implemented version of the SCSP and evaluated its scalability. The main features added consist of threading the IO tasks and creating an interfacing system for the program. A test of the final program concludes the study.

The new version of the protocol divides the data flows found in the functioning of the SCSP and manages them with different threads. This minimizes waiting periods and provides optimal throughput. In order to fulfill the current implementation, the program is provided with a message queue system that undertakes the IO tasks. Programs using the SCSP connect to the corresponding message queues associated with an SCSP instance and manage the replication through them. The new program executes as a system daemon that replicates transparently any input record.

The tests evaluate the scalability of the implementation. The evaluation is done in terms of data being replicated and configuration of the servers involved. In the former tests, we study the replication of possible models of caches. The caches are typified according to number of records, size of records, and number of updates received per second. These tests are executed for server groups with only two elements.

In the latter tests, we analyze the protocol from the view of different SG configurations. In this case, the program is analyzed for several numbers of servers replicating among themselves. The topology of the SG is important to obtain an effective replication. Therefore, two generic configurations of the connections between servers are studied. First, an SG in which all of the servers are connected to the same server (Star topology). The shared peer server within the group becomes a bottleneck and a critical point of replication. Second, an SG in which each server is connected to two servers. No circles are generated in the topology, thus the group consists of a sequence of servers (Inline topology). The aim of this test is to evaluate the influence of the number of servers traversed by a replica in the replication time.

The analysis concludes with the necessity of further improvements in several features. The access to buffers performs poorly and is the main cause of some unexpected results, i.e. List_Buffer and NewCSA buffers. The information held in the program must be organized so the management of the records can be performed fast. The proposed solution consists of the use

of some indexing or ordering method. It is important to notice that the buffers keeping track of the messages sent to each peer must be provided with similar features, i.e. the resend buffers perform random access to their data when receiving acknowledgments and these searchs can be expensive tasks.

A second problem regards communications. When the protocol processes data at a high speed, the program tends to loose messages. It is caused by the overload of the communication buffers. Even though the protocol implements resending procedures which deal with this problem, the program must avoid leaks of messages when possible. The solution proposed here consists of the use of some windowing system in order to control the sending of messages up to a limit. This limit should be dynamically configured depending on the network capacity.

We found an important lack of performance in the behavior of the protocol in tests for the star topology. The replication time grows exponentially when it should do quadratically according to the previous tests for different loads of replication. The main reason for this problem was related above when commenting on the buffering problems. We found that further improvements could be done here by forecasting if the information arrived to the central server is going to be sent to others and cache it properly.

During the development of the program, we faced situations not anticipated in the RFC document. Those situations were solved but they should be documented and reported.

In future works, in order to complete the implementation of the SCSP program, we recommend the development of a dynamically configured resend timing system, instead of the current static system. Finally, the program should be tested in a real application environment before being ready for release. This test must support the software product.

6. References

[1] Merrian Webster Dictionary. URL: http://www.m-w.com

[2] FOLDOC Computing Dicctionary. URL: http://foldoc.doc.ic.ac.uk/foldoc/index.html

[3] Fan, L., Cao, P., Almeida, J., "*Summary Cache: A scalable Wide-Area Web Cache Sharing Protocol*". Technical Report 1361, Department of Computer Science, University of Wisconsin-Madison, Feb 1998.

[4] Kantor, B., Lapsley, P. "*Network News Transfer Protocol*". RFC 977. February 1986.

[5] Postel, J., Reynolds, J. "*File Transfer Protocol (FTP)*". RFC 959. October 1985.

[6] Scalability in Distributed Multimedia Systems

[7] Internet Software Consortium. URL: http://www.isc.org

[8] GSM World. URL: http://www.gsmworld.com/membership/graph1.html

[9] Barish, G., Obraczka, K., "*WWW Caching: Trends and Techniques*". January, 2000.

[10] Danzig, B., De Lucía, D., Obraczka, K., "*Massively Replicating Services in Wide-Area Internetworks*". Computer Science Department, University of Southern California, January 1994. 13 pages.

[11] Melve, I., Tomlinson, G, "*Internet Web Replication and Caching Taxonomy*". Internet Draft. February, 26 1999.

[12] Bowman, C., Danzig, P., Hardy, D., Manber, U., Schwartz, M., "*The Harvest Information Discovery and Access System*". December 1995.

[13] Bestavros A. "*Demand-based Document Dissemination for the World Wide Web*". Technical Report BU-CS-95-002, Computer Science Department, Boston University, January 15, 1995 (Revised March 23, 1995). 19 pages.

[14] Michel S., Nguyen K., Rosenstein L., Zhang L., Floyd S., Jacobson V. "*Adaptive Web Caching: Towards a New Global Caching Architecture*". Computer Networks & ISDN Systems, 1998. 10 pages. URL: http://wwwcache.ja.net/events/workshop/25/3w3.html

[15] Gwertzman J., Setzer M. "*The Case of Geographical Push Caching*". Fifth Annual Workshop on Hot Operating Systems, 1995. 5 pages.

[16] Golding, R., "*Weak-Consistency Group Communication and Membership*". University of California, Santa Cruz, 1992.

[17] Hi-Tech Dictionary. URL: http://www.computeruser.com/resources/dictionary/noframes/index.html

[18] Stokes, E., Good, G. "*The LDUP Replication Update Protocol*". Internet draft, July 15, 2000. 13 pages.

[19] Merrells, J., Reed, E., Srinivasan, U. "*LDAP Replication Architecture*". Internet draft. April 22, 1998. 31 pages.

[20] Luciani, J., Armitage, G., Halpern, J., Doraswamy, N. "*Server Cache Synchronization Protocol (SCSP)*". RFC 2334. April 1998. 27 pages.

[21] Moy, J., "*OSPF Version 2*". RFC 2328. April 1998. 137 pages.

[22] Watterson, K., "*Database Replication Explained*" Earth Web, online <URL: http://datamation.earthweb.com/datab/09db-rep.html>, July 2000.

[23] Costa-Requena, J., "An Implementation of the Server Cache Synchronization Protocol". Helsinki University of Technology, Laboratory of Telecommunications Technology, 1999.

[24] Jepson, B., Hughes, D., "*Official Guide to MiniSQL 2.0*" John Wiley & Sons Inc., 1998.

[25] Cavanaugh, J., Spengler, M., Thomas, J., "*ATMARP/SCSP interface daemon*". Network Computing Services, Inc. April 1998.

[26] Hays M., "POSIX Threads Tutorial". *University of Arizona, Department of Mathematics*, <URL: http://www.math.arizona.edu/swig/pthreads/threads.html>, May 1998.

[27] Behforooz, A., Hudson, F., "*Software Engineering Fundamentals*". Chapter 5. Oxford University Press 1996.

[28] Mauro, J., "*Communicating the POSIX Way*". Sunworld online <URL: http://www.sunworld.com/sunworldonline/swol-12-1999-insidesolaris-2.html>

[29] "*SCSP daemon*". URL: http://www.gsp.com/cgi-bin/man.cgi?section=8&topic=scspd

[30]Luciani, J., "A Distributed NHRP Service Using SCSP", *RFC 2335*. Network Working Group, Standards Track, April 1998.

[31] Ronsenberg, J., Schulzrinne, H., "*A Framework for Telephony Routing over IP*". June 2000. 21 pages.

[32] Reilly, G., "*Server Performance and Scalability Killers*". Microsoft Web Workshop. URL: http://msdn.microsoft.com/workshop/server/iis/tencom.asp

[33] Obraczka, K., Danzig, P., "*Evaluating the Performance of Flood-d: A Tool for Efficiently Replicating Internet Informationn Services*". IEEE Journal on Selected Areas in Communications, VOL. 16, NO. 3, April 1998. 14 pages.

[34] Lewis B., Berg D. J., "How to Program with Threads. An introduction to Multithreaded Programming". *Sunworld Online*, <URL: http://www.sunworld.com/sunworldonline/swol-02-1996/swol-02-threads_p.html>, Feb. 1996.

[35] *Andrae Muys*, "A Pthreads Tutorial". New Mexico State University, Dept. of Computer Science. URL: http://www.cs.nmsu.edu/~jcook/Tools/pthreads/pthreads.html

[36] Olsen A., Reed R., Smith J.R.W., "Systems Engineering Using SDL-92". Elsevier.

[37] Bacon J., "Concurrent Systems. An Integrated Approach to Operating Systems, Database, and Distributed Systems". Addison-Wesley, 1993. Chapters 9-10.

[38] *Saraswati Balakrishna*, "Pthreads". Boston College, Computer Science Department. <URL: http://www.cs.bc.edu/~mc362/pthreads1.html>

[39] Rational Tutorial. <URL: http://www.rational.com/uml/resources/documentation/index.jsp>

[40] Wang, C., Verrilli, C., Luciani, J. "*Definitions of Managed Objects for Server Cache Synchronization Protocol Using SMIv2*". November 20, 1997. 26 pages.

[41] Mills, D., "*Network Time Protocol (Version 2)*". September 1989.

[42] Lamport, L., "*Time, Clocks, and the Ordering of Events in a Distributed System*". Communications of the ACM, July 1978, Volume 1, Number 7. 8 pages.

[43] Rosenberg, J., Schulzrinne, H. "*A Framework for Telephony Routing over IP*". RFC 2871. June 2000.