Environment for the simulation of different ACP and error recovery in Distributed Data Bases

A.C. Sebastián Ruscuni[1]
Lic. Rodolfo Bertone [2]

Laboratorio de Investigación y Desarrollo en Informática[3]
Facultad de Informática
UNLP

Key Words: distributed systems, distributed data bases, simulation, data integrity, protocols

Summary

This paper presents an evolution of a simulation environment where situations in which a DDB should maintain data integrity are modeled and implemented. These situations include failures taking place during transactions, and the focus is mainly on the utilization and later comparison of different atomic commit protocols (ACP). The implementation was made on Java due to different aspects such as ease of work, portability, etc..

The environment is mainly based on failure recovery of transactions in a distributed data environment, and it allows to choose for the simulation development between two-phase, three-phase, optimistic and pessimistic protocols. The task scheduler defined and implemented to carry out each simulation, based on an execution design establishing each specific problem to be solved, is also specified.

[1] Analista en Computación. Alumno avanzado de la Licenciatura en Informática. Facultad de Informática, UNLP.
Ayudante Diplomado con Dedicación Semi-Exclusiva.
E-mail sruscuni@lidi.info.unlp.edu.ar
[2] Prof. Adjunto con Dedicación Exclusiva, LIDI. Facultad de Informática, UNLP.
E-mail pbertone@lidi.info.unlp.edu.ar
[3] Calle 50 y 115 Primer Piso, (1900) La Plata, Argentina, Teléfono +54 221 422 7707
WEB: lidi.info.unlp.edu.ar

Introduction

A distributed system is defined as a collection of interconnected autonomous computers through a network, with a software designed to provide integrated computation facilities. Distributed systems are implemented on hardware platforms of varying size and connection.

The key characteristics of a distributed system are: resource sharing supports, concurrency, scalability, failure tolerance, and transparency. [COUL95]

There are several reasons for the development and use of distributed data bases: [HANS97]

- Organizations have divisions in different locations. For each location there might be a set of data which is frequently and exclusively used.
- Each location is allowed to store and keep their own local data base, which facilitates immediate and efficient access to the most frequently used data.
- Distributed data bases achieve improvements in system reliability, although they add other elements related to keeping information integrity and security.

A Distributed Data Base can be defined as an integrated collection of shared data which are physically distributed along the nodes of a computer network. A DDBMS is the software needed to manage a DDB in a way which is transparent to the user. [BURL 94]

In a distributed data base system, data are stored in several computers. The workstations of a distributed system are connected by different means, such as high speed cables or telephone lines. They do not share their main memory or their clock.

General DDB technology involves two different concepts, called **integration** through the elements that form a data base, and **distribution** through the elements of a network. **Distribution** is provided when data are distributed to the different locations of the network, whilst **integration** is given from the logical grouping of the distributed data, making them look as if they were a single unit to the end user of the DDB.

A centralized DBMS is a system that handles a single DB, while a DDBMS handles several DBs. The terms global and local are used when discussing DDBMS to distinguish between aspects referring to the single location (local) and to the system as a whole (global). Local DB refers to the DB stored at a location of the network, while global DB refers to the logic integration of all local DBs. [BELL 92]

The use of DDB presents a series of associated benefits and costs that should be evaluated when taking the decision as to whether use them or not. [ÖZSU 91]. Some of the advantages that can be mentioned are:

- Local autonomy: each site of the network in a distributed environment must be independent from the rest. Each local DB is processed by its own DBMS.
- Performance improvement.
- Information availability improvement.
- Economy: the equipment needed to set up a network to distribute information is comparatively much smaller than having a central location

with great processing capacity (associated with the concept of downsizing) [UMAR 93]
- Easily expandable systems.
- Sharing information.

Among the drawbacks and disadvantages, the following can be mentioned:

- Less experience in the development of distributed DB and systems
- Complexity
- Cost: additional hardware is required, as well as more complex software and a transmission channel to connect the distributed nodes of the network.
- Distribution control: this is in conflict with one of the advantages presented. From this viewpoint, the problems created by task synchronization and coordination are considered.
- Security: new and more complex problems related to the security of the information handled appear.

Transactions, integrity and security

A transaction is a collection of operations carried out by an only logic function. Each transaction is an atomicity unit (it occurs as a whole or does not occur at all). The algorithms to ensure the consistency of the DB, using transactions, include: [SILB 98]

- Actions taken during the normal processing of the transactions that ensure the existence of enough information so as to ensure failure recovery.
- Actions taken after a failure to ensure the consistency of the DB.

Transactions transform the data base from one consistent state into another consistent state. However, it should be taken into consideration that the data base can be violated during the execution of the transaction. The four basic properties of a transaction, referred to as **A.C.I.D.**, are:

1. Atomicity: the "all or nothing" property; a transaction is an indivisible unit
2. Consistency: transactions transform the data base from one consistent state into another, also consistent, state
3. Independence: transactions are executed independently from one another (partial effects of an incomplete transaction are not visible for the rest of the transactions)
4. Durability (also called persistency): the effects of the transactions that have already been completed (committed) are stored permanently in the data base and cannot be undone.

   In a distributed data base environment, a transaction can access data stored at more than one site of the network. Each transaction is divided in a series of sub-transactions, one for each location of the network where there are data that the original transaction needs to process.

There are two mechanisms used to ensure transaction atomicity:

- Log-based
- Double page

Atomic Commit Protocols (ACP)

A common model for distributed transactions is based on a process called *coordinator*, which is carried out at the location where the transaction is created, and a set of processes, called *locations*, which are run at the different locations that have to be accessed by the transaction.

In order to guarantee atomicity, all locations at which the distributed transaction *T* has been carried out should coincide with the end result of the execution. *T* must be executed or aborted at all *locations*. To ensure this property, the transaction *coordinator* in charge of *T* should execute a commit protocol.

In the last two decades, data base researchers have proposed a great variety of protocols. Among the most important, the *two-phase protocol (2PC)*, along with two variations called *Pessimistic protocol (PrA)* and *Optimistic protocol (PrC)*, and the *three-phase protocol (3PC)* are included here. According o the functionality of each of them, commit protocols require exchanging messages, through different phases, among the locations where the distributed transaction takes place. Therefore, several logs are generated in stable memory, some of which are synchronously recorded to the disk. Due to these additional costs, commit processing can result in an increment of the execution time of the transaction, thus making the election of the protocol a highly important design decision for distributed data bases systems.

Two-phase protocol

Each transaction is generated at one location, the local coordinator of transactions is in charge of controlling its execution. In the case of a local transaction, the procedure is the same as for centralized systems. Now, if the transaction accesses global data, the coordinator is in charge of subdividing the transaction into sub-transactions which will be run at the different locations of the network. Each participating location receives the sub-transaction it can handle and processes it as if it were a local transaction. [WEB1] [WEB2].

The two phases of the protocol are the following:

- **Phase 1**: the coordinator sends a message to each location involved asking if the transaction was finalized. In the case of a negative answer, or eventually, the lack of an answer, the transaction has to be aborted. If every location answers in the affirmative, the transaction can be committed.
- **Phase 2:** an abortion message (if there is at least one negative answer or after a pre-determined waiting time), or a finalization message (if all locations including the coordinator were able to finish the transaction) is sent. Each location commits the transaction and sends a recognition message to the coordinator.

The protocol used incorporates a *timeout*, that is, a maximum waiting time between two nodes is taken into account. If a sub-transaction does not receive a message from the coordinator , it asks every other one for a message of help. When the other location receives this message, it can help its pair according to its own status; that is, if it has received from the coordinator either a commit or abort message, it re-sends this message. The transaction kept on hold can now carry out the commit or the abortion at ease, since it has received the coordinator's secure message replicated by its partner. On the contrary, if the sub-transaction does not receive any message, it will continue to wait. [ZANC96] [BERN84]

Pessimistic Protocol

A variation of the two-phase protocol (2PC), called **pessimistic protocol**, attempts to reduce the number of messages between the participants of the distributed transaction by using the rule: "in case of doubts, the transaction is aborted". That is, if after a failure, a location asks the *coordinator* about the state of the transaction and it does not receive information about it, it is assumed that the transaction was aborted. With this supposition, it will not be necessary for locations to:

a) Send acknowledgment messages (ACK) to the coordinator for ABORT messages
b) Write the record ABORT on the log. And it will also not be necessary for the *coordinator* to write the ABORT nor the END records on the log for an aborted transaction.

Briefly stated, the pessimistic protocol is identical to the two-phase protocol for transactions which are committed, but it reduces the number of messages and log storage overhead for aborted transactions.

Optimistic Protocol

A variation of the pessimistic protocol is based on the observation that, in general, the number of transactions committed is higher than the number of transactions aborted. In this protocol, called optimistic protocol, the overhead is reduced for the transactions committed instead of for the ones aborted, requiring that all participants of the distributed transaction satisfy a rule stating "in case of doubt, commit". In this scheme, *locations* do not send the acknowledgement (ACK) in the global decision to commit, and do not register the COMMIT on the log. Therefore, the *coordinator* does not write the END record on the stable memory either.

Three-phase protocol

The main problem with the protocols described so far is that *locations* could be *blocked* in the case of a failure until the location that failed is recovered. For example, if the *coordinator* fails after the initiation of the protocol but before communicating its decision to each *location*, these will be blocked until the *coordinator* recovers and informs them of its decision. During the time the *locations*

are blocked, they keep using resources of the system, such as *locks* over some data items, thus preventing these resources from being used for other transactions.

It was to tackle the problem of blocking that the **three-phase protocol (3PC)** was proposed. It activates the ability not to be blocked by adding a new phase: "pre-commit" between the two phases defined by the two-phase protocol. Thus, a preliminary decision is obtained, which is communicated to every participating *location* of the transaction, which in turn allows to generate the global decision about the transaction independently from a possible failure of the *coordinator*. It should be noted that the cost of this non-blocking functionality is that there is a greater number of messages exchanged between the *coordinator* and the *locations*, since there is an extra phase. Therefore, additional records will also have to be written on the stable memory, during the "pre-commit" phase.

The three-phase protocol requires that:

- No network fragmentation is possible.
- There is at least one location working at any point.
- At any point, at most a number *K* of participants can fall down simultaneously (*K* being a parameter indicating the resistance of the protocol to failures at locations).

The protocol reaches this non-blocking property by adding an extra phase to make a preliminary decision about *T*. As a result of this decision, the participating locations know certain information that allows them to make a decision despite the failure of the coordinator.

- **Phase 1:** This phase is identical to phase 1 of the two-phase commit protocol.
- **Phase 2:** If the *coordinator* receives the **abort** *T* message (*T* being a transaction) from a participating location, or if it does not receive an answer from a participating location within a previously specified time interval, it then decides to abort *T*. The decision to abort is implemented in the same way as the two-phase commit protocol. If it receives a message *T* **ready** from each participating location, it will make the decision <<pre-execute>> *T*. The difference between pre-executing and executing *T* is that *T* can still be eventually aborted. The decision of pre-executing allows the coordinator to inform each participating location that all participating locations are <<ready>>. On this basis, it sends a message **pre-execute** *T* to all participating locations. When a location receives a message from the coordinator (be it **abort** or **pre-execute** *T*), it sends back an **acknowledgement to *T*** message to the coordinator.
- **Phase 3:** This phase takes place only if the decision at Phase 2 was to pre-execute. After the messages of **pre-execute** *T* have been sent out to all participating locations, the coordinator should wait for at least *K* messages of **acknowledgement to** *T*. Following this process, the coordinator makes a commit decision. According to this, it sends a message **execute** *T* to all participating locations.

The same as the two-phase commit protocol, a location where *T* has been executed can abort *T* unconditionally at any time before sending the *T* ready to the coordinator. The *T* **ready** message is, in fact, a promise made by the location to follow the order from the coordinator to either execute or abort *T*. As opposed to the two-phase commit protocol, where the coordinator can abort *T* unconditionally at any time before sending the execute *T* message, the message **pre-execute** *T* in the three-phase commit protocol is a promise made by the coordinator to follow the order of the participant to execute *T*.

Architecture used

Computers over the Internet are connected through the TCP/IP protocol. During the eighties, ARPA (Advanced Research Projects Agency) of the government of America, developed an implementation of the TCP/IP protocol under UNIX. A socket interface was then created. Nowadays, this socket interface is the most used method to access a TCP/IP network. [WEB4]

A socket is an abstraction representing a point-to-point link between two programs running over a TCP/IP network. It uses the Client/Server model. When two computers wish to communicate, each of them uses a socket. One of them acts as the "Server" opening the socket and waiting for some connection. The other is the "Client" that will call the server socket to initiate it. In addition to this, in order to establish the connection, only the Server address and the port number that will be used for the communication are needed. For this purpose, the java.net package is mainly used. Once both sockets are connected, data exchanges are carried out by means of InputStreams and OutputStreams. [WEB5]

The advantage of this model over other types of communication rests upon the fact that the Server does not need to have any knowledge about the place where the client is. On the other hand, platforms such as UNIX, DOS, Macintosh or Windows do not offer any restriction to carry out the communication. Any type of computer supporting the TCP/IP protocol will be able to communicate with other computer also supporting it through the sockets model. [WEB6]

Presentation of the research activities

Simulations were carried out over the execution of transactions in an environment of distributed data, indicating for each of them the atomic commit protocol (ACP) to be used for recovery for the different failures that follow, the consistency of the data base being kept at all moments. This environment was implemented on Java.

The different *failure situations* that may occur during the execution of a distributed transaction are:

- *Failure of a participating location.* For recovery from this failure, the log should be examined and the fate of the transaction should be decided based on that.
- *Failure of the coordinator.* In this case, the participating locations are the ones that make the decision about the fate of the transaction. If they do

not have enough information, they will have to wait until the coordinator recovers.

The implementation of the environment was done on Java. The JDK 1.3 was used, together with the Forte For Java 1.0, to build the interface with the user (GUI). Thus, hardware related interoperability is achieved, which allows any computer having JVM (Java Virtual Machine) to execute the environment with no inconvenient.

The use of JAVA presented advantages and improvements. These improvements are mainly related to the simulation environment, which presents a user-friendly interface. In addition to this, Java allows to handle connections via JDBC (Java DataBase Connectivity) with the data base, which is an advantage for the developer of the environment in comparison with other tools currently available in the market.

We considered a distributed system composed by a finite set of locations completely connected through a set of communication channels. Each location has a local memory and runs one or several processes. For the sake of simplicity, only one process per location is assumed. Processes communicate among themselves by asynchronously exchanging messages. At a given moment, a process can be in either of these two states: “Operational” or “Failure”.

When a process is in the “Operational” state, it will follow the actions specified by the transaction being executed. An operational process may fail, which turns it into the “Failure” state. We will consider that a process in such a state will eventually go back to the operational state.

A process that fails stops all its activities, including sending messages to other processes, up to the moment it recovers and goes back to its operational state. Each process has access to the stable storage (log), where the information needed for the recovery protocol is kept. This means that, during the recovery process, the process reestablishes its normal state by using the information on the log. In addition to this, in the case of an instruction to modify data, the technique used is that of data immediate modification, being stored both the old and the new values.

Another supposition taken into account is that if a message is sent from the process P_i to P_k , it is eventually received by P_k, if and only if P_i and P_k are not in "failure".

There are four different types of tasks:

- Transaction Manager
- Transaction Server
- Coordinator
- Agent

The processes simulating the coordinator and the agents participating of the transaction will be called “Threads”, which are created by the “Transaction Server” processes, which in turn are resident at each location.

In a distributed data base environment, a transaction may access data stored in more than one location of the network. Each transaction is divided into a series of sub-transactions, one for each location where there are data that the original

transaction needs to process. These sub-transactions are implemented by the "Agent" processes, which represent each location of the DDB.

Initially, the "Transaction Manager" process is executed, which provides the interface for the user to perform any kind of operation over the data base. This means carrying out a Query, which can be either a consultation or an insertion, deletion or update operation. To carry out these operations, the user should not enter SQL code directly; instead, by means of an interface, the information that will be used by the environment to generate the operation that will be later on executed in the data base, is defined. This definition includes mainly the tables upon which the operation will be carried out, the type of transaction to carry out, the location where the transaction is to be initiated, etc. This last choice allows the user of the simulation that a given operation over the DDB be initiated at different places, and thus be able to obtain and compare results later on, taking into account the initial distribution of data at each location. The location originating the transaction acts as its coordinator and, as such, is in charge of making all decisions related to its execution.

One of the parameters to be defined when carrying out a simulation is the identification of the commit protocol that will be carried out during the execution of the transactions. This will allow that, for the same execution design, the answer of the prototype to the different protocols can be analyzed in order to compare the results obtained.

Each location enabled to take part of the transaction within the simulation environment has a configuration file (still static) that shows the location and use of each table of the distributed environment. Based on this information, the coordinator will determine which locations will take part of the execution of the sub-transactions generated from the transaction to simulate.

Since the study cases chosen for this stage of the simulation environment consider a study of behavior in case of failure, the location (or the coordinator) that will undergo the failure should be indicated in the execution design, the environment under which the selected recovery protocol of the data base is simulated being thus produced. The simulation of a communication line going down and producing only a failure at the location connected through that line is considered in the study case.

Work Environment

The first experiences carried out with the defined simulation environment used a local network; our intention is to take this simulation to a geographically distributed network. The only difference between the experience carried out and the one which will be carried out is the use of workstations, but no change is needed in the defined environment. This is due to the fact that for the creation of the sockets, only the physical address of each location needs to be specified, then the "Transaction Server" processes are run at each of them.

In order to obtain the environment to carry out the experience, the physical location of each location is completely transparent; the problem of geographic distribution being solved by indicating as a simulation parameter the "cost" associated to a transmission between two locations, making it independent from the network scheme used.

Simulation Model

The transactions executed can be defined as local or global (normally within an 80-20 ratio). For the latter, the number of sub-transactions composing it is defined, and which involve, a priori, different locations where they will be executed. Each simulated location should be running a process called “Transaction Server”. In addition to this, the location where the transaction is initiated should run a process called “Transaction Manager”.

The process “Transaction Manager”, from transactions specifications (which can be on a design file or be defined on line), determines which “Transaction Server” process at some location of the simulation environment, will be the Coordinator of the transaction. Then, this thread determines, by consulting the locations table, who the intervening locations will be, and sends a communication to the corresponding “Transaction Server”.

When a “Transaction Server” receives a message from a coordinator indicating its participation as a location in the execution of the transaction, it creates a thread “Agent”.

After the establishment of the communication channels between the location originating the transaction (which acts as the coordinator) and each of the participating locations, the execution of the distributed transaction begins. Each “Agent” executes its part of the transaction, and indicates the coordinator when it finishes. The coordinator implements, as previously said, the atomic commit protocol (ACP) chosen at the beginning of the simulation in order to keep the integrity of the DDB.

Conditions for Comparisons

From the point of view of performance, commit protocols can be compared taking the following issues into account:

- **Effect on normal processing:** This is in relation to the way the protocol affects the processing performance of the normal distributed transaction (without failures). That is, what is the cost of providing atomicity when using this protocol?
- **Flexibility in failures:** A commit protocol is *non-blocking* if, in the event of a failure at a location, it allows the other sites to continue their executions without making them wait for the location that failed to recover. To allow this functionality, additional messages to those used by *blocking* protocols are usually required. In general, *two-phase commit protocols* are *blocking*, whereas the *three-phase commit protocol* is *non-blocking*.
- **Recovery Speed:** This is the time required by the data base to recover when a location fails. That is, the time passed until the transaction can be processed again at the location that has just recovered.

The two first issues are the most important ones, since they directly affect the processing of the transaction. Compared to these two, the last point is less critical due to several reasons, among others, the duration of a failure is usually larger than

the recovery time of the location. From this point of view, it is more important to focus on the mechanisms required during the normal recovery operation instead of its recovery time.

Comparison of protocols

The best atomic commit protocol (ACP) is the two-phase protocol (2PC), which was adopted by most transactional standards, for example DTP of X/OPEN and OTS of OMG; it was also implemented in several commercial data base systems. However, the 2PC is considered to be somewhat inefficient since it introduces a "substantial delay" (blocking) in the processing of a transaction. This delay is related to the time cost associated with the coordination between messages and forced log writings.

Let us suppose that n is the total number of participants (including the coordinator), the 2PC requires three communication stages (the requirement for the "vote", the "vote" and the "decision") and $2n + 1$ forced writings in stable storage, for a transaction with no failures. This introduces a considerable latency in the system, which increases the execution time of the transaction.

The probability of a blocking situation to take place in reality is small enough so as not to justify the extra cost of a three-phase commit. In addition to this, the three-phase commit is very vulnerable when it comes to link failures. This disadvantage can be saved with network-level protocols, but this increases the extra time.

Both protocols can be perfected to reduce the number of messages sent and to reduce the number of registers recorded in stable storage.

These limitations have led several researchers to work on optimized versions or alternatives to the 2PC. One of these is the "Optimistic Protocol " (PrC) and another one is the "Pessimistic Protocol " (PrA). These two protocols reduce the cost of the 2PC as regards log writing times and message complexity.

The latency of an ACP is determined by the number of forced log writings and the communication steps carried out during the execution of the protocol until each participant reaches a decision. Figure 1 compares the different protocols in terms of latency and message complexity needed to commit a transaction. When compared with the 2PC, it can be seen that the PrA does not reduce the cost of committing the transactions. The PrC requires less messages and forced log writings than the 2PC, but it does not reduce the number of communication steps required to commit a transaction.

Atomic Commit Protocol (ACP)	Number of Messages	Latency	Latency
		Log Writings	Number of communication stages
2PC	$4(n - 1)$	$1 + 2n$	3
PrA	$4(n - 1)$	$1 + 2n$	3
PrC	$3(n - 1)$	$2 + n$	3

Figure 1

Conclusions

Our goal was to define, model and implement a simulation environment for the maintenance and data recovery of a DDB system that can be studied, monitored, and later on results compared for the different commit protocols chosen for the same transaction.

Execution designs tested so far show that the variations of the 2PC (PrC and PrA) reduce the overhead of the protocol and provide an improvement as regards the 2PC only for very specific situations. The PrC presents a better performance when the distribution degree of the data is high, but current situations usually have a low distribution degree. On the other hand, the PrA offers an advantage as regards the 2PC when the probability of aborted transactions is low.

Briefly stated, in a DDB environment, it is advisable to use the 2PC, the PrC or the PrA, since they offer betteer benefits than the 3PC. However, if a non-blocking capacity is required, the three-phase protocol is the best option.

Bibliography

[BELL 92] *Distributed Database Systems*, Bell, David; Grimson, Jane. Addison Wesley. 1992

[BERN 84] *An algorithm for concurrency Control and Recovery in Replicated Distributed Databases* Bernstein, Goodman. ACM Trans. Database Systems, vol 9 no. 4, pp. 596-615, Dec 1984.

[BERS 92] *Client Server Architecture.* Berson, Alex. Mc Graw Hill Series. 1992

[BERT 99] *Ambiente de experimentación para Bases de Datos Distribuidas*. Bertone, Rodolfo; De Giusti, Armando; Ardenghi, Jorge. Anales WICC. Mayo 1999. San Juan Argentina.

[BHAS 92] *The architecture of a heterogeneous distributed database management system: the distributed access view integrated database (DAVID).* Bharat Bhasker; Csaba J. Egyhazy; Konstantinos P. Triantis. CSC '92. Proceedings of the 1992 ACM Computer Science 20th annual conference on Communications, pages 173-179

[BURL 94] *Managin Distributed Databases. Building Bridges between Database Island.* Burleson, Donal. 1994

[DATE 94] *Introducción a los sistemas de Bases deDatos*. Date, C.J. Addison Wesley 1994.

[DIPA 99] *Un ambiente experimental para evaluación de Bases de Datos Distribuidas.* Di Paolo, Mónica; Bertone, Rodolfo; De Giusti, Armando. Paper presentado (aún no evaluado) en la ICIE 99. Fac. Ingeniería. UBA. Buenos Aires. Argentina

[LARS 95] *Database Directions. From relational to distributed, multimedia, and OO database Systems.* Larson, James. Prentice Hall. 1995

[MIAT 98] *Experiencias en el análisis de fallas en BDD.* Miaton, Ivana; Ruscuni, Sebastián; Bertone, Rodolfo; De Giusti, Armando. Anales CACIC 98. Neuquén Argentina.

[ÖZSU 91] *Principles of Distributed Database Systems.* Özsu, M. Tamer; Valduriez, Patric. Prentice Hall 1991.

[SCHU 94] *The Database Factory. Active databse for enterprise computing.* Schur, Stephen. 1994.

[SHET 90] *Federated database systems for managing distributed, heterogeneous, and autonomous databases.* Amit P. Sheth; James A. Larson. ACM Computing Surveys.Vol. 22, No. 3 (Sept. 1990), Pages 183-236

[SILB 98] *Fundamentos de las Bases de Datos.* Silbershatz; Folk. Mc Graw Hill. 1998.

[THOM 90] *Heterogeneous distributed database systems for production use.* Thomas, Charles; Glenn R. Thompson; Chin-Wan Chung; Edward Barkmeyer; Fred Carter; Marjorie Templeton; Stephen Fox; Berl Hartman. ACM Computing Surveys. Vol. 22, No. 3 (Sept. 1990), Pages 237-266

[UMAR 93] *Distributed Computing and Client Server Systems.* Umar, Amjad. Prentice Hall. 1993.

[WEB1] *www.seas.gwu.edu/faculty/shmuel/cs267/textbook/tpcp.html*

[WEB2] *www.sei.cmu.edu/str/descriptions/dtpc_body.html*

[WEB3] *www.datanetbbs.com.br/dclobato/textos/bddcs/parte2.html*

[WEB4] *www.itlibrary.com/library/1575211971/ch45.htm*

[WEB5] *www.itlibrary.com/library/1575211971/ch26.htm*

[WEB6] *java.sun.com/docs/books/tutorial/networking/sockets/definition.html*

[ZANC96] *Análisis de Replicación en Bases de Datos Distribuidas*, Marcelo Zanconi, Tesis de Magister en Ciencias de la Computación, Univ. Nac. Del Sur, Bahía Blanca, 1996