DISTRIBUTED COMPUTING RESEARCH AT PRINCETON - 1986

Robert Abbott, Rafael Alonso, Louis Cova,
Hector Garcia-Molina, Boris Kogan, Kriton Kyrimis,
Frank Pittelli, Patricia Simpson,
Annemarie Spauster, Kenneth Salem

CS-TR-073-87

January 1987

DISTRIBUTED COMPUTING RESEARCH AT PRINCETON — 1986

Robert Abbott, Rafael Alonso, Luis Cova, Hector Garcia-Molina, Boris Kogan, Kriton Kyrimis, Frank Pittelli, Patricia Simpson, Annemarie Spauster, Kenneth Salem

Department of Computer Science
Princeton University
Princeton, N.J. 08544

DISTRIBUTED COMPUTING RESEARCH AT PRINCETON — 1986

Robert Abbott, Rafael Alonso, Luis Cova, Hector Garcia-Molina, Boris Kogan, Kriton Kyrimis, Frank Pittelli, Patricia Simpson, Annemarie Spauster, Kenneth Salem

Department of Computer Science
Princeton University
Princeton, N.J. 08544

1. Introduction

In this note we briefly summarize the distributed computing research we performed in the year 1986. In general terms, our emphasis was on studying and implementing mechanisms for *efficient and reliable* computing and data management. Our work can be roughly divided into seven categories: the implementation of a highly reliable database system, the implementation of a high availability database system, dynamic vote reassignment, a mechanism for dealing with long-lived transactions, real time database systems, caching in information systems, and load balancing.

Due to space limitations, we concentrate on describing our own work and we do not survey the work of other researchers in the field. For survey information and references, we refer readers to some of our reports.

2. A Highly Reliable Database System

One of the goals of our research is to understand reliable data management, so a natural question to ask is: how reliable can we ever hope to build a data management system and at what cost? In an attempt to answer this question we implemented an ultra high reliability database system.

The key to our system is a failure model that captures the worst possible behavior by a computer. In this model we have a collection of $2n + 1$ computers where at most n of them can fail. We make no assumption as to how a computer fails, and in this sense the model is the most general possible. We call a failed node fail-insane since it can send any message out, including misleading ones, it can refuse to send required messages, and it can even collaborate with other fail-insane nodes in an attempt to subvert the entire system. We assume that an insane failure is eventually detected and repaired.

If we wish to reliably perform a task in this environment we must execute

This work has been supported by NSF Grants DMC-8351616 and DMC-8505194, New Jersey Governor's Commission on Science and Technology Contract 85-990660-6, and grants from DEC, IBM, NCR, and Concurrent Computer corporations.

the task on all $2n + 1$ computers and look at all outputs. The output that $n + 1$ or more nodes agree on must be the correct one. This approach is well known and is called N-modular redundancy (NMR). It has been successfully used at the hardware level for many years. In our system, however, we are using NMR at a different level, the database level.

There are a number of advantages to operating at this level. The communication overhead can be greatly reduced. Instead of exchanging and comparing the result of, say, every addition or memory reference, the system operates at a much higher level, mainly comparing the outputs of user transactions. Given the reduced communications, it is easier to physically isolate the processing elements. Decoupling makes it less likely for a single environmental problem like a power failure or a fire to affect more than n processors.

Since remote sites in essence have backup copies of the database, processing at each site may be streamlined, eliminating local logging and dumping, for example. Actually, the database processing at each site is quite simple. Note that nodes execute each transaction independently from the rest of the system. Locks do not have to be requested from other nodes and there can be no global deadlocks. In addition, our system can be built with off-the-shelf equipment, as opposed to the specialized hardware usually required by NMR.

So in light of these advantages, and of the high reliability provided, the cost of database NMR may not be as high as one may initially imagine. It will never be cheap (because of the replicated hardware), but it may be a desirable alternative for critical applications where lives or money are at stake, or where users are simply tired of dealing with "temperamental" computers that lose or destroy their data.

As far as we know, ours is the first implementation of a database NMR system. In building it we had to address a number of challenging problems. One problem is the distribution of the input transactions. For the NMR scheme to work, not only do all nodes have to execute exactly the same transactions, but they have to execute them in the same order. A theoretical solution to this problem was known (Byzantine Agreement). We selected the most appropriate materialization of this solution, and built in several practical optimizations like null transactions and message batching. In [Pitt86a] we describe our approach to transaction scheduling and present experimental results that illustrate the performance of the optimizations.

Unlike conventional NMR systems, our processors carry a significant amount of "state" information (i.e., the database). After a failure has been repaired, the failed node(s) must recover this state from the operational ones. Since we cannot allow the failed nodes to halt the entire system, this recovery must take place without stopping the operational nodes. In [Pitt86b] we present a strategy for such a recovery. The process starts when a repaired node requests a snapshot of the database from the other nodes. Using probabilistic signatures, the portions of the database that were corrupted are identified and copied. While this is in progress, the repaired node saves all new transactions. These are executed once a consistent database state has been recovered. After this catch-up period, the node is fully recovered. The performance of the

system during the snapshot installation and catch-up periods was experimentally studied; the results are also given in [Pitt86b].

Our experimental system has also been used to study the inherent cost of ultra-high reliability [Pitt86c]. In particular, we compared a 3-node NMR system against a two node system, a one node system with local crash recovery, and a one node system with no protection. The structure of each system is similar, but the reliability, hardware cost, and performance all differ. Each system in the list provides a lower and lower degree of reliability, and a lower and lower hardware cost. However, our experimental results indicate that the performance is not always inversely proportional to the reliability. For example, the 3 node system may perform better than the single node with local crash recovery system. In summary, our results provide a very concrete performance and cost comparison of various degrees of reliability.

3. A Highly Available Database System

There are two important aspects to reliability: correctness and availability. The objective of the project described in the previous section is mainly correctness of the database operations, even at a cost in availability. For instance, if one computer (or even a group of n computers) is cut off from the rest of the system, it will be unable to process transactions because this could compromise the correctness of the results and the database. On the other hand, if availability were the main objective, it may be desirable to continue processing at the isolated node. After all, it does have a copy of the database and is capable of processing transactions. Of course, if isolated nodes are allowed to modify the database, then the copies will diverge. Hence, we need mechanisms for controlling these inconsistencies or for correcting them later.

We have continued our research effort in high availability database systems, trying to understand the choices and their implications. Since replicated data is the key to availability, we assume that the database (or at least the critical portion) is replicated at all nodes. We assume that nodes, when up, compute correctly (i.e., no arbitrary failures). However, nodes can be arbitrarily slow in responding to requests from other nodes. Furthermore, the communication network may lose messages and may even isolate some nodes totally (network partition). Given the undependability of other nodes, our view is that each node should be as *autonomous* as possible. A node should never get itself in a position where it cannot process transactions due to a failure of another node or of the communication network.

We believe that our autonomous node model is desirable in several applications. In some cases it is because communication failures occur at critical times. For instance, in military applications a conflict can make communications unreliable and it is precisely at this time that one wishes to have access to the data. In other cases, partitions are the normal mode of operation. For example, in the Unix UUCP network computers are usually disconnected from the network. Periodically, they dial other computers and exchange data. In such an environment one would like to process transactions even when there is no communication. In yet other cases, partitions are not as common, but

halting transaction processing causes a serious inconvenience or economic loss. For example, an unavailable airline reservations system means lost customers.

In the past we have studied strategies where nodes can indeed process any type of transaction any time they wish [Alon85, Abbo86a]. Inconsistencies among the copies were corrected using semantic knowledge when communications were restored. This approach works when the application is simple, but is harder to use as the applications semantics get more complicated.

We are currently exploring a slightly different approach: restrict the types of transactions that a node can execute in order to simplify the inconsistencies that can arise. A node can continue to process the transactions of its allowed types any time it wishes. To define the types of transactions that a node can run, we divide the database into fragments and introduce a controlling *agent* for each [Garc87]. A node can only run transaction that update a fragment if the agent resides at the node. By varying the way in which agents move, we can vary the availability provided. At the same time we can control the types of inconsistencies that arise, making it possible to have meaningful data in the database.

We have discovered that in some cases it is possible to have serializable schedules even with autonomous nodes. (If the execution schedule is serializable, then there are no anomalies or inconsistencies in the data.) To achieve this, transactions must have certain properties and updates must be propagated to the copies in a controlled fashion [Barb87]. Not all applications fall in this category; however, it may be possible to force the application to satisfy the properties in order to achieve serializability.

In summary, our new approaches give "controlled availability." They provide substantially more availability than conventional mechanisms, but they do restrict the types of operations that a node or a user may perform. However, through proper database design, a node or a user may be assigned precisely the transactions that he wishes to run. (It is rare that a user or nodes needs to run all possible transactions.) Thus, users (or at least many of them) will be able to perform the operations they want to whenever they want to.

4. Dynamic Vote Reassignment

A number of distributed algorithms require that at most one connected group of nodes be active at a time. This restriction can be enforced through a voting mechanism. Each node is assigned a number of votes; the group that has a majority of the votes knows that no other group can have a majority and thus can be active. (It is also possible that no group has a majority.) Such a mechanism can be used to elect a distinguished node, say to be a file server. Voting ensures that at most one node is elected, i.e., only the group with a majority of votes is allowed to have or to elect a file server.

To illustrate, consider a system with four nodes a, b, c, and d. Say we have initially assigned the votes $v_a = v_b = v_c = 1$ and $v_d = 2$, where v_i represents the votes assigned to node i. If a partition forms two isolated groups, $\{a, b, c\}$ and $\{d\}$, then only the first group will be active (e.g., have a

file server). (Nodes a, b and c have 3 out of 5 possible votes.) However, if a second partition occurs, separating node c from a and b, the system will be halted, i.e., no group will have a majority and no group will be active.

Since halted states are undesirable, we propose to reassign the votes after failures or repair occur. That is, after any failure or repair, the majority group (if any) dynamically reassigns the votes in order to increase its voting power and increase the system's chances of surviving subsequent failures. In our example, we can reduce the likelihood of halting if we increase the votes of group $\{a, b, c\}$ before the second partition occurs. For instance, a new vote assignment could be $v_a = v_b = v_c = 5$. Node d is unaware of the change and remains with $v_d = 2$ votes. (As a matter of fact, since d is not in the majority group it *cannot* change its votes.) In this way, the second partition described above will find nodes a and b with 10 votes out of a total of 17, forming a majority group that can continue to be active.

After the second partition, the new majority group $\{a, b\}$ could reassign itself new votes of $v_a = 15$ and $v_b = 5$ in order to tolerate even a third partition. When the partitions are repaired, the nodes that have proportionately less votes (e.g., d) can attempt to increase their votes. Equivalently, nodes that have increased their votes may decrease them to recapture the original assignment.

Notice that in our approach nodes operate quite autonomously, without requiring group consensus. Group consensus could select a better assignment but requires tighter coordination among the nodes. On the other hand, autonomous vote changes are much simpler and more flexible. Each node decides independently what its new vote value should be. The node does not need complete or accurate information about the state of the system. In a sense, the node makes an educated guess about the best number of votes to have, with its primary goal being to claim for itself all or part of the voting power of a node (or nodes) that have been separated from it.

There are two problems to solve in implementing the ideas we have sketched. The first involves the selection of a new vote value by a node that has detected the disconnection of another node. We refer to the mechanism for such a selection as the *policy*. The policy takes into account the current state of the system (who is up, who is down, who has how many votes), as determined (perhaps not accurately) by the node making the change and outputs the desired votes for this node. There are a number of different policy rules and these are surveyed in [Barb86a].

Once a node chooses a new value, it cannot vote with it right away. The second problem is to ensure that the node is part of a majority and authorized to make the change. The mechanism to do this is called the *protocol*. In [Barb86b] we describe a simple one-phase protocol that works as long as nodes only increase their votes. The basic idea is to collect acknowledgments from nodes with a majority of votes and in the process inform them of the change. In doing so, we must make sure the protocol does not get confused between the "old" and the "new" votes. Since nodes only increase their votes, the largest known vote value for a node represents the latest one. This makes it easy for

the protocol to identify "new" votes.

This protocol can be extended to handle decreasing as well as increasing vote changes [Barb86c]. Each vote value exchanged among the nodes must now be accompanied by a sequence number that represents the relative order of the votes a node has had. Although the protocol is not as simple, it still remains one-phase.

5. Sagas

As its name indicates, a *long lived transaction* is a transaction whose execution, even without interference from other transactions, takes a substantial amount of time, possibly on the order of hours or days. A long lived transaction, or LLT, has a long duration compared to the majority of other transactions either because it accesses many database objects, it has lengthy computations, it pauses for inputs from the users, or a combination of these factors. Examples of LLTs are transactions to produce monthly account statements at a bank, transactions to process claims at an insurance company, and transactions to collect statistics over an entire database.

In most cases, LLTs present serious performance problems. Since they are transactions, the system must execute them as atomic actions, thus preserving the consistency of the database. To make a transaction atomic, the system usually locks the objects accessed by the transaction until it commits, and this typically occurs at the end of the transaction. As a consequence, other transactions wishing to access the LLT's objects are delayed for a substantial amount of time. Furthermore, LLT have a high probability of encountering a deadlock or a system failure.

In general there is no solution that eliminates the problems of LLTs. However, for *specific applications* it may be possible to alleviate the problems by relaxing the requirement that an LLT be executed as an atomic action. In other words, without sacrificing the consistency of the database, it may be possible for certain LLTs to release their resources before they complete, thus permitting other waiting transactions to proceed.

To illustrate this idea, consider an airline reservation application. The database (or actually a collection of databases from different airlines) contains reservations for flights, and a transaction T wishes to make a number of reservations. For this discussion, let us assume that T is a LLT (say it pauses for customer input after each reservation). In this application it may not be necessary for T to hold on to all of its resources until it completes. For instance, after T reserves a seat on flight F_1, it could immediately allow other transactions to reserve seats on the same flight. In other words, we can view T as a collection of "sub-transactions" T_1, T_2, ..., T_n that reserve the individual seats.

However, we do not wish to submit T to the database management system (DBMS) simply as a collection of independent transactions because we still want T to be a unit that is either successfully completed or not done at all. We would not be satisfied with a DBMS that would allow T to reserve three out of five seats and then (due to a crash) do nothing more. On the other hand, we

would be satisfied with a DBMS that guaranteed that T would make all of its reservations, or would cancel any reservations made if T had to be suspended.

This example shows that a control mechanism that is less rigid than the conventional atomic-transaction ones but still offers some guarantees regarding the execution of the components of an LLT would be useful. We call a LLT that can be processed in this way a *saga* and in [Garc86b] we discuss how they can be executed by a database management system.

The basic idea is that a saga can be broken up into a collection of sub-transactions that can be interleaved in any way with other transactions. To amend partial executions, each saga transaction T_i should be provided with a compensating transaction C_i. The compensating transaction undoes, from a semantic point of view, any of the actions performed by T_i, but does not necessarily return the database to the state that existed when the execution of T_i began. In our airline example, if T_i reserves a seat on a flight, then C_i can cancel the reservation (say by subtracting one from the number of reservations and performing some other checks). But C_i cannot simply store in the database the number of seats that existed when T_i ran because other transactions could have run between the time T_i reserved the seat and C_i canceled the reservation, and could have changed the number of reservations for this flight.

In [Garc86b] we also study various implementation issues related to sagas, including how compensating transactions can be defined, how sagas can be aborted, and how they can be run on a system that does not support them directly. In addition we discuss some strategies that an application programmer may follow in order to write LLTs that are indeed sagas and can take advantage of our proposed mechanism.

6. Real Time Database Processing

Existing database management systems do not provide real time services. They process transactions as quickly as they can, but they never make any guarantees as to when a request will complete. Furthermore, most users cannot even tell the system what priority their request has. Hence, all transactions are treated as equal.

Many applications do have at the same time real time constraints and large data needs (e.g., aircraft tracking, hospital monitoring, reservations systems). Since database systems do not provide the required real time response, users have had to code their own special purpose data management systems. Although such systems seem to work, they are difficult to debug and to expand. Thus we believe it is time to investigate a general purpose real-time database system.

Such a system may very well be distributed, but we have decided to focus initially on a centralized one. The first problem we have addressed in this area is that of transaction scheduling [Abbo86b]. In the future we plan to study additional issues like the general architecture, how to trigger events efficiently, and the appropriate user interface.

Real time transaction scheduling differs from conventional scheduling in

that the transactions (or tasks) make *unpredictable* resource requests, mainly requests to read or write the database. In our case, the scheduling algorithm must be combined with the concurrency control algorithm (which guarantees that executions are serializable). To illustrate the interaction, consider a transaction T_1 that is being executed because its deadline is the nearest. Now assume that T_1 requests a lock that is held by transaction T_2. What should the system do? Abort T_2 so that the lock is released and T_1 can proceed? Or maybe suspend T_1 so that T_2 can complete and release its lock? Or maybe it is best to let T_1 proceed without aborting T_2, hoping that the schedule will still be serializable (optimistic control)?

To understand the choices we have classified the possible scheduling and concurrency control strategies [Abbo86b]. We have also studied the deadline or priority models that make most sense for a database system. Our next step is a detailed simulation. With it we expect to answer some of the following questions:

(a) What is the best scheduling/concurrency control mechanism for real-time database processing?

(b) Many real-time systems hold their data entirely in main memory. How are the scheduling decisions affected by this? In particular, transactions never wait for IO so scheduling decisions are made less often. Also, the cost of aborting a transaction may be less in this case.

(c) Many scheduling algorithms utilize a run-time estimate provided by the user. How do errors in this estimate affect scheduling? At what point is it better to ignore the estimates all together?

(d) There are various ways in which a user can specify the priority or deadline of a transaction. Are there scheduling strategies that are better suited to handling particular user constraints?

7. Data Caching in Information Retrieval Systems

Existing computer communication networks give users access to an ever growing number of information retrieval systems (IRS). Some of these services are provided by commercial enterprises (examples are Dow Jones and The Source), while others are research efforts (such as the Boston Community Information System). In many cases these systems are accessed from personal or medium size computers which usually have available sizable amounts of local storage. Thus, to improve the response time of user queries it becomes desirable to cache data at the user's site.

Caching can improve system performance in two ways. First, it can eliminate multiple requests for the same data. For example, consider an automobile manufacturing plant where a number of people are interested in news wire stories on trade and protectionism. In this case, it makes sense to cache the relevant articles at the company's local computer, eliminating redundant requests to the central IRS site. A second way in which caching can improve performance is by off loading work to the remote sites. For instance, if a user is interested in chemical companies he may store the latest stock prices of those

companies at his own computer. There he can run his own analysis programs on the data, without using any more central cycles.

Our work in this area has proceeded along two lines of inquiry. In one study, we are investigating the design of caches for IRS's in which updates are done only at the central machine and the frequency of changes is not too great. Secondly, we are also considering the possible performance benefits of relaxing cache coherency constraints in systems whose update pattern makes normal caching impractical. We explain further these two approaches in the subsections below.

7.1. Cache Design

Although well understood in the context of hardware memory caches, caching in an IRS environment has not been fully studied. We believe that there are several significant differences between standard memory caching and the kind of caching we are exploring. For example, since users do not perform updates, there is no need to keep track of changes and write modified data items back to the central site; when a cached item is to be replaced by another, it can simply be discarded (overwritten). On the other hand, in our case keeping a cache up to date is much more difficult because updates originate at the central site, not at the site holding the cache.

There are many issues to be studied in this context. For example, there are data conversion problems when moving data between the IRS mainframe and the user workstations. Software at the user site must understand the format of cached data and be capable of accessing and manipulating it. There are also choices to be made in the type of storage areas available for caching (i.e., should the cached data be placed in main memory or a disk, or perhaps on both). Finally, caching in IRS's is not constrained by the tight timing constraints of hardware caching (because of the latency of the communication lines); this (coupled with the fact that the items being cached may consist of many kilobytes of data) means that sophisticated caching strategies may be employed. (For a more detailed description of the problems in this area see [Simp86].)

Initially, we are focusing on the performance of caching schemes. We have developed simulation models to explore the effect of different parameters (such as workstation architecture, size of available caches, and communication bandwidth available) on the mean response time obtained under a variety of caching strategies. Our initial results indicate that, for users accessing data via relatively slow modems (300-1200 baud), caching will carry with it substantial overhead (mostly because it is expensive to fill the cache from the mainframe); a high hit ratio must be achieved if caching is to prove worthwhile. We are currently studying policies that restrict caching in some way (for example, by caching an item only after it has been accessed twice, thus taking better advantage of reference locality). We are also experimenting with strategies that cache only long-lived items (i.e., those that are accessed frequently enough that they should always be in the cache), such as the top levels of tree-structured menus, or reference works such as dictionaries or telephone directories.

7.2. Quasi-Copies

In principle, caching can off load work from the central site and reduce the communication traffic. However, caching has its price. Every time a cached value is updated at the central IRS, the new value must be propagated to the copies. Furthermore, the propagation must be done immediately if cache consistency or coherency is to be preserved. (A cached value for an object is consistent if it equals the value of the object at the central site.)

To reduce the overhead of maintaining multiple copies it may be appropriate to allow copies to diverge in a controlled fashion. This makes it possible to propagate updates to the copies efficiently, e.g., when the system is lightly loaded, when communication tariffs are lower, or by batching together updates. It also makes it possible to access the copies even when the communication lines or the central IRS are down. To illustrate, consider a user that is interested in the stock prices of chemical companies. The user may be satisfied if the prices at his computer are within five percent of the true prices. This makes it unnecessary to update the cached copy every single time a change occurs. When the deviation exceeds five percent, then a single update can bring the cached copy up-to-date. At the manufacturing company discussed earlier, users may tolerate a delay of one day in receiving the articles of interest. If the system takes advantage of this, it can transmit all the articles during the night when communication tariffs are lower. If a communication or central node failure occurs and its duration is less than 24 hours, then users can continue to access information that is correct by their standards.

We call a cached value that is allowed to deviate in a controlled fashion a *quasi-copy*. In [Garc86a] we study this notion in detail, suggesting various ways in which users can specify the allowed deviations. (The five percent numeric deviation and the one day delay in our examples are two ways in which this can be done.)

In the report we also study the available implementation strategies. For example, when the central database site wishes to inform remote sites of an update, it has several choices. It can send the new value. It can send an invalidation message that forces the old value out of the caches (but does not provide the new value). Or it can use implicit invalidation or *aging*. In this last case the original value is sent out with an expiration time. When that time arrives, the value is automatically purged from the cache.

There are also several choices regarding the time to send the update or invalidation. The update can be propagated as soon as it is installed, or at the last minute when the quasi-copy is about to exceed its divergence limit, or at some intermediate time. The tradeoffs related to these and other implementation strategies are discussed in the report.

8. Load Balancing

In many of today's computing environments, it is not uncommon to see a mix of idle and overloaded machines on the same network. This is specially true in local area networks of workstations, where users may use their machines

only sporadically. It is also the situation in systems where the workload requirements have a large variance throughout the day. This situation of load imbalance leads to a needless degradation in system throughput and to a large increase in mean response time. Although users may realize that there are cycles available elsewhere and individually execute their jobs remotely, we feel that there is a need for mechanisms that automatically perform this task.

Our work has consisted of implementing a load balancing mechanism [Alon86a] that runs on a local area network of SUN workstations. The software consists of a set of cooperating daemons that periodically transmit load information, and local shell programs that use that information to decide on the appropriate execution site for user jobs. Although we have used only very simple load balancing schemes, our measurements show that, even under conditions of relatively small load imbalance, sizable performance gains can be achieved, and that the overhead involved in running our system is very small (for both users and non-users of our mechanism). (See [Alon86b] for an analysis of the desirable properties of more complex load balancing strategies.)

Currently we are addressing two problems in this area. The first is that, since load balancing decisions are being made on the basis of broadcast load data, that data may be stale by the time the decisions have to be made. For example, if a machine is completely idle, all the other processors may decide to send their jobs to the idle processor. Very soon, that "victim" machine is overloaded, but if the broadcast interval is large, other machines will not find out about the change in load. (It should be noted that the frequency of broadcasts cannot be increased arbitrarily since there is a cost involved in obtaining, sending and receiving data.) We have explored a variety of techniques to minimize problems with stale data [Alon86c]. For example, to eliminate the "victim" problem described above, a machine can set up a high-load mark; if its load ever exceeds that level, it will refuse to execute any remote jobs.

A second area of research is motivated by the following observation. It seems to us that there are really two environments in which load balancing may be of use. One, a network where there are a number of machines owned by a single entity; there, it is desirable that load should be evenly balanced across all the machines. Two, a network of workstations, where individual users own their machines; while those owners may not mind that someone else is "borrowing" a few cycles while they are not fully utilizing their processors, they certainly are not willing to see their own response suffer in order to help the overall system response time. We are currently developing policies for both of these environments, and identifying techniques that can be used in systems that consist of a mix of the two environments [Alon86d].

9. Completed Thesis

Over the past year, one PhD thesis was completed as part of our project. Frank Pittelli's thesis [Pitt86c] deals with the highly reliable database system summarized in Section 2.

10. References

[Abbo86a] R. Abbott et al, "Distributed Computing Research at Princeton — 1985," Technical Report CS-29, Department of Computer Science, Princeton University, 1985.

[Abbo86b] R. Abbott, H. Garcia-Molina, "Real-Time Database Scheduling," in preparation, 1986.

[Alon85] R. Alonso et al, "Distributed Computing Research at Princeton (1984)," *IEEE Bulletin on Database Engineering*, Vol. 8, Num. 2, June 1985, pp. 68-75.

[Alon86a] R. Alonso, P. Goldman, P. Potrebic, "A Load Balancing Implementation for a Local Area Network of Workstations," *Proceedings of the IEEE Workstation Technology and Systems Conference*, March 18-20, 1986, Atlantic City, N.J., pp. 118-124.

[Alon86b] R. Alonso, L. Cova, "Load Balancing in Two Types of Computational Environments," to be submitted for publication.

[Alon86c] R. Alonso, P. Goldman, P. Potrebic, "An Experimental Study of Load Balancing Strategies," to be submitted for publication.

[Alon86d] R. Alonso, "The Design of Load Balancing Strategies for Distributed Systems," *Proceedings of the Army Research Office Future Directions in Computer Architecture and Software Workshop*, Seabrook Island, North Carolina, May 5-7, 1986.

[Barb86a] D. Barbara, H. Garcia-Molina, A. Spauster, "Policies for Dynamic Vote Reassignment," *Proc. 1986 International Conference on Distributed Computing Systems*, Cambridge, Massachusetts, May 1986, pp. 37-44.

[Barb86b] D. Barbara, H. Garcia-Molina, A. Spauster, "Protocols for Dynamic Vote Reassignment," *Proc. Fifth ACM Symposium on Principles of Distributed Computing*, Calgary, Canada, May 1986, pp. 195-205.

[Barb86c] D. Barbara, H. Garcia-Molina, A. Spauster, "Increasing Availability Under Mutual Exclusion Constraints with Dynamic Vote Reassignment," Technical Report CS-TR-056-86, Department of Computer Science, Princeton University, November 1986.

[Barb87] D. Barbara, H. Garcia-Molina, B. Kogan, "Maintaining Availability of Replicated Data in a Dynamic Failure Environment," *Proc. Sixth Symposium on Reliability in Distributed Software and Database Systems*, 1987, to appear.

[Garc86a] H. Garcia-Molina, R. Alonso, D. Barbara, S. Abad, "Data Caching in an Information Retrieval System," Technical Report CS-TR-065-86, Department of Computer Science, Princeton University, December 1986.

[Garc86b] H. Garcia-Molina, K. Salem, "Sagas," Technical Report CS-TR-070-86, Department of Computer Science, Princeton University, December 1986.

[Garc87] H. Garcia-Molina, B. Kogan, "Achieving High Availability in Distributed Databases," *Proc. Third International Conference on Data Engineering*, February 1987, to appear.

[Pitt86a] F. Pittelli, H. Garcia-Molina, "Reliable Scheduling in a TMR Database System," Technical Report CS-TR-028-86, Department of Computer Science, Princeton University, March 1986.

[Pitt86b] F. Pittelli, H. Garcia-Molina, "Recovery in a Triple Modular Redundancy Database System," submitted for publication.

[Pitt86c] F. Pittelli, "Experimental Analysis of a Triple Modular Redundant Database System," PhD Thesis, Department of Computer Science, Princeton University, October 1986.

[Simp86] Patricia Simpson, Rafael Alonso, "Data Caching in Information Retrieval Systems," submitted for publication.